DOT/FAA/TC-19/22

Federal Aviation Administration
William J. Hughes Technical Center
Aviation Research Division
Atlantic City International Airport
New Jersey 08405

Use of Virtual Machines in Avionics Systems and Assurance Concerns

October 2019

Final Report

U.S. Department of Transportation
Federal Aviation Administration

Technical Report Documentation Page

1. Report No. DOT/FAA/TC-19/22	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle USE OF VIRTUAL MACHINES IN AVIONICS SYSTEMS AND ASSURANCE CONCERNS		5. Report Date October 2019	
		6. Performing Organization Code	
7. Author(s) Bjorn Andersson, Sagar Chaki, Dionisio de Niz		8. Performing Organization Report No.	
9. Performing Organization Name and Address Software Solutions Division Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		10. Work Unit No. (TRAIS)	
		11. Contract or Grant No. DFACT-14-X-00010	
12. Sponsoring Agency Name and Address Federal Aviation Administration 950 L'Enfant Plaza Washington, DC 20024		13. Type of Report and Period Covered Final Report	
		14. Sponsoring Agency Code AIR-6B4	
15. Supplementary Notes The FAA William J. Hughes Technical Center Aviation Research Division Technical Monitor was Srini Mandalapu.			

16. Abstract

Virtual Machine (VM) technology uses a hypervisor that creates multiple virtual copies of a computer, each to be used by a single application as if it was the only one running. This virtual copy, known as a VM, must be isolated from the other VMs. The industry is considering VMs and hypervisors for efficiency and flexibility of computing resources. For the certification of avionics systems, this isolation is a valuable asset that potentially allows modularity of certification and recertification. As a result, this study evaluated current VM technology and similar technologies as they relate to the assurance of avionics systems.

The research was focused on assurance issues, verification in particular, in which virtualization technologies are implemented in avionics systems. Verification of new technologies is challenging to certification authorities. To investigate this aspect of VM, the research studied verification technologies for VMs. This study was divided into timing verification and logical verification. From the timing perspective, multiple verification techniques were proposed that are generic enough to model virtualization; the tradeoffs among them were documented. From the logical verification side, different techniques that require different degrees of human involvement, from fully automated (e.g., model checking) to more interactive (e.g., theorem provers), were documented. The research also highlighted the assurance data that can be affected by the porting of an application from single to multicore processors and the use of hardware emulation to try to preserve the behavior of the original hardware.

Virtualization in software architecture is an increasingly common practice in the software industry, and it is not surprising there is interest for airborne systems. This research provided several recommendations and conclusions on the use of VMs in airborne systems. The research results will be used in developing guidance and training material for the certification engineers. The research also recommends additional research on isolation and verification techniques for improved modular recertification and fostering corresponding standards, solutions for the impact of multicore on virtualization technology, and verification schemes for virtualization implementations.

17. Key Words Avionics systems, Virtual machines, Virtualization, Safety assurance, Hypervisor, Verification technologies, Worst-case timing		18. Distribution Statement This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 206	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorize

ACKNOWLEDGEMENTS

The SEI team thanks the FAA team, in particular Srini Mandalapu, for feedback on this report.

TABLE OF CONTENTS

LIST OF ACRONYMS

API	Application programming interface
BVT	Borrowed-Virtual-Time
CARTS	Compositional Analysis of Real-Time Systems
CFG	Control flow graph
CMAS	Certifiable Multicore Avionics Systems
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CRPD	Cache-related preemption delay
CS	Critical section
DAG	Directed acyclic graph
DM	Deadline monotonic
DMA	Direct memory access
DMS	Deadline monotonic scheduling
DRAM	Dynamic random-access memory
DTM	Dynamic thermal management
EDF	Earliest-Deadline First
EVM	Embedded virtual machine
EVT	Extreme value theory
EX	Execution
FIFO	First-in, first-out
FPS	Frame per second
FR-FCFS	First-ready, first-come, first-served
ID	Instruction Decoding
IF	Instruction Fetch
IMA	Integrated modular avionics
I/O	Input/output
IOMMU	Input/output memory management unit
IPC	Instructions per cycle
LCM	Least common multiple
MC/DC	Modified condition/decision coverage
MEM	Memory Access
MILS	Multiple Independent Levels of Security/Safety
MMU	Memory management unit
MPCP	Multiprocessor Priority Ceiling Protocol
MPSoC	Multiprocessor systems on chips
NOP	No operation
OS	Operating system
PALS	Physically Asynchronous Logically Synchronous
PCP	Priority Ceiling Protocol
PI	Priority Inheritance
PSO	Partial Store Order
QoS	Quality of service
rbf	Request bound function
RM	Rate monotonic
RMS	Rate-monotonic scheduling

RTES	Real-time embedded systems
RTOS	Real-time operating systems
RTV	Real-time virtualization
SAT	Satisfiability
sbf	Supply-bound function
SC	Sequential consistency
sEDF	Simple Earliest-Deadline First
SEU	Single event upset
SLA	Service-level agreement
SMT	Satisfiability modulo theories
SRAM	Scratchpad RAM
TD	Time division
TDM	Time-division multiplexing
TDM+FP	Time-division multiplexing plus fixed-priority scheduling
TLB	Translation lookaside buffer
TSO	Total Store Order
TTA	Time-triggered architecture
UDR	Utility degradation resilience
V&V	Verification validation
VC	Verification condition
VM	Virtual machine
vMCP	Vector Mixed-Criticality Packing
VMM	Virtual machine monitor
WB	Result Write Back
WCET	Worst-case execution time
WCRT	Worst-case response time
ZSRM	Zero-slack rate-monotonic

EXECUTIVE SUMMARY

Virtual machine (VM) technology uses a hypervisor that creates multiple virtual copies of a computer, each to be used by a single application as if it were the only one running. This virtual copy, known as a VM, must be isolated from other VMs. For the certification of avionics systems, this isolation is an invaluable asset that can potentially allow modularity of certification and recertification. This report evaluates current VM technology and similar technologies as they relate to the assurance of avionics systems.

This report is divided into seven sections. The first section provides a literature survey that tracks the origin of virtualization and explores related isolation techniques and verification technologies. This survey tracks the different motivations and goals of the virtualization technologies. In particular, the survey identifies how the initial virtualization objectives were focused on simplicity of development and use with a strong emphasis on throughput. Clearly, throughput has been a key driver of general-purpose computing but hinders the worst-case (timing) behavior needed to provide assurance of real-time systems. The section presents an alternative isolation mechanism focused on real-time behavior.

The second section on assurance issues discusses specific issues of assurance when virtualization technologies are used in avionics systems. In particular, the section discusses how the goals of assurance of avionics systems relate to different virtualization technologies and how different innovations in the hardware layers interact with virtualization. It also discusses how recent innovations in real-time systems deal with hardware innovations to ensure predictable timing behavior.

The third section focuses on verification technologies for VMs. This section is divided into timing and logical verification. For timing, the study presents multiple verification techniques that are generic enough to use to model virtualization and discuss the tradeoffs among them. For logical verification, it presents different techniques that require different levels of user involvement, from fully automated techniques, such as model checking, to more interactive techniques, such as theorem provers.

The fourth section discusses compositional techniques for verification. This section enhances the discussion of the previous section with a more focused discussion on compositionality and modularity aspects. In particular, it discusses the brittleness of the modules, the interfaces that these techniques offer, and their effect on modular certification.

The next two sections discuss the portability of assurance data from single to multicore, and the emulation of hardware. These sections are closely related. The first section highlights the assurance data that can be affected by porting an application from single to multicore processors; the second section covers the use of emulation hardware to try to preserve the behavior of the original hardware.

In the final section, conclusions and recommendations are presented. The section starts by presenting gaps in two main research areas: 1) missed opportunities in the modularity aspects to improve recertification, and 2) the challenges that multicore processors present to virtualization given that hardware shared across cores creates delays that break current isolation techniques. For

spatial virtualization, one of the key missed opportunities is the lack of verified hypervisor implementations that can support a verifiable isolation. As a result, our main recommendations are: 1) additional research on more modular isolation and verification techniques, and developing the corresponding standards; 2) additional research on solutions for the impact of multicore in virtualization technology; and 3) research on additional implementation-verification schemes to support verified isolation for full virtualization.

1. SURVEY OF LITERATURE RELATED TO VIRTUAL MACHINES

This report surveys related work on virtual machines (VMs) with the most common implementations present today. It complements the assurance issues document that discusses at length the implications of using VMs for the assurance of avionics systems (see appendix B).

This document starts by introducing the history of VMs dating back to the 1960s and defines the initial principles and goals. This is followed by reviewing the resurgence of VMs in the late 1990s, which continues today. The document then moves into a discussion of the products and systems that are available today with their main characteristics. The product discussion briefly highlights only the differences and points out the related concepts previously introduced. A more elaborate discussion will be presented in the assurance issues document.

Section 4 covers technology related to virtualization. This section discusses other technologies that also have the same or similar goals as virtualization: namely, to create some form of partition by which applications can execute in an isolated manner. In particular, two forms of virtualization are discussed: 1) temporal virtualization in section 4.1, and 2) spatial virtualization in section 4.2. Under temporal virtualization, a brief introduction on background in rate-monotonic scheduling is given to discuss processing servers that have recently been integrated in VMs aimed at providing real-time guarantees. Similarly, variations of resource reservation that are being investigated for mixed-criticality scheduling are presented, followed by a brief discussion of more traditional forms of temporal partitioning (virtualization) based on time slots.

Under spatial virtualization, section 4.2 discusses different variations of spatial protection that are traditionally implemented as variants of kernels. The discussion includes the traditional monolithic kernel, the microkernel, and two kernels that are more specialized: the security kernel and the separation kernel. All of this is put into the perspective of today’s VMs in section 4.2.5. This is followed by a brief discussion of technologies for development partitioning that, although not providing full virtualization, help the developer to isolate its work and the configuration of its system. This is discussed in section 4.3.

To understand the impact on certification that VMs may have, there is a section on analytic technologies that covers temporal analysis and logical analysis. The overview of temporal analysis focuses on the technology related to analysis of systems that use VMs. Each of the works in this section involve modifications to the mechanisms to make the timing behavior predictable and analyzable. The overview of logical analysis starts with testing, then moves to exhaustive verification with techniques like model checking and theorem proving, and discusses the related tools. Finally, section 5 highlights some implications related to safety standards, in particular to DO-178B/C.

2. ASSURANCE ISSUES ON VIRTUAL MACHINES IN AVIONICS SYSTEMS

Virtualization is an old concept originating in the early days of mainframe computing to hide the physical details of the hardware platform from applications. It has regained popularity recently with the advent of cloud computing, which essentially turned mainframe computing from a concept of physically centralized computing to virtualized computing. The original benefits remain. New benefits include relief from the maintenance and operational responsibilities associated with owning and managing a data center.

Advances in virtualization technology have led to the development of efficient hypervisors and minimal VMs. This has led to considering VMs as vehicles for deploying services that require different operating environments on a single platform managed by a hypervisor. This approach can enable security, interoperability, evolvability, and modifiability in many types of systems. However, their use in safety- and timing-critical environments, such as avionics systems, has not been fully studied.

Virtualization provides both a computing platform that mimics specific hardware and operating system (OS) platforms and isolates applications operating within a VM from affecting the VM or any other application under the control of the VM. The question for safety- and timing-critical systems is whether the isolation is strong enough that applications executing within a VM have suitably predictable behavior. Specifically, VMs need to allow for making strong guarantees about applications' timing and spatial access correctness.

With this in mind, this report discusses the key concepts of the structure and function of VMs and virtualization technology, identifies the characteristics of VMs and virtualization that support and are inimical to achieving timing and access correctness, and exposes the benefits and challenges that VMs and virtualization pose for certifying systems.

This report is organized as follows: Section 1 introduces VMs. Section 2 discusses the goals of VMs and the properties they exhibit in terms of isolation and partitioning with respect to different hardware resources. It goes particularly deep into how different innovations at different levels of hardware work together to improve throughput but may hinder worst-case response time (key for real-time systems). It then discusses the adaptations and analysis innovations that the real-time community has developed to make the hardware innovations predictable and allow it to build temporal isolation on top. Section 3 discusses the additional certification complexity that the use of VMs can incur, from both the logical and the timing correctness points of view. Section 4 compares the isolation characteristics provided by VMs with other technologies to explain VMs' advantages and limitations. The development process issues are discussed in section 5, highlighting how the partitioned development environment can help distribute development evenly across organizations. Section 6 then puts all the previous discussion within the context of certification standards. Section 7 presents conclusions.

3. EVALUATION OF VERIFICATION TECHNOLOGIES FOR VIRTUAL MACHINES

In this report, we evaluate verification technologies for VMs. These technologies cover both timing and logical verification. This report complements the "Assurance Issues on VMs in Avionics Systems" report (appendix B). As such, it does not include technologies already included in that report, such as rate-monotonic scheduling, but instead focuses on technology not previously discussed. Because only a few of the current verification technologies have been tailored to VMs, we take a wider approach, discuss general techniques, and highlight adaptations or potential uses for VMs where appropriate.

This report is divided into sections: “Timing Verification” and “Logical Verification.” The timing verification section covers real-time calculus, timed automata-based analysis, and worst-case execution time (WCET) analysis. The WCET analysis subsection is further subdivided into measurement-based analysis and model-based analysis to present methods suited for different confidence levels, like the ones related to DO-178B/C Software Levels.

The logical verification section starts with techniques that require more user involvement, such as testing and theorem proving, and moves into techniques that are fully automated, like model checking. A subsection of abstract interpretation is included in both the timing and logical verification sections, but the former is tailored to bound WCET, whereas the latter is tailored to verifying logical properties.

For each of the techniques covered in this report, we discuss their benefits, limitations, and applicability using examples where deemed helpful.

4. COMPOSITIONAL VERIFICATION FOR VIRTUAL MACHINES

Virtualization offers the possibility of isolating components and potentially verifying them independently from one another. Certification standards, like DO-178C, require such an isolation to allow the verification of an individual component without the need to recertify all components. However, the isolation among components offered by virtualization is not absolute, either because they need to interact with each other or because they share hardware resources. This means that users of these techniques for verification must be aware of the interactions across components. The research community calls this type of verification compositional verification.

Compositional verification defines components whose behavior can both be affected by other components and affect other components only through an interface. With this definition, the verification process is decomposed into two parts. First, components are verified by taking into account their interactions with other components observable through the interface. Second, the whole system is verified as a collection of components connected through their interfaces, whose behavior is limited to what is observable through the interface. These interfaces are influenced by both the verification technology and the mechanisms that restrict the behavior, such as a VM and its hypervisor.

In this report, we study the compositional technologies that support the use of different types of virtualization to enable independent component verification both from the timing and logical perspectives. Section 2 discusses virtualization interfaces from both the perspective of timing and functional correctness. Because timing is changed as a result of virtualization, we emphasize timing in our discussion. Section 3 presents mitigation strategies. Section 4 gives recommendations.

5. SINGLE-TO-MULTICORE PORTABILITY OF ASSURANCE DATA

The use of multicore processors in avionics has received increasing interest. There are two reasons for this: 1) commercial availability and 2) performance. With respect to commercial availability, single-core chips are simply not available from many chip vendors; if buying processor chips from such a vendor, then a multicore chip is the only option. With respect to performance, multicore processors offer advantages over single-core processors. These advantages include: 1) the potential for parallel execution of threads in multicore processors, and 2) lower power consumption and lower thermal dissipation, therefore reducing the need for advanced cooling and power generation. The potential for parallel execution is particularly helpful for software systems that are already multithreaded; this is typically the case for avionics. Reducing the needs for advanced cooling and power generation are important for application domains in which size, weight, and power requirements are important; this is also typically the case for avionics.

This increasing interest in using multicore processors in avionics raises a question: How do we certify aircraft that use multicore processors? This report discusses this question—particularly, what can go wrong when porting software originally developed for a single-core processor now must execute on a multicore processor?

APPENDIX A—SURVEY OF LITERATURE RELATED TO VIRTUAL MACHINES

A.1 INTRODUCTION

Virtualization is a broad term for software used to create, from one resource, software entities so that each software entity behaves like the resource. Different types of resources can be virtualized, such as processors, memory spaces, network interface cards, communications links, or entire networks. In this report, we focus mostly on virtualization of one or more processors. Virtualization offers the following benefits:

- Lower hardware cost: Virtualization allows for creating multiple machines from one physical machine.
- Support for multiple machines with slight differences: If many virtual machines (VMs) have many instructions in common, then they can execute natively, and only a small number (but different for each VM) of instructions needs to be simulated.
- Support for legacy hardware: In some cases, software has been developed in the past for old hardware that is no longer commercially available. It may be, however, that new hardware supports most of the instructions from the old hardware, and therefore a virtual machine monitor (VMM)[1] can be used running on the new hardware to create a VM of the old hardware.
- Security[2]: If malware infects one operating system (OS), other OSs for other VMs are unaffected (assuming that the VMM is not affected). For example, if you want to download some code from the web or visit a website that you are suspicious of, then you can do so in a VM. If the VM gets infected, it does not affect other VMs. You just kill your VM, and the malware will have no lasting effect on your machine (assuming that it does not write to a durable medium).
- Privacy: Because VMs have non-overlapping address spaces, it holds (under certain assumptions[3]) that execution within one VM cannot infer behavior of another VM.

In the next section, we will discuss the history of VMs.

A.2 HISTORY

The history of VMs can be traced back to the 1960s. In this section, we focus on three seminal papers that capture the historic roots of this technology.

[1] Also called a hypervisor.

[2] There is a known defect in some dynamic random-access memories (DRAM), known as row hammer. The defect allows a program to access one memory location and thereby modify other memory accesses. In this way, one VM could modify memory used by another VM or the VMM. See [A-1].

[3] So-called timing attacks are possible. Specifically, VMs execute instructions that read from and write to memory; when they do so (for most processors), they attempt to read from or write to a so-called cache memory (a small, fast memory that stores frequently accessed data items). A memory access from VM A may bring its data into the cache so that it can access this memory address again. In the meantime, it may happen that VM B accesses a memory address in a way that evicts the data that VM A brought. Therefore, VM A can detect the existence of another VM; it can even infer that a memory access to a certain subset of memory addresses has been made.

A.2.1 POPEK AND GOLDBERG

One of the most authoritative works on virtualization is the article by Popek and Goldberg [A-2]. This work aims to state conditions for virtualization, but it also provides a tutorial and a survey. It states, "a virtual machine is taken to be an efficient, isolated duplicate of the real machine." Because the VM must be efficient, it follows that using one processor to simulate two or more processors will lead to a large slowdown, and therefore it is not virtualization. The authors point out that most of the instructions must be executed natively on the physical processor; for virtualization to take place, software is allowed to simulate only a few instructions. They state explicitly that:

> The second characteristic of a virtual machine monitor is efficiency. It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella.

The authors describe virtualization with the following concepts: 1) a VMM is a piece of software that provides an environment for programs that is essentially identical with the original machine, 2) programs executing on this VM show at worst only minor decreases in speed, and 3) the VMM is in complete control of system resources.

Popek and Goldberg [A-2] point out that a VM will be slower because: 1) the VMM requires time, and 2) other VMs consume system resources. Popek and Goldberg [A-2] provide us with the following definition of a VM:

> **Definition:** A virtual machine is the environment created by the virtual machine monitor.

Later they provide another definition of a VM:

> **Definition:** Then functionally, the environment which any program sees when running under the control of a virtual machine monitor present is called a virtual machine.

From this definition, the article by Popek and Goldberg [A-2] points out that:

> … a VMM as defined is not necessarily a time-sharing system, although it may be.
>
> Note that isolation is not mentioned as one of the three properties of a virtual machine. It is because it is implied; we state that it provides an environment for execution; it executes on the virtual machine as if it were on a real machine and this implies isolation.

The focus of the work by Popek and Goldberg [A-2] was on so-called third-generation machines. This term is not used today, but it may be instructive to know what it means. First-generation machines used vacuum tubes; the second generation used transistors. Third-generation machines are characterized by the use of integrated circuits, and they have linear addressable memory. Examples are the IBM 360, Honeywell 6000, or DEC PDP-10. The fourth generation uses microprocessors, and the fifth generation provides artificial intelligence. Today, we have fourth-generation computers, but the work by Popek and Goldberg [A-2] on third-generation computers is still relevant to our discussion.

Popek and Goldberg [A-2] assumed that processors have two modes of operation: 1) supervisor mode and 2) user mode. The former has access to all instructions of the machine, but the latter does not necessarily have that. The authors also assumed that the computer has a memory-relocation register and that the instruction set consists of the usual operations: arithmetic, testing, branching, and moving data in memory. Computers today use multilevel paging and sometimes segmentation rather than relocation registers, but today's paging mechanism can function in a similar way as relocation registers. The authors assume that the VMM consists of three modules: 1) a dispatcher, 2) an allocator, and 3) an interpreter. The dispatcher "catches" traps. The allocator decides which memory addresses to allocate a VM. The interpreter executes so-called sensitive instructions (these are instructions that the hardware cannot execute directly). The authors specify how a sensitive instruction can be performed. The idea is that a machine can be modeled as having a state, and the instruction can be modeled as transitioning from one state to another. The authors state that we can describe a sensitive instruction with a set of pairs of states (from state, to state). The implementation of a sensitive instruction can be done with a table lookup. This method is not supposed to be used (because it is not efficient with respect to memory), but it is presented to show that sensitive instructions can be simulated.

The article by Popek and Goldberg [A-2] points out that a VM differs from a real machine in only two ways: timing and resource control. Therefore, later parts of this document will describe methods for virtualization while satisfying timing requirements.

The main result of Popek and Goldberg's work [A-2] is a sufficient condition for virtualization on third-generation computers. The condition is that if each sensitive instruction is a privileged instruction, then virtualization is possible. This is a sufficient condition, but it is not necessary.

Popek and Goldberg [A-2] end the paper by defining a "hybrid virtual machine." This is a VM in which all virtual-supervisor mode instructions are interpreted. This results in lower efficiency (lower execution speed), but it has the advantage of allowing virtualization on architectures in which the Popek and Goldberg sufficient condition is false.

A.2.2 BUZEN AND GAGLIARDI

Buzen and Gagliardi [A-3] provide a tutorial and survey the evolution of VMs in the 1960s and 1970s. They argue that one should build the system software as one small part that is assumed correct and another part that is everything else. They stress that a VM behaves as a real machine, but its timing may be different because 1) the VMM has overhead and 2) other VMs may be running concurrently. They also point out that the VMM does not perform instruction-by-instruction interpretation of programs. They emphasize that reliability is an important aspect of VMs; an operating system can be tested in one VM, and if it crashes then it does not crash other VMs. They stress isolation as one goodness property. They mention that in virtualization, VMM intervention of input/output (I/O) devices can make it possible to create I/O devices that have no physical analog, such as a tape that behaves like a disk. Buzen and Gagliardi [A-3] defined a VM as:

> A basic machine interface which is not supported directly on a bare machine but is instead supported in a manner similar to an extended machine interface is known as a virtual machine.

This definition is similar to the definition by Popek and Goldberg [A-2], but it emphasizes the interface rather than the environment.

Buzen and Gagliardi [A-3] discuss early VMs from the perspectives of 1) processor state mapping, 2) memory mapping, and 3) I/O mapping. With respect to 1), they mention that the VMM has a virtual status indicator so that when the VM executes an instruction that reads the status indicator, it traps and the VMM reads the virtual status indicator in the VMM and returns the value. Writes to the virtual status indicator are handled analogously. With respect to 2), they mention that the VMM maps pages for processes running under VMs. They point out that for interrupt service routines, it is often the case that the address where instructions of interrupt service routines start must be stored at a fixed location in memory, and a VM should not be allowed to change that. Therefore, the VMM must set up the virtual-to-physical address translation of the VM. With respect to 3), they mention a situation in which a program executing on a VM initiates the execution of a channel program. This is implemented by the VMM copying this program to certain fixed addresses and then executing this program; after getting the result, it copies it back to the VM. It is noteworthy that Buzen and Gagliardi [A-3] point out that paging systems are not necessary for virtualization; they state that any memory relocation mechanism that can be made invisible for nonprivileged execution is enough. They stress (as did Popek and Goldberg [A-2]) that trapping of sensitive instructions is an absolute requirement for virtualization.

Buzen and Gagliardi [A-3] mention that later generations of VMs ran on computers that supported paging. This made it possible to have recursive virtualization; that is, one VM can provide multiple VMs. Buzen and Gagliardi [A-3] point out that, in later systems that support paging, it may be necessary to perform address translation twice: from application processes' virtual address to the perceived physical address of the operating system to the actual physical address. To perform this double mapping efficiently, the VMM computes a composed table. Buzen and Gagliardi [A-3] discuss Type-2 virtualization, in which a VM is created on top of a normal operating system. When a new VM is created, the VMM informs the underlying operating system that traps should be directed to the VMM.

A.2.3 GOLDBERG

Goldberg [A-4] presents two taxonomies about virtualization: Type-1 virtualization and Type-2 virtualization. In the former, the VMM runs directly on the hardware. In the latter, the VMM runs on top of a privileged software nucleus. For the latter, when the VMM starts a new VM, *M*, the VMM needs to notify the privileged software nucleus that when a trap is generated from within *M*, it should not be dealt with by the privileged software nucleus; instead, it should be directed to the VMM.

> The term emulation is often used if the virtual machine is very different from the machine executing the VMM.

Goldberg [A-4] does not define a simulator, but this term is typically used to mean the same thing as an emulator except that the system is not simulated with as high fidelity. The separation kernel is a related concept, but it focuses more on security and confidentiality (see sections 4.2.3 and 4.2.4).

A.3 PRODUCTS AND SYSTEMS

The concept of VMs dates back to the 1960s, but had a revival in the late 1990s and early 2000s because of the increasing use of data centers and data warehouses, particularly for server processing of Internet applications. In this era, software developers wanted to offer applications accessible through a web browser and run the applications on a web server. This could be achieved if the software developer ran his or her web server. However, many software developers preferred to outsource server administration to focus on developing more and better software functionalities. Therefore, a new breed of companies for hosted service emerged. These companies found VMs useful because they could run multiple VMs, each for different clients, on a single physical machine. This approach saved hardware costs and cooling costs and allowed migrating one VM to another VM in case of overload (or hardware failure) of one physical machine.

From this era, two products are noteworthy: VMWare™ [A-5] and Xen™ [A-6]. They differ in how they deal with sensitive instructions. VMWare uses binary translation, and Xen uses para-virtualization. These products are discussed in section A.3.1.

A.3.1 VMWARE

VMWare is a private corporation in Palo Alto, CA, that sells VMMs and related products. It was founded in 1998 and later acquired. It claims to be the first company to virtualize x86 processors, which are hard to virtualize.

The cofounders of VMware wrote an article describing their first product VMware workstation [A-5]. This product was based on two ideas: 1) the VMM identifies basic blocks in a VM and detects whether, in a basic block, there is a sensitive instruction (if so, it calls a subroutine in the VMM); and 2) the product used software emulation of I/O devices. The former implies that it can run unmodified operating systems. The latter implies that all VMs experience the same hardware; this brings the benefit that if a VM named V is hosted by a real computer R1, then one can stop V and restart it on another computer, R2, even if R1 and R2 have different I/O devices. This facilitates load balancing.

Today, VMWare offers a wide range of products: both Type-1 and Type-2 VMMs and products for both desktops and servers. Among the products from VMWare, ESX/ESX is perhaps the best known.

A.3.2 XEN

Xen was developed in Cambridge, UK, in the early 2000s. It was developed as an open-source VMM of Type 1 (i.e., the VMM executes directly on the processor), and it changed certain operating systems (Windows® XP, Linux® kernel, and NetBSD) to operate together with this VMM. The changes to these operating systems were made to improve performance; the authors of [A-6] mention that their VMM can support 100 VMs. This idea of modifying an operating system running in a VM so that it performs well when running under a VMM is called para-virtualization; it is an old idea that had a revival in the early 2000s. Xen was originally developed for the x86 processor architecture, and its creators [A-6] list the difficulties in virtualizing x86. Some of the difficulties include the following: 1) Like most processors, some instructions are intended to be used only in privileged mode; 2) normally, such an instruction is executed in non-privileged mode, then it generates a trap, but in x86 they fail silently; and 3) because these sensitive instructions fail silently, the VMM cannot intercept them. The paper [A-6] mentions that VMWare ESX can handle this, but at the expense of lower performance, and VMWare ESX modifies the code in the guest operating system if there are instructions that exhibit this silent failure[4].

The authors of the paper [A-6] refer to their VMM as a hypervisor because it executes at a higher privilege than the normal operating system in a VMM (which can be thought of as supervisor mode). Xen took the following design decisions:

- Each guest operating system is responsible for memory management.
- The VMM is copied in the address space of each VMM, which avoids flushing of the translation lookaside buffer (TLB).
- Whenever the guest OS needs a new memory page, it allocates one from its own reservation and notifies the VMM.
- On x86, there are four privilege levels. Applications run in privilege level 1, and the operating system runs in privilege level 4. However, with Xen, the operating system runs in privilege level 3, and Xen runs in privilege level 4; this ensures that Xen runs with a higher privilege level than respective guest operating systems.
- When a processor invokes a system call (by generating a trap; i.e., software interrupt), then it yields control directly to its corresponding guest operating system (i.e., not to Xen). This is done because system calls are common, and it is important that they are performant.
- When a process generates a page fault, then it transfers control to Xen, to privilege level 4; the reason for this is that a page fault handler needs to know the address that generated the page fault, and this information is only accessible in privilege level 4.
- When a new VM is started, admission is performed, and this admission control is not executed in Xen; it is executed in a guest operating system (the reason for this is to keep the hypervisor small and to separate policy from mechanisms: mechanisms should be implemented in the Xen hypervisor, and policies should be implemented elsewhere).
- A guest operating system can invoke services from Xen using a hypercall (similar to a system call), and Xen can notify a guest operating system using events (similar to UNIX signals).

Xen refers to VMs as domains. There is one special domain called Domain0; it handles policy matters, I/O, and scheduling parameters. In the original version of Xen, central processing unit (CPU) scheduling of domains was done with the algorithm Borrowed-Virtual-Time (BVT) [A-7]. This algorithm works as follows:

[4] Intel has introduced a new mode in its x86 processors to avoid this.

1. Each processor has an actual virtual time and an effectual virtual time.
2. The process with the lowest effectual virtual time is selected.
3. At each time quantum, the actual virtual time of a process is incremented by an amount that is equal to the elapsed time multiplied by the share proportion of the process.
4. The effectual virtual time of a process is set to actual virtual time of the process, but the process can temporarily get a boost by getting this effectual virtual time decreased further.

The BVT is claimed to be designed for low-latency applications, but there is no known schedulability analysis for it (i.e., there is no method for proving that processes meet real-time deadlines when this scheduler is used). Later versions of Xen added two new schedulers: simple Earliest-Deadline First (sEDF) [A-8] and credit scheduler [A-9]. The sEDF scheduler characterizes a process with three parameters (period, execution time, and a flag stating whether it can reclaim time) and schedules processes with Earliest-Deadline First (EDF). The credit scheduler operates with virtual timers that are updated in a similar manner as BVT.

As mentioned, Xen was originally developed for x86, but today it supports many other architectures (ARM, PowerPC, MIPS). Recently, it has been claimed[5] that it offers superior performance compared to other open-source hypervisors.

With the popularity of virtualization, a large number of VMM products came out. Some are listed in sections A.3.4–A.3.7.

A.3.3 HYPER-V™

Hyper-V [A-10, A-11] is a hypervisor developed by Microsoft®. It is a Type-1 hypervisor, and it is released as a component of Windows Server®. It supports VMs with Windows or VMs with the Linux kernel. Hyper-V runs on X86-64 processors and uses the VT-X x86 virtualization feature.

A.3.4 KVM™

KVM [A-12] is open-source software that turns the Linux kernel into a hypervisor. It is a Type-1 hypervisor. KVM can create VMs for a wide range of processors—x86, S/390, PowerPC, and IA-64—and a wide range of guest operating systems—Linux, BSD, Solaris, Windows, and OS X. It supports para-virtualization for Ethernet cards and disk controllers.

A.3.5 VIRTUALBOX™

VirtualBox [A-13] is a Type-2 hypervisor that is partially available as open source. A core package is open source, but support for certain I/O devices is proprietary software. VirtualBox can run on top of many of the common desktop operating systems, and it can create VMs with many of the common operating systems.

[5] See "Ubuntu 15.10: KVM vs. Xen vs. VirtualBox Virtualization Performance," http://www.phoronix.com/scan.php?page=article&item=ubuntu-1510-virt

Hypervisors have also been developed for real-time, safety-critical, and secure systems. Two are listed in sections A.3.6 and A.3.7.

A.3.6 RT-XEN

RT-Xen [A-14] is a project to add real-time scheduling algorithms to Xen. RT-Xen supports global Earliest-Deadline First and global fixed-priority preemptive scheduling. These are schedulers used within RT-Xen, not schedulers within guest operating systems. RT-Xen also supports four server-based mechanisms (a polling server, periodic server, sporadic server, and deferred server); these are discussed in more detail in section 4.1.1.1. These mechanisms allow the attachment of a server period and server budget to a VM so that the VM will consume only a certain amount of processing power within a certain time window (how these time windows are generated depends on the specific server mechanism used). Today, parts of RT-Xen have been incorporated in the normal Xen hypervisor.

A.3.7 XMHF

XMHF [A-15] is a research prototype developed at Carnegie Mellon University. It is a hypervisor that supports a single VM. Its merit is that formal verification has been performed to prove memory integrity; that is, the hypervisor's memory can be modified only by instructions that are an intended part of the hypervisor.

A.4 RELATED TECHNOLOGY

VMs have not been the only technology development that has pursued the goals of system partitioning, isolation, and protection. In this section, we discuss related technologies that share similar goals as VMs.

A.4.1 TEMPORAL VIRTUALIZATION

Temporal virtualization in VMs is traditionally achieved with a two-level scheduler assigning a time slice in a round-robin fashion to each VM and letting the scheduler inside the VM schedule its processes internally [A-6]. This approach is aimed at fairness, but it may cause problems in real-time settings because it ignores deadlines. Some versions of real-time double scheduling exist today [A-14, A-16].

In real-time systems, where meeting thread (or task) deadlines are important, temporal virtualization takes the form of temporal protection. More specifically, real-time tasks interact with physical processes in a continuous fashion (they execute forever) through some form of periodic interactions or reactions to physical events. The execution of each interaction/reaction is known as a job. The interval between job activations (also called arrivals) is known as period or minimum inter-arrival time. These jobs are required to finish their execution within a fixed time from its activation, known as a deadline. Real-time systems use real-time schedulers that determine when tasks execute and analysis to verify whether all tasks will always finish by their deadlines; that is, whether the task set is schedulable. Temporal protection in this context implies that a task's misbehavior, such as executing longer than its specified worst-case execution time (WCET) or activating more frequently than its specified period, will not make other tasks miss their deadlines.

A.4.1.1 Reservations

Resource reservation is a scheme to provide temporal guarantees and isolation of tasks that use time-shared resources. This applies not only to CPU time but also to other resources, such as network, memory bandwidth, cache, and disk bandwidth.

Resource kernels [A-17, A-18] combine generalized rate-monotonic scheduling (RMS) theory [A-19] (that includes deadline monotonic scheduling, or DMS), and processing servers provide resource reservation and achieve temporal protection. Specifically, RMS relies on three elements. First, a scheduling mechanism known as fixed-priority scheduling ensures that the highest-priority task that is ready to execute is given the processor; if a higher-priority task becomes ready, it preempts[6] the currently running task. Second, an offline priority assignment gives higher priority to tasks with smaller periods (or deadlines, if they are not at the end of the period). Third, analysis algorithms verify that tasks meet their deadlines. These three elements—classified here as mechanism, configuration, and analysis—guarantee that tasks meet their deadlines if their specified parameters (WCET and period) are not violated (e.g., because of misbehavior).

For RMS, the mechanism is fixed-priority scheduling, the configuration is the assignment of higher priorities to tasks with shorter periods, and one of the analysis methods is the response-time test. The response-time test [A-20] uses the recurrence equation:

$$R_i^k = C_i + \sum_{j<i} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil C_j, \tag{A-1}$$

where the N tasks (τ_i) of the system are indexed with indices 1 to N in increasing order of period (or deadline, if different) that matches the increasing order of priority in RMS and defined with the following parameters: period (T_i), WCET (C_i), and deadline (D_i). The equation may need to be evaluated multiple times as it is defined in terms of the previous value of itself (i.e., R_i^k is defined in terms of R_i^{k-1}). For instance, for two tasks $\tau_1 = (T_1 = 4, D_1 = 4, C_1 = 2), \tau_2 = (T_2 = 8, D_2 = 8, C_2 = 3)$, we can calculate $R_1 = C_1 = 2$, given there are no tasks with higher priority (smaller index) than τ_1. R_2 can be calculated as $R_2^0 = C_2 = 3, R_2^1 = (C_2 = 3) + \left\lceil \frac{R_2^0 = 3}{T_1 = 4} \right\rceil (C_1 = 2) = 5, R_2^2 = (C_2 = 3) + \left\lceil \frac{R_2^1 = 5}{T_1 = 4} \right\rceil (C_1 = 2) = 7, R_2^3 = (C_2 = 3) + \left\lceil \frac{R_2^2 = 7}{T_1 = 4} \right\rceil (C_1 = 2) = 7$. In this case, multiple repetitions were necessary to make the equation converge in step 3 ($R_2^2 = R_2^3$). Given that in both cases the response time is smaller than the deadlines ($(R_1 = 2) \leq (D_1 = 4)$ and $(R_2 = 7) \leq (D_3 = 8)$), the tasks are schedulable.

It is worth noting that RMS analysis does not require that the tasks activate at the same time or in any particular sequence of time offset between them (such as in phasing). This means that tasks can be added or removed while other tasks are running if the offline analysis deems the modified task set schedulable with the new priority configuration (priority assignment) without making the old tasks miss their deadlines. This assumes that there is enough idle CPU cycles to reassign priorities.

6 When a task *x* is preempted by a task *y*, the execution of *x* is paused until task *y* finishes. Then the execution of task *x* is resumed.

To understand why RMS is not sensitive to activation phases between tasks, it is useful to compare figures A-1 and A-2.

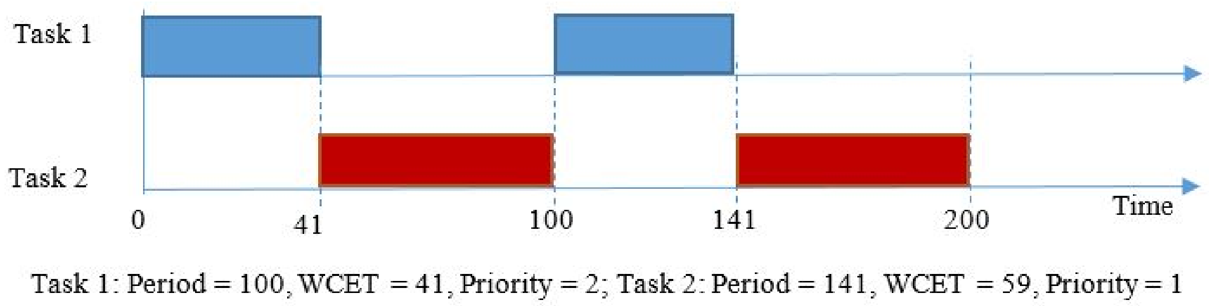
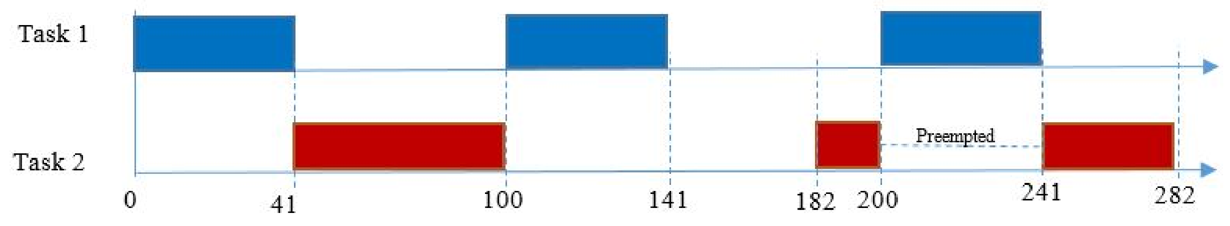


Figure A-1. Worst-case phasing of two tasks (both tasks start at zero)

Figure A-1 shows a Gantt chart of two tasks. The colored boxes represent the time tasks take to execute on the processor. In this case, both tasks are ready to execute at time 0, and the fixed-priority scheduler chooses Task 1, which has a higher priority (assigned according to RMS). We can see that after Task 1 completes at time 41, Task 2 starts and finishes its WCET (59) at time 100, just in time for the second arrival of Task 1 (after its period of 100 elapses). Task 1 finishes at time 141 just in time for the second arrival of Task 2 (after its period of 141 elapses). It is worth noting that the pattern exhibits execution of the tasks one after another without leaving any idle time in the processor. This means that it is not possible for any of the tasks to execute for longer periods or to add any other task. This was proven [A-21] to be the worst-case situation for any set of two tasks for any period and any WCET. Moreover, it is possible to observe that the ratio of the period of Task 2 to Task 1 is roughly equal to the square root of 2: $\frac{141}{100} \cong 2^{\frac{1}{2}}$. The sum of the utilizations (utilization = $\frac{WCET}{Period}$) is roughly equal to the square root of 2 minus 1 multiplied by 2: $\frac{59}{141} + \frac{41}{100} \cong \frac{82}{100} \cong 2\left(\frac{141}{100} - 1\right)$ (in the proof they are equal). This calculation gives a bound that can be used to perform a fast test on any task set of two tasks: $2(2^{\frac{1}{2}} - 1)$. For instance, we can test whether the two tasks in figure A-1 are schedulable by evaluating $\frac{41}{100} + \frac{59}{141} \leq 2\left(2^{\frac{1}{2}} - 1\right)$. This bound was generalized to task sets of any size N: $N(2^{\frac{1}{N}} - 1)$.

To complete the discussion on phasing, figure A-2 shows the same two tasks but with Task 1 starting at time 0 and Task 2 at time 41. In this case, the idle time from 141 to 182 (and a preemption of Task 2 by Task 1 from 200 to 241) is shown. This means that, if we consider the worst-case phasing, processing time may be wasted, but there is a gain in simplicity of evaluation and flexibility to start the tasks at any time. That is, to take advantage of this idle time, it is necessary to force the phasing shown in figure A-2.



Task 1: Period = 100, WCET = 41, Priority = 2; Task 2: Period = 141, WCET = 59, Priority = 1

Figure A-2. Another phasing (Task 1 starts at 0 and Task 2 at 41)

A.4.1.1.1 Servers

Software defects can lead to timing misbehavior in a task by causing it to execute beyond its WCET or more often than anticipated. Mechanisms known as servers [A-22–A-25] were developed to prevent this misbehavior from affecting the timing guarantees of other tasks. Whereas these mechanisms are known as servers in the real-time literature, the term processing servers will be used in this report to avoid conflicts with other type of servers (e.g., web, database) and just servers when the context is clear. When processing servers are used, tasks are assigned to servers, and the scheduler runs the tasks assigned to the server based on the server parameters. Each server has a budget and a replenishment interval or period. With these parameters, the server behaves as a periodic task with its deadline at the end of its period (with the budget as the WCET and the replenishment interval as the period), allowing the use of RMS analysis to verify the schedulability of a set of servers. This is the behavior of the periodic server [A-25]. In this case, as a task assigned to a server executes, it decrements the budget. If the budget reaches zero, the task is paused. The budget is replenished after the replenishment interval has elapsed, and the paused task (if any) resumes executing.

Servers have priorities and the same priority scheduling applies as in tasks in RMS. Tasks assigned to a server execute with the server's priority. When only one task is assigned to each periodic server, the server replenishment interval can be synchronized with the task's periodic activation, and the server budget is equal to the WCET of the task. The server guarantees the following properties: 1) it allows the task to execute for its WCET, but 2) prevents it from executing beyond; 3) it allows two activations of the task to be as close as one replenishment interval apart but 4) prevents shorter intervals. Properties 1 and 3, combined with a successful scheduling analysis of the servers, guarantee the schedulability of the tasks. Properties 2 and 4 ensure that any misbehavior of the task is prevented from affecting the schedulability of other tasks. Given that servers are not required to have synchronized activations, servers can be added or removed without inducing deadline misses just as with the scheduling of tasks.

When servers are used with multiple tasks, more complex schedulability analysis is required whether or not the tasks' WCET and periods are synchronized with the servers' budgets and periods. To understand this, how the server budget is consumed should first be discussed. Specifically, for the periodic server to behave as a periodic task, it needs to start consuming its budget as soon as it is activated if it has the highest priority among the active servers. This is an assumption of the RMS analysis. This means that if the task is running (highest-priority ready server), the budget must be decremented as time passes, even if no task is executing. This creates “blackout” intervals for tasks if a task arrives just after the server’s budget was exhausted (taking into account the worst-case response time calculation), and the blackout will last until after the next replenishment and all higher-priority servers execute. These unsynchronized effects also apply when multiple tasks are run on a single server. New scheduling algorithms have been created to take this into account [A-26]. Motivated by the drawbacks of the unsynchronized periodic server, three other variants of servers were created: the deferrable server, the sporadic server, and the polling server. The deferrable server [A-22] decrements the budget only when a task is executing. Whereas this prevents blackouts, the server does not behave as a periodic task anymore because it creates a back-to-back preemption for lower-criticality servers. That is, a task running on a high-priority server can execute at the end of the replenishment interval, exhausting the budget at exactly the same time the next replenishment is due. Then, because the budget is replenished at that time, the same task can continue executing. This means that lower-priority tasks will experience a preemption equivalent to two back-to-back WCET within an interval equal to the period, breaking the original assumption of periodic tasks (i.e., there is only one WCET within a period). A modified schedulability analysis has been created to take this into account [A-22].

To correct back-to-back execution, the sporadic server was created [A-24]. In this case, the budget is again consumed only when a task executes, but the replenishment is modified. Specifically, the portion of the budget consumed by the execution of a task is replenished only after an interval equal to the server period has elapsed from the start of executing such a task. This way, if the task consumes the budget at the end of the replenishment interval (as in the back-to-back case of the deferrable server), the replenishment does not happen immediately when the budget is exhausted (as in the deferrable server) but instead one full server period after the task started executing. This mechanism recovers the behavior of the periodic task but makes the replenishment mechanism more complex. To reduce this complexity, the polling server was created [A-27]. The polling server periodically checks whether a task is ready to execute and, if so, it executes the task with its budget. In this case, it also creates a blackout, but it is tailored to simplify the mechanisms.

Resource kernels [A-17, A-18] generalized processing servers into what is called resource reservations. A resource reservation is a reservation to use a resource. When the resource is CPU cycles, it maps directly to processing servers, and the reservation is described in terms of CPU time budgets with some periodicity, deadline, and other parameters. Resource reservations are implemented in resource kernels as an operating system object with a handler in the same fashion as a file and its file handler. A reservation handler is “attached” to a process ID for that process to run within the “processing server” implemented by the reservation. This means that the execution of the process is restricted to consume only the CPU assigned to the reservation and, therefore, the same scheduling analysis that we have discussed for processing servers can be applied.

Multiple processes can be attached to a reservation, and all the processes attached to a reservation will share the budget allocated to this reservation. At the same time, the temporal protection offered by the reservation prevents processes running in one reservation from interfering with the processes running in another reservation. It is worth noting that when a reservation is created, the resource kernel runs an internal schedulability test to verify that it is possible to execute the reservation within the requested parameters (budget, period). This test is called an admission test, ensuring that any new reservation is guaranteed by the timing analysis theory. Reservations can be created at runtime if the timing analysis theory does not require the reservations to be synchronized. This simplifies updates because parts of the system that need to be added or removed on the fly can be without disturbing the other parts of the system. This can be ensured by letting the admission test run in the idle time left by the running reservations (i.e., running this code at a lower priority than the reservations).

Resource reservations go beyond CPU cycles. In particular, they allow the reservation of other resources, such as network [A-28], disk [A-29], cache [A-30], and even RAM memory [A-31]. Both cache and RAM memory are critical resources that must be properly managed in multicore processors. This is because a task running in one core can delay the execution of another task running in a different core due to access to the common cache, RAM memory, or both. This approach has shown increases of up to six times when compared to a task running alone in legacy applications and more than14 times for synthetic cases [A-31].

The protection of resource kernels is based on ensuring that task deadlines are always respected without over-constraining the execution to specific time slots. Multiple issues must be addressed when tasks interact with each other, such as when they use mutexes or semaphores to synchronize. The reserve inheritance is one of these solutions [A-32]. When multiple resources are used together, such as when the disk scheduler needs CPU time to process a disk request, it is necessary to verify the combined requirements [A-29].

A.4.1.2 Mixed-Criticality Isolation

Temporal isolation typically implies protecting a task from delays imposed by other tasks to prevent a deadline miss. This isolation is identified as a symmetric protection, given that protecting Task A from Task B also implies protecting Task B from Task A. However, certification documents assign different criticalities to different tasks. This implies that it is necessary to verify that the probability of failure of a high-criticality task should be smaller than those of lower criticality, and the corresponding verification should be imposed. In real-time scheduling, all the scheduling algorithms are proven mathematically, assuming that a task cannot execute more than its stated WCET. Because the WCET is very difficult to calculate, it is a common engineering practice to obtain it through conservative experimental measurements. If variations from the external environment can be isolated, it is possible to use abstract interpretation [A-33] to obtain a better bound. In the end, a probability of not exceeding the WCET can be calculated, and different methods (and their probabilities) can be applied to tasks with different criticalities.

This has given rise to a new scheduling approach in which tasks with different criticalities are assigned different execution times. The schedulability of tasks is then verified at their assigned level of criticality, assuming that their WCET and the tasks that can delay them are bound by the WCET at this criticality level [A-34]. This way, an asymmetric temporal protection is achieved (i.e., a high-criticality task is protected, assuming that only tasks with criticality levels the same or higher can interfere with it). This applies to each criticality level; when the schedulability of a lower-criticality level (e.g., Level 1) is verified, it is assumed that higher-criticality tasks (Level > 1) execute for the WCET calculated at Criticality Level 1.

A number of scheduling approaches have been developed [A-35, A-36]. Zero-slack rate-monotonic (ZSRM) scheduling is one of these approaches [A-37]. ZSRM scheduling is based on RMS but adds a mechanism that suspends low-criticality tasks at what is called a zero-slack instant calculated offline. This zero-slack instant Z_i divides the execution of a task τ_i into parts or modes:

the normal mode and the critical mode. During the normal mode, all tasks run; in the critical mode, only tasks that have higher or equal criticality than τ_i execute. ZSRM scheduling uses the response time equation A-1 to calculate the zero-slack instant and to verify the schedulability of the tasks. However, tasks that interfere with task τ_i in equation A-1 also must include all lower-priority but higher-criticality tasks τ_j, given that τ_j may enter its critical mode and suspend τ_i. In this case, we need to take into account only the portion of τ_j that runs in critical mode. In ZSRM scheduling, tasks can be classified in any number of criticalities but only have two WCET parameters, known as nominal execution time (C_i) and overloaded execution time (C_i^o), in addition to period (P_i), deadline (D_i), and criticality (ζ_i).

To illustrate how ZSRM scheduling works, consider the following two tasks: $\tau_1 = (P_1 = D_1 = 4, C_1 = C_1^o = 2, \zeta_1 = 1)$ and $\tau_2 = (P_2 = D_2 = 8, C_2 = 2.5, C_2^o = 5, \zeta_2 = 2)$ with zero-slack instants $Z_1 = 4, Z_2 = 5$. [7] The response-time test of τ_2 can then be divided in nominal mode R_2^n (before $Z_2 = 5$) and the critical mode R_2^c (after Z_2) with the execution time split into nominal execution time ($C_2^n = 2$) and critical execution time ($C_2^c = 3$) adding to its overloaded execution time ($C_2^o = 5$). This is calculated as: $R_2^{n(0)} = (C_2^n = 2), R_2^{n(1)} = (C_2^n = 2) + \left\lceil \frac{R_2^{n(0)}=2}{T_1=4} \right\rceil (C_1^o = 2) = 4, R_2^{n(2)} = (C_2^n = 2) + \left\lceil \frac{R_2^{n(1)}=4}{T_1=4} \right\rceil (C_1^o = 2) = 4$ and $R_2^{c(0)} = (C_2^c = 3), R_2^{n(1)} = (C_2^c = 3) + \min\left(Z_2 = 5, \left\lceil \frac{R_2^{c(0)}=3}{T_1=4} \right\rceil (C_1^o = 2)\right) = 8,$ $R_2^{n(2)} = (C_2^c = 3) + \min\left(Z_2 = 5, \left\lceil \frac{R_2^{c(1)}=8}{T_1=4} \right\rceil (C_1^o = 2)\right) = 8$. In this case, equation A-1 was modified to include the *min* function that captures the fact that after Z_2, τ_1 is suspended and does not add to the response time of task τ_1. By design Z_2 is calculated to make τ_2 finish at its deadline. Now, because there are no tasks with lower criticality than τ_1, its zero-slack instant is set to its deadline: $Z_1 = D_1 = 4$. This means that it never executes in critical mode. However, when calculating its response time, it needs to be taken into account that the fraction of τ_2 runs in critical mode because, during that execution, $\tau_1$is suspended. In addition, only the WCET from τ_2 at the criticality level of τ_1 is considered. In ZSRM, this is the nominal execution time ($C_2 = 2.5$), which applies to any criticality level lower than ζ_2. Furthermore, the execution time that τ_2 was able to complete in its nominal mode before Z_2 ($C_2^n = 2$) can be discounted; that is, only $(C_2 = 2.5) - (C_2^n = 2) = 0.5$ units of execution are considered. As a result, R_1^n can be calculated as $R_1^{n(0)} = (C_1^o = 2), R_1^{n(1)} = (C_1^o = 2) + \left\lceil \frac{R_1^{c(0)}=2}{T_2=8} \right\rceil (C_2 - C_2^n = 0.5) = 2.5, R_1^{n(2)} = (C_1^o = 2) + \left\lceil \frac{R_1^{c(1)}=2.5}{T_2=8} \right\rceil (C_2 - C_2^n = 0.5) = 2.5.$

It is worth noting that, because only the nominal execution time of tasks with higher criticality than τ_i are considered to calculate its response time, ZSRM scheduling guarantees τ_i's schedulability only if no task τ_j with higher criticality than τ_i exceeds its nominal execution time C_j.

ZSRM scheduling has been extended for multiprocessors [A-38], multimodal systems [A-39], utility-based systems [A-40], and tasks that synchronize with mutexes [A-41].

[7] See [A-16] for the calculation of the zero-slack instant.

A.4.1.3 Time-Triggered Architecture

Time-triggered architecture (TTA) [A-42, A-43] is a design methodology and supporting mechanism that allows the building of a distributed real-time system based on a globally synchronized clock completely activated (triggered) by a time-triggered bus. This synchronization allows some time-fault detection and recovery mechanisms based on the absence of messages expected to occur at specific times and the presence of unexpected messages at specific times. TTA is implemented with a bus guardian mechanism that follows a strict schedule of message receptions and transmissions that are specified at design time. The time of the bus is the global time to which all nodes are synchronized. The guardian prevents message transmissions or receptions that are not part of the schedule (of each application). This feature is what implements the temporal virtualization and protection. Given that perfect synchronization is not possible, a time granularity is selected according to the capability of the hardware implementation. For instance, in the implementation discussed in [A-42], this granularity is 60 ns.

To build a system with TTA, it is necessary for each node to specify its schedule of communication (transmission and reception) that satisfies the requirements from the applications, such as periodicity, deadlines, and execution times. The node schedules are then integrated into the global bus schedule to ensure that all the nodes' timing constraints are honored. The final schedule is typically computed with constraint-solving approaches like mixed-integer linear programming or satisfiability modulo theories (SMTs) [A-44]. Such methods often take a long time. Furthermore, modifications to the applications frequently require a global recomputation of the task schedule and the bus schedule, making this method very sensitive to changes.

TTA has a number of derived technologies, mostly related to networks. Among them we can find the time-triggered Ethernet [A-45], the time-triggered controller area network [A-46], and—to an extent—FlexRay [A-47].

A.4.1.4 Integrated Modular Avionics

Integrated modular avionics (IMA) [A-48] is a reference architecture developed to allow the incremental acceptance of modules integrated into an avionics system. It uses partitions that are isolated in time and space. Partitions are typically implemented by dividing a global time frame (known as a major frame) into a sequence of time slots for the execution of a number of applications in a processor. These time slots are then grouped into (usually interleaved) subsequences that are called partitions and assigned to different applications. A runtime mechanism is in charge of executing these applications according to the time slot in their partitions (switching from one to the other). The major frame is repeated continuously, defining the continuous time sharing of the processor.

As with time-triggered approaches, the use of a timeline to schedule tasks imposes important restrictions to both the verification of the timing behavior (meeting deadlines) and the flexibility to modify partitions. More specifically, it has been proven that finding the optimal schedule for periodic tasks when they need to be executed strictly periodically is an NP-complete problem in the strong sense [A-49], except for special cases in which the periods and execution times are divisible. As a result, finding the schedule in these cases requires constraint-solving approaches or suboptimal heuristics, such as in [A-50].

One of the best-known standards that implement IMA is ARINC 653 [A-51]. This standard allows the use of fixed-priority scheduling within a time slot. This means that multiple threads from the same application can use a time slot, and they will be scheduled according to their priority. Using fixed-priority scheduling within a partition increases the complexity of verifying the schedulability of tasks because of hierarchical scheduling decisions (i.e., when a partition starts and when a task is allowed to run in a partition). Instead, hierarchical scheduling approaches can be used. An additional challenge is the inevitable modification of tasks within a partition, which necessitates more complex schemes to figure out what kind of changes (e.g., to periods, deadlines, or execution times) can be tolerated. Some approaches, like [A-52], investigate this issue by using linear programming to explore potential variations.

It is worth highlighting that techniques based on processing servers do not exhibit the restrictions of the strict periodic execution of timeline-based scheduling. It is enough to run a test, like the response-time test in equation A-1, to verify the schedulability, and the processing server can be started at any time.

A.4.2 SPATIAL VIRTUALIZATION

In this section, virtualization techniques are examined from a logical/spatial perspective. In some sense, a primitive form of logical virtualization exists even in commodity operating systems. The concept of a process is meant to provide an abstraction so that a specific computation can use a shared resource (CPU, memory, file system, devices) while being logically isolated. In this setting, the operating system kernel acts as the VMM. Of course, this isolation is weak in traditional operating systems. Processes have visibility across each other. Moreover, a runaway process can take down the entire OS, including other active processes within it. Over the years, different types of OS kernels have been proposed and implemented. They are briefly surveyed next.

A.4.2.1 Monolithic Kernel

A monolithic kernel implements all the functionalities of a full-fledged OS. This includes memory management, device drivers, file systems, network stacks, threads and processes, and inter-process communication. Such a kernel has high performance because of fewer switches between components of different privilege levels. However, the kernel can be hard to maintain and modify because many complex pieces are tightly interconnected. Kernels are also harder to verify because of their complexity. Nevertheless, the vast majority of successful operating systems today, including Windows and Linux, are based on monolithic kernels.

A.4.2.2 Microkernel

A microkernel is a minimalist approach to develop an OS kernel. The microkernel provides only fundamental, low-level functionalities, such as managing memory and address spaces, thread management, and inter-thread communication. All other functionality—such as device drivers, network protocols, and file systems—are delegated to user-space modules. Microkernels trade off toward modularity and modifiability at the cost of performance. Few microkernels are actively developed today, and none has been commercially successful or widely adopted. This is unfortunate, because by isolating functionalities in well-defined modules, microkernels also make verification more tractable. For example, the security of the L4 microkernel has been verified formally [A-53].

Hypervisors have much in common with microkernels. However, hypervisors are focused on supporting VMs and do not have minimality as a first-class goal. There are two main types of hypervisors: 1) native or bare metal, which run directly on the hardware, with each guest operating system running as a process on the hypervisor, and 2) hosted, which run on a full-fledged OS and abstract the guest OS from the host. In some cases, the distinction is not clear. For example, KVM is a Linux kernel module that effectively converts the host Linux OS into a bare-metal hypervisor. However, the host can have other processes running and competing for resources with the guest.

A.4.2.3 Security Kernel

A security kernel is an OS kernel (or component thereof) in which all security-relevant functionality is isolated. The idea is that if we can assure the security kernel, then all other components of the system become irrelevant from a security perspective. Traditionally, monolithic security kernels always have *trusted processes* that must *themselves violate the security policy to implement it*. An example is a printer spooler that must be able to read spools of different levels of classification. This means that to ensure security, these trusted processes must be verified as well, which can be difficult because they lie outside the kernel and are often unknown during the verification phase. It is noteworthy that a security kernel is a mechanism for isolating security-relevant functionality only from other functionalities, whereas a VM isolates an entire OS from other OSs executing on the same hardware.

A.4.2.4 Separation Kernel

To overcome the challenge posed by security kernels, Rushby [A-54] proposed the concept of a separation kernel, which is a security kernel implemented as a distributed system. The idea is that by decomposing the security kernel into smaller pieces, some of which implement the functionality of trusted processes, verification becomes more manageable. However, the entire separation kernel can still run on a single processor. The separation kernel is essentially a component that ensures that the security-critical components are separated from each other in a way so that they believe each is running on a separate processor in a distributed system. This must be achieved from both logical and timing perspectives, although Rushby's original paper was concerned mainly with the logical aspect. A separation kernel is a mechanism for isolating security-relevant functionality only from other functionalities, whereas a VM isolates an entire OS from other OSs executing on the same hardware. A separation kernel is one of the separation mechanisms that can be used to implement the Multiple Independent Levels of Security/Safety (MILS) architecture. In addition to a separation kernel, a MILS architecture implementation may also use a partitioning communication system and physical separation. MILS is, therefore, a more general concept.

A.4.2.5 Virtual Machines

In the early days of virtualization, there were hardware mechanisms such as relocation registers and segments for partitioning memory into logically isolated fragments. However, these have given way to page tables, which are the de facto memory fragmentation mechanisms for modern hardware platforms. Most modern VMs implement spatial partitioning through nested virtual pages [A-55], allowing the guest OS to also manipulate the page tables of its own processes. This represents the most advanced form of spatial partitioning in virtualized systems. Originally, nested page tables were implemented in software and incurred a performance penalty, despite optimizations, such as delayed loading. However, many modern processors [A-56] have started to support nested page tables natively, so they can lower virtualization performance cost significantly.

A.4.3 DEVELOPMENT PARTITIONING/ISOLATION

New lightweight isolation mechanisms called containers allow developers to experience an isolated development environment. These containers also allow developers to configure the deployment of their applications (including databases and web servers) in an isolated manner. Once the application is configured within a container, this container can be moved to different hosts. Because different containers share the host OS and kernel, this scheme offers isolation in terms of space, but temporal isolation is limited. Such an isolation is enough for IT systems, but no efforts exist for real-time systems.

Examples of container technologies include Docker [A-57], Linux Containers [A-58], and OpenVZ [A-59]. OpenVZ is the basis for VMs in a hypervisor called Virtuozzo [A-60] that offers full virtualization of the OS to increase protection against errors in the kernel.

Container technology offers even less protection than regular VMs. It is unlikely they can be used alone to simplify the certification process.

A.4.4 ANALYTIC TECHNOLOGY

A.4.4.1 Temporal Analysis

The real-time systems research domain is a scientific area studying timing of computer systems. It is common to describe a system as having a set of jobs in which each job arrives at a certain time and needs to finish a certain computation within a certain time after its arrival. Typical questions studied in real-time systems include:

- What is the maximum time that a program can take to finish if it executes in isolation?
- What is the maximum time that a piece of code can take to finish if it executes in isolation?
- How can programs share computer resources (processors, memory) so that jobs meet deadlines?
- Given descriptions of a software system and its hardware and methods for sharing, how does the outside environment trigger job arrivals in the system, and the deadlines of jobs? How can it be proven that all deadlines are met under all scenarios considered possible?

The first two questions are typically called WCET analysis. The third question is typically called real-time scheduling. The fourth question is typically called schedulability analysis.

In this section, these questions will not be discussed in general; instead, it will focus on known results specifically for VMs. For this scope, only a few results are known.

Park et al. [A-61] considered real-time systems in the automotive domain and studied the problem of allocating resources (e.g., putting two pieces of code into one thread or putting two message transfers into a single message frame and assigning software to processors) so that deadlines are met. Their paper mentions the concept of the virtual execution platform, but it is not a VM. It is a concept used by software developers to analyze software running on hardware that is not yet built. Therefore, this paper is not relevant for the purpose of this survey, but it is mentioned because of the similarity of the concepts it presents.

Hyoseung Kim et al. [A-62] considered a system with a multi-core virtualization environment and multiple VMs. For each VM, there is one or multiple virtual CPUs, and each virtual CPU is mapped to some physical CPU. When software shares non-CPU resources (e.g., data structures or I/O devices), there are certain known situations in which a thread can experience a very long (and counterintuitively long) delay; this is known as priority inversion. There are known methods for countering this. Kim et al. take one of these methods and apply it on the VM setting mentioned.

Considering the same setting mentioned above, Kim et al. [A-63] also observed that many real-time systems are interrupt-driven; that is, a thread wakes up because of an event (e.g., a message is received on the computer network). Interrupt storms are problematic; this refers to situations in which a very large number of interrupts are received, and they consume processing resources. Therefore, Kim et al. present a solution that avoids interrupt storms.

Kim and Segall [A-38] do not claim to explicitly study VMs; they study binary-to-binary translation techniques that maintain timing equivalence. One could imagine this being useful for the following setting: a software developer used to execute software S on hardware H1, but many years later, hardware H1 is no longer commercially available. The software developer now wants to execute software S and maintain the same timing, but on a new hardware H2. The developer might employ a VMM that uses binary translation, using the result of Kim and Segall [A-38].

Lee et al. [A-64] considered the Xen VMM and a set of real-time tasks in VMs using the so-called periodic server (see section 4.1.1.1) to analyze the timing of the real-time tasks. The work is related to the CARTS [A-65] toolkit.

Danish et al. [A-66] presented a new operating system, named Quest, which is not labeled as a VMM but it is labeled as a separation kernel (see section 4.2.4). Quest offers virtual CPUs with server mechanisms.

Pajic and Mangharam [A-67] introduced the concept of embedded virtual machines (EVMs). They consider a distributed computer system, specifically a wireless sensor and actuator network. The authors point out that wireless connectivity changes, and it is desirable to have software continuing delivery services. Therefore, the authors introduce the EVM, which allows code migration for the setting just mentioned.

Kaldewey et al. [A-68] presented a method to virtualize a disk. This allows the creation of two virtual disks from one physical disk and assigns bandwidth for each virtual disk.

Bruns et al. [A-69] evaluated the L4/Fiasco microkernel with respect to context-switching times and interrupt latencies. They find that the use of microkernel-based virtualization creates a need for larger cache memories.

A.4.4.2 Logical Analysis

From the logical-analysis perspective, virtualization presents both unique opportunities and challenges. However, research and tool development for logical analysis of virtualized software and VMMs are nascent. This section is therefore organized in two parts: logical analysis techniques and tools for software in general, which has a rich history, and applications of these tools for virtualized systems, which has started recently and already produced some promising tools and results. Logical analysis of (general) software takes two broad forms: testing and exhaustive verification. Both have pros and cons, and they are discussed individually.

A.4.4.2.1 Testing

Testing can be done directly on executable code, with a very realistic environment. For example, avionics (and other safety-critical) systems are often tested via hardware-in-the-loop simulations. The diagnostic feedback from testing is correspondingly quite detailed. Finally, testing can be applied to complete systems (integration testing). However, testing suffers from poor coverage. Whereas various types of coverage metrics (such as modified condition/decision coverage, or MC/DC [A-70]) are used in practice, they are ultimately syntactic and do not provide exhaustive coverage in terms of the program's semantic behavior. MC/DC testing was proposed initially by the FAA for Level A software, but full semantic coverage via testing is practically infeasible. For example, to test a C program with a single "int" input exhaustively, more than 10^{18} different test cases would be needed, assuming a 64-bit hardware platform.

Techniques such as combinatorial testing [A-71] can reduce the number of needed test cases, under the assumption that most bugs are triggered by a small number of inputs (even when the software as a whole has a large number of inputs). However, the growth in the number of inputs needed for combinatorial testing is still exponential in the worst case. Moreover, even if no bugs are found with combinatorial testing, it does not mean that the program is bug free because combinatorial testing is incomplete.

Fuzz (or random) testing [A-72] is another commonly used brute-force testing technique. The general idea behind fuzzing is to test the software under a randomly selected set of inputs. The big advantage of fuzzing is its simplicity, which means it can be applied to large-scale software with relatively less effort. It has been extremely effective in finding security vulnerabilities, especially in poorly developed or complex software with a large number of shallow bugs (i.e., those triggered by a single unanticipated input). However, the advantage of fuzzing diminishes as bugs become more complex (i.e., can be triggered by only a small number of carefully selected inputs). It is therefore used in combination with other testing techniques and verification and validation (V&V) techniques.

The incompleteness of testing worsens for concurrent (e.g., multi-threaded) software because, in addition to the large number of possible inputs, testing now also has to cope with a combinatorially large number of possible thread/component execution interleavings. In practice, therefore, state-of-the-art testing tools and techniques still cover only a miniscule fragment of a system's possible behaviors. This is despite the fact that V&V is becoming the dominant component of a system's development cost, and testing is usually done until the budget is exhausted or the deadline for project delivery is reached. The result is that critical bugs are often uncovered late in the V&V

phase (e.g., during integration testing), which leads to cost escalation. Even worse, many bugs are left undetected and are found only after they have led to some expensive or catastrophic failure in the deployed system.

A.4.4.2.2 Testing Tools

Testing is still by far the technique that is used the most during software V&V. Not surprisingly, therefore, a large number of testing tools are available, and enumerating them all is beyond the scope of this report. Many commercial and open-source software integrated development environments include some form of testing infrastructure. For example, Eclipse™ comes with the JUnit [A-73] testing framework for Java, and CUTE [A-74] supports testing in C++. Visual Studio comes packaged with Microsoft's unit testing framework [A-75], which can be used for supported languages like C++ and C#. Even domain-specific languages, such as MATLAB®/Simulink, have their own testing infrastructures [A-76]. Finally, a number of commercial entities [A-77, A-78] produce testing tools. In addition, all major software vendors have custom-made testing infrastructures, developed either in house or outsourced to an external organization. Fuzz testing tools have also seen wide development, covering a number of different application domains, such as web services [A-79], Windows ActiveX controls [A-80], and general-purpose software [A-81]. This report is not endorsing any of the tools mentioned here. They are only meant to be a sample of what is available today.

A.4.4.2.3 Exhaustive Verification

Exhaustive verification, as its name implies, aims at full semantic coverage of a program's behavior. We distinguish among three classes of exhaustive verification: theorem proving, abstract interpretation, and software model checking. The last two techniques are often clubbed together into a single category known as static analysis. However, for our purposes, it makes sense to discuss them separately and highlight their similarities and differences. Exhaustive verification techniques make different tradeoffs along two dimensions: 1) precision (number of false warnings) versus scalability (how many lines of code can be analyzed with commodity computing resources), and 2) power (how rich is the set of properties that can be analyzed) versus automation (to what extent is the technology push-button).

A.4.4.2.3.1 Theorem Proving

Theorem proving is the most powerful (therefore, least automated) and most precise (therefore, least scalable) exhaustive verification technique. It is also the oldest form of exhaustive logical verification of software. In the early days, program verification was synonymous with some form of theorem proving [A-82]. The general approach is to model the target's program semantics, and the property to be proved, as a logical formula ϕ and then use a theorem prover to prove the validity of ϕ.

Originally, Floyd [A-83] showed how ϕ can be constructed (as a verification condition) for a sequential program without loops. Subsequently, Hoare [A-82] generalized this approach to programs with loops and function calls, which require a user to provide loop invariants and function contracts. Owicki and Gries [A-84] generalized this approach further to concurrent shared memory programs, which require a user to provide thread-environment assumptions and guarantees. Since then, there has been a wide body of work on applications to different types of programing languages (e.g., object-oriented [A-85], with dynamic memory allocation).

Theorem proving is extremely powerful, and limited largely by the human user's ability to supply invariants, contracts, and proof hints. It has even been used to prove critical properties of low-level system software, such as the L4 microkernel [A-53], because it allows for reasoning not just about the software but also the hardware environment to a high level of precision. It is also manually intensive and, therefore, best suited for verifying critical software components (e.g., VMMs) for which the high cost is justified by the benefits and gets amortized over multiple applications as the software is updated. This line of work is also commonly referred to as deductive verification or auto-active verification.

A.4.4.2.3.2 Abstract Interpretation

Abstract interpretation [A-86] is a form of static program analysis for which an over-approximation of the program's behavior is analyzed using an "abstract domain." For example, a commonly used abstract domain is the interval domain [A-87]; instead of keeping track of the precise values of program variables, the only thing tracked for each variable v is an interval $[min, max]$ such that the value of v always lies between min and max. By abstracting multiple program states into an interval, the state space explored by the analysis is reduced significantly. Abstract interpretation is very scalable and automated, and it has been applied to millions of lines of industrial code [A-88]. Conversely, abstract interpretation can lead to false warnings because, by definition, it analyzes an over-approximation of the target software, and is therefore imprecise. Careful use of application-specific abstract domains [A-89] can reduce the number of false warnings, but this requires more manual effort (in designing the domain) and can make the analysis more expensive (due to increased precision).

A.4.4.2.3.3 Software Model Checking

Software model checking [A-90] is a technique that combines model checking with automated abstraction and refinement to analyze programs. Model checking [A-91] is an automated and algorithmic approach to verify whether a finite-state machine (usually a Kripke structure) satisfies a temporal-logic (e.g., computation tree logic or linear temporal logic) formula. At a high level, a Kripke structure is a finite-state machine in which each state is labeled with atomic propositions denoting facts that are true in that state. Formally, a Kripke structure is a triple (S, R, L) where S is the set of states, R is the transition relation between states, and L is the labeling function. For example, figure A-3 shows a Kripke structure with three states $S = \{s_1, s_2, s_3\}$. There are two atomic propositions, p and q, and states are labeled with the atomic propositions true in them. Note that this means q is false in s_1 and p is false in s_2. Arrows denote transitions between states. A big advantage of model checking is its ability to return a counterexample demonstrating the failure of a property concretely. Counterexamples are valuable as diagnostic feedback to find and fix bugs.

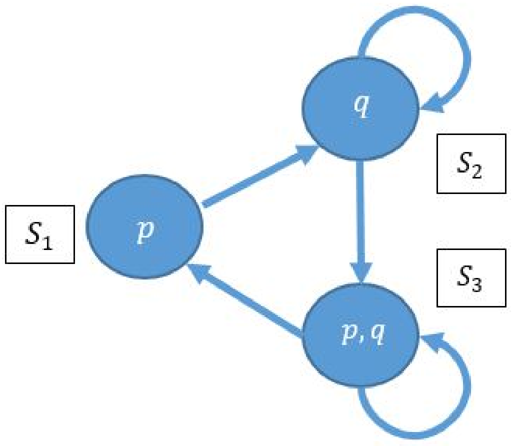


Figure A-3. A Kripke structure

In its original formulation, model checking was limited to verifying only finite-state systems. This makes its direct use for software analysis challenging because software has a large (or potentially infinite) number of states. In software model checking, an abstraction technique (such as predicate abstraction [A-92]) is first used to construct a finite conservative model of the program. This model is then verified with a model checker with respect to a target property. If the model has no bugs, then it is known that the original program is also correct (because the model is conservative). Otherwise, the model checker returns a counterexample. Because the counterexample was found by verifying the abstract model, it must be checked whether it corresponds to a real program behavior. If it does, a real bug has been found that can be reported to the software developers. Otherwise, the counterexample is spurious. It is used to refine the model, and the process is repeated with the improved model. This iterative procedure is called counterexample-guided abstraction refinement [A-93], or CEGAR, and was a crucial breakthrough that has led to the further development of software model-checking techniques and tools since the early 2000s.

A.4.4.2.3.4 Concolic Testing

Concolic testing [A-94] is a recent development that tries to bridge the gap between testing and exhaustive verification. The term concolic is a shortened version of "concrete + symbolic." As its name implies, concolic testing combines two techniques: concrete execution (i.e., testing) and symbolic execution. The basic idea is to start with a random input, and execute the software on that input. However, also keep track of the execution of the software symbolically. In particular, keep track of the variable assignments and branch decisions during the execution. This requires treating the software as a whitebox; therefore, other names of concolic testing are whitebox fuzzing and directed random testing. Next, use the variable assignments and branch decisions to construct another input on which the program would take a slightly different path (e.g., take the opposite decision on the last branch). Usually this is done by constructing a logical formula, checking for its satisfiability, and constructing the new input from a satisfying solution. The process is repeated until some threshold is exceeded. Note that concolic testing also aims for branch coverage like MC/DC but is semantically deeper. For example, it can leverage relationships between various branch conditions.

A.4.4.2.4 Exhaustive Verification Tools

Theorem-proving tools for software verification fall into two categories. The first is general-purpose theorem provers, such as Coq [A-95]. These can be applied to verify any kind of software if its semantics can be represented in the language of the prover. The other category of tools is auto-active verifiers [A-96] that are tailored to specific programming languages (e.g., Esc-Java [A-97] for Java, Dafny [A-98], and Frama-C [A-99] for ANSI C). Auto-active verifiers automate the process of constructing and proving verification conditions (using satisfiability modulo theory, or SMT, solvers) once the user has provided sufficient information in terms of loop invariants and function contracts. They are somewhat easier to use than general-purpose theorem provers because the target language's semantics are already encoded by the verifier. However, they are less generally applicable.

One of the most widely known abstract interpretation tools is ASTREE [A-100]. A large number of academic and commercial static analysis tools also use abstract interpretation. Notable examples are CodeSonar™ [A-101], Coverity [A-102], and Klocwork [A-103]. Again, this list is a sample of a large number of static analysis tools in existence. These tools are very scalable and have been applied to millions of lines of code. Their big drawback remains false warnings. Moreover, they can usually analyze only sequential code.

Software model checking is also supported by a wide range of tools. The most industrial example is Microsoft's Static Driver Verifier [A-104], which was a successful technology transfer of the SLAM [A-105] project. The SLAM project was motivated by the realization that a vast majority of Windows' "blue screens of death" were caused by bugs in device drivers over which Microsoft had no direct control. The success of SLAM indicates the applicability of software model checking to verify critical system-level code once domain-specific restrictions are used to focus the analysis. This is likely to be true for VM code as well. Other software model checkers include Java Pathfinder [A-106] from NASA. Also, a large number of academic tools have been and continue to be developed. Examples include BLAST [A-107], CBMC [A-108], MAGIC [A-109], and tools that participate in the annual Software Verification Competition [A-110].

Despite its recent development, concolic testing has already received considerable work in terms of tool development. For example, KLEE [A-111] is a concolic tester built on top of the LLVM [A-112] infrastructure. CUTE and JCUTE [A-113] are concolic testers for C and Java. PEX [A-114] is a concolic tester developed for the .NET framework at Microsoft. SAGE [A-115] is another concolic tester developed at Microsoft aimed at finding security vulnerabilities.

A.4.4.2.5 Logical Analysis of Virtualization Software

This section concludes with an overview of tools and results for logical analysis of virtualization software (i.e., software that implements both VMMs and guest Oss). The work in this area is nascent. All the tools and techniques described earlier in this section are applicable to virtualization software as well. Therefore, testing is applied routinely to VMM source code, and model checking and static analysis can be used to verify device drivers inside guest operating systems. However, logical analysis can also be applied to prove that VMMs implement isolation correctly. This requires overcoming two main challenges:

1. Hardware modeling: It is no longer sufficient to treat the VMM as a regular program. VMMs rely on critical hardware support to implement isolation. Examples include dynamic root of trust, nested page tables, and I/O memory management units (MMUs) to control access to shared devices. To verify the VMM, these hardware primitives must be modeled at a sufficient level of detail.

2. Guest/host concurrency: The execution of a virtualized system consists of a sequence of switches between guest and host modes. In each mode, certain restrictions hold in terms of memory and device access. Moreover, the switch can happen only when specific instructions are executed. This means that the guest and host cannot simply be treated as two threads interleaving with each other non-deterministically. Only specific points of context switching are allowed.

So far, overcoming these challenges has required careful decomposition of a virtualized system into a number of different components, each with its own interface in terms of its interaction with the rest of the system, including the hardware. This is followed by selective application of verification tools to each component to show that its implementation satisfies its interface. The final piece is an argument that the individual components and their interfaces imply the overall isolation property required. This approach has led to successful verification of hypervisors [A-116, A-15].

A.5 SAFETY STANDARDS

This section focuses on the literature addressing those issues that the use of VMs in avionics systems may have with respect to safety standards and related guidance documents. Perhaps the most important document to discuss is DO-178B/C [A-117].

A number of real-time operating systems (RTOS) compliant with ARINC 653 (e.g., Lynx [A-118]) implement some form of VM and address issues of test coverage as required by DO-178B Level A. In particular, DO-178B Level A software requires verification with MC/DC. This means that: 1) every entry and exit point has been exercised at least once, 2) every option in every decision point in a program must have been taken at least once, and 3) each condition in a decision must had been shown to affect the decision independently. This requirement affects how VMs and their hypervisors are implemented and verified. It is worth mentioning that the type of VMs implemented by ARINC 653-compliant RTOS is different from the type implemented by general-purpose VMs because RTOS focus on the partition implementation (temporal and spatial) described by the ARINC standard.

The DO-178C standard mentions a number of issues that must be addressed by VMs to provide the appropriate isolation demanded by the partition concept presented in the document. In particular, section 2.4.1 of the document lists the following relevant requirements for a partition [A-117]:

- A partitioned software component should not be allowed to contaminate another partitioned software component's code, I/O, or data-storage area.

- A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.

- Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components.

- Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components. (This requirement in particular is what forces Level A designation on hypervisors in compliant RTOS that implement some form of VM.)

- Any hardware providing partitioning should be assessed by the system-safety assessment process to ensure that it does not adversely affect safety.

Sections 6.3 and 6.4 of DO-178C discuss the need to verify the timing characteristics of software, specifically its WCET and worst-case response time. It is important for a VM implementation to be backed up by an appropriate method and analysis that could obtain both of these figures. Similarly, the use of low-level requirements, such as bus loading, must also be verified.

Of particular importance is the consideration needed for interrupts (DO-178C, section 11.7) with respect to their timing and interference with different partitions. In particular, VMs virtualize interrupts, and this may delay their timely delivery to the target partition. It is necessary to ensure that the implementation of virtualized interrupts does not violate the requirements of an application. Similarly, it is necessary to ensure that an interrupt intended for one partition does not interfere with another partition.

Finally, DO-178C requires that software be checked to ensure that "the protection mechanisms for exceeded frame times respond correctly" [A-117]. For VM implementation, it should be clear what the response would be and that the partitions with higher criticality levels are not affected.

Of particular importance is fault identification, isolation, and resolution. A fault raised in one partition shall not affect the error handling in another partition.

A.6 REFERENCES

A-1. Ducklin, P. (2015). "Row hammering" – how to exploit a computer by overworking its memory. Retrieved from https://nakedsecurity.sophos.com/2015/03/12/row-hammering-how-to-exploit-a-computer-by-overworking-its-memory/

A-2. Popek, G. J., and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM, 17*(7), 412–421.

A-3. Buzen, J. P. and Gagliardi, U. O. (1973). The evolution of virtual machine architecture. *AFIPS '73 Proceedings of the June 4-8, 1973, national computer conference and exposition*, (291–299).

A-4. Goldberg, R. P. (1971). *Virtual machines --- semantics and examples,* Proceedings from the IEEE International Computer Society Conference, Boston, MA.

A-5. Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. (2012). Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems, 30*(4), Article 12.

A-6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., … Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review - SOSP '03 37*(5), 164–177.

A-7. Duda, K. J. and Cheriton, D. R. (1999). *Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*. Proceedings from Proceedings of the seventeenth ACM symposium on Operating systems principles. (261–276) Charleston, SC.

A-8. Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D, … Hyden, E. (2006). The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications, 14*(7), 1280–1297.

A-9. Waldspurger, C. A. (1995). *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management,* (PhD thesis), Massachusetts Institute of Technology Cambridge, MA.

A-10. Lowe, S. (2012). Hyper-V™ vs. vSphere™: Understanding the differences Retrieved from http://www.apmdigest.com/sites/default/files/images/VMvSphereHyperV_Whitepaper.pdf

A-11. Microsoft. (2013). Why Hyper-V? Competitive Advantages of Microsoft Hyper-V Server 2012 over the VMware vSphere Hypervisor. Retrieved from http://download.microsoft.com/download/5/7/8/578e035f-a1a8-4774-b404-317a7abcf751/competitive-advantages-of-hyper-v-server-2012-over-vmware-vsphere-hypervisor.pdf.

A-12. Dall, C. and Nieh, J. (2014). KVM/ARM: the design and implementation of the linux ARM hypervisor. In ASPLOS '14: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems. (333–348).

A-13. Oracle Corporation. (Copyright 2004-2015). *Oracle VM VirtualBox User Manual*. Retrieved from https://www.virtualbox.org/manual/UserManual.html.

A-14. Xi, S., Wilson, J., Lu, C., and Gill, C. (2011). RT-Xen: towards real-time hypervisor scheduling in xen. In Proceedings of the ninth ACM international conference on Embedded software (EMSOFT '11).

A-15. Vasudevan, A., Chaki, S., Jia, L., McCune, J.M., Newsome, J., Datta, A. (2013). Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. *2013 IEEE Symposium on Security and Privacy*, 430–444.

A-16. Xi, S.; Xu, M.; Lu, C.; Phan, L.T.X.; Gill, C.; Sokolsky, O.; Lee, I. Real-time multi-core virtual machine scheduling in Xen. Proceedings from the 2014 International Conference on Embedded Software (EMSOFT). Jaypee Greens, India.

A-17. Mercer, C. W., Savage, S., and Tokuda, H. (1994). Processor capacity reserves: Operating system support for multimedia applications. In *1994 Proceedings of IEEE International Conference on Multimedia Computing and Systems*. Piscataway, NJ: IEEE.

A-18. Oikawa, S., and Rajkumar, R. (1999). Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium.* Piscataway, NJ: IEEE.

A-19. Sha, L.; Rajkumar, R.; Sathaye, S. S. (1994). Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. *Proceedings of the IEEE, 82*(1), 68–82.

A-20. Joseph, M., Pandaya, P. (1986). Finding response times in a real-time system. *The Computer Journal, 29*(5), 390–395.

A-21. Liu, C. L.; Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment, *Journal of the ACM, 20*(1), 46–61.

A-22. Strosnider, J.K.; Lehoczky, J.P.; Sha, L., (1995). The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers, 44*(1), 73–91.

A-23. Lehoczky, J. P., Sha, L., Strosnider, J. K. (1987). Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings from the Real Time Systems Symposium 1987* (261–270). Piscataway, NJ: IEEE.

A-24. Sprunt, B., Sha, L., Lehoczky, J. P. (1989). Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems, 1*(1), 27-60.

A-25. Sha, L., Lehoczky, J.P., & Rajkumar, R. (1986). Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. *RTSS.*

A-26. Shin, I., Lee, I. (2004). Compositional real-time scheduling framework. 25th IEEE International Real-Time Systems Symposium, 57-67.

A-27. Sprunt, B., Sha, L., & Lehoczky, J.P. (1989). CMU / SEI-89-TR-11 ESD-TR-89-19 Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.

A-28. Ghosh, S., Rajkumar, R. R. (2002). Resource management of the OS network subsystem. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*. Piscataway, NJ: IEEE.

A-29. Saewong, S., Rajkumar, R. (1999). Cooperative scheduling of multiple resources. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*. Piscataway, NJ: IEEE.

A-30. Kim, H., Kandhalu, A., & Rajkumar, R. (2013). A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. *2013 25th Euromicro Conference on Real-Time Systems*, 80–89.

A-31. Kim, H., Niz, D.D., Andersson, B., Klein, M.H., Mutlu, O., & Rajkumar, R. (2014). Bounding memory interference delay in COTS-based multi-core systems. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 145–154.

A-32. Niz, D.D., Saewong, S., Rajkumar, R., Abeni, L. (2001). Resource Sharing in Reservation-Based Systems. *IEEE Real Time Technology and Applications Symposium*.

A-33. Ferdinand C., Heckmann R., Wilhelm R. (2006) Analyzing the Worst-Case Execution Time by Abstract Interpretation of Executable Code. In: Broy M., Krüger I.H., Meisinger M. (eds) *Automotive Software – Connected Services in Mobile Networks*. ASWSD 2004. Lecture Notes in Computer Science, vol 4147. Springer, Berlin, Heidelberg.

A-34. Vestal, S. (2007). Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 239–243.

A-35. Burn, A., Davis, R. I., (2017). Mixed-Criticality Systems – A Review. Retrieved from https://www-users.cs.york.ac.uk/burns/review.pdf

A-36. Baruah S.K., Bonifaci V., D'Angelo G., Marchetti-Spaccamela A., van der Ster S., Stougie L. (2011) Mixed-Criticality Scheduling of Sporadic Task Systems. In: Demetrescu C., Halldórsson M.M. (eds) Algorithms – ESA 2011. ESA 2011. Lecture Notes in Computer Science, vol 6942. Springer, Berlin, Heidelberg.

A-37. Niz, D.D., Lakshmanan, K., & Rajkumar, R. (2009). On the Scheduling of Mixed-Criticality Real-Time Task Sets. *2009 30th IEEE Real-Time Systems Symposium*, 291–300.

A-38. Lakshmanan, K., Niz, D.D., Rajkumar, R., & Moreno, G.A. (2010). Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems. *2010 IEEE 30th International Conference on Distributed Computing Systems*, 169–178.

A-39. Niz, D.D., & Phan, L.T. (2014). Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 111–122.

A-40. Niz, D.D., Wrage, L., Rowe, A., & Rajkumar, R. (2013). Utility-based resource overbooking for Cyber-Physical Systems. *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, 217–226.

A-41. Lakshmanan, K., Niz, D.D., & Rajkumar, R. (2011). Mixed-Criticality Task Synchronization in Zero-Slack Scheduling. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 47–56.

A-42. Kopetz, H. (1998). The Time-Triggered Architecture. *Proceedings of the IEEE, 91*(1), 112–126.

A-43. Kopetz, H., & Grünsteidl, G. (1993). TTP - A time-triggered protocol for fault-tolerant real-time systems. *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 524–533.

A-44. Steiner, W. (2010). An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. *2010 31st IEEE Real-Time Systems Symposium*, 375–384.

A-45. Suethanuwong, E. (2012). Scheduling time-triggered traffic in TTEthernet systems. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 1–4.

A-46. Leen, G., Heffeman, D. (2002). TTCAN: a new time-triggered controller area network. *Microprocessors and Microsystems*, *26*(2), 77–94.

A-47. FlexRay (2010). *FlexRay Consortium*. FlexRay Communications System Protocol Specification Version 3.0.1.

A-48. RTCA. (2005). Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. (RTCA DO-297).

A-49. Korst, J. H. M., Aarts, E. H. L., and Lenstra, J. K. (1996). Scheduling Periodic Tasks. *INFORMS Journal on Computing, 8*(4), 428–435.

A-50. Kim, J., Yoon, M., Bradford, R.M., Sha, L. (2014). Integrated Modular Avionics (IMA) Partition Scheduling with Conflict-Free I/O for Multicore Avionics Systems. *2014 IEEE 38th Annual Computer Software and Applications Conference*, 321–331.

A-51. Samolej, S. (2009). ARINC Specification 653 Based Real-Time Software Engineering. *e-Informatica Software Engineering Journal, 5*(1), 39–49.

A-52. Kim, J., Abdelzaher, T.F., Sha, L. (2015). Schedulability bound for integrated modular avionics partitions. *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 37–42.

A-53. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, …Winwood, S. (2009). seL4: formal verification of an OS kernel. *SOSP*, 207–220.

A-54. Rushby, J.M. (1981). Design and Verification of Secure Systems. *SOSP*.

A-55. Waldspurger, C.A. (2002). Memory resource management in VMware ESX server. SIGOPS Oper. Syst. Rev. 36, SI.

A-56. Advanced Micro Devices, (2008). *Inc.AMD-V Nested Paging*. Retrieved from http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf

A-57. Docker. (n.d.). https://www.docker.com

A-58. Linux Containers. (n.d.). https://linuxcontainers.org/

A-59. OpenVZ. (n.d.). https://openvz.org/

A-60. Virtuozzo. (n.d.). http://www.virtuozzo.com/

A-61. Park, S., Olds, W., Shin, K.G., Wang, S. (2007). Integrating Virtual Execution Platform for Accurate Analysis in Distributed Real-Time Control System Development. *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 61–72.

A-62. Kim, H., Wang, S., Rajkumar, R. (2014). vMPCP: A Synchronization Framework for Multi-core Virtual Machines. *2014 IEEE Real-Time Systems Symposium*, 86–95.

A-63. Kim, H., Wang, S., Rajkumar, R. (2015). Responsive and Enforced Interrupt Handling for Real-Time System Virtualization. *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, 90–99.

A-64. Lee, J., Xi, S., Chen, S., Phan, L.T., Gill, C.D., Lee, I., Lu, C., Sokolsky, O. (2012). Realizing Compositional Scheduling through Virtualization. *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, 13–22.

A-65. CARTS: Compositional Analysis of Real-Time Systems. (n.d.). https://rtg.cis.upenn.edu/carts/index.php.

A-66. Danish, M., Li, Y., West, R. (2011). Virtual-CPU Scheduling in the Quest Operating System. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 169–179.

A-67. Pajic, M., Mangharam, R. (2010). Embedded Virtual Machines for Robust Wireless Control and Actuation. *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 79–88.

A-68. Kaldewey, T., Wong, T.M., Golding, R.A., Povzner, A., Brandt, S.A., Maltzahn, C. (2008). Virtualizing Disk Performance. *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, 319–330.

A-69. Bruns, F., Traboulsi, S., Szczesny, D., Gonzalez, M.E., Xu, Y., Bilgic, A. (2010). An Evaluation of Microkernel-Based Virtualization for Embedded Real-Time Systems. *2010 22nd Euromicro Conference on Real-Time Systems*, 57–65.

A-70. Rajan, A., Whalen, M.W., Heimdahl, M.P. (2008). The effect of program and model structure on mc/dc test adequacy coverage. *2008 ACM/IEEE 30th International Conference on Software Engineering*, 161–170.

A-71. Kuhn, D.R., Kacker, R., Jia, H., & Hunter, J. (2009). Combinatorial Software Testing. *Computer, 42*(8), 94–96.

A-72. Sutton, M., Greene, A., Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Boston, MA: Addison-Wesley. (ISBN 0-321-44611-9).

A-73. JUnit. http://junit.org/

A-74. CUTE. http://www.cute-test.com/

A-75. How to: Create and Run a Unit Test. https://msdn.microsoft.com/library/dd286656%28v=vs.100%29.aspx

A-76. Similunk Test. http://www.mathworks.com/products/simulink-test/

A-77. Reactive Systems Inc. http://www.reactive-systems.com/

A-78. Vector Software. https://www.vectorcast.com/software-testing-products

A-79. OWASP WSFuzzer. https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project

A-80. Dranzer. https://www.cert.org/vulnerability-analysis/tools/dranzer.cfm?

A-81. PeachFuzzer. http://www.peachfuzzer.com/

A-82. Hoare, C.A. Retrospective: An Axiomatic Basis for Computer Programming. *Communications of the ACM, 12*(10), 576–580.

A-83. Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics, 19*, 19–31.

A-84. Owicki, S.S., Gries, D. (1976). Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM, 19*(5), 279–285.

A-85. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R., Schulte, W. (2004). Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology, 3*(6), 27–56.

A-86. Cousot, P., Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *POPL*. 238–252.

A-87. Cousot, P., Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proc. of the 2nd International Symposium on Programming*. 106–130.

A-88. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X. (2011). Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes, 36*(1), 1–8.

A-89. Chen, L., Miné, A., Cousot, P. (2008). A Sound Floating-Point Polyhedra Abstract Domain. *APLAS*. 3–18.

A-90. Jhala, R., Majumdar, R. (2009). Software model checking. *ACM Comput. Surv., 41*(4), 21:1-21:54.

A-91. Clarke, E.M., Emerson, E.A. (1981). Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs*. 52–71.

A-92. Graf, S., Saïdi, H. (1997). Construction of Abstract State Graphs with PVS. *CAV*. 72–83.

A-93. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM, 50*(5), 752–794.

A-94. Sen, K. (2007). *Concolic testing*. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. (571–572). Atlanta, GA.

A-95. The Coq Proof Assistant. https://coq.inria.fr/

A-96. K. Rustan M. Leino and Michał Moskal: Usable Auto-Active Verification. http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf.

A-97. Flanagan, C., Leino, K.R., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R. (2013). PLDI 2002: Extended static checking for Java. *SIGPLAN Notices, 48*(4S) 22–33.

A-98. Leino, K.R., & Wüstholz, V. (2014). The Dafny Integrated Development Environment. *F-IDE*. 3–15.

A-99. Frama-C. http://frama-c.com/

A-100. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X. (2005). The ASTREÉ Analyzer. *ESOP*. 21–30.

A-101. GrammaTech CodeSonar. http://www.grammatech.com/products/codesonar

A-102. Coverity. http://www.coverity.com/

A-103. Klocwork. http://www.klocwork.com/

A-104. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J. (2010). The Static Driver Verifier Research Platform. *CAV*. 119–122.

A-105. Ball, T., Cook, B., Levin, V., Rajamani, S.K. (2004). SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. *IFM*. 1–20.

A-106. Visser, W., Havelund, K., Brat, G. P., Park, S., Lerda, F. (2003). Model Checking Programs. *Automated Software Engineering, 10*(2), 203–232.

A-107. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G. (2002). Lazy abstraction. *POPL*. 58–70.

A-108. Clarke, E.M., Kroening, D., Lerda, F. (2004). A Tool for Checking ANSI-C Programs. *TACAS*. 168–176.

A-109. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H. (2003). Modular verification of software components in C. *IEEE Transactions on Software Engineering, 30*(6), 388–402.

A-110. SVCOMP 2016. http://sv-comp.sosy-lab.org/2016/

A-111. Cadar, C., Dunbar, D., Engler, D.R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI*. 209–224.

A-112. The LLVM Compiler Infrastructure. http://llvm.org

A-113. Sen, K., Agha, G.A. (2006). CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper). 419–423.

A-114. Pex development team. Pex (2007), http://research.microsoft.com/Pex

A-115. Godefroid, P. (2007). Random testing for security: blackbox vs. whitebox fuzzing. Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). Atlanta, GA.

A-116. Leinenbach D., Santen T. (2009) Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti A., Dams D.R. (eds) *FM 2009: Formal Methods*. FM 2009. Lecture Notes in Computer Science, vol 5850. Springer, Berlin, Heidelberg.

A-117. RTCA. Software Considerations in Airborne Systems and Equipment Certification. (RTCA DO-178B).

A-118. Lynx Software Technologies. http://www.lynx.com/products/real-time-operating-systems/lynxos-178-rtos-for-do-178b-software-certification/

APPENDIX B—ASSURANCE ISSUES ON VIRTUAL MACHINES IN AVIONICS SYSTEMS

B.1 INTRODUCTION

Virtualization is a technique that enables the creation of new resources that behave mostly like real resources. For example, virtualization allows forming, from a single processor, one or multiple virtual processors. Any program can then execute directly on a virtual processor just as on a real processor. Typically, it is done as follows: one piece of software called a hypervisor (also known as a virtual machine monitor, or VMM) executes directly on the processor and forms the virtual processors. When a program executes a nonprivileged instruction in the virtual processor, it executes directly on the processor. However, if a program executes a privileged instruction, then it generates a trap, and the hypervisor takes over and emulates the instruction. In this way, a virtual machine (VM) allows the execution of an operating system (OS). Furthermore, it is also possible to execute different OSs on different VMs.

The notion of virtualization was originally formed in the 1970s for mainframe computers, and it was revived in the late 1990s and early 2000s because of its use in server farms for Internet applications and data warehousing. In the mid-2000s, virtualization became more widespread in desktop computers. Recently, virtualization has received increasing attention in safety-critical systems (e.g., avionics and automotive).

Virtualization brings a number of potential advantages:

- Improved security: The hypervisor is typically small and is therefore more amenable to formal analysis or testing with high coverage. If malware is introduced in one VM, it does not (unless the hypervisor is also compromised) infect the other VMs.
- Greater flexibility potentially leading to better resource usage: A program that is started in one VM can be stopped and checkpointed, allowing it to resume on another VM.
- Better support for legacy hardware: Consider a program written for an old OS and for an old processor. Assume that this program offers functionality of great value, and the user would like to have this functionality in a new airplane, but this old operating system and old hardware are no longer available. The old program can be run by using modern hardware and a hypervisor to build a VM for the old processor under this hypervisor. The old operating system and old application are then run in this VM.
- Potential for simpler integration: With the trend of shifting from federated architectures to integrated architectures, it is necessary to put multiple applications (from potentially different suppliers) into a single computer. However, each application may have real-time requirements and is designed with the assumption that it has its own processor. Virtualization is potentially of value in fulfilling this assumption.

For avionics, despite these advantages, there is a potential roadblock because assurance issues are critical. It is not obvious how the use of virtualization affects assurance and current guidance for certification of airplanes and associated approval of software (e.g., DO-178C). This is the subject of this report.

The remainder of this report is organized as follows. Section 2 presents goals and properties of VMs. Section 3 presents additional certification complexity due to virtualization. Section 4 discusses isolation, contrasting VMs with other technologies. Section 5 discusses development process issues. Section 6 discusses assurance of avionics systems. Finally, section 7 gives conclusions.

B.2 GOALS AND PROPERTIES OF VIRTUAL MACHINES

In this section, the goals of the different implementations of VMs and the properties of these implementations are discussed. The subtleties of these properties and to what extent they are achieved will also be discussed.

B.2.1 TEMPORAL ISOLATION/PROTECTION/PARTITIONING

Discussions of temporal isolation can be traced back to the beginning of multiprogramming environments. When computers were only used to execute multiple batch jobs at the same time, the main metric of performance was the number of jobs completed per unit of time. Once multiple users were connected to the same computer, response time became an issue. In particular, the response time experienced by a user was satisfactory if it was possible to preserve the illusion that he or she was the only one using the computer. In this context, temporal isolation was interpreted as an effective illusion of having one dedicated computer per user.

Batch processing and interactive use are two usage types that define metrics for system utilization and application requirements, respectively. These two types of metrics play an important role in the implementation and properties of virtualization mechanisms.

For general-purpose computing, both system utilization and application requirements are evaluated in an average-case fashion. Specifically, system utilization defined as the average number of jobs per unit of time is known as throughput. However, response time plays a role in the specification of service-level agreements (SLAs) used now to contract virtualized computing services from cloud providers. In some cases, a maximum response time is specified in an SLA [B-1], but it does not have the same character as the worst-case response time that real-time systems demand. In particular, SLAs also include language that specifies availability, how things are measured (e.g., by probes), and even procedures for dispute resolution. The result is that the response time has the character of an average case within a window of time. In the end, the fact that general-purpose computing workloads are never clearly defined leaves system designers with only the possibility to predict future workloads based on (average) history and dynamic mechanisms that react to that history. This is the case, for instance, of scheduling policies that favor interactivity by giving the processor to perform short computations and quickly return to wait for input (e.g., assumed to be from a user in a terminal).

The average-case character of timing metrics in information systems has created a trend that permeates the design of mechanisms for sharing resources across all the layers of the computer infrastructure. These layers include hardware, OSs, networks, and ultimately VMs. More importantly, design decisions at all these levels have implications for the properties of temporal isolation offered by different variations of VMs, which is the main topic of this section. Given the large influence of general-purpose computing on the processors' design, trends in this area and the consequences for the goals and properties of temporal isolation mechanisms will first be discussed. This discussion will be followed with one about real-time systems.

B.2.1.1 General-Purpose Computing

In the hardware area, the influence of the average timing metrics can be better understood from the processor design point of view. In particular, the main performance metric of a processor architecture is the instructions per cycle (IPC) that, with its cycles per second (Hertz), define its processing speed. It is worth noting that IPC is measured as the average number of instructions per cycle. This focus has driven the processor innovations toward improving the IPC of single threads. Three types of innovations are important for our discussion: 1) speculative execution, 2) locality-based memory access speedup, and 3) asynchronous I/O.

B.2.1.1.1 Speculative Execution

The innovations in speculative execution are aimed at keeping all the stages of the pipeline of a processor full. In particular, the execution of instructions in most processors today is organized in a pipeline that is divided into a number of stages. For instance, consider a pipeline divided into five stages: Instruction Fetch (IF), Instruction Decoding (ID), Execution (EX), Memory Access (MEM), and Result Write Back (WB) [B-2]. With this pipeline, a processor can execute a sequence of instructions (numbered 1 to *N*) from a program as follows: At Time 1, Instruction 1 executes in stage IF (i.e., the instruction is fetched). As soon as Instruction 1 moves to stage ID, then Instruction 2 can start in IF. The third instruction can start in IF as soon as Instruction 2 moves to stage ID and Instruction 1 moves to stage EX. Following this pattern, in steady state, the processor executes five instructions in parallel (at different stages) at the same time. In this case, the processor can finish one instruction every cycle, assuming that each stage can be executed in one cycle.

Unfortunately, the pipeline execution assumes an uninterrupted sequence of instructions. This sequence can be interrupted by four type of events: 1) jumps to an instruction other than the next one (branch instructions), 2) data dependencies between instructions, 3) instruction dependencies from memory access, and 4) thread (context) switch.

Branching: To deal with branch instructions, processors use branch prediction. For an unconditional branch, the prediction is basically certain, and it is only necessary to fetch instructions from the branch target address. However, conditional branches are more challenging. Conditional branch instructions implement programming language constructs such as if, while, and for. Branch prediction schemes use different heuristics to guess the next instructions to fetch. For instance, for a “while” implementation, a good guess is to assume that the program will remain in the loop and keep fetching from inside the while loop. In this way, as soon as the prediction is correct, the pipeline continues to complete the execution of one instruction per cycle without interruption. However, when the prediction is wrong, the processor needs to discard the half-executed instructions fetched from the wrong address (e.g., the inside of the while when the while terminates) without causing side effects. This means that the pipeline execution is stopped in the middle, intermediate results are eliminated, and new instructions from the correct address are started. Obviously, this correction has a time penalty that must be considered. In the end, branch prediction provides significant improvement to the IPC but makes it difficult to figure out the worst-case execution time (WCET) of a set of instructions.

Data Dependencies: Data dependencies between two consecutive (or close) instructions also cause delays in the pipeline. For instance, a program has the following two instructions (one after the other): "ADD R1, R2, R3; MUL R5, R1, R7," in which MUL reads register R1 that is modified by the ADD instruction. The MUL instruction cannot execute until the ADD instruction has written back its result. For our sample pipeline processor, this may mean that the MUL instruction cannot execute stage ID until ADD finishes stage WB. It is worth noting that, in steady state, ADD would arrive at stage EX at the same time that MUL arrives at stage ID. Therefore, MUL will have to wait for ADD to finish stages EX, MEM, and WB (three "idle" cycles) before it can continue executing. These waits (or delays) are known as stalls.

Memory Access Dependencies: Memory access dependencies are more costly and have deeper implications for temporal isolation. To show this, consider a seven-instruction program as follows: "LOAD R1, [100]; INCR R1; INCR R2; LOAD R3 [200]; ADD R1,R1,R2; STORE [100],R1; LOAD R2,[100]." In this case, the first load cannot complete until the memory access to address 100 is completed. This creates a stall, depending on whether the data in address 100 is loaded in one of the caches or main memory; that can take up to four cycles for L1 cache, tens of cycles for L2 and L3 caches, and hundreds of cycles for dynamic random-access memory (DRAM) access. This stall affects other dependent instructions like the second instruction (INCR R1). To minimize the effect of the stall, processors try to keep executing nondependent instructions, such as the third instruction (INCR R2), that do not depend on previous instructions.

This can be done by reordering the instructions to keep non-stalled instructions executing in the pipeline while stalled instructions wait for the memory access to complete. Reordering has two effects. First, multiple reads can be issued back to back. In the seven-instruction program, this will happen if we move the LOAD R3 [200] instruction to the second place. This generates what is known as multiple pending memory operations, or multi-issues. These include not only multiple reads but also multiple writes. More importantly, because writes can be delayed until there is another instruction set that accesses the same address, it is quite common to have a write queue to store and defer writes. In the example, the STORE [100] R1 instruction could have been delayed if it was not followed by the last LOAD R2 [100] that depends on it. Write queues have a significant impact on temporal isolation mechanisms and their properties, as will be discussed further in this section.

Context Switch: A thread context switch clearly disrupts the sequence of instructions fed into the pipeline. This means that the pipeline is "restarted," clearing all the stages and creating a stall at least as long as the size of the pipeline. Given that context switches are relatively infrequent in comparison to the speed of the pipeline, they do not play a big role. However, thread contexts play a bigger role in a related concept. Specifically, different instructions use different processor functional units, like the arithmetic logic unit or the floating-point unit, and a modern processor will have more than one unit of each type to speed up execution. However, the number of instructions that can use these units simultaneously is limited by the dependencies that these instructions have on one another within the executing thread.

As a result, processors implement what is known as simultaneous multithreading (known commercially as hyperthreading) to have multiple sequences of instructions executing simultaneously. This means that a larger number of instructions (from different hyperthreads) will be in the EX stage and can execute in different types of functional units (e.g., one in an arithmetic logic unit and another in a floating point unit) or multiple instances of the same unit. Hyperthreading, however, basically gives the scheduling decision to the processor (what thread runs first), diluting the predictability of the WCET of a thread. A second issue with thread context switches is the fact that data loaded into the cache by a thread can be evicted by the thread that is replacing it in the context switch. This discussion will be deferred to the next section.

B.2.1.1.2 Memory Access Speedup

Some memory access speedup techniques are well known by most computer users. Cache memory plays a critical role in achieving the processing speed of current processors. Specifically, whereas processor speed doubled every 2 years some decades ago, memory speed increased only linearly. As a result, to keep feeding instructions to the processor fast enough to prevent it from going idle (memory stalls), a hierarchy of cache memory was designed to keep the most frequently used data in faster (but smaller) memory areas. The cache hierarchy is organized into levels known as Level 1 (L1), Level 2 (L2), Level 3 (L3), and so on, of increasingly slower but larger memory (with L1 running at the same speed as the processor). Automatic and transparent data replacement algorithms take care of loading the most frequently used data in a successive manner from memory into L3, L3 into L2, and L2 into L1. Cache replacement algorithms exploit the data and time locality of program executions that are broken whenever a thread context switch occurs. With two orders of magnitude difference between memory and processor speed, a context switch not only can be costly in terms of time but also may induce unpredictable behavior that can affect the temporal isolation properties if not taken into account.

The DRAM memory system includes a number of mechanisms tailored to improve the IPC of the processor. The three main components of the DRAM memory system are the memory bus, the memory controller, and the DRAM chips, as shown in figure B-1 [B-3, B-4]. The memory controller schedules memory read/write requests that are sent to the DRAM chips through the memory bus. The DRAM chips then retrieve the data and send it back through the memory bus. The internal mechanisms of these components that influence timing and the effectiveness of temporal partitioning will be briefly discussed.

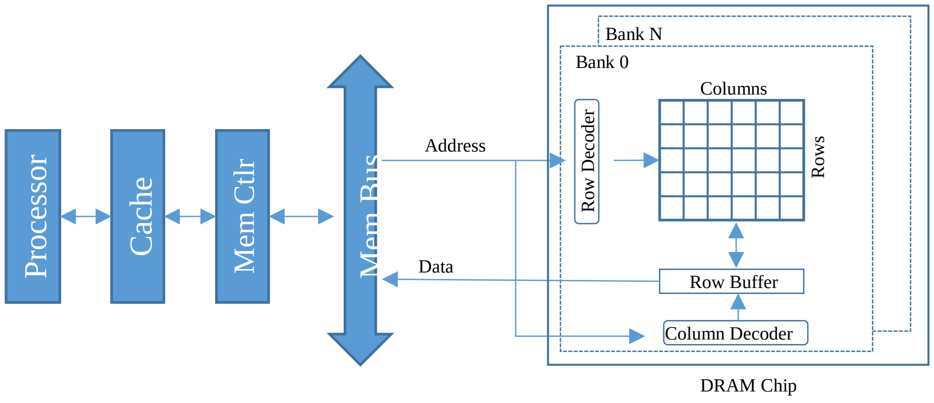


Figure B-1. DRAM memory system organization

DRAM Chip: The DRAM chip is organized as a set of banks, and each bank is organized in a matrix of rows and columns. More specifically, when a memory location is read from DRAM, the bank number *B*, row number *R*, and column number *C* are extracted from the memory address. Next, a request to load the row *R* from the memory matrix of the bank B is sent. Then another command is sent to read the column *C* from the row buffer. This row buffer acts as another caching strategy to reduce the time needed to access the memory. In this way, any additional memory request (column) from the same row can avoid sending the load row command. Obviously, this strategy assumes that memory is accessed in a sequential fashion within a bank. Furthermore, following this strategy, the mapping of addresses to banks and rows is organized in a row-interleaving fashion. For instance, memory address 0 would start in bank 0, row 0, and column 0. For instance, every 8 bytes it will move to the next column; once the columns of the row are exhausted, instead of moving to the next row, it moves to the next bank. This allows the creation of a "virtual" row that puts all the rows of all the banks back to back in a large row buffer, increasing the row-buffer locality.

Memory Controller: The memory controller schedules read/write requests through a policy known as first-ready, first-come, first-served (FR-FCFS). FR-FCFS uses one queue per bank, in which the memory controller puts the requests in a particular bank as they arrive. Then, each bank queue is reordered by moving to the back of the queue the requests that are "not ready" or when a new request arrives to a row that is currently in the row buffer. A request is considered not ready if executing it would violate the timing restrictions of the memory system (e.g., inter-bank row-activate). For requests to the row in the buffer, the controller is able to put these requests to the front of the queue but imposes a limit to prevent starvation. However, the end result is that requests to rows other than the loaded one can suffer significant delays.

As expected, switching to another thread interrupts the "sequential" memory access assumed by the DRAM memory system. However, thanks to the cache, the effects on single-core processors are not significant. Unfortunately, this is not the case for multicore systems, in which the execution time of a program running in one core can increase up to 12 times [B-4] or more because programs running in other cores and the interference in the memory system. Clearly, this is a critical issue when temporal isolation is the key.

B.2.1.1.3 Asynchronous I/O

I/O devices that transfer a large amount of data, such as video acquisition cards (e.g., camera) and network interfaces, use the direct memory access (DMA) mechanism to move data from their internal buffers directly into main memory. Unfortunately, this means that such a transfer can happen at any time the I/O device is ready and can interrupt or slow down the memory access of a program running in the processor. From the temporal isolation point of view, this means that a thread may be slowed down because of the DMA transfer for an I/O request from another thread.

B.2.1.1.4 Thermal Management

Thermal dissipation in today's commercial off-the-shelf (COTS) processors is performed with a combination of passive and active mechanisms. The active mechanism is known as dynamic thermal management (DTM). Specifically, DTM allows the processor to run at frequencies that can potentially drive the temperature of the processor to unsafe levels. DTM complements this approach with a dynamic monitoring of the temperature. When it reaches a certain threshold, it reduces the speed and, therefore, the temperature of the processor to prevent it from burning out. This approach allows the processor to speed up on a burst of workload, increasing the average-case performance.

B.2.1.1.5 Operating System

In general-purpose OSs, the same average-case metrics are considered. Research in these systems has produced another metric that is also applied at a lower level: fairness. A system is considered fair when it allocates the same proportion of resources to all clients (e.g., threads). For instance, a processor scheduling policy is consider fair if it allocates the same proportion of central processing unit (CPU) time to all the threads in a system. The objective of a round-robin scheduler is to give turns to all the threads in a round-robin fashion (as the name implies), giving a specific CPU time slice to a thread in each turn. In the ideal case, if the time slice is made as close as possible to zero, then it can be claimed the system is completely fair. However, a nonzero time slice will allow a difference of one time slice of CPU time between threads. The size of the time slice offers a tradeoff in system design between fairness and overhead of the processor (i.e., the smaller the time slice, the fairer the system), but it incurs more context switches that waste CPU time. Whereas this approach provides some average-case temporal isolation, it cannot distinguish different requirements from different tasks and cannot provide strong guarantees.

B.2.1.1.6 Virtual Machines

VM implementations for general-purpose computing, such as those used in cloud computing (VMWare, Xen, Virtual Box), focus on the fairness-overhead tradeoff, but at the level of individual VMs. More specifically, a hypervisor gives a time slice to a VM and, inside the VM, the host OS gives turns to the threads. With today's processor speeds, this scheme provides a reasonable response time to users of information systems but also limits the number of VMs that can be supported while providing a reasonable response time.

In the final analysis, the cumulative layers of scheduling decisions made to optimize the average-case response time and throughput can create critical variations that in turn make the evaluation of the worst-case response time both very difficult and very pessimistic in a general-purpose VM.

For the same reason, the temporal isolation that can be achieved within a general-purpose VM can be expected to satisfy human interaction or average throughput but can have no reasonable guarantee on worst-case behavior.

B.2.1.2 Real-Time Embedded Systems

Real-time embedded systems (RTES) are in some sense simpler systems than general-purpose computing systems. In particular, the workload of an RTES is well defined with a specific task set (set of threads) and clear parameters (such as WCET, execution rate) and timing requirements (deadlines). As a result, the mechanisms, metrics, and evaluation methods can be more easily defined. At the same time, the validation criteria is more stringent. Instead of judging whether an expected response time should be met most of the time, as happens in general-purpose computing, RTES ensures that it should be met all the time under any circumstance (i.e., tasks must always meet their deadlines). Because the exact timing characterization of an RTES is not always possible, over-approximation techniques of worst-case behavior (e.g., WCET, worst-case delays, and worst-case response time) are used to ensure that meeting deadlines can still be guaranteed.

As expected, the worst-case mindset has created different mechanisms at all layers of the computing infrastructure. These mechanisms are aimed at providing a more predictable execution time. However, sometimes predictability comes at the cost of a slower average execution time. As a result, innovations and products used to improve IPC and throughput sometimes are also used in RTES. In this case, it is important to develop complementary mechanisms and evaluation that properly over-approximate the worst-case behavior.

B.2.1.2.1 Execution Time Predictability Versus Speedup

In the past, and in low-end embedded processors, predictability was the driving force. As a result, these processors were designed as single-issue (only one memory request can be pending) with non-speculative execution [B-5]. However, new high-end embedded processors implement complex pipelines with speculative execution to reach the required execution speed demanded by today's applications. This is the case, for instance, of the ARM11 MPCore processor [B-6] that implements an eight-stage pipeline to get close to an IPC of 1. As expected, this processor implements speculative execution techniques such as branch prediction.

The use of complex mechanisms to improve a processor's IPC can complicate obtaining the WCET of a task. For simple architectures (e.g., ARM Cortex-M3), formal techniques like abstract interpretation are capable of obtaining a tight WCET [B-7]. For complex architectures, more traditional measuring techniques are used.

B.2.1.2.2 Memory Access Speedup

Whereas embedded processors use a cache hierarchy to speed up memory access, some processors also implement what is known as scratchpad memory. Scratchpad memory uses the same fast memory technology as scratchpad RAM (SRAM), but in this case, the developer is in charge of loading (and replacing) the memory that is needed to run faster.

Just as for general-purpose computing, context switching between tasks can require reloading data into the cache (and evicting the data from the previous task), incurring a time delay. This delay is known in the literature as cache-related preemption delay (CRPD) [B-8]. CRPD can be observed both in single-core processors and in multicore processors. In general, this means that temporal isolation cannot be achieved in the presence of a shared cache because, even if a task is prevented from executing beyond a specified budget (or time slot), it can still evict data from the cache used by the other tasks imposing a CRPD on it.

To achieve temporal isolation in the presence of a cache, a technique called cache coloring has been developed to partition the cache into noninterfering partitions (called colors) and give one partition to each task. Cache partitioning uses the internal division of the cache into sets that are mapped to different parts of the memory. This division is known as cache-set associativity, and it is designed to prevent consecutive memory locations from evicting each other. Using this division, cache partitioning maps the memory of different tasks to different cache sets, ensuring that they do not evict each other's memory.

The mapping of cache sets to tasks in cache partitioning is implemented using the virtual memory scheme in the memory management unit (MMU) of a processor (some simple processors may not have one). Specifically, virtual memory is implemented by translating a virtual memory address used by a program into a physical memory address. This translation allows a program to address (virtually) a large memory space that may not fit in the available physical memory by loading to the physical memory (typically from disk) only the parts of the virtual memory that are needed at any given time in a similar fashion to cache. This is accomplished by dividing the memory into pages and splitting the address of a location into a page number and an offset within the page for that location. With this division, the OS uses a page table to map virtual pages to physical pages and mark the virtual pages that are not loaded into the physical memory.

Virtual memory is combined with set associativity to take advantage of the set associativity's implementation. It uses a specific subset of bits of the memory address of a location to identify the cache set number where the location should be loaded in the cache. This enables the mapping of virtual pages of two (or more) tasks to physical pages that do not map to the same cache set and, therefore, do not evict each other. The number of bits used for the page number and the cache set are not typically the same (as can be seen in figure B-2), but they share enough bits to make it practical.

Bit index	…	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Virtual Address	Virtual Page Number									Page Offset											
Physical Address	Physical Page Number									Page Offset											
Cache Mapping					Cache Set Index											Cache Offset					
Bank Mapping	Row				Rank/Bank				Column									Offset			

Figure B-2. Page to cache set and bank mapping—Intel i7 2600

Cache Partitioning: Cache partitioning is a key technique to achieve temporal isolation not only for single-core processors [B-9, B-10] but also for multicore processors in which the last-level cache is shared across processors [B-11]. Moreover, because the number of partitions that can be created with this technique is not always large enough for assigning one partition per task, practical solutions may require two or more tasks to share a partition. In this case, the temporal isolation can only be achieved at the task group level (i.e., one group may be isolated from another group). It is then necessary to modify the timing analysis verification method to accommodate the interference due to the shared cache. In [B-11], the authors describe a cache partitioning scheme and implementation with cache sharing for rate-monotonic scheduling (RMS).

DRAM Partitioning: The interference from the DRAM system needs to be addressed to implement temporal isolation in multicore processors. Specifically, memory interference can increase the execution time of a task more than 12 times [B-4] because of memory requests to the same memory bank from other cores, as mentioned in section 2.1.1.2. To prevent tasks from using the same bank, the cache coloring scheme can be applied to memory banks to take advantage of the fact that banks are addressed by a subset of address bits, as shown in figure B-2.[8] In [B-4], the authors present a bank-partitioning approach for real-time systems with RMS. In this paper, the authors also recognize that partitions may need to be shared and provide a timing verification approach for tasks that share partitions.

Combined Partitions: Because both cache and bank partitioning use the virtual memory mechanism, they may interfere with each other. Specifically, whereas figure B-2 shows a case in which the bank bits are a subset of the cache set bits, this is not always the case. For instance, consider a processor in which address bits 10 and 11 are used for the bank number and 11 and 12 for the cache set number. This configuration may lead to the belief that four cache partitions and four bank partitions could be configured. Unfortunately, this is not the case. If bit 11 is changed, that not only changes the cache set number but also the bank number that is used for this memory location. In the end, there are only eight independent partitions instead of the ideal 16. This is shown in figure B-3. In [B-12], the authors study this problem and provide partition allocation algorithms for real-time systems to solve these issues.

Bank Bits

Cache Bits		01	10	11
00	X		X	
01	X		X	
10		X		X
11		X		X

Figure B-3. Limited partitions due to shared bits (existing partitions are marked with an X)

Scratchpad Memory: Scratchpad memory can also be partitioned to implement temporal isolation. A number of papers had been published on this topic, including [B-13, B-14]. In this case, this partitioning can be done independently from bank partitioning schemes.

Translation lookaside buffer (TLB): The TLB caches virtual-to-physical page translations to speed up the translation and prevent a single memory access from translating into multiple memory

8 Ranks are sets of banks that we do not discuss here for simplicity of presentation. Refer to [B-0, B-0] for more details.

accesses because of the access to the page table. The TLB can also be colored with a similar technique to cache coloring. In [B-15], the authors show such a scheme. In this paper, the authors also point out the benefits of modifying the heap allocation scheme to improve the performance benefits of TLB coloring.

B.2.1.2.3 Asynchronous I/O

Asynchronous I/O requests using DMA cause interference from two main sources: 1) shared memory banks, and 2) memory buses. To deal with the problem of shared memory banks, bank partitioning can be used to achieve temporal isolation. However, dealing with memory bus interference can be more challenging. This was recognized by Betti et al. in [B-16]. To solve this problem, the authors developed an I/O bridge and peripheral scheduler that coordinates the memory transfer of these peripherals with the transfer requirements of the task set running in the processor. Without this solution, a DMA-enabled peripheral is allowed to start a memory transfer at any time and either completely block the bus or steal bus cycles to perform its transfer in an interleaved fashion with the processor memory accesses. Unfortunately, the peripheral activity delays the memory access from the processor, incurring memory stalls that enlarge the execution time of the tasks running on it. The end effect is an indirect violation of temporal isolation because a task that is waiting for the completion of I/O (i.e., not running) can indirectly interfere with the currently running task because of a DMA transfer related to the request of the former task.

B.2.1.2.4 Thermal management

As discussed in section 2.1.1.4, DTM allows the processor to run at frequencies that can potentially drive the temperature of the processor to unsafe levels and reduce its speed when the temperature reaches a threshold to prevent it from burning out. Unfortunately, this change of speed can enlarge the WCET of tasks and, therefore, invalidate the analysis performed assuming a fixed speed. Moreover, the temperature increase incurred in one temporal partition could induce a speed-down on another partition, breaking temporal isolation if DTM is not taken into account. Some initial work has appeared that accounts for heat when verifying schedulability in real-time systems. For instance, in [B-17], the authors focus on minimizing the peak temperature in a multicore processor while ensuring that all deadlines are met.

B.2.1.2.5 Operating Systems

Real-time operating systems (RTOS) are designed to ensure temporal determinism to provide timing guarantees. RTOS typically provide three types of scheduling policies: 1) time-division multiplexing (TDM), 2) priority scheduling, and 3) combinations of the two.

Time-Division Multiplexing: An example of TDM is the time-triggered architecture (TTA) discussed in [B-18]. Whereas in [B-18] the properties and challenges of this policy for implementing temporal isolation were discussed, this section will highlight the context of the other shared resources. In particular, to ensure temporal isolation in a TDM RTOS, it is necessary to ensure that the sharing policies for 1) cache (or scratchpad memory), 2) DRAM, 3) asynchronous I/O, 4) thermal management, and 5) TLB are properly coordinated with the OS scheduler and its supporting analysis. For TDM, the research on this coordination has a stronger focus on specialized hardware (e.g., systems on chips) than on COTS hardware. For instance, in [B-19, B-20], the authors propose a cache partitioning approach for multiprocessor systems on chips (MPSoC) with reconfigurable caches. In particular, their framework generates a time-triggered, non-preemptive schedule and a set of cache configurations for a specific task set at compile time. The use case for this solution is the generation of different numbers of cores and cache modules on a field-programmable gate array. Memory partition has received less attention in TDM scheduling. One of the few papers on this topic is [B-21]: the authors suggest the use of memory partitions to provide temporal isolation. However, no discussion on the coordination of partitioning and scheduling is provided.

Fixed-Priority Scheduling: In fixed-priority scheduling (including RMS), deadline monotonic scheduling (DMS), Earliest-Deadline-First (EDF) scheduling, a number of combined solutions have been explored. For instance, the work in [B-4] was implemented in the context of Linux/RK, a resource reservation kernel (already discussed in [B-18]) that combines multiple temporal partitions from different resources, including cache and memory banks, with the corresponding schedulability analysis. Specifically, in [B-4], the authors use the worst-case response time presented in equation 1 of [B-18] and modify it to include the interference that can be suffered from shared memory partitions. This results in the following equation:

$$\begin{aligned} R_i^k = C_i + \sum_{j<i} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil C_j + \min(H_i \cdot RD_p \\ + \sum_{j<i} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \cdot H_j \cdot RD_p, JD_p\left(R_i^{k-1}\right)) \end{aligned} \tag{B-1}$$

Here, the minimum of two terms is added to the response time equation of a task. The first of these two terms is a request-driven memory interference bound composed of the number of requests that a job of the task under analysis (τ_i) performs (H_i), multiplied by the worst-case delay that each request can impose (RD_p), plus the worst-case number of preemptions τ_i may suffer ($\sum_{j<i} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil$), multiplied by the number of memory requests per preemption (H_j) of each task τ_j, multiplied by the cost of each memory request (RD_p). The second term is a job-driven approach that takes into account the response time of the job being analyzed (R_i^k). It obtains the maximum possible delay that can be caused by the worst activity within the memory system during that time interval. The minimum of these two terms is used to reduce pessimism given that, depending on the task and memory system parameters, one term may be larger than the other. Additional details of the calculation of each element of this equation is beyond the scope of this document; see [B-4] for more information.

For I/O, [B-16] includes a scheduling framework that co-schedules tasks in the processor and DMA memory transfer requests. The authors propose the use of real-time calculus [B-22] to account for all possible delays and jitter in the system. In real-time calculus, arrival curves provide an upper bound (maximum possible) on the number of requests (e.g., jobs) that arrive over time. They are used in conjunction with service curves that provide a lower bound (minimum possible) on the available "computation" in a processing unit (e.g., processor, network, I/O device, data bus) to evaluate if there is enough computation to process the requests in a window of time. Multiple arrival curves (e.g., tasks) can be combined with special operations that ensure that the upper bound is preserved. Similarly, multiple service curves can also be combined (e.g., processing in a CPU, network, I/O unit) to provide an end-to-end schedulability analysis.

Combined TDM and Fixed-Priority Scheduling: The combination of TDM and fixed-priority scheduling (TDM+FP) is part of the ARINC 653 standard [B-23]. A TDM schedule is used to create time partitions and, within each partition, fixed priority is used to schedule the tasks assigned to it. To verify the schedulability of this combination, it is necessary to take into account the parameters selected at both levels—time slices and priorities—and the combined behavior of these schedulers. Some work had been done in this area in what is called compositional analysis. For instance, in [B-24], the authors refined the periodic server model [B-25], in which the time slice is modeled as a fixed amount of computation provided periodically with a fixed interval (major frame). This allowed the authors to build a supply-bound function (sbf) that describes the minimum computing resource (processor) available within an interval of time. For the periodic server (and ARINC 653), it takes into account what is called a "blackout" interval—the worst-case interval that a task will have to wait before it can get the processor (because the processor is attending other partitions). Intuitively, this can be defined as the period of the server (the major frame) minus the size of the budget of the server (time slot) given to this partition. Similarly, a request bound function (rbf) describes the maximum computation time requested by all the applications in a partition. A simplified version of this rbf for a task τ_i can be described as:

$$rbf_i(t) = \sum_{j<i} \left\lceil \frac{t}{T_j} \right\rceil C_j \qquad \text{(B-2)}$$

Note that equation B-2 is similar to equation B-1, except that t is used instead of R_i^k. The schedulability of the task set can then be evaluated by testing whether the sbf is always larger than the rbf. For a more-detailed discussion and more-elaborate equations of this scheme, see [B-24].

Unfortunately, analysis of the TDM+FP combination that includes other resources such as cache, memory, and I/O has not been properly developed. However, it is possible to use a simplified version of cache and memory partitioning to ensure that each partition obtains its own partition and to extend the periodic server model analysis to include the interference of shared partitions. However, this needs to be carefully studied to avoid critical pitfalls.

B.2.1.2.6 Virtual Machines

From VMs, it is necessary to account for interference from all the different shared resources, as explained in section 2.1.2.5. In addition, because scheduling decisions are made both at the hypervisor level and at the OS level, an analysis that combines these two levels of decisions must be performed. This is similar to the TDM+FP combination presented in the previous section. For instance, in [B-26], the authors present an approach to compositional scheduling for periodic servers. For the RT-Xen, this approach takes into account the time quantization of the Xen hypervisor.

In another work, Kim et al. [B-27] present a synchronization framework for multicore VMs implemented in KVM. In this case, they also provide options for hierarchical scheduling with two server models (periodic and deferrable) and a resource synchronization (e.g., mutexes) that take into account virtualization.

B.2.2 SPATIAL ISOLATION/PROTECTION/PARTITIONING

Spatial isolation technologies are, in general, more mature than their temporal isolation counterparts. In particular, today's spatial isolation is implemented using the hardware MMU. The MMU is used to provide a virtual address space to each process in the system, as explained in section 2.1.2.2, mapping these virtual pages to whatever physical pages are convenient. If the virtual pages of two processes map to different physical pages, then they will be completely isolated from each other (from the memory point of view).

Complete spatial isolation is sometimes not convenient or even possible when these processes need to communicate or use common services from the OS. For communication, two processes may use inter-process communication mechanisms provided by the OS or a shared memory region that is mapped to both processes. This offers different degrees of isolation, and the specifics of the communication determines how isolated they are.

Common OS services can potentially weaken spatial isolation. This is a particular concern for security and a common way to bridge the isolation. For instance, a well-known attack is to call an OS service in a way that malicious code is injected into the call (e.g., as part of a buffer that is larger than the OS expects, also known as a buffer overflow attack), forcing it to "return" to the injected code. This attack allows the malicious code to run inside the kernel and gain access to all the physical memory. Furthermore, the OS code in charge of manipulating page tables (e.g., changing page tables when a context switch from one process to another occurs), known as the virtual memory system, may contain bugs. As a result, the strength of the spatial isolation also depends on the correctness of the implementation of the virtual memory system in the OS.

When VMs are used, the hypervisor creates a hierarchical virtual memory scheme, in which it gives each VM its own page table and the OS gives each process a page table. In general, the implementation of the virtual memory system at the hypervisor level tends to be simpler than the OS implementation. This is because new VMs are not created as frequently as processes (they are typically only created at boot time), and they do not need the flexibility that processes need.

Reduced complexity of spatial isolation at the hypervisor level has enabled the use of formal methods to verify its implementation. This is the case in [B-28], in which the authors verified the implementation of the isolation between hypervisor modules.

B.2.3 I/O ISOLATION

In addition to the temporal interference caused by I/O devices with DMA access, I/O could create problems in a DMA device using memory that does not correspond to the request at hand. To prevent this, the I/O memory management unit (IOMMU) was created. The IOMMU creates a virtual memory system for I/O devices with DMA access with its own page tables. Hypervisors use the IOMMU to do DMA memory remapping, ensuring that the device has access only to the memory partition of the request at hand, therefore preserving spatial isolation [B-29, B-30].

The IOMMU is also designed to reroute interrupts. This is important for temporal isolation, given that a task may be preempted by an interrupt, even though such an interrupt is related to a lower-priority task's I/O request. This is a twofold problem. First, interrupts must be directed to the core that is related to the I/O request; second, it should be ensured that the interrupt affects the timing of only the partition related to that request. Otherwise, interrupts would break the temporal isolation.

Temporal interference from interrupts is a well-known problem in real-time systems. The problem originates from the fact that CPUs give higher priority to interrupts than any other task that may be running. Whereas in general-purpose computing this may be acceptable, in real-time systems, tasks may have higher priorities than some interrupts and, therefore, should not be preempted. Moreover, if we follow RMS, if the interrupt has a minimum inter-arrival time larger than that of a task, it should be given a lower priority than the task. This problem was addressed in [B-31]. In this paper, the authors created an integrated priority space where interrupts can be given any priority, even lower than that of a task. This was implemented in COTS i386 hardware using the programmable interrupt controller to disable the interrupts that had lower priority than the currently running task. This was necessary to circumvent the hardwired priorities of the interrupts. IOMMUs simplify the interrupt disabling. However, further research is necessary to align these solutions to the temporal protection needed in real-time systems and coordinate it with other shared resources.

B.2.4 HETEROGENEOUS HOST OPERATING SYSTEMS

Hosting heterogeneous operating systems in different VMs does not impose more challenges for isolation than the ones already discussed. However, ensuring that the worst-case timing behavior of an application in different operation system is preserved can potentially increase the number of OS/hypervisor combinations that need to be validated.

B.2.5 SUMMARY OF GOALS AND PROPERTIES

The economies of scale of general-purpose computing have triggered a series of innovations (such as cache, pipelined processors, memory bank parallelization, and multicore processors) that improve the average-case performance at all levels all the way up to VMs. The need to incorporate more functionality in embedded systems, and specifically in avionics systems, demands the use of these innovations in the RTES realm. However, two challenges arise from this use: 1) the need to adapt innovations in general-purpose computing to make their timing behavior predictable, and 2) the need to create an analytic approach that integrates the behavior of all the innovations (cache, memory, buses, processors, OS, and hypervisors) while determining the schedulability of task sets running in these platforms. Table B-1 shows some of the innovations and their adaptations to RTES computing.

Table B-1. Innovation properties and adaptations

Innovation	Adaptation	Properties/ Characteristics	Integration
Pipelined processor	Abstract interpretation	Increased IPC Decreased predictability	WCET Measurement plus over-approximation
Cache memory	Cache coloring Scratchpad memory	Faster memory access Eviction isolation	Memory partitioning and scheduling
Memory banks	Bank coloring	Memory interference isolation	Coordination with cache plus scheduling
DMA transfer	Core co-scheduler	Reduced and predictable DMA memory interference	With schedulability Missing additional integration
Interrupts	Unified interrupt plus task priorities	Elimination of unpredictable unrelated interrupts	Only at the priority level Missing temporal isolation integration
Hypervisor scheduling	Compositional scheduling	Analyzable but inefficient timing	Some initial integration OS and memory hierarchy

B.3 ADDITIONAL CERTIFICATION COMPLEXITY DUE TO VIRTUALIZATION

B.3.1 ADDITIONAL TIMING VERIFICATION COMPLEXITY DUE TO VIRTUALIZATION

In this section, the additional complexity added to the timing verification due to the use of VMs is discussed. The additional complexity of timing verification comes from four sources: 1) more complicated WCET analysis caused by interference between VMs, 2) hierarchical scheduling, 3) shared software resources within the VMM, and 4) overhead of the VMM.

With respect to more-complicated WCET analysis caused by interference between VMs, the following is noteworthy. A thread executing in one VM can evict a cache block that a thread in another VM has fetched. In this way, the execution of one thread in one VM can impact the execution time of another thread in another VM. In multicore processors, there are additional effects that are similar. A thread can evict a row in a memory bank that another thread in another VM has fetched to this memory bank; this causes the other thread to execute more slowly. In addition, a thread in one VM can contend for the memory bus and prevent another thread in another VM from accessing the memory bus at a certain instant; this delays the memory access time of the other thread. These memory interferences happen because one thread makes a memory access that is based on a memory instruction (e.g., load or store) that is in the program. However, a program may also make other memory accesses that are not directly seen from the executable binary code. For example, a thread that generates a page fault may generate a memory access (a lookup in a page table). Whereas a thread performing no-operation (NOP) instructions generates no page faults, the processor may still generate memory access because of hardware prefetching. Speculative execution can cause similar additional memory accesses.

With respect to hierarchical scheduling, the following is noteworthy. Many VMMs schedule VMs; if one VM does not need to execute, then another VM will execute. Therefore, the speed of execution for one VM depends on whether another VM is busy or idle. Although a thread in one virtual processor depends on whether another virtual processor is idle or busy, bounds on the timing of a thread can still be proven. The real-time systems community has developed a theory called hierarchical scheduling that can do this. It computes a so-called supply-bound function for each VM (this provides a function that states for each *t*, a lower bound on the amount of processing time that the VM receives for each time interval of duration *t)*. Given this supply-bound function for a VM, timing analysis can be performed of all threads in this VM. This analysis typically makes simplifying assumptions, such as ignoring interference from the memory system (mentioned in the previous paragraph).

With respect to shared software resources within the VM, the following is noteworthy. When a thread makes a system call to perform I/O, it typically copies data to/from the guest operating system kernel. In turn, these data need to be copied from/to the VMM. Therefore, there are shared data structures in the VMM. Typically, it is necessary to protect those data structures from concurrent accesses (one entity should not write to a data structure in which writes are already in progress). Usually, locks are used to prevent that. However, an I/O operation performed by one thread in one VM can then impact the timing of one thread in another VM.

With respect to overhead of the VMM, the following is noteworthy: the VM generates additional context switches that may increase the number of cache misses. Finding bounds on these cache misses is nontrivial and not well studied.

B.3.2 ADDITIONAL LOGICAL VERIFICATION COMPLEXITY DUE TO VIRTUALIZATION

This section discusses the additional complexity added to logical verification due to the use of VMs. Such additional complexity arises from new interactions between the hypervisor and guests, and between the guests themselves. These interactions open up the possibility that a system consisting of several OSs will reach new states when executing in a virtualized environment, as opposed to the case when each operating system executes on a physically separate hardware-computing platform. In other words, virtualization exacerbates the “state-space explosion” problem, which in turn leads to increased complexity of exhaustive logical verification techniques (such as model-checking and theorem-proving) that must analyze all possible reachable states and executions.

What is the essential difference between a virtualized system (one in which multiple OSs share one or more CPUs) and a federated system (one in which each OS executes on its own CPU) in terms of logical complexity? The answer is that in the virtualized system, the interface between OSs is increased significantly. In a federated system, OSs can only interact via a communication medium, such as a network, or a shared resource, such as a networked file system. The interfaces for interacting in this manner are limited and well understood. For example, network-based communication can be abstracted as channels, whereas files can be thought of as shared memory.

Synchronization primitives (such as blocking function calls and file locks) are used to avoid concurrency issues, such as deadlocks and race conditions. These synchronization primitives are also used to restrict possible executions (and therefore the reachable state space). For example, if a file is guarded by a lock, then the system can never have an execution in which multiple OSs write to the same file at the same time. The "air gap" between OSs in a federated system limits its state space and, therefore, its logical verification complexity.

In contrast, multiple OSs in a virtualized system can interact directly through the shared hardware because there is no air gap between them. The most obvious way to interact is via the memory subsystem. Unless appropriate protection mechanisms are implemented, one OS could simply overwrite the memory of another. Even worse, a guest OS could overwrite the memory of the hypervisor, rendering the system completely unpredictable. In the same way, a guest could modify the state of a device that is also accessed by another OS or the hypervisor, unless appropriate protection mechanisms are employed (e.g., by appropriately programming the IOMMU at system boot time). Specific types of such interaction include:

- Interference by one guest OS on another due to bugs in the VM/hypervisor that allow logical isolation between guests to be broken. The interference can lead to data corruption or data leakage, and changes in control flow that allow new code to be executed.
- Interference by a guest on the VM due to logical bugs. This can cause complete takeover of the system by a guest, or one guest can cause the entire system to crash or steal CPU cycles from others.
- Interference by the VM on guests over and above those required for virtualization. For example, a buggy VM can leak information from guests, corrupt their data and control flow, and prevent them from being scheduled properly.

Ensuring proper isolation between guests, and between the guest and the hypervisor, is possible. The challenge is verifying that these isolation mechanisms are implemented correctly. It is complicated by the fact that hypervisors must involve at least some low-level programming (e.g., assembly), and verifying low-level programs is notoriously complex because of the need to model the hardware state and bit-level semantics. Classical program verification techniques that treat programs as transformers of numeric variables (e.g., int variables as unbounded integers) are no longer adequate in such cases. Traditional state-space reduction techniques, such as abstraction, are also not applicable because the low-level code (e.g., programming the IOMMU, or setting the value of the register pointing to the base of a nested page table) is crucial to correct implementation of the isolation mechanism and cannot be abstracted away without losing the ability to prove the system correct.

The other source of complexity is concurrency. Because the guests and the hypervisor execute in parallel, the logical isolation mechanisms (page tables, IOMMUs) must be constantly updated by the hypervisor to ensure that the executing guest is able to access its memory and devices, but nothing else. The logical invariants needed to prove that this is done correctly are complicated, because they involve the states of all the guests. They are difficult to discharge and not amenable to compositional reasoning without considerable manual effort (e.g., to apply assume-guarantee reasoning by manually constructing appropriate assumptions).

Note that all this logical verification complexity is over and above the complexity of verifying a virtualized system as if it were federated. For example, we still have to verify that shared files are locked properly and network communication is appropriately encrypted. However, verifying a virtualized system is not impossible, provided the right combination of architecture-guided analysis decomposition is used with software verification that includes an appropriate hardware model. Whereas this effort is manually intensive, it may be justified by the safety-critical nature of such systems and the fact that the verification effort is a one-time affair that is then amortized over many successful deployments of the target system.

B.4 ISOLATION COMPARISONS WITH OTHER TECHNOLOGIES

In this section, other technologies developed to provide similar properties as the ones provided by VMs are discussed. Their differences and the implications for certification, when appropriate, are also discussed.

B.4.1 OPERATING SYSTEMS

Whereas general isolation properties of OSs were discussed in section 2, in this section the focus will be on variations of OS kernels and their different protection schemes.

B.4.1.1 Monolithic Kernels

Monolithic kernels are the most common ones, like the different variants of Unix, including Linux; Windows NT; and its successors XP, 7, 8, and 10. These kernels basically present two main protection spaces, user and kernel, with the user space partitioned into processes. The monolithic kernel partitioning can be considered fairly stable and strong with respect to spatial partitioning. However, they can provide only average-case guarantees for temporal protection, as previously discussed in section 2.1.1.5.

Monolithic kernels have the particularity that all kernel activity lives in the same protection space and has access to all areas, including the memory of all processes and physical devices (therefore the “monolithic” character). This simplifies implementation but also makes it very easy for a fault in one area of the kernel to propagate to any other area. Also, because all the functionality of the kernel lives in this protection space, the complexity of the code running at this level is significant and not very amenable to verification.

B.4.1.2 Microkernels

Microkernels were developed to address the complexity problem of monolithic kernels. Specifically, microkernels push most of the functionality into modules that execute with a low level of privilege (user level). These modules communicate and are scheduled through a very small privileged layer known as the microkernel. The microkernel also provides low-level access functions to physical devices. For instance, the file system lives in a server running in user space. When a user process wants to write to a file, it sends the request to the file system server through the microkernel. Then, the file system server identifies the disk blocks it needs to read/modify and asks the microkernel to perform these low-level operations. The advantage of this structure is that when a server fails, this failure can be contained to the process where the server runs. Properties of temporal and spatial protections implemented by the microkernel can be more easily verified.

Two examples of microkernels are the QNX Neutrino Microkernel [B-32] and seL4 [B-33]. The QNX kernel is a real-time commercial kernel owned by Blackberry. However, seL4 is a research microkernel originally developed for general-purpose computing with a particular emphasis on security; seL4 has been formally verified from the logical perspective (as opposed to timing). The project eChronos has been actively working to create a verified RTOS based on seL4. In particular, eChronos includes a fixed-priority scheduler, and its authors are actively engaged in making the execution of the kernel predictable, such as by verifying the maximum number of iterations that the loops in the kernel execute [B-34]. As discussed in [B-18], microkernels have a lot in common with VMs; they can keep only one copy of the kernel and OS services or potentially multiple copies and different implementations (e.g., Unix, Linux, Windows). This concept was introduced by the Mach microkernel [B-35].

B.4.1.3 Separation Kernels (MILS)

As discussed in [B-18], separation kernels modularize the kernel functionality and create security levels (known as Multiple Independent Levels of Security/Safety, or MILS) that simplify the security verification. The focus on security differs from the safety certification concerns. Also, Rushby's original paper does not mention timing separation [B-36]. As a result, MILS have different objectives from safety-oriented isolation.

B.4.2 ARINC 653

A number of commercial RTOSs have implemented the ARINC 653 as already discussed in [B-18]. In this section, the partitioning characteristics that make it amenable for certification are highlighted. In particular, given that temporal partitions are created as a set of time slots over a major frame (see [B-18], section 4.1.4), it is easier to match this scheme with certification standards language. For instance, [B-37], section 2.3.3 states that "A software partition should be allowed to consume shared processor resources only during its allocated time." Similarly, in section 3.2, the document reads, "For example, any service performed on behalf of an application should be executed in its allocated time and not during the time allocated to another application." Similar words are used in [B-38]. For instance, section 2.4.1 reads, "A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution." In all these cases, validating these requirements against a static timeline (over a major frame) can potentially be performed manually or with a simple tool.

It is worth noting that, because the ARINC 653 standard includes a compositional approach in which tasks within a partition are scheduled with fixed-priority scheduling, verifying that tasks do not miss their deadlines requires a hierarchical verification approach, as discussed in [B-18], section 4.1.4.

As discussed in section 2, the combined used of resources—including I/O, cache, and memory—must also be evaluated to ensure the proper temporal isolation.

B.4.3 TTA

The TTA approach follows a time partitioning approach similar to ARINC 653 by dividing a major frame into slots where different applications are run. However, TTA does not have a hierarchical approach and only schedules the processor based on time slots. Moreover, TTA was born in a distributed environment, and a time-slotted bus transmission is used as a global clock that triggers all the activities in the distributed system. As mentioned in [B-18], sections 4.1.3 and 4.1.4., defining the time slots of a system with this type of scheduling is an NP-complete problem and, therefore, does not scale well.

B.4.4 RESOURCE RESERVATIONS

The temporal partitioning implemented by resource reservations ensures that if a task is deemed schedulable (to meet its deadline), it is guaranteed to stay schedulable even if other tasks try to misbehave (execute longer than their WCET). Two elements are involved in this guarantee: 1) a worst-case timing analysis (e.g., response time), and 2) a WCET enforcement mechanism. The timing analysis typically assumes a priority scheduling, in which a task with high priority is allowed to preempt lower-priority tasks whenever it arrives and a minimum inter-arrival time of the tasks. Using these elements, the worst-case timing behavior of a task, such as its worst-case response time (WCRT), is evaluated (taking into account preemptions from higher-priority tasks). If it is shorter or equal to its deadline, it is deemed schedulable. For instance, the WCRT of a task in a task set scheduled under RMS or DMS can be obtained with equation 1 from [B-18].

Resource kernels track the execution of a task, discounting all the preemptions it may suffer from higher-priority tasks to monitor it. During this monitoring, if the kernel discovers that the task will exceed its WCET, the kernel stops it, typically reactivating it once its next arrival time elapses. This ensures that the calculated WCRT of the tasks is preserved.

The mechanisms, assumptions, and theory backing up the resource reservation scheme for temporal isolation are not as easily relatable to the language in certification documents. More specifically, it is not possible to manually identify the “allocation time” of a partition, as stated in [B-37, B-38]. Instead, tasks are allowed to preempt each other based on priorities, and mathematical properties are used to evaluate the worst-case timing behavior. However, verifying whether a task set is schedulable is as easy as evaluating equation 1 in [B-18], which runs instantaneously, as opposed to solving a complex NP-complete problem to figure out static time slots for ARINC 653 or TTA.

B.4.5 MIXED-CRITICALITY SCHEDULING

Mixed-criticality scheduling is a variant of the resource reservation scheme that allocates different WCETs to tasks at different criticality levels and assigns each task a criticality level. These criticality levels are aimed at matching the software level based on the failure condition categorization of certification standards like DO-178. Similarly, the rigor with which each level is verified is mapped to the degree of certainty that a task will not exceed a particular WCET (at the task criticality level). Because, in practice, a more pessimistic WCET (that allows for a larger margin of error) leads to more certainty, the WCET of a task at a higher criticality level will be larger than at a lower criticality level. Then, when it is verified whether a task meets its deadline, all the tasks in the system are assumed to run with a WCET at the same or a lower criticality level as required by the task being verified. Low-criticality tasks are prevented from interfering with higher criticality tasks, but a higher criticality task is allowed to steal cycles from lower criticality tasks when the higher criticality task runs at a WCET with a criticality that exceeds that of the lower criticality tasks.

Clearly, different partitioning variants are possible to implement the protection that VMs aim to provide. These variants provide different validation approaches; some are easier to match to certification documents but may suffer from a more complex configuration/validation scheme. Others are easier to configure and validate but may not be readily relatable to certification language.

B.5 DEVELOPMENT PROCESS ISSUES

In this section, the development process benefits from the isolation or partitioning properties offered by VMs will be discussed. Virtualization has become a commodity technology today. At the individual user level, products like VMWare, Xen, and VirtualBox are well supported and easily deployable on commodity desktops and laptops. Even lightweight container technologies, such as Docker, have reached a significant level of easy and widespread use. At the large-scale distributed level, well-established server farms, such as Amazon's EC2 and Microsoft's Azure, can now be found. These enable CPU cycles to be traded in bulk and backed by sophisticated service-level agreements. The most common format in which CPU cycles are made available is VMs. The user rents one or more VMs on which he or she can execute arbitrary software. The supplier manages the hardware infrastructure on which a farm of such VMs can be executed while respecting the service-level agreements.

The availability of cheap and efficient virtualization technology has ushered in some radical changes in which software is developed, tested, and maintained:

- VMs allow much easier setup of a specific development and test environment for a specific project. This includes not only the correct operating system but also the right editors, compilers, libraries, IDEs, test harnesses, and more. A developer working on multiple projects can easily set up one VM per project, each with the appropriate environment. This completely sidesteps issues such as software incompatibility between different projects, which is a major obstacle if conflicting projects have to be developed on the same OS. It also provides a portable development environment that can be transferred easily from one machine to another. Because VMs provide the same interface as hardware, they do not suffer from portability issues caused by library emulation, such as Cygwin and Wine.

- VMs provide a natural isolation between multiple projects. Often, this isolation is just a precaution to avoid confusion. However, in many cases the isolation is necessary, such as when two projects contain proprietary information from different customers or credentials (such as software licenses and encryption keys) that should not be mixed. VMs also provide network isolation in a natural way and, therefore, are useful to set up protected enclaves that are unreachable from the corporate network, LAN, WAN, or Internet. This reduces the likelihood of cyber attacks on individual VMs and the spread of contagion from one VM to another.

- The portability of VMs also provides a clean way to share development and testing among multiple project members. Developers (e.g., working across different time zones) can work on the same product by sharing a VM. A tester can provide feedback to a developer by passing back the VM. Teams can demonstrate and share their products with management and customers by deploying them on VMs with all necessary software installed. This reduces chances of missing dependencies and failures during demonstrations.

- The DO-178 standard requires isolation between applications with different levels of criticality. VM technology can provide an effective way to achieve this isolation while executing such applications on the same hardware. However, isolation must be rigorously assured. At the same time, the loss of physical redundancy can hamper fault-tolerant characteristics. For example, a hardware fault can affect multiple applications, where previously this was impossible because of the "air gap."

- DO-178 also requires a mapping between high-level and low-level requirements. Virtualization can help construct this mapping by enabling the development of systems in a compositional and hierarchical manner. This is particularly critical for RTES, such as avionics, that are being developed by large groups of teams working on individual subcomponents. Often, the development of subcomponents is subcontracted to other organizations, and the resulting products are integrated by the main company. Given the increasing trend toward model-based development and virtual integration, real-time properties at the system level can be ensured at the system level only by using compositional timing verification techniques (such as those presented in section 6.2).

- Last, the availability of large server farms means that companies have the option of outsourcing hardware maintenance to other parties so they can focus their energy on core product development. This allows for more specialization and increased productivity and innovation of the software ecosystem.

All these benefits come with the caveat that VMs must be properly designed, developed, and deployed, which is by no means a trivial task. However, to a large extent, these problems have been addressed in the IT space, if not in the real-time embedded domain.

B.6 ASSURANCE OF AVIONIC SYSTEMS

In this section, VM issues related to assurance in avionics systems will be discussed. The section focuses on DO-178C and related documents. It begins with a brief overview of DO-178C and then considers specific parts of DO-178C. Then, it covers DO-333, which is a supplement to DO-178C on formal verification. Then, the section discusses DO-254, which covers complex electronic hardware. Finally, a document (CAST) that is not a guidance document but a position paper on multicore processors is discussed. This is included because one of the likely usage scenarios of virtualization is for multicore processors.

B.6.1 DO-178C OVERVIEW

General overview of DO-178C: By law, in the United States of America, an airplane must be airworthy; that is, it must be suitable for safe flight. The airworthiness of an airplane is assessed by aircraft certification authorities (e.g., FAA). As part of the certification of airworthiness of an airplane, the national certification authority approves the software. DO-178C [B-38] is a document that provides guidance for such approval. It discusses the applicant (e.g., an organization that wants to develop a new airplane) and the certification authority (e.g., FAA).

Section 10 in DO-178C [B-38] describes the certification process; it involves three major steps: 1) the certification basis, 2) software aspects of certification, and 3) compliance determination. Certification basis means that the certification authority establishes the basis for the product to be certified in consultation with the applicant. Then the applicant submits a "Plan for Software Aspects of Certification" to the certification authority. The certification authority assesses whether the plan complies with the certification basis agreed on in the first step. Finally, the certification authority determines whether the product to be certified (including its software) complies with the certification basis. It does so by reviewing a "Software Accomplishment Summary." The last two steps will be described in this section.

The "Plan for Software Aspects of Certification" is described in section 11 in DO-178C [B-38]. It includes a system overview that describes the function of the system, hardware, and software. It also includes a description of the software. It includes a schedule for giving the certification authority visibility into the activities of the software life cycle so reviews can be planned. Finally, it can contain a large number of plans, including plans for requirements, plans for design, plans for coding, plans for integration, and plans for verification.

The "Software Accomplishment Summary" is described in section 11.20 in DO-178C [B-38]. It is similar to the "Plan for Software Aspects of Certification," but it describes what actually happened and deviations from the plan.

Overview of DO-178C relevant for VMs: According to DO-178C, there should be a description of how a function is implemented, and a system should be described with components. A function has a failure condition category; it describes the consequences of a failure of a function. For example, the category "Catastrophic" means that a failure may result in multiple fatalities, usually with the loss of the airplane. There are four other failure condition categories for less severe consequences. A software component has a level indicated by a letter A–E, in which A is the most critical and E is the least critical. The level of a software component is assigned based on the failure condition category of the function it provides.

DO-178C uses the term "partition." Partitioning is a way to ensure that one software component does not impact another software component. If partitioning is not used, then the software must be assigned a level equal to the highest failure condition of the functions that it provides. Traditionally, partitions were formed using two techniques: 1) the virtual memory system, and 2) ARINC 653. With a virtual memory system, whenever a program accesses memory, it accesses a virtual memory address. This virtual memory address is translated to a physical memory address, and then data are read from or written to this physical memory address. In this way, the OS can set up the virtual memory translation so that a write of one program cannot modify memory that another program may access.

With ARINC 653, a scheduler in the OS forms time partitions (i.e., the scheduler uses time-triggered scheduling with a table that specifies the starting time and duration of a time partition). In this way, a component can be assigned to one time partition and another component can be assigned to another time partition. Ideally, they will not influence one another's timing. In many practical settings, however, execution of one component in one partition will influence the timing of another component in another partition. There are many causes for this (as explained earlier in this report). Cache eviction is one of the causes. Another is that one component may initiate DMA transfer, and this transfer is performed at a time when another time partition is active (i.e., when another component executes). Despite these drawbacks, ARINC 653 is used in practice and it provides some degree of isolation. With ARINC 653 and each component having its own dedicated time partition, the timing of one component tends to be less influenced by the execution of another component than would have been the case if no time partitions existed.

DO-178C specifies that for each system requirement on a function that is allocated to software, the software component should have high-level requirements. Then low-level requirements should be traceable to high-level requirements. The low-level requirements are inputs to the coding process (specified by the Software Development Plan and Software Coding Standard); the low-level requirements should be sufficiently detailed that coding can be done without further information.

B.6.2 DO-178C-SPECIFIC REMARKS RELATED TO VIRTUAL MACHINES

This subsection will make detailed remarks on DO-178C with respect to virtualization.

Section 2 in DO-178C covers system aspects related to software development. Section 2.3.1 discusses the relationship between software errors and failure conditions:

> It is important to realize that the likelihood that the software contains an error cannot be quantified in the same way as for random hardware failures. [B-38, p. 12]

This statement was made in the context of a system with hardware and software, and discussing errors and failure conditions. However, because a VM behaves, from the perspective of application software, as hardware but is being implemented in software, it is important to note that the simulated hardware (i.e., the VM) can suffer from a fault not only because its underlying hardware suffers from a fault but also because of design defects in the software that generates a VM.

Page 15 mentions that:

> A partitioned software component should not be allowed to contaminate another partitioned software component's code, I/O, or data storage areas.

From a functional perspective, this can often be achieved if each component has its own VM. However, the timing (and, in this case, timing of I/O) can be impacted by the fact that a partitioned software executes on a VM. To understand this, consider a legacy system that does not use virtualization. In this legacy system, there is an old processor. The software uses a sensor, and the software does not use an OS; it performs I/O directly (i.e., I/O code is intertwined with the application code—not a recommended practice these days). Assume the functionality is still highly valued, but the processor is no longer commercially available. Therefore, a VM equivalent to this old processor is needed. It could be replaced as follows: Consider a VMM that simulates even I/O instructions, and assume that the software performs programmed I/O. Consider that this programmed I/O is done as follows: read I/O port, perform five NOP operations (to cause a delay), and then repeat the above a fixed number of times.

A software practitioner may have written the software that performs I/O assuming that the code executes on a computer with a physical processor with some bounds on its speed. For this assumption, the software practitioner can know that the time it takes to execute the five NOP instructions is at most a certain known time and at least a certain known time. The choice of using five NOPs may be intentional to ensure that the time between two readings of I/O ports is upper and lower bounded. However, now when this software performing I/O executes on a virtual processor, it can happen that the scheduler in the VMM allocates one time quantum (e.g., 10 ms) for VM; at the end of this time quantum, the software performing I/O performs one I/O instruction. Then the VMM switches to another VM for one time quantum and switches back again. In this case, the time between two I/O operations can be approximately 10 ms, which is typically much longer than the time to execute five NOP on a physical processor. Therefore, when a software practitioner ports existing software from a physical processor to a VM, the software practitioner needs to check if this type of programmed I/O with delays achieved with NOP instructions exists, and that the timing between such I/O operations will be satisfactory in the new configuration (i.e., the VM).

Page 15 mentions that:

> A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.

As was mentioned previously in this document, the use of virtualization can cause one software component in one partition to evict cache blocks that another software component in another partition has fetched. Therefore, one software component can influence the timing of software components in other partitions.

There is another risk related to this and virtualization: it is common in avionics to use triple or quadruple modular redundancy to achieve better reliability. The main idea is that hardware faults are independent. Therefore, if one processor experiences a fault, then the other processors are not likely to experience a fault. However, if these processors are formed as VMs out of a single physical processor, then if the physical processor experiences a fault, it will clearly impact all the virtual processors (or VMs) running on this physical processor. Therefore, if one takes a single physical processor, uses virtualization to form four virtual processors, and runs a quadruple redundancy scheme, this cannot be expected to work just like on physical processors. For this reason, a system uses replication; the replicas should be assigned to different physical processors (i.e., replicated software should be assigned to virtual processors that map to different physical processors).

Section 4 in DO-178C covers the software planning process. Section 4.4.3 discusses the software test environment:

> a. The emulator or simulator may need to be qualified as described in section 12.2. [B-38, p. 29]

This sentence appears to have been written assuming that the deployed system does not use virtualization; it appears to discuss emulation and simulation only for testing purposes. However, if virtualization is used, then the runtime system also has a simulator. It is worth asking whether this runtime system should be considered (within DO-178C terminology) as a tool and therefore require tool qualification. Alternatively, perhaps the runtime system that provides VMs should instead be considered as the deployed software and therefore be verified rather than qualified. The FAA considers that anything that executes at runtime should be considered as part of the deployed systems and therefore should be verified.

Section 5 in DO-178C covers the software development process. Section 5.4.2 discusses integration process activities:

> a. Software integration should be performed on a host computer, a target computer emulator, or the target computer. [B-38, p. 36]

It is worth asking what “target computer” means here. If the system in operation uses virtualization to form two VMs, then the target computer is one of the VMs. The FAA considers that the VM should be tested on the actual target hardware given that the combination (VM + hardware) is part of the runtime environment required to be tested.

Section 6 in DO-178C covers the software verification process. Section 6.3 discusses software reviews and analysis:

> For example, a combination of reviews, analyses, and tests may be developed to establish the worst-case execution time or verification of the stack usage [B-38, p. 41].

It is worth noting that virtualization can influence both WCET and stack usage. As already mentioned, a component in one VM can evict a cache block fetched by another component in another VM. Therefore, one component in one VM can impact the execution time of another component in another VM. Stack usage is also influenced by the number of preemptions; the number of preemptions is influenced by the response time; and the response time is influenced by the WCET. Therefore, migrating software from a physical processor to a virtual processor can influence both WCET and stack usage. Even without cache eviction, the execution of one component in one VM can still impact another component in another VM. This can happen if a VMM that does not use time partitioning is used. Most VMMs for IT systems use a variant of round-robin scheduling. Then the amount of processing cycles that one VM will get in a time interval depends on whether there are other VMs in the system. Therefore, it can happen that the software practitioner tests software in one VM, and its timing is acceptable, but when a new VM is started (for running software of another component), the scheduler gives less processing time to the former, and its timing requirements are violated. This can happen (i.e., execution on one VM can impact the timing of software executing in another VM) even if the VMM uses an event-triggered real-time scheduler (e.g., global EDF or global Rate-Monotonic). Such real-time schedulers typically have an analysis so that the impact of one component on another can be bounded. Yet the dependency of timing of one component on the execution of another component is still there; it is just that it is bounded.

Section 6 in DO-178C covers the software verification process. Section 6.3.1 discusses reviews and analyses of high-level requirements:

a. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially system response times and I/O hardware [B-38, p. 41].

With virtualization, what is referred to as the target computer may be a VM. Once again, with virtualization, the response time and stack usage of one thread/component in one VM can be influenced by execution of another thread/component in another VM. Yet, once again, programmed I/O with NOPs is sometimes used to generate a delay between I/O operations. The timing of this execution can be very different in a VM compared to a physical processor.

Section 6 in DO-178C covers the software verification process. Section 6.3.3 discusses reviews and analyses of software architecture:

a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirement, especially functions that ensure system integrity, for example, partitioning schemes. [B-38, p. 42]

Some virtualization software for real-time systems (e.g., RT-Xen) uses an event-triggered scheduler with a reservation mechanism for each VM. In this way, the execution of one component in one VM can impact the timing of a component in another VM. Typically, one configures parameters of the reservation mechanism (e.g., VM 1 should receive 30% of the processing capacity, and the granularity of allocation should be 10 ms) so that each VM receives a lower bound on service for a time interval of given duration. In this way, there is a new type of linked requirements. There are requirements on the reservation mechanism, which relates to a VM, and on response times of threads within a VM. It is important to trace those requirements.

Section 6 in DO-178C covers the software-verification process. Section 6.3.3 discusses reviews and analyses of software architecture:

a. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. The relationship exists via data flow and control flow. If the interface is to a component of lower software level, it should also be confirmed that the higher software level has [an] appropriate protection mechanism in place to protect itself from potential erroneous inputs from the lower software level component [B-38, p. 42]. The messaging between partitions is itself partitioned, so that the work done by the sending partition is done in the context of the sender, and the work done to perform the receiving operations is done by the receiving partition. The MMU will perform any switching required to ensure that each part of the transaction is performed by the appropriate actor.

Shared data structures in the VMM can cause timing dependencies between components in different VMs. Consider two components in different VMs, and each VM is allocated to its own physical processor (assume that the physical computer is a multicore processor). One component makes a system call to its guest operating system and this, in turn, calls the VMM (for example, for memory management). The other component does exactly the same thing. However, if their execution in the VMM requires holding a data structure under mutual exclusion, then only one system call to the VMM can be done at a time; the other has to wait. This results in a timing dependency.

Section 6 in DO-178C covers the software verification process. Section 6.3.3 discusses reviews and analyses of software architecture:

> c. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer. [B-38, p. 43]

It is worth mentioning that in this language, the "target computer" could refer to a VM and a VMM.

Section 6 in DO-178C covers the software verification process. Section 6.3.4 discusses reviews and analyses of source code:

> f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed-point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on worst-case execution timing, and this impact should be assessed. [B-38, p. 42–43]

As mentioned above, porting software from running on a physical processor to an "equivalent" virtual processor can lead some of these issues to violate timing requirements and stack usage requirements.

Section 6 in DO-178C covers the software verification process. Section 6.4 discusses software testing:

> e. The executable object code is compatible with the target computer [B-38, p. 44].

It is worth mentioning that in this language, the "target computer" could refer to a VM.

Section 6 in DO-178C covers the software verification process. Section 6.4.2.1 discusses normal test range cases:

> b. For time-related functions, such as filters, integrators, and delays, multiple iterations of the code should be performed to check the characteristics of the function in context [B-38, p. 46].

Porting software from running on a physical processor to an "equivalent" virtual processor can lead some of these issues to violate timing requirements and stack usage requirements. This impacts the above statement in DO-178C. Specifically, certain implementations of differentiation of a signal with respect to time are performed by taking a sensor reading of the current time and then reading the sensor and current time again. The differentiation can then be approximated by the difference between the sensor readings divided by the difference between the times read. For systems in which time is quantized, it can happen that although some time has elapsed between the two sensor readings, and the difference in time is therefore strictly greater than zero, the computed time difference is equal to zero. This leads to a division by zero and throws an exception. As a result, code that computes differentiation of a signal in this way can compute the wrong result because of incorrect timing. This scenario can occur if the timer API (application programming interface) used before porting did not use quantized time, but the time API used after porting used quantized time.

Section 6 in DO-178C covers the software verification process. Section 6.4.2.2 discusses robust test cases:

e. A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly [B-38, p. 47].

Porting software from running on a physical processor to an “equivalent” virtual processor can lead some of these issues to violate timing requirements. This impacts the above statement in DO-178C. In this case, because it cannot be guaranteed when the temporal protection code will run in a VM, it cannot be ensured that a partition can be stopped from exceeding its frame time.

Section 6 in DO-178C covers the software verification process. Section 6.4.2.2 discusses robust test cases:

f. For time-related functions, such as filters, integrators, and delays, test cases should be developed for arithmetic overflow protection mechanisms [B-38, p. 47].

As mentioned above, porting software from running on a physical processor to an “equivalent” virtual processor can lead some of these issues to violate timing requirements. This impacts the above statement in DO-178C.

Section 6 in DO-178C covers the software verification process. Section 6.4.2.2 discusses robust test cases:

a. Failure to satisfy execution time requirements [B-38, p. 47].

As mentioned above, porting software from running on a physical processor to an “equivalent” virtual processor can lead some of these issues to violate timing requirements. This impacts the above statement in DO-178C. In the same list, DO-178C mentions other timing issues. Once again, the use of VMs can impact these requirements.

Section 6 in DO-178C covers the software verification process. Section 6.4.2.2 discusses robust test cases:

c. Incorrect responses to missing or corrupted data. [B-38, p. 48]

One cause for missing data is that a component is designed as a chain of different threads operating at different rates. The missing data are caused by so-called undersampling. That is, thread B reads data from a data structure that thread A writes to, and thread B has the same period as thread A. If the arrivals of these threads are unsynchronized, and if the response times of threads are variable (which is usually the case for event-triggered real-time systems), then thread A sometimes produces a new data item for thread B to read; sometimes thread A produces zero new data items for thread B to read; and sometimes thread A produces two data items for thread B to read. In the latter case, if there is buffer space for only one data item, then one data item is lost. The timing of arrival of B and the response time of A influence whether data are lost. As already mentioned, a VM can influence the response time. Therefore, migrating software to a VM can cause this problem (missing or corrupt data).

Section 7 in DO-178C discusses software configuration management process. VMMs typically rely on configuration parameters when creating VMs. For example, if the VMM uses an event-triggered real-time scheduler with reservations (which is the case for RT-Xen), then the parameters of the reservations become an issue of configuration management. If the VMM uses cache coloring, bank coloring, or some other scheme to achieve memory accesses with more time predictability, then these coloring schemes also rely on parameters, which add additional configuration management issues to the use of VMM. Parameter Data Items are an addition to DO-178C that requires lifecycle data. The difficulty this creates is that data do not have the typical equivalence classes that algorithms use, making data much more difficult to verify.

Section 11 in DO-178C covers software life-cycle data. Section 11.1 discusses planning for the software aspects of certification:

b. Software overview. This section briefly describes the software functions with emphasis on the proposed safety and partitioning concepts. Examples include resource sharing, redundancy, fault tolerance, migration of single event upset, and timing and scheduling strategies. [B-38, p. 48]

Porting software from running on a physical processor to an “equivalent” virtual processor can lead some of these issues to violate timing requirements. This impacts the statement in DO-178C.

In addition, DO-178C states that the software system should be described with components, each component should be assigned a level, and the software practitioner may want to use partitioning to achieve isolation, as discussed above. However, if virtualization is used, then it needs to be determined how a VM relates to a component. For instance, three different component assignment cases could be considered: 1) assigning each component to its own VM, 2) assigning multiple components with the same level to one VM, or 3) assigning multiple components with different levels to the same VM. In this report, these questions are not answered definitively. However, they relate to architecting, and therefore they relate to the above statement in DO-178C.

Section 11 in DO-178C covers software lifecycle data. Section 11.3 discusses planning for software aspects of certification:

b. Partitioning consideration: If partitioning is used, the method used to verify the integrity of the partitioning. [B-38, p. 48]

If virtualization is used, then this statement relates to verification of the VMM. It is worth pointing out once again that the typical time partitioning does not offer complete timing isolation because of cache evictions and other effects. However, as mentioned in section 2.5, it is possible to provide resource-sharing mechanisms and the corresponding analysis methods to guarantee time partitioning. Robust partitioning will ensure that the caching policies, or other mechanisms used, are bounded. The hard part is how to verify the robust partitioning requirements when they are "negative requirements." It can be shown when one partition is required to change another partition, but how is it shown that one partition "shall not change the memory of another partition unless configured to do so." "Shall not" requirements are very difficult to verify.

Section 11 in DO-178C covers software life-cycle data. Section 11.4 discusses the software configuration management plan:

b. 9. Software life cycle environment controls: Controls for the tools used to develop, build, verify, and load the software, addressing sections 11.4.b.1 through 11.4.b.4. This includes control of tools to be qualified [B-38, p. 73].

Again, some virtualization solutions for real-time systems (e.g., RT-Xen) rely on configuring reservations. Typically, tools are used to dimension reservations so that all tasks served from a reservation are guaranteed to meet their deadlines. Such tools may need to be qualified. This typically means that the tools are TQL-1 (i.e., need a lot of work to qualify). This depends on the tool and the transformations the tools perform. This is not an easy process. These configuration files could be 750,000 lines of XML.

Section 11 in DO-178C covers software life-cycle data. Section 11.6 discusses software requirements standards7:

d. The method to be used to provide derived requirements to the system processes [B-38, p. 74].

In some systems, there is a timing requirement on the maximum allowed latency from when one software component receives an event to when another software component has finished. If these two components are in different VMs, then the timing analysis can be complex. One way (not necessarily the best way, but sometimes convenient) to deal with this is to divide the end-to-end deadline into a set of sub-deadlines, one for each component, so that the sum of sub-deadlines equals the end-to-end deadline. Then, it is sufficient to check only each sub-deadline of components in each VM. This is mentioned because it requires that one derive requirements, and this relates to the statement above in DO-178C.

Section 11 in DO-178C covers software life-cycle data. Section 11.7 discusses software design standards:

b. Conditions imposed on permitted design methods, for example, scheduling, and the use of interrupts and event-driven architecture, dynamic tasking, re-entry, global data, and exception handling, and rationale for their use [B-38, p. 74].

Porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 11 in DO-178C covers software life-cycle data. Section 11.9 discusses requirements data:

b. Timing requirements and constraints [B-38, p. 75].

As mentioned, porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 11 in DO-178C covers software life-cycle data. Section 11.10 discusses design description. Pages 75–76 mention the importance of scheduling and timing in many items in a bulleted list. As mentioned, porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 11 in DO-178C covers software life-cycle data. Section 11.20 discusses the software accomplishment summary:

i. Software characteristics: This section states the Executable Object Code size, timing margins including worst-case execution time, memory margins, resource limitations, and the means used for measuring each characteristic [B-38, p. 79].

Porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 12 in DO-178C covers additional considerations. Section 12.1.1 discusses modifications to previously developed software:

i. The area affected by a change should be determined. This may be done by data-flow analysis, control-flow analysis, timing analysis, traceability analysis, or a combination of these analyses [B-38, p. 81].

Porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 12 in DO-178C covers additional considerations. Section 12.1.2 discusses change of aircraft installation:

a. The system safety assessment process assesses the new aircraft installation and determines the software level and the certification basis. No additional effort will be required if these are the same for the new installation as they were in the previous installation. [B-38, p. 82]

Porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-178C.

Section 12 in DO-178C covers additional considerations. Section 12.1.3 discusses change of application or development environment:

a. Use and modification of previously developed software may involve a new development environment, a new target processor, or other hardware, or integration with other software than used for the original application [B-38, p. 82].

In this phrase, “a new target processor” may be a VM. As mentioned, porting software from running on a physical processor to an “equivalent” virtual processor can cause some of these issues to violate timing requirements.

One vulnerability that should also be mentioned is the use of synchronizing instructions.

For example, we have two partitions running on one core. Partition A is at level A, and partition C is at level C. Partition A issues a system call to send a message to partition C. Partition C cannot write to a memory location in the kernel because it lacks the privilege.

In the kernel, instructions are being executed in a pipeline. Just before the context switch instruction, the memory prefetch is getting memory locations through the pipeline using its privileged mode. However, the memory fetches are further up the instruction stream and should be blocked by the user state, but they are pre-executed in system state. Once the context switch instruction is executed and the instructions fetches are committed (from virtual to embedded logic analyzer register and then to memory) the memory violation is undetected.

The RTOS must be analyzed for these context-sensitive instructions, and they need to be preceded by synch instructions (data synch, instruction synch, or both) The RTOS suppliers need to conduct careful analysis and identify which data or instruction sequences can be affected and which really matter.

B.6.3 DO-333 OVERVIEW

DO-333 [B-39] partly deals with use of formal methods for verification. This document follows the same structure as DO-178C and has the same sections as DO-178 (one for each process). However, for most of the sections, there is simply a statement: follow the guidance in DO-178. The main exception is the section that discusses verification.

DO-333 points out that formal methods can be used for different types of requirements. They can be used to prove a property between a set of low-level and high-level requirements, and between a low-level requirement and source code. Formal methods can also be used to prove a property between a low-level requirement and the executable code. DO-333 lists three general classes of formal methods: 1) theorem proving, 2) model checking, and 3) abstract interpretation. Theorem proving means that a proof is constructed (typically by a human user and with some aid from a proof assistant), and then checked automatically. Model checking is an automated, algorithmic, and exhaustive technique for verifying properties of finite-state machine models of systems. Because software has a very large (potentially infinite) state space, model checking cannot be applied to it directly. Abstract interpretation is typically used to construct finite models of software, so that it can be analyzed by techniques such as model checking or direct reachability analysis.

Formal methods can be used to show compliance with requirements, but DO-178C also requires that one show the reason for the existence of a requirement. DO-333 shows that the former can be achieved with formal methods, but the latter cannot. To show the reason for the existence of a requirement, another method (reviewing) must be used.

B.6.4 DO-333-SPECIFIC REMARKS RELATED TO VIRTUAL MACHINES

Section FM.1.0 in DO-333 introduces the document. Section FM.1.6.2 discusses formal analysis:

> These properties are either created or embedded in specific tools that implement the formal analysis (for example, worst-case execution timing) [B-39, p. 4].

Porting software from running on a physical processor to an "equivalent" virtual processor can cause some of these issues to violate timing requirements. This impacts the statement in DO-333.

Section FM.4 in DO-333 covers the software planning process. Section FM.4.3 discusses software plans:

> All assumptions related to the formal analysis should be described and justified, especially those associated with the target computer [B-39, p. 11].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been discussed on previously).

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.0 introduces this section and mentions:

> As a minimum, testing will still be required to ensure that the Executable Object Code is compatible with the target computer [B-39, p. 15].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software verification process. Section FM.6.2.1 discusses considerations for formal methods:

> All assumptions related to each formal analysis should be described and justified: for example, assumptions associated with the target computer or about the data range limits [B-39, p. 17].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.1 discusses reviews and analyses of high-level requirements:

> c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially system response time and I/O hardware. If the high-level requirements and hardware/software features of the target computer are formally

modeled, then potential conflicts can be detected through formal analysis [B-39, p. 19].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software verification process. Section FM.6.3.1 discusses reviews and analyses of high-level requirements:

f. Traceability. The objective is to ensure that the functional, performance, and safety-related requirements of the system allocated to software were developed into the high-level requirements. If formal methods are used to meet the objective of section FM.6.3.1, item a, this can provide evidence in support of traceability. [B-39, p. 19]

This statement will be discussed with respect to traceability and timing requirements. Software executing on a VM may have different timing compared to the timing that the software would have if it executed on a physical machine. One way to deal with this difference is to use a formal method for proving the timing requirements of the software. The research literature offers a technique called hierarchical (or compositional) scheduling (which is used in RT-Xen—one of the most popular open-source virtualization solutions, extended for real-time systems), as discussed in section 2.1.2.6. Hierarchical scheduling works as follows: A root scheduler schedules subsystems, and in each subsystem, there is a local scheduler. Whenever a subsystem is selected by the root scheduler, the local scheduler is invoked to select one thread within the subsystem. For each subsystem, there is an interface that describes the resource consumption of the threads in the subsystem (e.g., subsystem 1 will need 20% of the processor, and this processing capacity will be distributed with a resolution of 10 ms). A schedulability test is applied on the root scheduler; this schedulability test is a formal method that can prove that each subsystem receives enough processing cycles as indicated by the interface. A schedulability test is applied on each subsystem; this schedulability test is a formal method that can prove that all threads within a component meet their timing requirements (deadlines) assuming that the subsystem receives processing time as specified by the interface of the subsystem. The concepts of hierarchical scheduling can be applied for systems that use virtualization in the following way: Let the root scheduler be the scheduler in the VMM, let each subsystem be a VM, and let each local scheduler be the scheduler in the guest operating system. For such a use, there may be a need to perform traceability between the two schedulability analyses (i.e., those associated with the root and local schedulers).

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.2 discusses reviews and analyses of low-level requirements:

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the low-level requirements and the hardware/software features of the target computer, especially the use of resources such as bus loading, system response times, and I/O hardware. If the low-level requirements and hardware software features of the target computer are formally modeled, then potential conflicts can be detected through formal analysis [B-39, p. 20].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.2 discusses reviews and analyses of low-level requirements:

> f. Traceability. The objective is to ensure that the high-level requirements were developed into the low-level requirements. If formal methods are used to meet the objective of section FM.6.3.2, item a, this can provide evidence in support of traceability. [B-39, p. 20]

Timing analysis of systems with virtualization can be performed with theories for hierarchical scheduling. It may then be necessary to trace the analysis of a local scheduler to the analysis of a root scheduler.

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.3 discusses reviews and analyses of software architecture:

> a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes. If high-level requirements and software architecture are formally modeled, then formal analysis can be used to show compatibility [B-39, p. 20].

As already mentioned, in many virtualization solutions, the execution in one VM can impact the timing of execution of a thread in another VM. Many virtualization solutions do not offer complete timing isolation. This is true even for RT-Xen, which was designed for real-time systems. As already mentioned, though, one can use an analysis to obtain bounds on delay even if complete isolation is not achieved. With these bounds, it is (under certain assumptions) possible to make changes of software in one VM and reprove its timing properties without having to reprove the entire system. This is done with compositional reasoning—not timing isolation.

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.3 discusses reviews and analyses of software architecture:

> c. The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer. If the software architecture and hardware/software features of the target computer are formally modeled, then potential conflicts can be detected through formal analysis [B-39, p. 21].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software verification process. Section FM.6.3.3 discusses reviews and analyses of software architecture:

f. The objective is to ensure that partitioning breaches are prevented. If the software architecture is formally modeled, some aspects of partitioning integrity can be verified by formal analysis [B-39, p. 21].

It is worth mentioning that with most virtualization solutions, the execution of one thread in one VM can impact the timing of another thread in another VM. If time partitioning (ARINC 653) is used, then this dependency can be caused by cache eviction. If time partitioning is not used, but an event-triggered real-time scheduler is used in the VMM, then this can still happen simply because of CPU scheduling.

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.3.4 discusses reviews and analyses of source code:

f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed-point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing, and this impact should be assessed. If mathematically defined syntax and semantics exist for the Source Code, then these characteristics can be checked using formal analysis. The mathematically defined syntax and semantics may need to take into account the programming language standards, compiler information (for example, default behavior and configuration options), and characteristics of the target computer [B-39, p. 21].

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

Section FM.6 in DO-333 covers the software-verification process. Section FM.6.7 discusses formal analysis of the executable object code:

> Verification of the Executable Object Code is primarily performed by testing. This can be assisted by reviews and analyses, as indicated in DO-178C. Formal analyses exist to establish specific properties, such as worst-case execution time and stack usage. These analyses can replace some testing aspects of the Executable Object Code but not all. Tests executed in target hardware are always required to ensure that the software in the target computer will satisfy the high-level requirements as defined in DO-189C section 6.4.3, item a.
>
> The verification activities should verify: Correct operation of the software in the target computer environment… [B-39, p. 22].

Section FM.6.7 also discusses “time-related functions” [B-39, p. 24–25], “compatibility with [the] target computer” [p. 26], and the use of formal methods to compute safe bounds on resource consumption (e.g., stack usage and worst-case execution timing) [p. 27]. It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been discussed on many occasions above).

Section FM.12 in DO-333 covers additional considerations. Section FM.12.35 discusses coverage analysis when using a combination of formal methods and testing:

> Even when a combination of formal methods and testing is used, functional tests executed in target hardware are always required to ensure that the software in the target computer will satisfy the high-level requirements as defined in DO-178C section 6.4.3, item c. [B-39, p. 43]

It is worth mentioning that with virtualization, the target computer may be a VM. This may impact timing (which has been previously discussed).

B.6.5 DO-254 OVERVIEW

DO-254 [B-40] is a document that gives guidance for developing airborne electronic hardware. Hardware design is not within the scope of this report, but a VM can be used to replace a physical computer; for this reason, DO-254 is briefly discussed.

DO-254 can be thought of as DO-178C, except for hardware. DO-254 is process driven and requires the applicant to submit plans. DO-254 uses similar concepts as DO-178C. It discusses high-level requirements and low-level requirements. The system is composed of components, each component has a level, and the levels are the same as in DO-178C. However, DO-254 refers to the levels as hardware assurance design levels. They play the same role as software levels in DO-178C.

DO-254 was written in 2000. Today, there is hardware that was not available or not common at that time. Specifically, the following hardware is available today that was not available in 2000: multicore processors, processors with simultaneous multithreading (also called hyperthreading), processor chips with integrated graphics processor units, processor chips with x86-compatible processors, and integrated FPGAs. Virtualization is also frequently used in desktop computing and server farms today but not in 2000.

A hardware unit is categorized as simple or complex. A hardware unit is simple if "a comprehensive combination of deterministic tests and analyses appropriate to the design assurance level can ensure correct functional performance under all foreseeable operating conditions with no anomalous behavior" [B-40]. A hardware item that cannot be categorized as simple should be categorized as complex. DO-254 discusses complex hardware items.

B.6.6 DO-254-SPECIFIC REMARKS RELATED TO VIRTUAL MACHINES

Section 2 in DO-254 covers system aspects of hardware design assurance. Section 2.1.1 discusses information flow from the system development process to the hardware design life-cycle process:

> The information flow may include: … 3. Allocated probabilities and at risk exposure times for hardware functional failures [B-40, p. 11].

It is worth mentioning that if a physical processor is replaced with a VM, then the fault probabilities of the VM will depend on the underlying physical processor that provides the VM.

Section 2 in DO-254 covers system aspects of hardware design assurance. Section 2.1.2 discusses information flow from the hardware design life-cycle process to the system development process:

> The information flow may include: … 3. Implementation architecture, including fault containment boundaries. [B-40, p. 11]

If a system is designed so that each software component has its own dedicated processor, the system is modified so that each software component has its own dedicated VM, and all VMs execute on a single physical processor, then fault containment boundaries change.

Section 2 in DO-254 covers system aspects of hardware-design assurance. Section 2.1.3 discusses information flow between the hardware design life-cycle process and the software life-cycle process:

> The information flow may include: 1. Derived requirements needed for integration, such as definition of protocols, timing constraints, and addressing schemes for the interface between software and hardware [B-40, p. 12].

With most virtualization solutions, the execution of one thread in one VM can impact the timing of another thread in another VM. If time partitioning (ARINC 653) is used, then this can be caused by cache eviction. If time partitioning is not used, but an event-triggered real-time scheduler is used in the VMM, then this can still happen simply because of CPU scheduling.

Section 2 in DO-254 covers system aspects of hardware design assurance. Section 2.3.1 discusses hardware safety assessment considerations:

> If a hardware item contains functions that individually have different design assurance levels, such situations may be addressed by either of the following methods:
>
> * The entire item may be assured at the highest design assurance level.
>
> * The individual hardware function may be assured separately at the respective hardware design assurance levels as defined by the hardware safety assessment, if their function, interfaces and shared resources can be protected from adverse effects of functions of lower design assurance levels. Design assurance of shared resources should be the design assurance level of the function with the highest level [B-40, p. 15].

The second bullet mentions shared resources in hardware. If two hardware items are replaced by two VMs, then there will be additional shared resources in the memory system (cache, memory buffer row) of the processor that executes the VMM.

Section 2 in DO-254 covers system aspects of hardware design assurance. Section 2.3.3 discusses qualitative assessment of hardware design errors and upsets:

> Redundancy management techniques and quantitative assessment methods to be used should be selected so that potential common mode faults and the effects of upsets are precluded or mitigated [B-40, p. 16].

If two hardware elements are replaced with two VMs, then the software of the VMM becomes a source of common mode faults. From a timing perspective, the memory system also becomes a source of common mode faults.

B.6.7 CAST-32

CAST-32 is a position paper discussing multicore processors. It stands for Certification Authorities Software Team. The paper assumes that a computer system has two processor cores and only one application and that the processor does not use hyperthreading.

The first 11 pages of the document stress that 1) thermal management may change the clock frequency of a processor core, 2) multicore processors have undocumented features, 3) multicore processors have not been used in safety-critical systems and, therefore, there is no service history, and 4) cores that are not used should be deactivated. Then the document describes VMMs (referred to as hypervisors) and states three objectives:

1. MCP_Determinism_4: An applicant should state whether hypervisors are used.

2. MCP_Determinism_5: An applicant should show "how they intend to show compliance of the software hypervisor with the certification authority's applicable guidance and has successfully conducted those activities that they planned."

3. MCP_Determinism_6: Some multicore processors include a hypervisor built into the hardware. If this is used, the applicant should state how the applicant intends to verify the activated functionality of the multicore processor.

The document continues by stressing that 1) multicore processors typically have implicitly shared resources (e.g., caches) that create interference channels between processor cores from the perspective of timing analysis; 2) many hardware features (e.g., interconnection networks on a chip) are undocumented; 3) some interconnection networks can cause transactions to be lost or served in another order than the one in which they were requested; 4) in existing certified systems, each software application is statically allocated to a processor core; and 5) there is not enough guidance on multicore processors. The latter is expressed as follows:

> Because of the lack of industry experience with MCPs, the certification authorities are concerned that when applicants attempt to conduct software verification using only the existing guidance, their approach to verification may not provide enough assurance that all the software would comply with its requirements when executing in parallel with the mechanisms of the MCP permitting interference between the software hosted on the two cores.

B.7 CONCLUSION

Operating systems (OSs) serve the double role of creating a high-level abstraction of the hardware (OS services like the File System) and providing the illusion of a dedicated machine to each of the programs that run simultaneously in a computer. Early implementations of virtual machines (VMs) (e.g., in the IBM 360) aimed at separating the dedicated machine illusion from the high-level hardware abstraction. In recent years, VM implementations have taken a new role in supporting the "cloud" industry that sells platform as a service, in which different companies package their systems in VMs that can be hosted in shared hardware. These VMs are isolated from each other in space and time. However, the temporal isolation is limited to average-case response time and throughput without hard guarantees. Such isolation has been enough for information systems. Furthermore, these metrics are compatible with the advances in hardware, such as cache, memory systems, and pipelined processors. In other words, because these different layers focus on the average-case response time, they do not need to have a strict coordination, because their combined behavior averages out to a reasonable average-case response time and throughput.

Whereas the spatial isolation implemented by VMs works fine for real-time embedded systems (RTES) and in particular for avionics systems, temporal isolation does not. More importantly, the number of performance improvements in hardware create a twofold problem. They focus on average-case response time that can complicate the worst-case response time evaluation necessary for RTES. However, the combined effect of the different layers of hardware improvements (e.g., pipeline plus cache plus memory access scheduling/bank parallelism) cannot be easily understood from the worst-case point of view. Instead, it is necessary to develop adaptations that restrict and coordinate their behavior across layers to make VMs predictable and amenable to worst-case analysis, as was discussed in sections 2.1.2 and 2.5.

VMs are not the only way to provide isolation between components, as discussed in section 4. Other implementations are more amenable to providing worst-case temporal isolation to system components. In addition, the techniques presented in section 4 can also be used in combination with VMs.

As mentioned in section 5, VMs can be helpful to simplify the development processes when the system is developed in a distributed fashion across organizations. However, to make this possible, it is necessary to support the development with appropriate analysis techniques that can evaluate properties at the individual group level and at the system integration level, typically configured as compositional techniques.

Finally, the use of VMs in avionics systems requires careful consideration of certification standards. These issues are covered in section 6. Different aspects of the standards can be affected by the use of VMs in avionics systems. In some contexts, VMs have the potential to help, but in a larger number of contexts, they can be the cause of concern because they hide decisions in the hypervisor that may hinder temporal determinism and isolation.

B.8 REFERENCES

B-1. Diaz, A. L. (2011). Service Level Agreements in the Cloud: Who cares? Wired.com Retrieved from http://www.wired.com/insights/2011/12/service-level-agreements-in-the-cloud-who-cares/

B-2. Hennessy, J. and Patterson, D. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc. (ISBN:012383872X 9780123838728).

B-3. Jacob, B., Ng, S.W., Wang, D.T. (2007). *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

B-4. Kim, H., Niz, D.D., Andersson, B., Klein, M.H., Mutlu, O., Rajkumar, R. (2016). Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, *52*(3), 356–395.

B-5. Park, S., Shrivastava, A., Paek, Y. (2008). Hiding Cache Miss Penalty Using Priority-based Execution for Embedded Processors. *2008 Design, Automation and Test in Europe*, 1190–1195.

B-6. ARM. (2017). *ARM11 MPCore Processor Technical Reference Manual*. Retrieved from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360e/I1002919.html.

B-7. http://www.absint.com

B-8. Lee, C., Hahn, J., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M., Kim, C. (1996). Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, *47*(6), 700–713.

B-9. Bui, B.D., Cacao, M., Sha, L., Martinez, J. (2008). Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 101–110.

B-10. Basumallick, S. and Nilsen, K. (1994). Cache issues in real-time systems. In *ACM Workshop on Language, Compiler, and Tools for Real-Time Systems*.

B-11. Kim, H., Kandhalu, A., Rajkumar, R. (2013). A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. *2013 25th Euromicro Conference on Real-Time Systems*, 80–89.

B-12. Suzuki, N., Kim, H., Niz, D.D., Andersson, B., Wrage, L., Klein, M.H., Rajkumar, R. (2013). Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. *2013 IEEE 16th International Conference on Computational Science and Engineering*, 685–692.

B-13. Takase, H., Tomiyama, H., Takada, H. (2010). Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, 1124–1129.

B-14. Gauthier, L., Ishihara, T., Takase, H., Tomiyama, H., Takada, H. (2010). Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multi-task systems. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (CASES '10).*

B-15. Panchamukhi, S.A., Mueller, F. (2015). Providing task isolation via TLB coloring. *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 3–13.

B-16. Betti, E., Bak, S., Pellizzoni, R., Caccamo, M., Sha, L. (2013). Real-Time I/O Management System with COTS Peripherals. *IEEE Transactions on Computers, 62*(1), 45–58.

B-17. Fisher, N., Chen, J., Wang, S., Thiele, L. (2009). Thermal-Aware Global Real-Time Scheduling on Multicore Systems. *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 131–140.

B-18. FAA Report. (2016). Survey of Literature Related to Virtual Machines.

B-19. Richt, B., Chen, G., Hu, B., Huang, K., Knoll, A. (2014). Cache Management and Time-triggered Scheduling for Hard Real-time MPSoCs. Technical Report TUM.

B-20. Chen, G., Hu, B., Huang, K., Knoll, A., Liu, D., Stefanov, T. (2014). Automatic cache partitioning and time-triggered scheduling for real-time MPSoCs. *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 1–8.

B-21. Cilku, B., Puschner, P.P. (2013). *Towards Temporal and Spatial Isolation in Memory Hierarchies for Mixed-Criticality Systems with Hypervisors*. Proceedings from the ReTiMiCS, RTCSA 2013, Taipei, Taiwan.

B-22. Thiele, L., Chakraborty, S., Naedele, M. (2000). *Real-time calculus for scheduling hard real-time systems*. Proceedings from the 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353). Geneva, Switzerland.

B-23. Samolej, S. (2011). ARINC Specification 653 Based Real-Time Software Engineering. *e-Informatica, 5*(1), 39–49.

B-24. Easwaran, A., Lee, I., Sokolsky, O., Vestal, S. (2009). A Compositional Scheduling Framework for Digital Avionics Systems. *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 371–380.

B-25. Shin, I., Lee, I. (2003). Periodic Resource Model for Compositional Real-Time Guarantees. *RTSS*. In *Proceedings of IEEE Real-Time Systems Symposium*.

B-26. Lee, J., Xi, S., Chen, S., Phan, L.T., Gill, C.D., Lee, I., Lu, C., Sokolsky, O. (2012). Realizing Compositional Scheduling through Virtualization. *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, 13–22., Beijing.

B-27. Kim, H., Wang, S., Rajkumar, R. (2014). vMPCP: A Synchronization Framework for Multi-core Virtual Machines. *2014 IEEE Real-Time Systems Symposium*, 86–95.

B-28. Vasudevan, A., Chaki, S., Jia, L., McCune, J.M., Newsome, J., Datta, A. (2013). Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. *2013 IEEE Symposium on Security and Privacy*, 430–444.

B-29. Hypervisor IOMMU Architecture. logicpundit.com/blog/wp-content/uploads/2015/07/Hyper-V-IOMMU.pptx

B-30. Ben-Yehuda, Xenidis, J. (2006). Utilizing IOMMUs for Virtualization in Linux and Xen Muli. Presented at OLS’06: The 2006 Ottawa Linux Symposium. Ottawa, Canada.

B-31. Leyva-del-Foyo, L.E., Mejía-Alvarez, P., & Niz, D.D. (2012). Integrated Task and Interrupt Management for Real-Time Systems. *ACM Trans. Embedded Comput. Syst., 11*(2), 32:1–32:31.

B-32. The QNX Neutrino Microkernel. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html

B-33. The seL4 Microkernel. https://sel4.systems/

B-34. Sewell, T., Kam, F., & Heiser, G. (2016). Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1–11.

B-35. Accetta, M.J., Baron, R.V., Bolosky, W.J., Golub, D.B., Rashid, R.F., Tevanian, A., & Young, M. (1986). Mach: A New Kernel Foundation for UNIX Development. *USENIX Summer*.

B-36. Rushby, J.M. (1981). Design and Verification of Secure Systems. *SOSP*. (12–21).

B-37. RTCA. (2005). Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. (RTCA DO-297).

B-38. RTCA. (2011). Software Considerations in Airborne Systems and Equipment Certification. (RTCA DO-178C).

B-39. RTCA. (2011). Formal Methods Supplement to DO-178C and DO-278A (RTCA DO-333).

B-40. RTCA. (2000). Design Assurance Guidance for Airborne Electronic Hardware. (RTCA DO-254).

APPENDIX C—EVALUATION OF VERIFICATION TECHNOLOGIES FOR VIRTUAL MACHINES

C.1 TIMING VERIFICATION

Recall from the previous report, “Assurance Issues on VMs in Avionics Systems,” that avionics software interacts with the physical world by reading sensors and computing actuation commands; this requires that the timing be correct (appendix B). For example, if timing is incorrect, then a feedback controller may become unstable. Another case in which incorrect timing is undesirable is in a sensor fusion system. Such a system may take multiple different types of sensor readings as input, and its software is intended to draw a conclusion based on these sensor readings, assuming that they are taken at approximately the same time. However, if these sensor readings relate to different times, then the conclusion drawn may be incorrect. For these reasons, this section discusses timing verification.

C.1.1 NETWORK CALCULUS

Network calculus [C-1] is not intended for analyzing virtual machines (VMs) or delays of software in processors. However, because it deals with timing, it can be thought of as complementary to other methods described in this report. Network calculus considers a computer network with network elements (e.g., multiplexers with queues, demultiplexers, fixed-delay elements), links, and traffic flows. With this information, network calculus computes 1) upper bounds on the delay of traffic flows and 2) upper bounds on buffer space needed for queues. Network calculus can be distinguished from other methods as follows: Network calculus describes each traffic flow with two parameters—burst size and throughput—and the computations needed for the analysis run fast. In contrast, other techniques such as schedulability analysis describe software with more details; a task may be described with three or more parameters—such as period, deadline, and execution time—providing more accurate predictions at the cost of a longer analysis time. Schedulability analysis performed with timed automata (described later in this report) provides analysis of software with even more detailed models, and this comes at the cost of even longer analysis time.

An example of a result in network calculus is shown in equation C-1: Consider a multiplexer with two input flows, one output flow, and first-in, first-out (FIFO) queuing discipline. Then the input-to-output delay of a bit in flow 1 is upper bounded by:

$$\frac{1}{C_{out}} * max_{u \geq 0}(b_1(u) + b_2(u + L/C_2) - C_{out} * u) \qquad \text{(C-1)}$$

In the expression above, subscript 1 relates to traffic flow 1, and subscript 2 relates to traffic flow 2. C_{out} is the speed of the output link (in terms of bits per second). L is an upper bound on each packet (in terms of bits). $b_1(u)$ is an upper bound on the number of bits required to be transmitted from traffic flow 1 among all packets received in a time interval of duration u. This is analogous for flow 2. Typical network calculus assumes that the function $b(u)$ is represented by two parameters; specifically, $b(u)$ is represented as $b(u) = const_1 + const_2 * u$.

Network calculus also allows computations of burstiness of output queues. In this way, if the rate and burstiness of two flows is known, and these two flows are incoming flows to a multiplexer, then we can compute the burstiness of the outgoing flow; this helps when computing further delays in a network. In addition, network calculus allows us to compute the maximum buffer space needed in a multiplexer.

Network calculus introduces the concept of a regulator as an element that takes a traffic flow as input and produces a traffic flow as output, and this traffic flow is shaped. Specifically, a regulator has two parameters, burstiness and rate, and ensures that the outgoing flow from the regulator complies with the two parameters regardless of the input flow. In this way, a regulator can be thought of as having a similar function as the reservation/server mechanism mentioned in the previous report—"Assurance Issues on VMs in Avionics Systems" (appendix B). Note, however, that the model used in network calculus is less expressive than those typically used in schedulability analysis.

Network calculus has been extended to analyze not only networks with single-network elements but also networks with multiple network elements. For details, see [C-2].

The ideas of network calculus have been used in real-time calculus, which is intended for processor scheduling. Real-time calculus is covered in the next section.

C.1.2 REAL-TIME CALCULUS

Real-time calculus [C-3] defines a mathematical framework (calculus) based on network calculus that models recurrent (e.g., periodic) tasks (i.e., threads), resource consumption (central processing unit, or CPU, cycles), and processor resource production (CPU cycles) as functions. It defines special operations to add and subtract these functions. These operations allow for evaluating 1) whether there are enough resources (CPU cycles) to finish the execution of the set of tasks, 2) the response time of these tasks, 3) the number of task activations that may be pending (e.g., buffers for packets to be processed), and 4) the remaining resource (CPU cycles) after servicing these tasks.

Tasks are modeled as request functions $R(t)$ that represent the total amount of computation (CPU cycles) that the task has requested up to time t. Processors, however, are modeled as capacity functions $C(t)$ that represent the computation that the processor can deliver up to time t. Using these two functions, it is possible to evaluate how many requests have been "delivered" within an interval of time with the equation:

$$R'(t) = \min_{0 \leq u \leq t} \{R(u) + C(t) - C(u)\} \qquad \text{(C-2)}$$

and the remainder capacity function $C'(t) = C(t) - R(t)$. For example, consider a periodic task that executes for 10 ms every 100 ms starting its execution at time zero. The processor is modeled as a capacity function $C(t)$ that provides one unit of execution capacity every unit of time. This is an obvious way to model the processor, but if its frequency is reduced to half, it could provide only half a unit of processing capacity per unit of time. Now, it will be possible to evaluate $R'(t)$ at different times as presented in Table C-1.

Table C-1. $R'(t)$ at different times

t	$R(u)$	$C(t)$	$C(u)$	$R'(t)$	$C'(t)$
5	5	5	5	5	0
10	10	10	10	10	0
20	10	20	20	10	10
100	10	100	100	10	90
105	15	105	105	15	90
110	20	110	110	20	90
120	20	120	120	20	100
200	20	200	200	20	180

It is worth noting that at any time between 0 and 10, u can take any value because $R(u)$ and $C(u)$ will cancel each other.

The request and capacity functions previously defined work only for a specific time interval; in the example, they work only for time 0–200. This is not enough to guarantee schedulability, because in a different interval, even of the same length (e.g., from 200–400) the conditions may be different (e.g., another task may arrive). Therefore, to guarantee schedulability, boundary functions are defined to make them work for any interval of a particular length. To do this, Thiele et al. [C-3] define a minimum request bounding function $\alpha_r(t)$ such that, for an interval between two time instants s and t and any subinterval within its bounds, the request function is bounded from above: that is, $(t) - R(s) \leq \alpha_r(t - s)\ \forall s \leq t$. For instance, if $s = 0$ and $t = 200$, then $R(5) - R(0) \leq \alpha_r(5 - 0)$ but also $R(15) - R(5) \leq \alpha_r(15 - 5)$ and any other subinterval. The request bound function then can be calculated as:

$$\alpha_r(\Delta) = \max_{u \geq 0}\{R(\Delta + u) - R(u)\} \tag{C-3}$$

Similarly, a maximum delivery curve β in any subinterval given a processor capacity function is defined such that $C(t) - C(s) \geq \beta(t - s)\ \forall s \leq t$. This can be calculated as:

$$\beta(\Delta) = \min_{u \geq 0}\{C(\Delta + u) - C(u)\} \tag{C-4}$$

The remaining processing capacity C' then can be bounded by the function:

$$\beta'(\Delta) = \max_{0 \leq u \leq \Delta}\{\beta(u) - \alpha_r(u)\} \tag{C-5}$$

C.1.2.1 Rate-Monotonic Example

Now apply this to a set of periodic tasks scheduled under fixed-priority scheduling with rate-monotonic priority assignment. Consider two tasks: task τ_1 with period $T_1 = 100$ and maximum execution time $C_1 = 10$, and task τ_2 with period $T_2 = 200$ and maximum execution time $C_2 = 20$. The processor is initially modeled providing a continuous capacity of 1 for each unit of time. It then will calculate:

1. the request bound function of task τ_1 as $\alpha_r^1(t) = \left\lceil\frac{t}{T_1}\right\rceil C_1 = \left\lceil\frac{t}{100}\right\rceil 10$
2. the request bound function of task τ_2 as $\alpha_r^2(t) = \left\lceil\frac{t}{T_2}\right\rceil C_2 = \left\lceil\frac{t}{200}\right\rceil 20$
3. the maximum processor delivery curve $\beta(t) = t$

There functions are derived from the response time calculation in [C-4]. Now, the remaining processing capacity can be calculated after scheduling τ_1:

$$\beta'(t) = \max_{0\le u\le t}\{t - \left\lceil\frac{t}{100}\right\rceil 10\} \tag{C-6}$$

This residual can be used to schedule τ_2 (given that it runs only when τ_1 does not). This gives us:

$$\beta''(t) = \max_{0\le u\le t}\{\left(t - \left\lceil\frac{t}{100}\right\rceil\right) 10 - \left\lceil\frac{t}{200}\right\rceil 20\} \tag{C-7}$$

The request and capacity functions can now be evaluated as noted ing Table C-2:

Table C-2. Capacity function evaluation

t	$\alpha_r^1(t)$	$\alpha_r^2(t)$	$\beta'(t)$	$\beta''(t)$
5	10	20	–5	–25
10	10	20	0	–20
20	10	20	10	–10
30	10	20	20	0
40	10	20	30	10
100	10	20	90	70

By time $t = 30$, both τ_1 and τ_2 have finished, validating that they are schedulable. In addition, the negative numbers in columns $\beta'(t)$ and $\beta''(t)$ can be seen as a CPU cycle debt (still needed to satisfy the demand).

C.1.2.2 Compositional Verification

Capacity functions can also be used to encode how a scheduler gives processing time to a component with a set of tasks. For instance, it is possible to model the way an ARINC 653 partition scheduler gives slots to a partition. As an example, consider a system with two partitions of 2 ms each; first, 2 ms are given to partition 1 and the next 2 ms to partition 2, repeating this pattern forever. Then from the point of view of one partition, a delivery curve will be seen:

$$\beta(t) = \begin{cases} 0 \; if \; t < 2 \\ \frac{t-2}{2} \; if \; t \ge 2 \end{cases} \tag{C-8}$$

The delay at the beginning of the partition time ensures that the worst case (minimum processor capacity) is observed. Then, if the two tasks in the rate-monotonic example are shown, in Table C-3 results will be:

Table C-3. Rate-monotonic example

t	$\beta(t)$	$\alpha_r^1(t)$	$\alpha_r^2(t)$	$\beta'(t)$	$\beta''(t)$
2	0	10	20	–10	–30
4	1	10	20	–9	–29
6	2	10	20	–8	–28
8	3	10	20	–7	–27
10	4	10	20	–6	–26
20	9	10	20	–1	–21
22	10	10	20	0	–20
24	11	10	20	1	–19
26	12	10	20	2	–18
28	13	10	20	3	–17
30	14	10	20	4	–16
40	19	10	20	9	–11
42	20	10	20	10	–10
60	29	10	20	19	–1
62	30	10	20	20	0
64	31	10	20	21	1

By time 62, both tasks have finished, validating that they are schedulable.

This compositional verification technique is needed to verify the hierarchical arrangement of schedulers within VMs (i.e., a hypervisor scheduler giving CPU time to a VM and the VM host operating system [OS] scheduler giving CPU time to a particular task).

C.1.2.3 End-to-End Timing Verification

To verify end-to-end timing of a sequence of tasks $(\tau_1, \ldots, \tau_n)$ that execute one after another (τ_i followed by τ_{i+1}), real-time calculus uses the completion of a task τ_i as the arrival of the task τ_{i+1}. These completions are encoded as arrival curves (or functions). These curves are then used as a generalization of any task arrival, even the periodic ones that are triggered by a timer. However, in this case, it is not only important to evaluate the latest instant when a task can complete but also the earliest. This is because if a task τ_i completes earlier than in the worst case, it will trigger the successor task τ_{i+1}, which may preempt earlier another task τ_j running on the same processor as τ_{i+1}. As a result, now both upper and lower arrival bounding functions are calculated. This is automatically supported in the Matlab real-time calculus toolbox created by Thiele et al. [C-5]. In this tool, arrival curves are specified in two parts:

1. an initial aperiodic part that specifies a sequence of intervals with a start and an end and a slope that indicates the rate of capacity time given (e.g., one unit of processing per unit of time, or half a unit)

2. a periodic part that is another sequence of intervals with their slope, followed by a periodicity specification

For a deeper discussion of this topic, see the Real-Time Calculus Tutorial [C-5], from which additional references can be found.

C.1.2.4 Limitations

Real-time calculus is a powerful framework and tool to validate the timing of arbitrarily complex schedulers and systems. However, the real challenge is in the specification of the request and capacity curves that are not straightforward to encode and prove that they really represent true upper and lower bounds. Depending on the precision of the specification, the approximation can be tight or not, potentially leading to a significant waste of resources. From the rate-monotonic example, it is clear that the curves can be developed from previous results that can be used independently from real-time calculus. Furthermore, solving the scheduling of a system specified in real-time calculus can easily get into nonscalable territory.

C.1.3 COMPOSITIONAL ANALYSIS OF REAL-TIME SYSTEMS

Compositional analysis of real-time systems (CARTS) [C-6] is a tool for computing interfaces of components in hierarchical systems. The main idea is that a system is composed of components, and in each component there are real-time tasks. A real-time task is described with three parameters: period, deadline, and execution time. The tool computes an interface for each component. This interface describes the total resource consumption of the tasks in the component. With this knowledge, it is then possible to check that the scheduler that allocates processing times to components (root scheduler) has enough capacity to supply enough processing time to each component.

C.1.4 TIMED AUTOMATA ANALYSIS

In this section, schedulability analysis using timed automata will be discussed. The section begins by describing the original notion of timed automata. Then it covers a more modern notion of timed automata. Finally, it shows how this modern notion can be used to solve various schedulability analysis problems.

Alur and Dill [C-7] presented the original theory of timed automata. This theory describes a system as a set of states, a set of clocks, and a set of transitions between some pairs of states; conditions that must be true for a transition to occur; and statements on which clocks should be reset when certain transitions occur. A timed word is a sequence of pairs that consist of a symbol and a time (this indicates a sequence of events, in which each symbol indicates the type of event occurring, and the time indicates the time at which the event occurred). The set of timed words that a system can generate is called the timed language of the system. The correctness of a system can also be specified as a timed language. Then the correctness can be checked through language inclusion: Is the timed language of the implemented system a subset of the timed language of the specification? If yes, then the system is correct. If no, the system is incorrect. To specify behavior of a system, it is important to not only describe which transitions are allowed but also to specify that certain transitions should take place (otherwise a system may stay in the initial state forever). Therefore, it is important to specify progress. The original theory of timed automata specifies progress by requiring that timed words are infinite; that is, the system never stops. It also specifies that certain states must be visited infinitely often.

Henzinger et al. present a simplified notion of timed automata called "timed-safety automata" [C-8, section 3.4]. The idea is to describe progress in another way. Instead of requiring that timed words are infinite, and certain states must be visited infinitely often, it allows finite timed words and instead adds invariants to states. An invariant is a condition stating that the automata is allowed to visit the state only if the invariant is true. For example, if x is a clock and an invariant of a state if $x \leq 5$, then this forces the automaton to make progress; if the automaton is in this state and x becomes 5, then the automation cannot stay in this state and must perform a transition. Today, timed-safety automata is used more often than the original theory of timed automata; therefore, the term-timed automata is often used to refer to timed-safety automata. An example of a timed automata is shown in figure C-1.

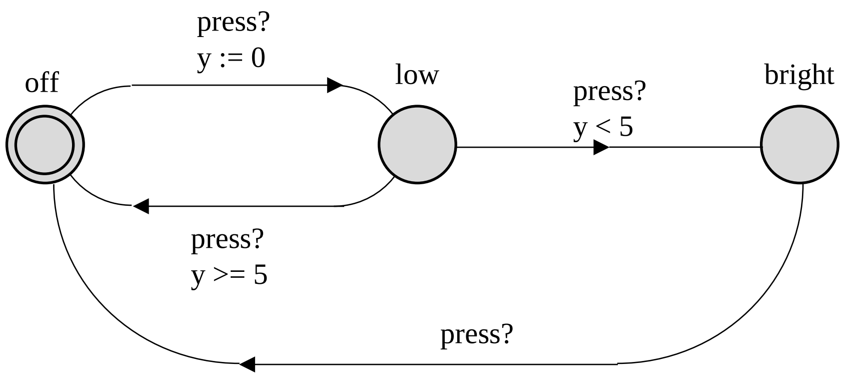


Figure C-1. Example of timed automata

This example shows a lamp that can be off or on. If it is on, then it can be in one of two possible states (low light or bright light). Initially, the lamp is in the off state (shown with a circle inside a circle). If a user presses a button when the lamp is in this state, then the clock y is reset (i.e., the value of the clock is set to zero) and the lamp transitions to another state (marked "low" in figure C-1). The value of the clock y progresses. After two time units, it holds that $y = 2$. If a user presses a button again, then the lamp transitions to a new state. If $y < 5$, then it transitions to the state "bright." If $y \geq 5$, then the lamp transitions to the "off" state. This example makes use of clocks (y is a clock). It also makes use of guards on transitions ($y \geq 5$ is a guard on the transition from low to off). It does not, however, make use of invariants. The theory of timed-safety automata uses location as a discrete state. Therefore, in the example above, it would be said there are three locations rather than three states.

Numerous tools have been developed for proving properties of timed automata; the most popular one is Uppaal [C-9]. This tool allows a software practitioner to specify a system as a set of different timed automata and describe their interaction. The latter is achieved through synchronizations of transitions. For example, timed-automaton A has a transition 1, and timed-automaton B has a transition 7, and these transitions should be made to happen simultaneously. Transition 1 of timed-automata A can be attached "mysynch!" and transition 7 of timed-automata B can be attached “mysync?” In this way, two automata make these two transitions simultaneously. Uppaal also comes with a verification engine that can be used to prove that a certain state is not reachable. For example, consider a protocol for achieving mutual exclusion in a distributed system and how to express that this protocol works correctly. One state could be named CS (meaning critical section) and then expressed for each pair of timed automata; it should hold that if one automaton is in CS, then the other is not in CS. To simplify modeling, Uppaal also allows variables. These can be assigned values during transitions, and they can be used as guards.

Norstrom et al. [C-10] were the first to express schedulability analysis using timed automata. They consider nonpreemptive scheduling of tasks (not necessarily periodic or sporadic) in which the execution times of a job are equal to its worst-case execution time (WCET). The paper shows that the schedulability analysis problem can be formulated as reachability with a timed automata. The idea is to introduce the notion of extended automata with tasks. It works like a normal automaton, but for each transition, there is a symbol attached, and this symbol indicates that a job arrives. For example, if there is a transition from state 1 to state 2 in automaton A, and this transition is associated with “b,” it means that when the automaton takes this transition, then a job of task b arrives (i.e., it requests to be executed). There is also a queue associated with the system; this queue keeps track of the jobs of each task that has arrived, the time until the deadline, and the remaining execution time. For such a queue, unschedulability can be defined; this occurs when there is a deadline relative to the current time such that this deadline is smaller than the sum of remaining execution times of jobs with deadlines smaller than this deadline. This condition expresses that the amount of execution required before a deadline is so large that not all of the execution can be finished before the deadline—therefore a deadline miss results.

Can the extended timed automaton with tasks generate jobs so that the queue becomes unschedulable? The paper presents an approach to answer this question; this is done by expressing the dynamics of the queue as a normal timed automata with synch points so that when the timed automaton extended with tasks makes the transition with “b!”, then the automaton implementing the queue makes the transition with “b?”. Therefore, there ends up being two automata: one timed automaton that expresses job arrivals and one timed automaton that expresses scheduling decision. In the latter, there is a state “error” that represents unschedulability (as mentioned above). Therefore, schedulability is checked by asking the question: Is it possible for the system to reach the state “error”? This question can be answered with a tool; the paper shows how to do it with Uppaal. Figure C-2 shows an example of a timed automaton that expresses job arrivals.

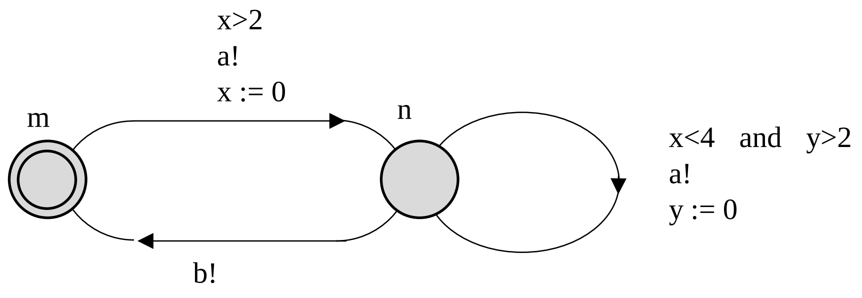


Figure C-2. Job arrival timed automaton

In this example, there are two states: "*m*" and "*n*." In state *m*, it is possible for a job of task *a* to arrive. In state *n*, it is possible for a job of task *b* to arrive, and it is also possible for a job of task *a* to arrive. Clocks (*x* and *y*) place restrictions on how frequently jobs can arrive. For example, if a transition from *m* to *n* is made (and therefore a job of task *a* arrives), then the clock *x* is reset (i.e., *x* is set to zero). If a job of task *a* arrives again (through the self-loop of state *n*), then it is necessary to wait two time units until a job of task *a* can arrive again. The timed automaton that expresses scheduling will now be discussed. Norstrom et al. show a timed automaton for the scheduler [C-10, p. 5]. To avoid clutter, it will not be shown; this automaton is shown in figure C-3 with only a few of the transitions.

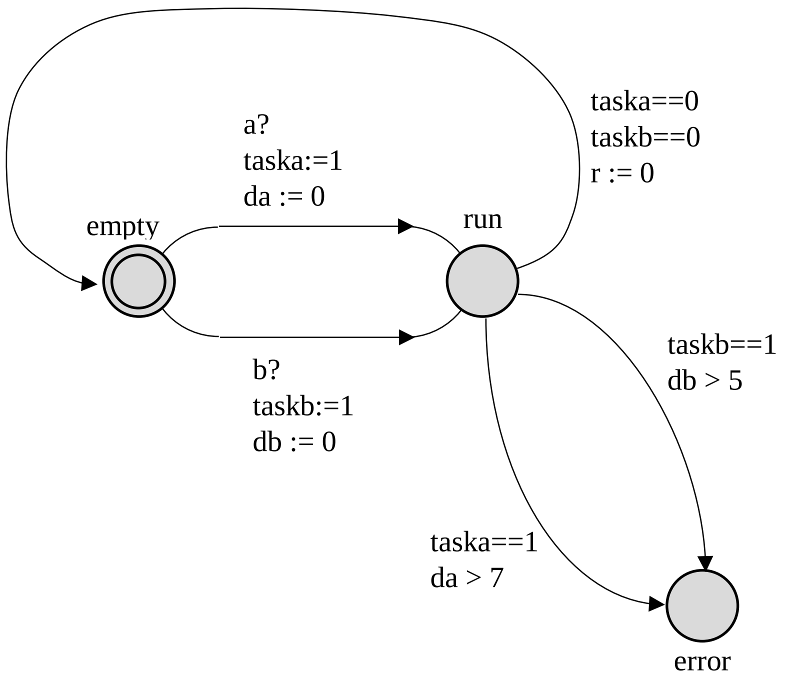


Figure C-3. Scheduler automaton

Although figure C-3 does not show the entire scheduling automaton, some of its behavior can be understood. The scheduler deals with two tasks: task *a* and task *b*. The deadline of task *a* is 7. This means that whenever a job of task *a* arrives, it must finish within 7 time units; otherwise its deadline is missed. The deadline of task *b* is 5; its meaning is analogous. The system starts in an empty state (shown as "empty"). If the task automaton generates a job of task *a* (i.e., performs a transition with *a*!), then the scheduler automaton performs the transition *a*?. With this transition, two variable assignments are performed. The variable *taska* is assigned 1 and the clock *da* is assigned 0. The variable *taska* is used to indicate whether a job of task *a* is ready. The clock *da* is used to indicate the amount of time from the arrival of the job of task *a* until now. If *da* exceeds the deadline of a job of task *a*, and this job is still executing, then we know that there was a deadline miss. There is a transition *taska* == 1 *da* > 7 that performs this; it transitions from the state "run" to "error." See page 5 of the paper [C-10] for further details (transitions for the case when a job arrives when the processor is already busy and transitions for the case that a job has finished its execution).

Whereas Norstrom et al. consider only nonpreemptive scheduling, Fersman et al. [C-11] deal with preemptive scheduling and discuss decidability and undecidability. There is a more powerful formalism called stop-watch automata; it differs from timed automata, in which clocks are always increasing unless they are reset. However, in stop-watch automata, a clock can be stopped so that it does not increase. Stop-watch automata are potentially useful for checking schedulability of a system that uses preemptive scheduling because the amount of execution that a job has performed needs to be tracked; it is natural to do so with a clock. Each job could be associated with a clock to keep track of the amount of execution it has performed and then let each clock increase when the job executes. However, when a job does not execute, the clock corresponding to this job should not increase. In this way, stop-watch automata could be used to perform schedulability analysis of a system that uses preemptive scheduling.

Unfortunately, reachability analysis for stop-watch automata is undecidable (i.e., there is no algorithm for this problem that is guaranteed to deliver a correct result and terminate). As previously mentioned, it would be natural to model preemptive scheduling with stop-watch automata. Therefore, analyzing preemptive scheduling for general arrivals may be thought to be undecidable. Fersman et al. [C-11] show, however, that schedulability checking for preemptive scheduling with a general arrival model is decidable. It is achieved without stop-watch automata. Specifically, it is achieved by using timed automata that are update automata, in which a clock can have its value subtracted by another clock to model that a task does not execute during the time at which a higher-priority task executes.

This work assumes that a job executes according to WCET. It is actually not stated explicitly in the paper that a job executes according to its WCET, but it can be inferred from the paper in three ways [C-11, pp. 8, 12]:

1. An upper bound on the number of jobs of task P is given by D(P)/C(P). This statement assumes that a job executes according to its WCET.

2. "$c(i,j)$ (a computing clock) is … subtracted with $C(k)$ when the running task, say P_{kl}, is finished if it was preempted after it was started." However, because timed-automata with subtraction of a constant is used, this will only work if $C(k)$ is a constant; therefore, the approach will only work if a job executes according to its WCET.

3. "Note that the maximum number of instances of P_i appearing in a schedulable queue is ceil($D(i)/C(i)$)." This statement assumes that a job executes according to its WCET.

The paper [C-11] shows how to perform schedulability analysis of tasks in which job arrivals are described with a timed automaton. The method in the paper achieves this by performing reachability with a subtraction automata. The main drawback of the approach is that it does not allow a job to execute at less than its WCET. The main advantage of the approach is that it allows a software practitioner to model systems that could not be modeled with normal scheduling theory (normal real-time scheduling theory assumes that jobs from a task arrive periodically or sporadically with a minimum inter-arrival time).

The main idea of the approach in the paper is as follows: For each task, there is an upper bound on the number of jobs that are active. Let $<i,j>$ denote the jth job of process i. An automaton describes the arrival of tasks; this automaton has actions on edges and associates a job arrival for each action. A scheduler automaton is then also formed. If *n* is an upper bound on the number of jobs that could be active, then the scheduler automaton has *n* + 2 locations; one location is idle, another location is an error, and for each possible job, there is also one location running$_{i,j}$. Edges are described between these locations; for each edge, a guard, an action, and an update of variables (in the paper, the updating is called *reset*) is described. For each potential job, there is a tuple describing elapsed time since arrival, total execution time performed so far since arrival, and its status (if it is running, preempted, released, or free). When a job arrives, two clocks are set to zero: one for elapsed time since arrival and one for total execution time. These clocks increase with the progress of real time. This is fine for the clock that keeps track of the elapsed time since job arrival.

The other clock, however, should stop when the job is not executing (in this case, because it has been preempted). This could be done with so-called stop-watch automata, but (as already mentioned) reachability for such automaton is undecidable. The solution used in the paper is as follows: whenever a job <*i*,*j*> finishes, the following is done: for each other job that has been preempted, decrement its clock that records total execution time so far by the execution performed by <*i*,*j*>. This works in the model assumed in the paper because the execution performed by <*i*,*j*> is equal to its WCET. Therefore, reachability analysis on the automata construction used by the authors is equivalent to schedulability testing. The authors show that reachability testing of this type of automata is decidable. The reason for this is that for all reachable states, there is an upper bound on the clock variables. Reachability analysis of automata in which clock values can be subtracted by a constant is decidable (if such an upper bound does not exist, then the problem is undecidable).

Amnell et al. [C-12] present a tool that is based on the schedulability test in [C-11]. This tool not only performs schedulability testing but also some code generation. They extend this tool by allowing the execution time of a job to be specified with a lower bound (C_B) and an upper bound (C_W) [C-13]. This extension is done as follows: Each job is associated with two clocks—*c* and *w*. When a job arrives, its *c* clock is initialized to zero, and its *w* clock is initialized to the WCET of the job minus the best-case execution time of the job. For a job, there is a guard on the event that the job finishes, and this guard is $C_B <= c < = C_B + w$. Whenever a job *J* finishes, all jobs *J′* that were preempted have their *c* and *w* updated as follows: $c_{J'} := c_{J'} – C_{J,B}$ and $w_{J'} := w_{J'} – w_J$. The idea is to maintain bounds on the execution times that a job has performed (*w* indicates the difference between these bounds).

Fersman et al. [C-14] show that schedulability analysis can be performed with *n* + 1 clocks, where *n* is the number of tasks. They also show how to model tasks in which the arrival time depends on a condition on data. Both of these results assume that execution times of a job are constant (i.e., its upper bound is equal to its lower bound).

Pavel and Wang [C-15] show that the problem of determining schedulability (using reachability analysis) is undecidable if all of the following three conditions are true: 1) the execution time of a task has an upper bound and a lower bound, which may be different; 2) the arrival time of a job may depend on the completion time of another job (e.g., precedence constraint); and 3) a task may preempt another task. This is shown through transformation of the halting problem of a two-counter machine to the problem of determining schedulability. The paper also shows that if at most two of the above conditions are true, then schedulability is decidable.

The main idea of the proof that schedulability testing is undecidable if the above three conditions are true is as follows: Consider a two-counter machine. This machine has two registers; each register stores a non-negative number (there is no upper bound on this integer). A control unit executes a program, and this program has three types of instructions: 1) decrement a counter, 2) increment a counter, and 3) branch conditionally depending on whether a counter is zero. The paper constructs a timed automata extended with tasks and shows that it can simulate a two-counter machine; specifically, it shows that if and only if the two-counter machine reaches a halting state, then the extended timed automata reaches a state with a deadline miss. Therefore, with this construction, it is shown that if and only if the two-counter machine reaches a halting state, then the task set is unschedulable. Because determining whether a program on a two-counter machine can reach a halting state is undecidable, it follows that determining schedulability with an extended timed automata is undecidable too.

A key idea in this construction is that a correspondence can be formed between a counter in the two-counter machine and a clock in the timed automata with tasks. If the counter is *v*, the clock is 2^{1-v}. In this way, any value of the counter has a corresponding value of the clock. The paper shows that for each state of the program in the two-counter machine, a state can be formed in the timed automata with tasks. The paper observes that if the program executes a decrement on a counter, it doubles the corresponding clock. The paper constructs transitions in the timed automata such that if and only if the program decrements a counter, the clock value is doubled. Incrementing a counter and making the clock value half are analogous. The program may contain a conditional branch instruction. This can be implemented in the timed automata by having a state with two transitions: one for branch-taken and another for branch-not-taken. For these transitions, there are guards. If the condition for the branch in the program is $c = 4$, then the guard for the corresponding transition in the timed automata is 2^{1-4}. The paper also shows that the undecidability result for the three conditions is true even if the task set is such that only a single preemption can occur.

Fersman and Wang [C-16] extend the approach in Fersman et al. [C-14] to deal with precedence constraints (i.e., thread A must finish before thread B executes because thread A produces data that thread B needs) and resource constraints (e.g., thread A and thread B share a data structure, and operations on this data structure should not be done concurrently). Note that this paper assumes that the execution time of a job is constant.

Guan et al. [C-17] do not consider timed automata but present a schedulability analysis with an expressive model; it achieves this analysis using abstraction and refinement. The idea is as follows: Each task is described with a number of concrete behaviors. There is an operation that can form an abstraction of two concrete behaviors. To understand this merging, consider the following example: A task 1 has a behavior 1 that in a time interval of duration 10, there can be at most 5 units of execution, and in a time interval of duration 20, there can be at most 15 units of execution. Consider that task 1 also has another behavior (behavior 2) stating that in a time interval of duration 10, there can be at most 6 units of execution, and in a time interval of duration 20, there can be at most 14 units of execution. Then an abstraction of these two behaviors considers the worst case from each behavior; the abstraction states that in a time interval of duration 10, there can be at most 6 units of execution, and in a time interval of duration 20, there can be at most 15 units of execution. In this abstracted behavior, the worst case has been taken for each concrete behavior. It is shown that two concrete behaviors can be described (through an over-approximation) with an abstract behavior. However, two abstract behaviors can also be described with a single abstract behavior. In this way, one abstract behavior can be obtained for each task. This allows for performing schedulability analysis rapidly.

This comes at the cost of pessimism, however; it may be that the task set was actually schedulable, but the schedulability test could not guarantee that. Guan et al. [C-17] describe an approach to reduce pessimism through refinement as follows: Initially, all tasks have a single behavior; if there are *n* tasks, when there is a queue with a single element, this single element is a vector with *n* behaviors. The schedulability analysis is run as described above. If it fails, then the following is done: Choose one task (task *i*); its behavior is an abstract behavior that was formed from two other (potentially abstract) behaviors. The two behaviors offer refinements of the abstract behaviors. Remove the head element in the queue, and form two elements that are added to the queue. Each of these two elements are vectors; a vector differs only in that for one of the elements of the vector, the refined behavior is used. With this new queue, choose the head element and check schedulability. If it cannot guarantee schedulability, then perform refinement again.

In [C-18], Fersman et al. repeat results from previous papers. They state that if the execution time of a job is fixed, job arrivals may be data-driven, the computer has a single processor, and job arrivals are described with timed automata, then the schedulability analysis problem is decidable. As already mentioned in a previous paper, when execution times are given in a range (best-case execution time, WCETs), then the schedulability analysis problem is undecidable. For this reason, this paper presents an over-approximation technique for this case. The idea is that the actual execution time of a job *J* is used in two ways: to determine when the job may finish (this execution time is used as a guard), and to update a lower- and upper-bound variable for lower priority jobs. The update can be performed in a pessimistic way; the lower bound can be updated by a lower bound on the execution of the higher priority task and, analogously, the upper bound can be updated by an upper bound on the execution time of the higher priority task.

In [C-19], Pavel et al. extend a previous paper [C-15] to multiprocessors. They consider a set of tasks in which each task is assigned to a processor (i.e., no migration) and assume that there is a timed automata such that for an edge of the timed automata there is a label indicating the task that arrives when this edge fires. There is also a scheduling automaton such that some guards fire when a task arrives. For the case that the finishing of a task cannot influence the arrival of other tasks, the paper shows that the schedulability analysis problem is undecidable (note that from the perspective of decidability, this makes multiprocessor analysis different from single-processor analysis).

C.1.5 WORST-CASE EXECUTION TIME ANALYSIS

The schedulability analysis developed for real-time systems takes as an input the WCET of the function that is executed periodically by the task. More specifically, WCET analysis is the problem of finding an upper bound on the execution time of a function when the function executes by itself on a processor. Researchers started studying this problem in the 1980s [C-20], and it has now received significant attention. See [C-21] for a good summary. In general, there are two approaches to obtain the WCET of a task: 1) a measurement-based approach, and 2) a model-based approach. Measurement-based approaches are first discussed, then model-based approaches.

C.1.5.1 Measurement-Based WCET Analysis

Existing measurement-based WCET analysis combines measurements with some type of model that can include statistical models or execution path models.

C.1.5.1.1 Statistical-Based WCET Estimation and Validation

Hansen et al. [C-22] present an approach in which execution time measurements are used to create a statistical model based on extreme value theory (EVT) [C-23] to 1) predict the probability of exceeding a specific WCET that could be matched to the application requirements, 2) obtain WCET beyond the measurements used for a given target probability of exceeding this WCET, and 3) validate that the measurement data match a specific statistical model (probability distribution).

The authors in [C-22] use EVT to avoid the large amount of data required to obtain good estimates of the tail behavior (worst case) when using traditional methods and allow them to generate an estimate higher than the highest observed value. EVT models data as a random variable $Y = \max X_1, \ldots, X_n$ and uses the generalized extreme value distribution (that converges as n approaches infinity) to obtain the desired worst case. More specifically, this distribution converges to any of three forms: *Gumbel*, when the underlying distribution is not heavy tailed (like the normal distribution); *Frechet*, when the underlying distribution is heavy tailed; and *Weibull*, when the distribution has a bounded upper tail (like the uniform). To use this approach, the Y distribution needs to be constructed by dividing the data into sample blocks (considered to be iid—independent and identically distributed) of size b that match each of the X_i variables. Then, the maximum of each block is obtained, and the best-fit parameters of the selected distribution are calculated, followed by a fitness test. Once the model is constructed, the WCET can be estimated for a particular probability.

In [C-22], only the Gumbel distribution is explored. This distribution has two parameters: a location parameter μ, and a scale parameter β. These parameters are then used to build the cumulative distribution function:

$$F_G(y) = e^{-e^{\frac{y-\mu}{\beta}}} \tag{C-9}$$

This function can then be used to compute the percent-point function:

$$F_G^{-1}(q) = \mu - \beta \log(-\log(q)) \tag{C-10}$$

That is used to compute the WCET.

The estimation of the parameter for the Gumbel distribution involves applying linear regression to the quantile plot (QQ-plot [C-23]) of the data against that of the distribution. For the Gumbel distribution, this takes the form of a line with a positive slope that intersects the *y*-axis on the positive side. From this line, μ can be obtained from the y intersect and β from the slope of the line.

Once μ and β are obtained, a chi-square test is conducted to verify the fitness of the distribution to the data. From the results of this test, there may be a need to restructure the sample data to create sample blocks of different sizes that can be tested again.

Once the properly fit distribution is obtained, it is possible to calculate the WCET for a particular probability of exceeding this WCET p_e as follows. First, compute the probability q that all b samples in a block are less than the WCET:

$$q = (1 - p_e)^b \tag{C-11}$$

Then use equations C-10 and C-11 to calculate the WCET ω:

$$\omega = \mu - \beta \log(-\log(1 - p_e)^b)) \tag{C-12}$$

<u>C.1.5.1.1.1 Limitations</u>

This approach can provide WCET estimates with a strong theoretical support for smaller measurement samples than non-EVT distributions and theory. However, no variable execution paths are considered. As a result, the approach needs to be complemented with execution path exploration if the software exhibits multiple paths.

<u>C.1.5.1.2 Rapita RapiTime</u>

Rapita RapiTime [C-24] is a tool to automate the measurement of WCET of embedded software. The tool automatically instruments the code to obtain execution time measurement of blocks of code and builds a structural model to relate those blocks to the general structure of software and the different execution paths. This structure is also used to predict the worst-case path that the software can exhibit and derive a WCET based on it and the measured execution time blocks.

RapiTime allows the addition of annotations to code to eliminate infeasible paths that can make the WCET estimate pessimistic. RapiTime also identifies the hot spots where execution time improvements can be more significant.

The tool is built with DO-178B in mind. DO-178B requires the qualification of verification tools like this one. Therefore, the company that commercializes this tool provides the qualification data to satisfy this requirement.

<u>C.1.5.1.3 Limitations</u>

RapiTime does not offer any probabilistic characterization of the WCET obtained. As a result, the obtained WCET is as good as the data used to obtain it. Depending on how many samples the user collects, the WCET can be more conservative or not.

C.1.5.2 Model-Based WCET Analysis

C.1.5.2.1 Abstract Interpretation

This section starts with some history of WCET analysis. Then it discusses abstract interpretation [C-25] and AiT [C-26] because these are the most popular methods and tools.

Shaw's work [C-27] served as a catalyst for the research community to begin studying WCET analysis. It presented a framework for reasoning about time that involved two specifications: 1) timing of individual lines of code and how the time required for executing multiple lines of code can be derived from single lines of code and 2) the arrival times and deadlines of programs. The former is relevant for WCET. The method can be understood with an example. Consider the following program:

```
if a>=7 then
 b := 7
 flag := 1
 else
 b := 0
 end if
```

The WCET of this program can be computed as follows:

```
 WCET of program =
 WCET of "if statement" +
 WCET of condition "a>=7" +
 max( WCET of "b:=7" + WCET of "flag := 1",
 WCET of "b:=0"
 )
```

Shaw [C-27] presents rules (based on the grammar of the language) to compute a WCET of a program based on its constituent parts. This method has two drawbacks. First, it assumes that the executable code directly maps to the source code. In some cases, this is not true; for example, a compiler may perform dead code elimination or move a computation out of a for-loop. Second, it describes the execution time of an individual part with an upper bound and a lower bound, and these bounds do not depend on other parts. However, modern processors today use caches, so the time to perform a load instruction depends on previous memory operations. For these reasons, the method in [C-27] is not used today.

Researchers realized that the hardware needs to be modeled. One approach was to use abstract interpretation; it was initially used for cache analysis. Abstract interpretation [C-25] is a general technique for proving properties of programs. The idea is as follows: A program consists of instructions, and each instruction performs a mapping of a before-instruction state to an after-instruction state. Describing these mappings for each instruction is cumbersome and would lead to time-consuming analysis. Therefore, abstract interpretation introduces the notion of abstraction. It can be best understood through an example. If there is an instruction "*z* := *x* * *y*," where *x*, *y*, and *z* are 32-bit integers, then every combination of *x* and *y* can be enumerated, and a table can express the value of *z*. However, later in a program, the exact value of *z* is not a concern—the only concern is whether it is negative, positive, or zero (for example, *z* may be used later in a conditional branch). In abstract interpretation, abstract values are used; values are represented with their abstraction, and instructions expressed on their abstractions as well. For example, multiplication can be expressed as follows:

```
if x is zero then z is zero.
if y is zero then z is zero.
if x is negative and y is negative, then z is positive.
if x is negative and y is positive, then z is negative.
if x is negative and y is don’t know, then z is don’t know
if x is positive and y is negative, then z is negative.
if x is positive and y is positive, then z is positive.
if x is positive and y is don’t know, then z is don’t know
if x is don’t know and y is negative, then z is don’t know.
if x is don’t know and y is positive, then z is don’t know.
```

Then, in a program, consider each basic block (a sequence of instructions with a single entry point and a single exit point), and describe the computation performed by this basic block for how it produces a next state (described with the abstractions) from its previous state (described with the abstractions). With this description, facts of a program can be derived by applying these operations iteratively, considering that the output for certain basic blocks is the input to other basic blocks, and considering that two abstractions can be merged. When two states are different, merging may result in a state of “don’t know.” Abstract interpretation is typically quite fast for program analysis, but it suffers from potential loss of precision; it may be that some facts of a program are true, but the method cannot prove them.

Typically, abstract interpretation was used for proving facts about the computed values of programs (e.g., to prove that after a certain basic block has finished execution, a certain variable has taken a certain value or is within a certain range). However, abstract interpretation can also be applied to other types of states, such as which cache blocks are in the cache memory after a basic block. Indeed, Alt et al. [C-28] describe how a memory instruction changes the state of the cache, and this can be used to classify memory instructions. For example, a memory instruction can be classified as “always hit,” meaning that whenever it executes, it results in a cache hit. Another category is “always miss,” meaning that whenever it executes, it results in a cache miss. Yet another category is “not classified,” which means that the analysis could not categorize the memory access. With such an analysis, other types of analyses can obtain more accurate bounds on the execution time of a basic block. Today, abstract interpretation is used in the tool AiT—one of the most popular WCET tools today [C-26].

C.1.5.2.2 Symbolic Execution

Another method to obtain the WCET is the use of symbolic execution that explores the micro-architectural features of the processor in which the program runs. An example of this approach is presented in [C-29]. In this paper, the authors develop a symbolic execution framework to explore the effect of cache in the WCET of a program that can easily be extended to other features, such as pipelines. In this paper, the authors model the execution of the program as a labeled transition system S as a tuple $(\mathcal{L}, \ell_0, \longrightarrow)$, where $\mathcal{L}$ is the set of program points with $\ell_0 \in \mathcal{L}$ as the unique initial program point, and $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times Ops$, where Ops is the set of operations of the program. These operations are restricted to be either assignments or assumptions. Assignments assign values to the variables of the system, and assumptions specify conditions that must be true to allow the execution to continue in this path. This transition system builds a directed acyclic graph (DAG) that is a simplification of a full-control flow graph.

Based on the transition system previously defined, a symbolic state is defined as a tuple (ℓ, c, σ, Π), where $\ell \in \mathcal{L}$ is the concrete program point of this state, c is the abstract cache state, σ is the current valuation of all the variables of the program, and Π is a first-order logical formula that encodes the conditions that must be true for the execution of the program to reach this point. With these definitions, as the program executes and new branches are created into the execution DAG, loops are unrolled, and infeasible paths are discovered, the cache state is also updated (as memory locations loaded/unloaded into cache). At the same time, summarizations of the branches in terms of conditions that encode infeasible paths and cache states are created to be reused in other branches. These summarizations are the core of a scalable analysis; they allow for determining that a branch analysis can be assumed to be dominated by another branch (e.g., has a "less"-worse execution time).

This approach first uses both the transition system representation of the program and the summarization to evaluate the worst-case timing of a branch of execution given a cache state. Then, it reuses the result in other branches if it can be proven that the first dominates the second, (i.e., that it still has the same infeasible paths and the same or worst cache state).

This framework was implemented in the LLVM IR framework with the Z3 SMT engine as the constraint solver. The paper presents a number of experiments over some benchmarks. Whereas this approach provides more precise WCET figures, the complexity of the approach remains super-linear with respect to the complexity of space and time.

Other work has explored similar approaches. For example, Banerjee et al. [C-30] use abstract interpretation and SAT solvers to explore infeasible paths and architectural features in the calculation of WCET.

C.1.5.2.2.1 Limitations

As mentioned before, this approach suffers from superlinear complexity in space and time. Moreover, the algorithm in this case considers only cache, and adding other architectural features might increase the verification time.

C.2 LOGICAL VERIFICATION

In this section, several verification technologies relevant to assuring logical correctness of systems that use virtualization techniques are discussed in more detail. Logical correctness is primarily concerned with ensuring that the software does the right thing (i.e., produces the expected output). All logical verification techniques require the user to specify the initial state of the software, the inputs, and the expected outputs. They differ in the specific way these artifacts are expressed and checked.

C.2.1 TESTING

Software testing is by far the most widely used logical verification technique in practice. Whereas a full treatment of testing is beyond the scope of this article, it will be described briefly. The basic idea is to express the initial condition, inputs, and outputs of the target software as a "test case." The testing tool then executes the software, starting from the specified initial state, and stimulates it with the specified inputs. Finally, it checks whether the outputs from the software match the ones expected in the test case. If they do, then the test passes. Otherwise, the test fails. The main advantage of testing is that it can be highly automated and run routinely. Various languages and software development environments include some form of testing infrastructure (e.g., JUnit).

One limitation of testing is its lack of coverage. Because software (including avionics software) has enormous state spaces, even a very large number (e.g., tens of millions) of test cases exercise only a minute fraction of their possible executions. This lack of coverage manifests in several ways. For example, many bugs are found during "integration testing" when multiple components are combined and then tested. This happens even when the individual components have passed unit tests. Second, bugs are detected during field trials despite the software having been tested heavily. Often, these are due to sequences of interactions with the physical environment and human operators that were not covered by testing. Whereas a number of criteria (such as MC/DC proposed by the FAA) have been developed to achieve higher syntactic coverage of software via testing, the problem of achieving adequate semantic coverage remains open.

Another limitation of testing is that it is biased toward the implementation side of the software engineering "V" process. These limitations of testing would also be applicable to software with virtualization machines. The state-space explosion problem would be exacerbated by the high degree of concurrency and interaction between the various VMs. It is recognized that defects found earlier are easier to fix and therefore reduce the overall software development cost. In the rest of this section, more exhaustive logical verification techniques will be described that not only provide higher coverage but also are better suited to the earlier (design and modeling) phases of software development.

C.2.2 THEOREM PROVING AND DEDUCTIVE VERIFICATION

The ideas behind applying theorem proving to verify logical correctness of software were developed in the early 1980s by Floyd and Hoare. The verification problem is stated in the form of a Hoare triple. Formally, a Hoare triple is of the form $\{p\}\, S\, \{q\}$, where S is a program (e.g., a sequence of statements, a loop, a function call) and p and q are conditions (they are known respectively as the pre-condition and the post-condition) expressed over the variables appearing in S. The triple is valid if whenever S is executed in a state that satisfies p, it terminates in a state that satisfies q. For example, consider the Hoare triple $\{x \geq 0\}\, y \coloneqq x\, \{x = y \wedge y \geq 0\}$. This triple is valid because if the assignment statement $y \coloneqq x$ is executed from any state that satisfies $x \geq 0$, the resulting state must satisfy the condition $x = y \wedge y \geq 0$ (note that $\wedge$ denotes logical conjunction). Similarly, consider the Hoare triple $\{x \geq 0\}\, y \coloneqq x - 1\, \{y \geq 0\}$. This triple is invalid because if the assignment $y := x - 1$ is executed from a state that satisfies $x \geq 0$ (in particular, suppose $x = 0$), then the resulting state may not satisfy $y \geq 0$ (in this case $y = -1$).

Once a Hoare triple is specified, its validity is checked by constructing a logical formula known as a verification condition (VC) and proving that the VC is valid. To verify complex programs with loops and function calls using this approach, Hoare triples for all functions, and loop invariants for all loops must be specified. Informally, the loop invariant for a loop is a condition that holds whenever an execution of the program is about to enter the body of the loop. Several tools automatically construct VCs from a program that is sufficiently annotated with pre- and post-conditions for functions and loop invariants and prove the validity of these VCs using off-the-shelf satisfiability modulo theory (SMT) solvers. In particular, the Frama-C [C-31] tools verify C programs with ACSL annotations in this way. For example, the C program shown in figure C-4 can be verified automatically by Frama-C.

```
/*@requires x == 0; @modified \nothing; @ensures \return == 10;*/
int foo (int x)
{
  int i = x;
  /*@loop invariant 0<=i<=10; @loop assigns i;*/
  while (i < 10) i++;
  return i;
}
```

Figure C-4. Example C code with ACSL annotations

Note that the function's pre-condition is $x = 0$, and its post-condition is that the return value is 10. The loop invariant is that the value of i must be between 0 and 10, inclusively. Other such deductive software verifiers include Boogie, Why3, and ESC-Java. Whereas Boogie and Why3 have their own target programming languages, the ESC-Java tool verifies Java programs. Deductive verifiers are extremely powerful in that complex safety properties about a program's behavior can be specified via pre- and post-conditions, but they are manually intensive. However, the cost is usually justified for verifying long-running system software (such as operating systems [C-32]) that are critical to get right and do not change frequently. For example, Frama-C has been used to verify avionics software [C-33]. Whereas there has been some effort to apply deductive verifiers to hypervisors [C-25], more work needs to be done. In particular, better hardware models need to be constructed so that hypervisors can be verified across a broader range of platforms.

C.2.3 ABSTRACT INTERPRETATION

Abstract interpretation [C-34] is another exhaustive verification technique for proving logical correctness of programs. The idea behind abstract interpretation is to perform reachability analysis over an abstract model of the program's semantics constructed via an abstract domain. The model is conservative in that if no unsafe state is reachable in the model's abstract state space, then no unsafe state is reachable in the program's concrete state space. However, the model is imprecise and can lead to false warnings. In other words, there may be a path to an unsafe state in the model, but no corresponding execution in the actual program. Consider again the example C program from figure C-4. The control flow graph (CFG) of that program is shown in figure C-5 (a).

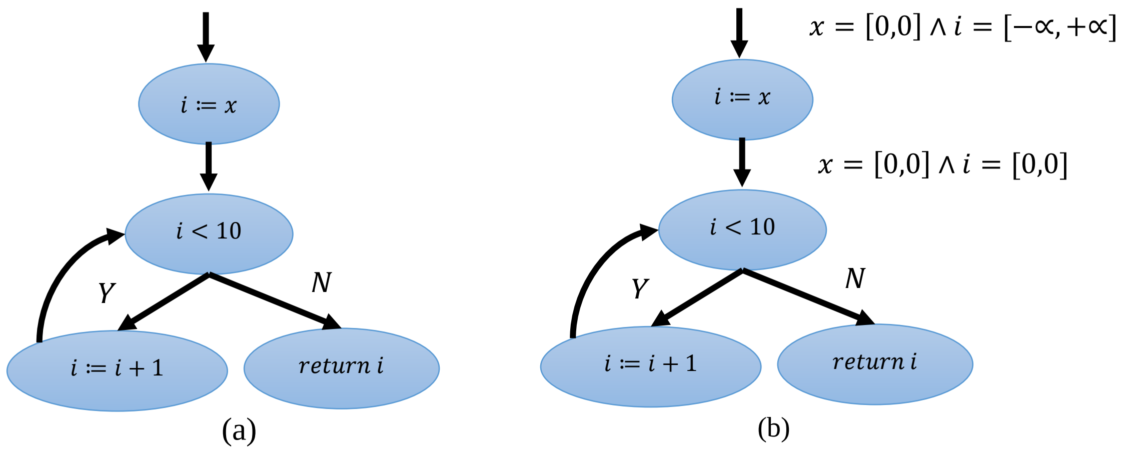
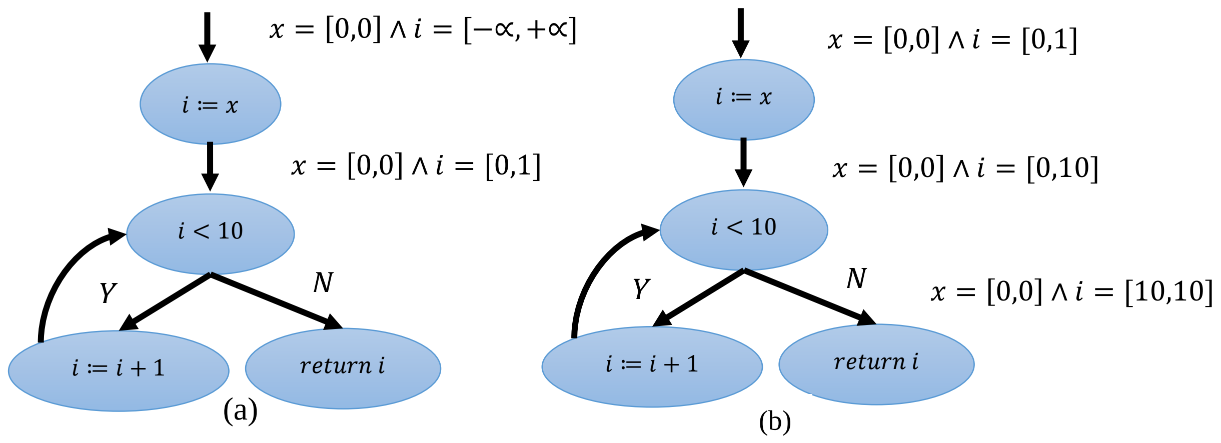


Figure C-5. Control flow graph and results of abstract interpretation

Informally, the CFG is a directed graph in which nodes represent program statements and edges represent flow of control from one statement to another. Suppose an abstract interpretation of this program was performed using the interval abstract domain [C-35]. This means that each abstract state will assign only an interval of possible values to the program variables (i and x). Note that each interval must over-approximate all possible values of the corresponding variable in concrete executions. Therefore, at the entry node of the CFG, the interval is $x = [0,0] \wedge i = [-\propto, +\propto]$. This means that although the value of x must be 0, the value of i is unconstrained. Assume here that i is a mathematical integer, not a machine integer. This is for simplicity of presentation and is not an inherent limitation of abstract interpretation.

Next, reachability is performed over the abstract state space, using the semantics of the program statements to produce next states from previous ones. Therefore, after the initial assignment $i \coloneqq x$, there is a new abstract state $x = [0,0] \wedge i = [0,0]$. After the assignment, the value of i is no longer unconstrained. Figure C-5 (b) shows the state of the abstract interpretation at this stage. At the conditional statement $i < 10$, the Y branch must be followed because the condition holds under the abstract state. Now the assignment $i \coloneqq i + 1$ is performed to get the abstract state $x = [0,0] \wedge i = [1,1]$. When the control-flow edge is followed to return to the conditional statement, the previous abstract state $x = [0,0] \wedge i = [0,0]$ must be "merged" with the new one to get an updated abstract state $x = [0,0] \wedge i = [0,1]$. Figure C-6 (a) shows the state of the abstract interpretation at this stage.



$x = [0,0] \wedge i = [-\propto, +\propto]$
$i := x$
$x = [0,0] \wedge i = [0,1]$
$i < 10$
Y
N
$i := i + 1$
$return\ i$
(a)
$x = [0,0] \wedge i = [0,1]$
$i := x$
$x = [0,0] \wedge i = [0,10]$
$i < 10$
Y
N
$x = [0,0] \wedge i = [10,10]$
$i := i + 1$
$return\ i$
(b)

Figure C-6. Progress of abstract interpretation

If this process is repeated by going through the loop 10 times, the abstract state $x = [0,0] \wedge i = [0,10]$ will be computed at the conditional statement. At this point, the N branch will be followed and the abstract state $x = [0,0] \wedge i = [10,10]$ obtained at the $return$ statement, which suffices to prove that the function returns 10.

The advantage of abstract interpretation is its scalability and efficiency. Most commercial static analysis tools, such as Coverity and Klocwork, perform abstract interpretation in one way or another. The scalability comes from the fact that the abstract domain can focus only on the variables of interest—and relationships between them—and eliminate all other program details. Moreover, a technique known as "widening" can be used to analyze loops that have large bounds without having to unwind them completely (as in the example). On the flip side, scalability comes at the cost of precision. Therefore, static analysis tools are plagued by false warnings, which limit their effectiveness. In practice, domain-specific abstract domains and heuristics are used to reduce the number of false warnings or weed them out. However, this limits practical applicability to multiple domains and increases manual cost. Nevertheless, abstract interpretation remains the most widely used formal verification technology for logical correctness over industrial code bases.

There are two potential ways in which abstract interpretation could be used to verify avionics software with VMs. First, it can be applied through an off-the-shelf static analyzer. Static analyzers can help find common programming errors, such as numeric overflows and buffer overflows, which plague all types of software. However, they do not provide any benefits that are specific to this type of software. Second, new abstract domains can be developed targeted toward low-level software involving concurrency and use of special hardware features. This is more challenging, and prior work on abstract interpretation of concurrent low-level programs has yielded limited benefits.

C.2.4 MODEL CHECKING

Model checking [C-36] is an algorithmic and automated technique for proving correctness of finite state models of systems against properties specified using temporal logic. The model is given as a Kripke structure, which is a finite state machine for which states are labeled with atomic propositions that denote facts that are true in that state. For example, figure C-7 shows the model of a microwave oven. There are four atomic propositions: $start$ means that the oven is turned on; $close$ means that the oven door is closed; $heat$ means that the food is being cooked; and $error$ means that some error condition has occurred because of incorrect operation.

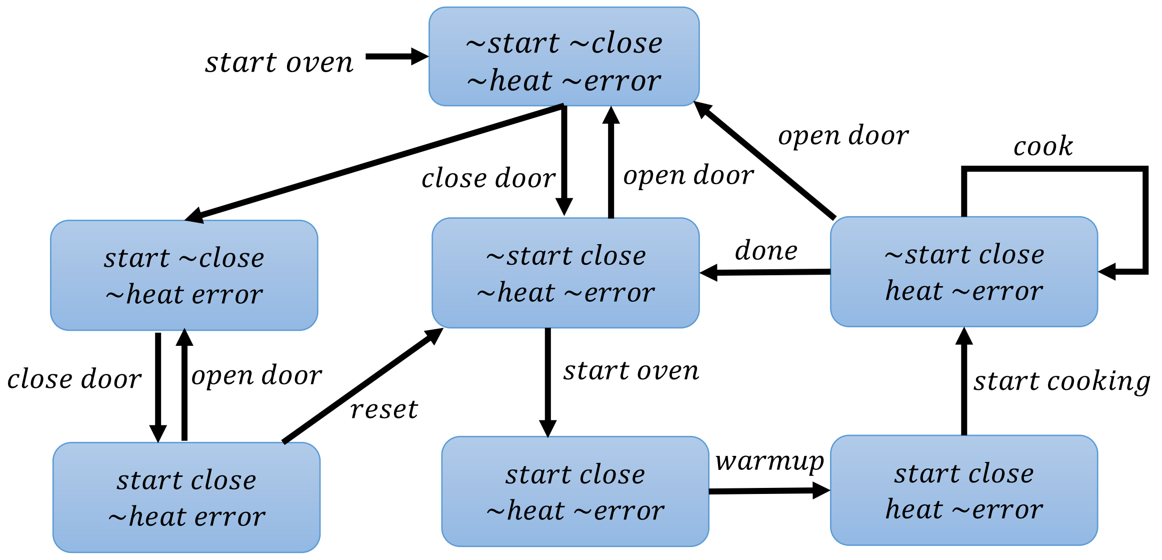


Figure C-7. A Kripke structure model of a microwave oven

The target properties are expressed using temporal logic. Here are three example properties expressed in computation tree logic:

- $AG(heat \Rightarrow close)$: If the oven is cooking the food, then the door is closed.
- $AG(start \Rightarrow AF\ heat)$: Whenever the oven is turned on, eventually it will cook the food.
- $AG((start \wedge {\sim}error) \Rightarrow AF\ heat)$: Whenever the oven is turned on properly, eventually it will cook the food.

Model checking involves an exhaustive exploration of the state space of the Kripke structure to prove the target property or find a counterexample (i.e., an execution of the model) that shows why the property does not hold. In the example, the first and third properties are valid, whereas the second one does not hold. The counterexample to the second property is an execution in which the microwave is turned on without closing the door first.

Model checking has been widely used to verify properties of concurrent systems, including software and hardware. The big challenge with model checking is with state-space explosion, and most of the advances in this area have been devoted to addressing this problem. The first prototype model checker was explicit state and could only verify systems with 10,000 or so states. Modern model checkers, such as NuSMV and SPIN, use a host of techniques, including symbolic analysis using binary decision diagrams and partial order reduction, and can verify systems with over 10^{20} states [C-37]. The development of more powerful computers has also improved the ability of model checkers to handle larger and more complex systems. A variant of model checking, known as bounded model checking [C-38], uses powerful propositional satisfiability (SAT) solvers to verify properties over an unwinding of a model's state space up to some specific bound. Whereas in general it does not perform complete verification, it has been extremely effective for finding bugs, which often have short counterexamples.

However, model checking is limited to verifying finite state systems, which is insufficient for programs that typically have infinite state spaces. The applicability of model checking to avionics software with virtualization is therefore most likely possible through a generalization of model checking to software, which is described next.

C.2.5 SOFTWARE MODEL CHECKING

The idea behind software model checking [C-39] is to use a counterexample-guided abstraction-refinement approach. This is an iterative procedure with the following steps:

1. Create a finite abstract model M of the program by, for example, using predicate abstraction.
2. Verify M using model checking.
3. If M satisfies the target property, then so does the original program because the abstraction procedure is conservative. In this case, the software model checker terminates by declaring the program to be verified.
4. Otherwise, let CE be the counterexample returned by the model checker in step 2.
5. Check whether CE corresponds to a concrete execution of the program. If so, then a counterexample has been found. The software model checker terminates and reports this counterexample as diagnostic feedback.
6. Otherwise, CE is a spurious counterexample. Use it to create a more refined model M and repeat from step 2.

This approach has been implemented in a number of commercial and academic tools. For example, Microsoft's® static driver verifier essentially uses this technique to prove that device drivers interact correctly with the Windows kernel. Software model checking has seen a lot of growth in recent years, and tools regularly compete in an annual competition [C-40]. Starting with the verification of safety properties of sequential C programs, it now includes concurrent programs with pointers, object-oriented programs, and verification of more complex properties, such as termination. Bounded model checkers for software [C-41] are also available.

However, most software model checkers are still targeted toward source code. To verify low-level software with virtualization, an appropriate hardware model must be developed and the model checker extended to reason about the hardware. In particular, model checkers will be required to reason about assembly code that is often a part of system software, such as hypervisors.

In this section, various verification techniques for logical verification are summarized. Whereas considerable progress has been made, application of these techniques to avionics software with virtualization remains an open challenge. Developing an appropriate hardware model that includes the semantics of virtualization primitives will be crucial. In addition, given the complexity of these systems, it is unlikely that any single verification technique will be able to handle an entire system. A compositional approach that combines multiple techniques in a sound manner will be necessary for success. The presentation of the verification evidence produced by various tools in a connected and traceable manner (from high-level requirements to low-level tool outputs), in the spirit of assurance and safety cases, will aid certification.

C.3 CONCLUSION

Verification technologies are key for the safe use of virtual machines (VMs) in avionics systems. Unfortunately, the state of the art is not well developed. Three broad challenges are faced. First, verification technologies need to be expressive; they must describe a system so that the description is sufficiently close to reality. Second, verification techniques need to run fast. Third, verification techniques need to be able to deal with the specifics of VMs. All of these verification techniques are now well understood. In general, there is a tradeoff between the speed of verification and the level of detail of the model on which the verification depends. One potentially useful idea for achieving greater fidelity of the models and sufficient speed in verification is the use of abstractions. This is true for both timing and logical verification.

The main motivation for the use of VMs in avionic systems is the possibility of isolating different parts of the system in VMs. However, appropriate technology is needed to verify these parts independently and compose the results (compositional verification). More specifically, the capacity to tolerate modification to one VM without requiring a reverification of the whole system is key and aligned to the use of partitions in standards like DO-178B/C. More research is needed to obtain practical compositional verification technologies for both the timing and logical aspects. The recommendations for certification of engineers will be deferred until the compositional verification techniques are discussed in the next section.

C.4 REFERENCES

C-1. Cruz, R.L. (1991). A calculus for network delay, Part I: Network elements in isolation. *IEEE Trans. Information Theory, 37*(1), 114–131.

C-2. Cruz, R.L. (1991). A calculus for network delay, Part II: Network elements in isolation. *IEEE Trans. Information Theory, 37*(1), 132–141.

C-3. Thiele, L., Chakraborty, S., Naedele, M. (2000). Real-time calculus for scheduling hard real-time systems. Proceedings from the 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353). Geneva, Switzerland.

C-4. Joseph, M., Pandya, P.K. (1986). Finding Response Times in a Real-Time System. *The Computer Journal, 29*(5), 390–395.

C-5. RTC Toolbox Tutorial. http://www.mpa.ethz.ch/static/Tutorial.html

C-6. https://rtg.cis.upenn.edu/carts/index.php

C-7. Alur, R., Dill, D.L. (1994). A Theory of Timed Automata. *Theoretical Computer Science, 126*(2), 183–235.

C-8. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S. (1992). Symbolic Model Checking for Real-time Systems. *Inf. Comput., 111*(2), 193–244.

C-9. http://www.uppaal.org/

C-10. Norström, C., Wall, A., Yi, W. (1999). Timed Automata as Task Models for Event-Driven Systems. *RTCSA.*

C-11. Fersman, E., Pettersson, P., Yi, W. (2002). Timed Automata with Asynchronous Processes: Schedulability and Decidability. *TACAS: 8th International Conference,* Budapest, Hungary.

C-12. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W. (2002). TIMES - A Tool for Modelling and Implementation of Embedded Systems. *TACAS: 8th International Conference,* Budapest, Hungary.

C-13. Amnell T., Fersman E., Mokrushin L., Pettersson P., Yi W. (2004) TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In: Larsen K.G., Niebert P. (eds) *Formal Modeling and Analysis of Timed Systems*. FORMATS 2003. Lecture Notes in Computer Science, vol 2791. Springer, Berlin, Heidelberg.

C-14. Fersman E., Mokrushin L., Pettersson P., Yi W. (2003) Schedulability Analysis Using Two Clocks. In: Garavel H., Hatcliff J. (eds) *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2003. Lecture Notes in Computer Science, vol 2619. Springer, Berlin, Heidelberg.

C-15. Krčál P., Yi W. (2004) Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In: Jensen K., Podelski A. (eds) *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2004. Lecture Notes in Computer Science, vol 2988. Springer, Berlin, Heidelberg.

C-16. Fersman, E., Yi, W. (2004). A Generic Approach to Schedulability Analysis of Real-Time Tasks. *Nordic Journal of Computing, 11*(2), 129–147.

C-17. Guan N., Tang Y., Abdullah J., Stigge M., Yi W. (2015) Scalable Timing Analysis with Refinement. In: Baier C., Tinelli C. (eds) *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2015. Lecture Notes in Computer Science, vol 9035. Springer, Berlin, Heidelberg.

C-18. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W. (2006). Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science, 354*(2), 301–317.

C-19. Krcal P., Stigge M., Yi W. (2007) Multi-processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times. In: Raskin JF., Thiagarajan P.S. (eds) Formal Modeling and Analysis of Timed Systems. FORMATS 2007. Lecture Notes in Computer Science, vol 4763. Springer, Berlin, Heidelberg.

C-20. Shaw, A.C. (1989). Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering, 15*(7), 875–889.

C-21. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, … Stenström, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems, 7*(3), 36:1–36:53.

C-22. Hansen, J.P., Hissam, S.A., & Moreno, G.A. (2009). Statistical-Based WCET Estimation and Validation. *WCET: 9th Intl. Workshop on Worst-Case Execution Time Analysis,* Dublin, Ireland, July 1–3.

C-23. Beirlant, J., Goegebeur, Y., Segers, J., and Teugels, J. (2004). *Statistics of Extremes: Theory and Applications.* Hoboken, New Jersey: Wiley Press.

C-24. Rapita Systems Ltd. (n.d.) "Automating WCET Analysis for DO-178B/C." White Paper. Retrieved from https://www.rapitasystems.com/downloads/automating-wcet-analysis-do-178bc.

C-25. Cousot, P., Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *POPL*. 238–252.

C-26. https://www.absint.com/ait/

C-27. Shaw, A.C. (1989). Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering, 15*(7), 875–889.

C-28. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R. (1996). Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program., 35*, 163–189.

C-29. Chu, D., Jaffar, J., & Maghareh, R. (2016). Precise Cache Timing Analysis via Symbolic Execution. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* 1–12.

C-30. Banerjee, A., Chattopadhyay, S., & Roychoudhury, A. (2013). Precise micro-architectural modeling for WCET analysis via AI+SAT. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 87–96.

C-31. Frama-C Tool website. http://frama-c.com/

C-32. Moskal M. (2012) From C to Infinity and Back: Unbounded Auto-active Verification with VCC. In: Madhusudan P., Seshia S.A. (eds) *Computer Aided Verification*. CAV 2012. Lecture Notes in Computer Science, vol 7358. Springer, Berlin, Heidelberg.

C-33. Vasudevan, A., Chaki, S., Jia, L., McCune, J.M., Newsome, J., Datta, A. (2013). Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. *2013 IEEE Symposium on Security and Privacy*, 430–444.

C-34. Cousot, P., Cousot, R. (1976). *Static determination of dynamic properties of programs*. Proceedings of the 2nd International Symposium on Programming, Paris, France. Dunod. 106–130.

C-35. Clarke, E.M., & Emerson, E.A. (1981). Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs*. 52–71.

C-36. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., & Hwang, L.J. (1992). Symbolic Model Checking: 10^20 States and Beyond. *Information and Computation*, *98*(2): 142–170.

C-37. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y. (2003). Bounded model checking. *Advances in Computers 58*, 117–148.

C-38. Jhala, R., Majumdar, R. (2009). Software model checking. *ACM Comput. Surv., 41*(4), 21:1-21:54.

C-39. Competition on software verification (SVCOMP). https://sv-comp.sosy-lab.org/2016/

C-40. CBMC model checker. http://www.cprover.org/cbmc/

APPENDIX D—COMPOSITIONAL VERIFICATION FOR VIRTUAL MACHINES

D.1 INTRODUCTION

Virtualization offers the possibility of isolating components and potentially verifying them independently from one another. Certification standards, such as DO-178C, require an isolation to allow the verification of an individual component without the need to recertify the rest. However, either because of their need to interact with each other or because they use the same hardware, the isolation among components offered by virtualization is not absolute. This means that the techniques used for verification must be aware of these interactions across components. The research community has identified this type of verification as compositional verification.

Compositional verification defines components whose behavior can be affected by other components and can also affect other components, only through an interface. With this definition, the verification process is decomposed into two parts. First, components are verified taking into account the interactions with other components observable through the interface. Second, the whole system is verified as a collection of components connected through their interfaces, whose behavior is limited to what is observable through the interface. Clearly, these interfaces are influenced by both the verification technology and the mechanisms that restrict the behavior, such as a virtual machine (VM) and its hypervisor.

In this report, the compositional technologies that support the use of different types of virtualization to enable independent component verification will be studied, both from the timing and logical perspectives. Section 2 discusses virtualization interfaces from both the perspective of timing and functional correctness. Because timing is changed as a result of virtualization, timing is emphasized in the discussion. Section 3 presents mitigation strategies. Section 4 gives recommendations.

D.2 VIRTUALIZATION INTERFACES: A VERIFICATION PERSPECTIVE

Different virtualization techniques offer different interfaces that can be mapped to different compositional verification techniques. In the following, this mapping is studied and the tradeoffs that they offer are discussed.

D.2.1 VIRTUALIZATION TIMING INTERFACES CHARACTERIZATION AND MODELING

In this section, the focus is on temporal virtualization, the different ways to define the timing interfaces they provide, and the corresponding compositional verification technology.

D.2.1.1 Task Model

To set up the discussion, an initial model of the tasks that can be hosted in a VM will be provided. Specifically a task τ_i is defined as:

$$\tau_i = (O_i, J_i, T_i, D_i, C_i) \qquad \text{(D-1)}$$

where O_i is the time of the first job arrival of the task, J_i is the jitter that a task can suffer on activation, T_i is its period, D_i its deadline, and C_i is its worst-case execution time. This model describes a task τ_i that arrives at times $kT_i + O_i$ for non-negative integer values of k and a job is released $[kT_i + O_i, kT_i + O_i + J_i]$, capturing that J_i is an upper bound on the release jitter. Figure D-1 shows this.

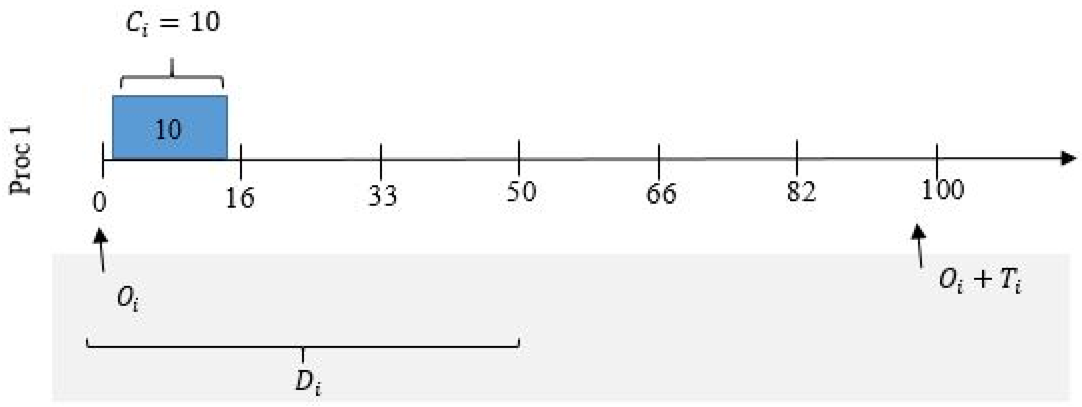


Figure D-1. Concepts in the model: O$_i$, J$_i$, T$_i$, C$_i$

It is worth noting that when applications have tasks that execute across multiple processors (e.g., reading sensor data in one processor, performing control computation in another processor, and actuating in yet another processor), they need to be decomposed into the different segments that execute in each of the processors with the corresponding deadlines and offsets that will ensure that their dependencies will be satisfied. In this example, one time unit represents one millisecond. Consider an example with an end-to-end task τ_1^{e2e} with a period equal to its deadline $T_1^{e2e} = 100$ and $D_1^{e2e} = 50$ that must execute three segments $C_{1,1,} = 10, C_{1,2} = 10, C_{1,3} = 10$ in three different processors. It can be decomposed into independent tasks such that:

$$D_{1,1} + D_{1,2} + D_{1,3} = D_1^{e2e} = 50 \quad \text{(D-2)}$$

and

$$D_{1,1} = O_{1,2} \quad \text{(D-3)}$$

and

$$D_{1,1} + D_{1,2} = O_{1,3} \quad \text{(D-4)}$$

One such decomposition is the following[9]:

- $\tau_{1,1} = (T_{1,1} = 100, O_{1,1} = 0,\ J_{1,1} = 0, C_{1,1} = 10, D_{1,1} = 17)$
- $\tau_{1,2} = (T_{1,2} = 100, O_{1,2} = 17, J_{1,2} = 0, C_{1,2} = 10, D_{1,2} = 17)$
- $\tau_{1,3} = \left(T_{1,3} = 100, O_{1,3} = 34, J_{1,3} = 0, C_{1,3} = 10, D_{1,3} = 16\right)$

Figure D-2 shows a Gantt chart of the decomposition with the corresponding parameters.

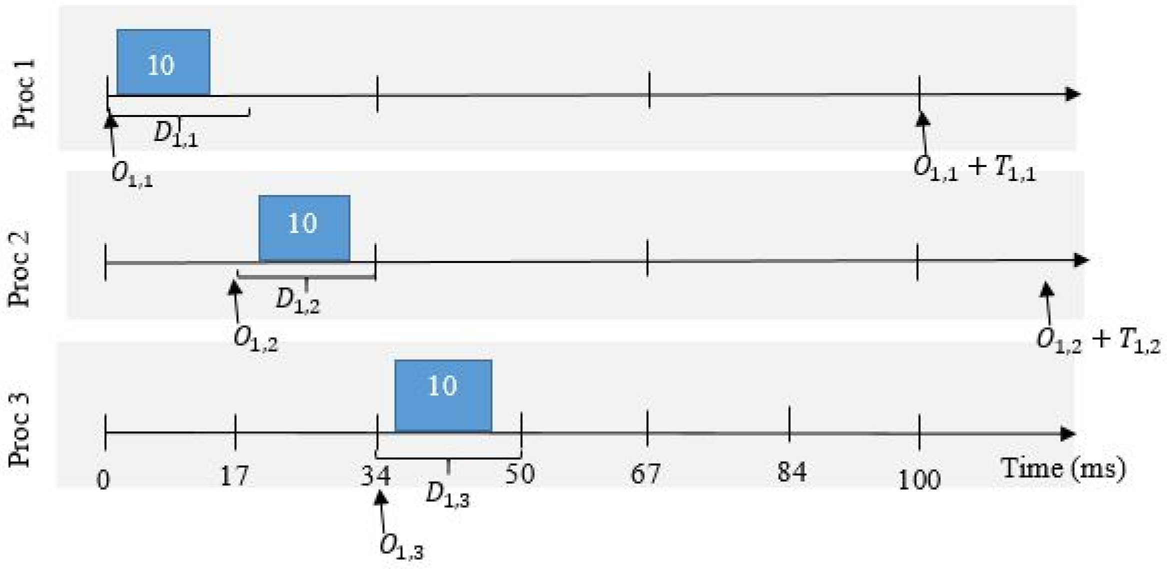


Figure D-2. Deadline decomposition

D.2.1.2 Brittleness

One of the key objectives of a timing interface is to isolate changes in a component. The research literature offers one approach for this [D-1]. It considers a system with components, and each component has an interface. Normal compositional schedulability analysis takes the interface as input. Here, however, a question arises: Is there an assignment of tasks to each component such that for each component, the tasks in this component respect the interface and the resulting task set is unschedulable? If the answer is "no," then the task set is schedulable. Brittleness will now be discussed more generally.

To characterize the brittleness of an interface, the following degrees of brittleness that match the degrees of freedom of a component task specification are defined.

9 The research literature offers several approaches for decomposing a task in a distributed system with an end-to-end-deadline into many independent tasks. One such method is available in [D-2].

1. Period Brittleness: Period brittleness defines the sensitivity of an interface to changes in the periodicity of a task inside a component. If the period is modified and all the other factors remain equal, it can also modify the utilization ($U_i = \frac{C_i}{T_i}$) of the task. Therefore, this degree of brittleness is divided into

   a. Utilization-Neutral Period Brittleness, in which the budget is adjusted to keep the same utilization, and
   b. Utilization-Agnostic Period Brittleness, in which the budget is not adjusted and it will affect the utilization.

   It is worth noting that utilization can be discussed only if the deadline is the same as the period. Otherwise, the focus needs to be on the density as described in the next paragraph. If the deadline is shorter than or equal to the period even across modifications, the density will remain the same.

2. Deadline Brittleness: The deadline brittleness has a similar effect to period brittleness with respect to density of the task ($\delta_i = \frac{C_i}{D_i - J_i}$). As a result, we also divide this brittleness into:

   a. Density-Neutral Deadline Brittleness, in which the budget is adjusted to keep the density constant, and
   b. Density-Agnostic Deadline Brittleness, in which the budget remains unchanged allowing the density to be modified.

   Note that when period and deadline are equal (and the offset is zero), the utilization and the density are the same, and when that is not the case, the relevant figure is density. Therefore, for the rest of the degrees of brittleness, the focus will be on density.

3. Offset Brittleness: This also modifies the density of the task and, therefore, is also divided into:

   a. Density-Neutral Offset Brittleness, and
   b. Density-Agnostic Offset Brittleness.

   Note that even though the deadline could be modified to keep the density the same (or the offset in the case of deadline brittleness), the focus in this case is on the adjustment of the budget.

4. Budget Brittleness: Similar to period and deadline brittleness, it is possible to modify the period and deadline to keep the utilization or density equal and, therefore, have both:

   a. Density-Neutral Budget Brittleness, and
   b. Density-Agnostic Budget Brittleness.

5. Jitter Brittleness: Jitter also affects density and, therefore, it is divided into:

   a. Density-Neutral Jitter Brittleness, and
   b. Density-Agnostic Jitter Brittleness.

The duration of the active interval of a task τ_i is defined as the deadline of the task minus its jitter parameter. This computed quantity is denoted A_i.

D.2.1.3 Time-Division Partitioning Interfaces

Time-division (TD) partition interfaces are offered by virtualization technology that provides time slots of CPU time to a component at regular intervals. The two most common virtualization technologies of this type are time-triggered architecture (TTA) [D-3] and ARINC 653 [D-4].

The interface defined by TTA consists of a frame-length specification ($\mathcal{F}$) and a set of time slots within the slot, typically of equal length. This can be specified as beginning offset and size ((b_i, z_i)). Putting it all together, a time-division partition interface can be expressed as:

$$I_{TD} = (\mathcal{F}, \{(b_1, z_1), (b_2, z_2), \ldots, (b_n, z_n)\}) \tag{D-5}$$

Here, it is assumed that the indices of b_i are given in time order; that is $b_1 \leq b_2 \leq \cdots \leq b_n$.

Time interfaces and interfaces in general have the purpose of isolating the component from the rest of the system to allow it to evolve independently. Specifically, this has two points of view. First, from the component perspective, if some other part of the system is modified, the interface of the component should not be modified. Second, from the rest of the system point of view, if the component is modified, then its interface should not be modified. From the certification perspective, the interface should allow for performing the certification against this interface and, therefore, if such an interface remains stable across changes it should block the propagation of changes.

TD interfaces are typically used in two ways: 1) to directly execute a task and 2) to execute a set of tasks for which another scheduler is used within the time slots of the interface. The former is the traditional way TTA systems are developed, and the latter is the way ARINC 653 systems are developed. In particular, it is common for ARINC 653 systems to use fixed-priority scheduling within the partitions.

Time interfaces must be evaluated in conjunction with the compositional verification techniques that would allow them to match their isolation objective. In other words, a compositional verification technique allows for evaluating whether a component with a specific interface can fulfill its timing specification (e.g., meet deadlines) after a change without the need to change the interface. The compositional techniques and their limitation are now presented, which can help answer this question.

D.2.1.3.1 Optimization and Constraint-Solving Approaches

Optimization and constraint-solving approaches—such as integer linear programing, mixed-integer linear programing, satisfiability (SAT) solvers, and satisfiability modulo theory (SMT) solvers—are typically used to either evaluate whether a TD interface can satisfy a task specification or to define a TD interface to satisfy such a specification. Whereas there are a number of approaches to optimize the combined schedule of tasks and messages in a time-triggered architecture based on these techniques, such as [D-5] [D-6], schemes for compositionality have not been proposed. However, it is common for this approach to identify the time slot table of a node as the interface of the node that has the same structure of the I_{TD} defined before. Therefore, a few aspects of this interface will be discussed.

D.2.1.3.1.1 Human Readability

Having a set of time slots allows the designer to easily check if the deadlines are met and that the precedence constraints of the different segments running on different nodes are respected. This is true if the schedule is small. Note, however, that if the least common multiple (LCM) of periods of tasks is large (e.g., because periods of tasks are relative prime numbers), then the schedule becomes very large, and therefore it is not so easy to "see" that deadlines are met and precedence constraints are satisfied.

D.2.1.3.1.2 Change Isolation

There are two brittleness limits that are worth discussing for this type of interface:

1. Partition initial blackout limit: An interface $I_{TD} = (\mathcal{F}, \{(b_1, z_1), (b_2, z_2), \dots, (b_n, z_n)\}$ has an initial blackout interval equal to b_1. This means that no task can be modified to have a period smaller than b_1.

2. Partition additional blackouts limits: For any two consecutive time slots $(b_i, z_i), (b_{i+1}, z_{i+1})$ of an interface $I_{TD} = (\mathcal{F}, \{(b_1, z_1), (b_2, z_2), \dots, (b_n, z_n)\}$ if $b_{i+1} > b_i + z_i$ then there is a blackout slot (b_i^{bo}, z_i^{bo}) such that $b_i^{bo} = b_i + z_i$ and $z_i^{bo} = b_{i+1} - (b_i + z_i)$. This means that any modification to a task period that increases a period while preserving the same density, and in which the period increment falls within a blackout, will not result in an increment in available CPU time and that the budget adjustment may therefore lead to a system that is not schedulable.

D.2.1.3.1.3 Internal fragmentation

Given that the interface describes a series of time slots that can be given to different tasks within the component (processor), if a new task is added, the interface will have to be added, assuming that all the time slots of the interface have already been assigned. Similarly, if a task changes one of the parameters that define its active interval, the blackouts discussed before can prevent it from receiving its required CPU time. Any slight increase in the C_i of the task can also render the task unschedulable if it goes beyond its assigned time slot.

D.2.1.3.1.4 Component Integration

Given that there is no modular schedulability per se, verifying that all the component interfaces can be satisfied (i.e., that they are schedulable system-wide) in this case is no different from the component-level schedule.

D.2.1.3.2 Priority-Based Delay-Calculation Approaches

A number of approaches exist that can be used for the compositional modeling of TD partitions when used in combination with fixed-priority scheduling that uses delay calculation [D-7][D-8] [D-9]. In this section, the focus is on [D-10], which presents a scheme to address the compositional analysis of ARINC 653 partitions with fixed-priority tasks within each partition. The timing interface of a partition is modeled as a single time slot that repeats periodically:

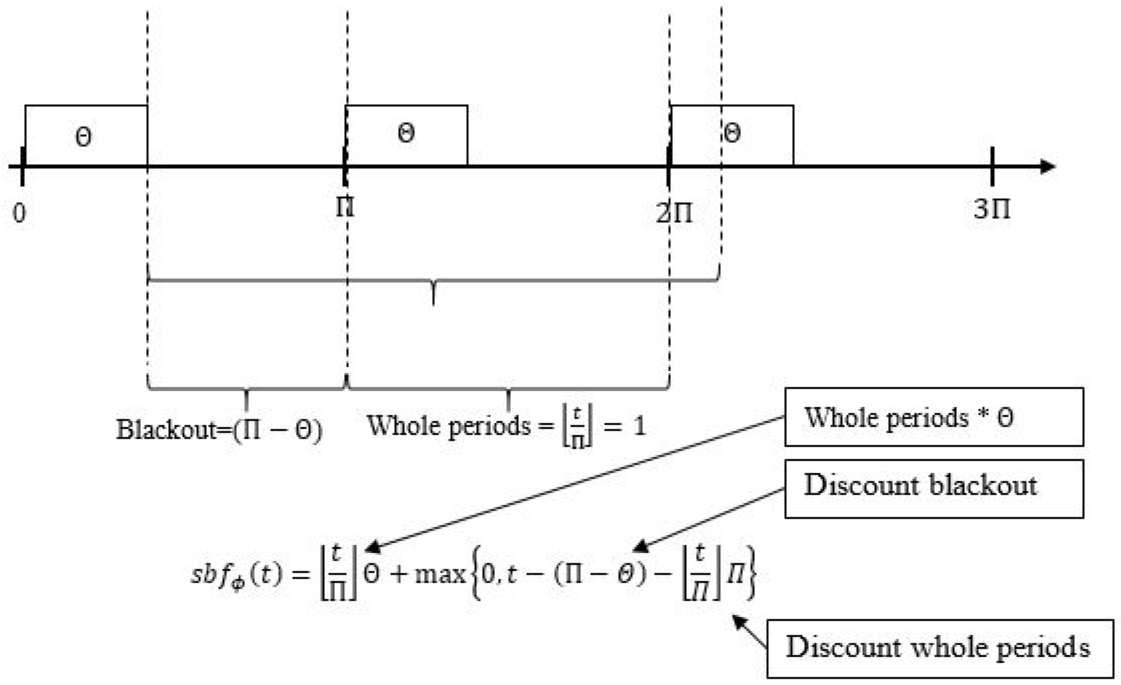
$$\phi = (\Pi, \Theta) \tag{D-6}$$

where Π is the period ($\mathcal{F}$ in our TD interface) and θ is the duration of the time slot ($\Theta = z_i$ of a time slot (b_i, z_i)).

With this definition, the minimum amount of CPU time that this interface provides is described with a supply-bound function $sbf_\phi(t)$:

$$sbf_\phi(t) = \left\lfloor \frac{t}{\Pi} \right\rfloor \Theta + \max\left\{0, t - (\Pi - \theta) - \left\lfloor \frac{t}{\Pi} \right\rfloor \Pi \right\} \tag{D-7}$$

This can be better understood by the example shown in figure D-3.



Θ
Θ
Θ
0
Π
2Π
3Π
Blackout=(Π − Θ)
Whole periods = $\left\lfloor \frac{t}{\Pi} \right\rfloor = 1$
Whole periods * Θ
Discount blackout
$sbf_\phi(t) = \left\lfloor \frac{t}{\Pi} \right\rfloor \Theta + \max\left\{0, t - (\Pi - \theta) - \left\lfloor \frac{t}{\Pi} \right\rfloor \Pi \right\}$
Discount whole periods

Figure D-3. Supply-bound function sample

Figure D-3 shows how an interval t is divided into a blackout interval in which the interface does not provide any CPU time, a number of whole periods in which the full Θ amount of CPU time is secured, and a remainder that can be obtained by subtracting the blackout period and the whole periods already accounted for from t.

The CPU time that a task τ_i requests within an interval t between two time instants t_1 and t_2 with all the tasks with higher priority than τ_i is described with the request function:

$$rf_{P,i}(t_1, t_2) = \sum_{j=1}^{i} \left(\left\lceil \frac{t_2 - O_j}{T_j} \right\rceil - \left\lfloor \frac{t_1 - O_j - J_j}{T_j} \right\rfloor \right) C_j \quad \text{(D-8)}$$

where tasks with lower index have higher priority.

With this function, it is then possible to evaluate the schedulability of a task set by testing one task at a time for the different intervals between 0 and the LCM of the tasks' periods. These intervals need to be incremented only with the arrival of the period. This can be captured with a variable t_x that is updated as: $t_x = O_i + J_i + xT_i$ for integer values of x between 0 and the value that makes $t_x + D_i - O_i - J_i \geq LCM$. Then a task is tested within each of these interval values as

$$\exists t \in (t_x, t_x + D_i - O_i - J_i]: rf_{P,i}(0,t) \leq sbf_{\phi}(t) \wedge rf_{P,i}(t_x, t) \leq sbf_{\phi}(t - t_x) \quad \text{(D-9)}$$

This basically means that there exists an interval t in which the requested CPU time is less or equal to the supplied CPU time. This is verified for each of the task arrivals (i.e., jobs) within the 0-to-LCM interval as discussed in this section.

D.2.1.3.2.1 Human Readability

The schedule of fixed-priority task sets running under TD interfaces is less readable than a full TTA-style interface. This is because it is not clear from the interface the exact instant when a task runs. However, it is possible to still relate the earliest time that a task can start to run and the latest time when it completes. This information can then be used to verify the end-to-end deadlines and the precedence constraints of distributed tasks. Whereas ensuring that these timing constraints are respected requires understanding this theory, once the theory is understood, the information observed in the interfaces does not grow with the number of tasks in a component as it does in a TTA interface.

D.2.1.3.2.2 Change Isolation

The change-isolation capacity of this interface is now discussed in terms of its brittleness limit. For TD interfaces analyzed with supply (see "1. Partition Blackout Brittleness Limit" and figure D-3) and demand bound functions the following limits are worth highlighting:

1. Partition Blackout Brittleness Limit: When the modification to the task parameters T_i, D_i, O_i, J_i leads to either $T_i \leq (\Pi - \Theta)$ or $(D_i - J_i) \leq (\Pi - \Theta)$, then the task will not be able to guaranteed the reception of any CPU time. This is because CPU time can be guaranteed only after the blackout period has elapsed, as can be seen in figure D-3. Notice that this limit is independent of the density and therefore cannot be overcome even if the budget is adjusted.

2. Task Active interval to Partition Period Ratio Limits: This ratio creates a rounding effect that limits the amount of CPU time that a task can receive. There are three cases:

a. When the task active interval A_i is larger than the blackout period but smaller than the partition period, i.e., $(\Pi - \Theta) < A_i \leq \Pi$: then the budget that the task is guaranteed to receive is limited by: $\max(A_i - (\Pi - \Theta), \Theta)$. This means that it is not guaranteed to receive the full Θ budget given that the blackout interval can delay the availability of the CPU budget.

b. When the task active interval A_i is larger than the partition period but smaller than twice the period minus the length of the budget: $\Pi < A_i \leq 2\Pi - \Theta$ then the maximum amount of CPU time that the task can be guaranteed is Θ. It is worth noting that even if the task parameters such as D_i, T_i, O_i, J_i are modified to enlarge the active interval to reduce the required CPU bandwidth consumption, this will not have an effect if the new active interval A_i^{new} stays within the range presented previously, i.e., $\Pi < A_i^{new} \leq 2\Pi - \Theta$. This is because within this interval no more CPU time can be guaranteed.

c. Generalizing the previous result when the task active interval is within the range $((n-1)\Pi, n\Pi - \Theta]$, i.e., $(n-1)\Pi < A_i^{old} < A_i^{new} \leq n\Pi$, the task is still guaranteed to receive only at most: $(n-1)\Theta$, i.e. no more CPU time can be guaranteed.

This means that if the active interval of the task τ_i is increased by a factor Δ, i.e., $A_i^{new} = \Delta A_i^{old}$ and the density is kept the same, the time demand in this interval would increase by the same factor: $C_i^{new} = \Delta C_i^{old}$ but the available CPU time would not be increased.

Reduced Internal Fragmentation: It is worth highlighting that using fixed-priority scheduling within a partition using this analysis technique reduces the internal fragmentation of a component as compared to techniques in which everything is driven by time (e.g., TTA). For instance, it is possible to add a task to the component keeping it schedulable with the same interface given that the budget of the interface is used (and shared) by all the tasks in the component instead of having individual time slots assigned to individual tasks. This also applies to changes to the parameters of component tasks.

D.2.1.3.3 Timed-Automata Approach

Recall from section 1.4 in our previous report “Evaluation of Verification Technologies for VMs” (appendix C) that:

1. A timed automata is a state-machine extended with real-valued clocks that can be tested and reset. Because the value of a clock now represents part of a state, the timed automata literature refers to an automata as having locations and transitions. A transition goes from one location to another location and a location may have a guard (i.e., a condition that must be true for the transition to be enabled). Guards can be of different types; one is based on clocks. For example, if *x* is a clock, then the transition from location *L1* to location *L2* might have the guard x≥5). A transition may have one or many actions; one possible action is to reset a clock. For example, if *x* is a clock, then setting *x* to zero might be an action of a transition).

2. To simplify modeling, a network of timed automata can be described. For such use, there is a primitive that specifies that two timed automata perform transitions simultaneously.

3. Timed-automata can be used to perform schedulability analysis. An error location can be introduced for each task (indicating that a job of the task has not yet finished and that the time since the job arrived exceeds the deadline). Then a so-called reachability analysis can be performed (i.e., to decide if it is possible to reach an error location). If "yes," then the system is unschedulable; otherwise the system is schedulable.

4. The arrival times of a task with a timed automaton can be described. This allows a software practitioner to model the software in a way that is very close to reality (e.g., modeling bursty arrivals).

5. With this approach, exact schedulability analysis can be decided if runtime scheduling is non-preemptive; with some minor adjustments, it is possible to also model some special cases of preemptive scheduling.

6. The advantage of performing schedulability testing this way is that it is very expressive. The drawback is that the time to perform the schedulability analysis grows rapidly with the number of tasks, and therefore only systems with a small number of tasks can be analyzed in practice.

The use of timed automata for schedulability analysis was previously discussed, but not for analysis of hierarchical systems. Therefore, in this section, schedulability analysis for analysis of hierarchical systems is discussed.

The paper [D-11] studies hierarchical scheduling on a single processor. The paper considers priority-based scheduling—fixed-priority or Earliest-Deadline-First (EDF)—of the global scheduler (the one that allocates processing time to components). The paper models hierarchical scheduling with timed automata with details on the operating system, proves correctness properties of these details, and also shows how to translate these details to C code so that it can run in an operating system. The paper considers preemptive scheduling and uses the TIMES tool [D-12] (already mentioned in a previous report) that models timed automata systems.

Because the TIMES tool supports only non-preemptive scheduling, the paper splits a task into fragments, each of one time unit. Note that this paper focuses on proving properties of the scheduler implementation—the paper does not perform schedulability analysis.

The paper [D-13] studies hierarchical scheduling on a single processor. The paper considers EDF scheduling of the global scheduler with reservations; this is called Constant Bandwidth Server. The paper considers fixed-priority preemptive scheduling as the local scheduler in each component (the paper refers to a component as an application). The paper performs schedulability analysis of each component (i.e., no analysis of the entire system is needed). This analysis is performed by modelling the system as a linear hybrid automata (a generalization of timed automata) and doing reachability analysis (i.e., a deadline miss corresponds to reaching a bad location, and the analysis aims to check if the system can reach this location). The reachability analysis is performed with a tool FORTS that the authors of the paper [D-13] have developed.

The idea for the modeling is as follows: Consider that there are *n* tasks in a component; this component is modeled with *n*+2 tasks. Each of these tasks is described with one timed automaton; for each of these timed automaton, there is a transition that indicates that a job arrives and there is a timer that serves as a guard on this transition that represents a job arrival. Because there are *n* tasks in the component, there are *n* such automata. There is also one dispatching automaton. For each possible subset of tasks that can be ready, there is one location to represent this subset, and for each such subset, there is a specified task that executes (because it is fixed-priority scheduling and it is assumed that priorities are unique). There is also one automaton that describes the server. It actually does not specify exactly the behavior of the server; instead, it specifies that the server execute in such a way that the server deadlines are met. With these *n*+2 automata, schedulability testing can be performed (using reachability analysis). This is accomplished with the stop-watch automata approach described by previous work mentioned in the report "Evaluation of Verification Technologies for VMs" (appendix C); that is, for each task, there are two clocks—one clock that is always incrementing and describes the amount of time since arrival and another clock that describes the total amount of execution that a job has done so far; this clock has a zero derivative when the job is preempted. The paper [D-13] shows examples of task sets for which traditional schedulability tests cannot prove schedulability but for which the method in [D-13] can. The risks of brittleness in the hierarchical scheduling scheme in [D-13] are the same as those described in section 2.1.3.2.2 (Change Isolation). There is, however, an additional risk of brittleness with this approach simply because the approach performs exact schedulability testing. It is believed that the brittleness of this approach can be reduced/assessed by modifying it in a fairly generic way as follows: Consider a component *k* and let orig denote the set of tasks in that component. Then check schedulability with the method mentioned in [D-13]. Assume that it resulted in deeming the task set schedulable. Now use binary search to find a number such that multiplying the server budget of the component makes the component schedulable but decreasing the server budget by some small amount makes the component unschedulable. This multiplication factor shows how much the budget can change without changing schedulability. A similar procedure can be used in case the original task set was unschedulable.

The paper [D-14] studies hierarchical scheduling. A previous work [D-15] performed schedulability analysis with timed automata and another studied schedulability analysis of a system with hierarchical scheduling using Petri nets. The paper [D-14] combines these ideas (but uses only timed automata; i.e., does not use the Petri nets from previous work). Thereby, the paper [D-14] presents a schedulability analysis of a system with hierarchical scheduling using timed automata. The idea is to analyze each component individually and for each individual analysis to use an automata that describes the supply of time from the global scheduler. The authors point out that hierarchical scheduling leads to some pessimism; that is, there are task sets such that the task set is schedulable if it was scheduled directly on the processor but because hierarchical scheduling is used, it is not possible to guarantee that deadlines are met (even if an exact schedulability analysis for hierarchical scheduling is used). Note that this pessimism is not caused by the schedulability test; it is caused by the use of hierarchical scheduling. Because the paper [D-14] performs exact schedulability analysis, the comment about brittleness made regarding paper [D-13] also applies to paper [D-14].

The paper [D-16] introduces the notion of stochastic supply to describe how much processing time the global scheduler supplies. For this model, a schedulability test based on timed automata is also presented. The paper [D-16] also compares a non-stochastic analysis against the CARTS tool (mentioned in this section) and finds an error in the CARTS tool. The paper [D-17] is a journal version of [D-16]; it also offers analysis of systems with multicore processors. For multicore processors, an important aspect of brittleness is the following: Consider a computer system with two processors and three components—component 1, component 2, and component 3. Assume that component 1 and component 2 are assigned to processor 1, and component 3 is assigned to processor 2. Assume that the system is schedulable. Now, change the system so that the software in component 1 requires much more processing, and component 3 requires much less processing. If the assignment of components to processors remains the same, the system may be unschedulable. A static assignment of tasks to processors may make the system more brittle. This could be mitigated by using a runtime scheduler with dynamic migrations. However, a runtime scheduler that allows tasks to migrate may generate additional overhead. This overhead is obviously problematic because it can delay execution of some tasks. More insidiously, however, it is often very hard to find an upper bound on this overhead. A better approach is to assign tasks to processors statically so that this assignment does not change at runtime, but at design time, if the software changes, to allow a new assignment of tasks to processors.

D.2.1.3.4 Real-Time Calculus Approach

Given that it depends on the underlying scheduling scheme being modeled, real-time calculus [D-7] does not have a fixed brittleness characterization. Specifically, recall from [D-18] that real-time calculus uses request functions $R(t)$ to describe the total amount of computation (CPU cycles) that a task requires and capacity functions $C(t)$ to describe the capacity of a processor to provide CPU cycles over a specific time interval t (e.g., from 0 to 100 ms). This interval is generalized to "any" interval of a specific length Δ with the request-bound function:

$$\alpha_r(\Delta) = \max_{u \geq 0}\{R(\Delta + u) - R(u)\} \tag{D-10}$$

that obtains the maximum CPU request for an interval of length Δ and any possible subinterval of length 0 to Δ. Similarly, the maximum delivery curve for any subinterval of length Δ is calculated with:

$$\beta(\Delta) = \min_{u \geq 0}\{C(\Delta + u) - C(u)\} \tag{D-11}$$

Then, it is possible to obtain the remaining processing capacity after the request-bound function of the tasks had been serviced with the equation:

$$\beta'(\Delta) = \max_{0 \leq u \leq \Delta}\{\beta(u) - \alpha_r(u)\} \tag{D-12}$$

This function is then used to verify whether the request bound of a task is schedulable if it can be ensured that for an interval Δ equal to the deadline of the task $\beta(\Delta) \geq 0$ (see example in [D-18]).

Real-time calculus enables the analysis of mixed scheduling policies. For instance, it is possible not only to evaluate pure fixed-priority scheduling or pure TDMA scheduling, but it is possible to evaluate a mixture of these two. Unfortunately, these mixtures can create inefficiencies.

This can be observed in the example presented in section 1.2.2 of [D-18] in which two TDMA partitions are used to schedule two periodic tasks. Specifically, the two partitions have a size of 2 ms, dividing the timeline in 2-ms segments as shown in figure D-4.

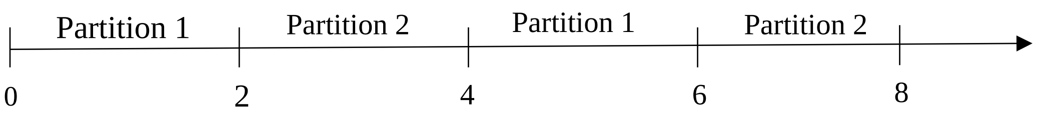


Figure D-4. TDMA partitions example (size = 2 ms)

This gives a task the delivery curve:

$$\beta(t) = \begin{cases} 0 \; if \; t < 2 \\ \frac{t-2}{2} \; if \; t \geq 2 \end{cases} \tag{D-13}$$

This curve can now be compared with the supply-bound function of figure D-3 to make the following observations:

1. They both have a blackout period. However, in figure D-3, this blackout repeats periodically, whereas in the delivery function of equation D-5, it does not.
2. Equation D-5 makes a linear approximation of the available CPU time by dividing the "rest" of the time (after discounting the blackout) by the size of the number of partitions (2). In contrast, in figure D-3, a more elaborate modeling of the periodic interleaving of blackout and CPU availability is used.

These two observations highlight the significance of the modeling step. In particular, equation D-5 shows a solution that was originally aimed at simplifying the explanation of real-time calculus that, in this case, also allows for discussing the penalty incurred for such a simplification. To see this, consider a task τ_1 with $T_1 = 10$, $C_1 = 3$, $O_1 = J_1 = 0$, and $D_1 = 5$. If the model provided in figure D-3 is used, there is a supply-bound function:

$$sbf_{\phi}(t) = \left\lfloor \frac{t}{4} \right\rfloor 4 + \max\left\{0, t - (4-2) - \left\lfloor \frac{t}{4} \right\rfloor 4\right\} \tag{D-14}$$

And a request-bound function for τ_1:

$$rb_1 = \left\lceil \frac{t}{10} \right\rceil 3 \tag{D-15}$$

Then, subtracting $rb_1 - sbf_{\phi}$ for $t = 5$, the remaining execution time to be completed is found:

$$\left\lfloor \frac{5}{4} \right\rfloor 4 + \max\left\{0{,}5 - (4-2) - \left\lfloor \frac{5}{4} \right\rfloor 4\right\} - \left\lceil \frac{5}{10} \right\rceil 3 = 0 \tag{D-16}$$

meaning that by this time the task has already finished. However, if the delivery curve in equation D-5 is used the result is:

$$\frac{5-2}{2} - \left\lceil \frac{5}{10} \right\rceil 3 = 1.5 \tag{D-17}$$

meaning that there is still part of the task that needs to be executed (from the model point of view).

<u>D.2.1.3.4.1 Request and Delivery Curves Modeling</u>

To understand the effects of the curve modeling, how these curves are described is first discussed. Specifically, a curve is specified in two dimensions (*X* and *Y*) where the *X*-axis defines elapsed time and *Y* defines the amount of computation or CPU time. In this setting, a curve is composed of a set of curve segments each of which has a starting point (*x*,*y*) and a slope indicating the amount of computation time per elapsed time. For instance, the curve presented in equation D-5 consists of two segments as: $\beta = \left(\left[(0,0,0), \left(2,0,\left(\frac{1}{2}\right)\right)\right]\right)$. This is shown in figure D-5.

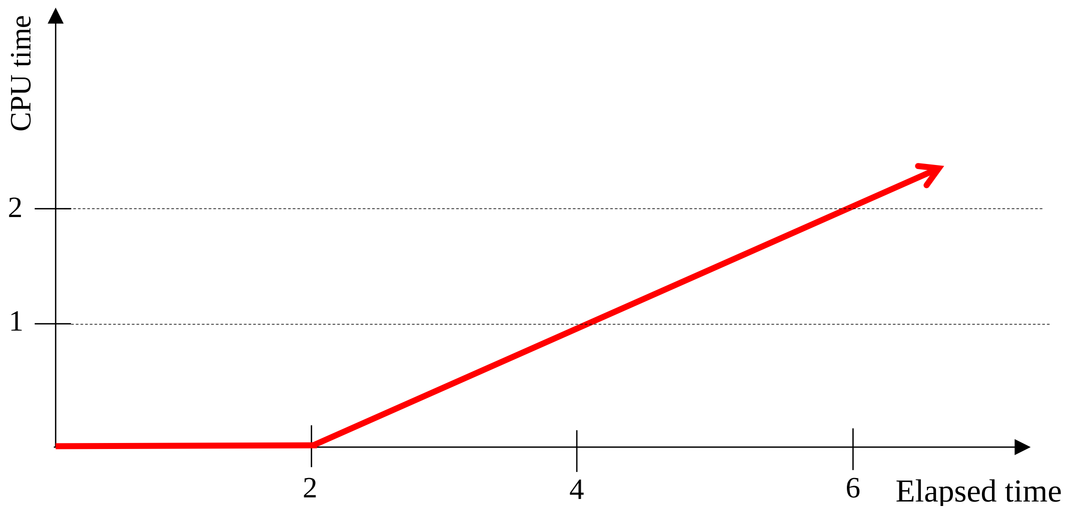


Figure D-5. Sample delivery curve

Delivery curves can also have a repeating part at the end that is specified as another set of curve segments with the starting *X* and *Y*, its period and offset. For instance, a periodic part at the end of β with a blackout segment could be created followed by an interval of full dedication of the CPU that repeats every 3 ms to create: $\beta' = \left(\left[(0,0,0), \left(2,0,\left(\frac{1}{2}\right)\right)\right], [(4,1,0),(5,1,1)], 3,0\right)$. This is shown in figure D-6.

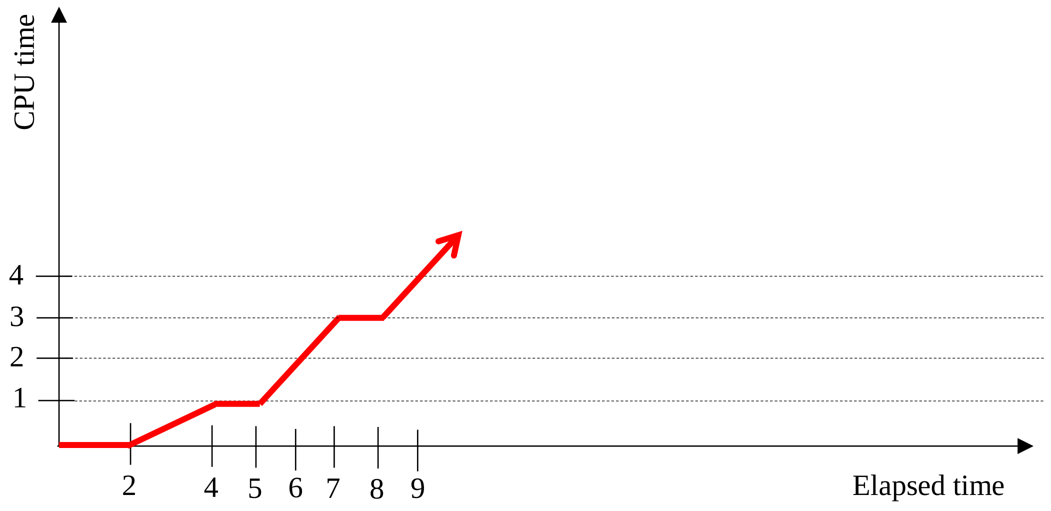


Figure D-6. Delivery curve with periodic segments

Generalizing for the purpose of discussion, a delivery curve can be defined as: $\beta_i = ([(x_1, y_1, s_1), \ldots, (x_k, y_k, s_k)], [(x_{k+1}, y_{k+1}, s_{k+1}), \ldots, (x_n, y_n, s_n)], T_i, O_i)$ where the periodic part is optional.

D.2.1.3.4.2 Change Isolation

The effect that the model of the request and delivery curves has on the brittleness of the model is now discussed with respect to the:

- Blackout Brittleness Limit: Delivery curves in real-time calculus may encode two types of blackout—aperiodic and periodic. However, the limit comes from the way the residual delivery function is calculated in equation D-4, which includes calculating the minimum available CPU cycles from the previous delivery curve in equation D-3 and the maximization of the CPU cycles request from equation D-2. Clearly, any modification to task periods where T_i or $(D_i - O_i - J_i)$ is smaller than the maximum blackout of the delivery curve will make that task unschedulable.
- Delivery Curve Fragmentation Brittleness: This fragmentation occurs when the sequence of the delivery curve segments that minimizes the CPU in the A_i interval (see equation D-3) that ends with a blackout segment. Then, the part of the blackout segment beyond A_i is an interval that causes brittleness. In other words, if the blackout curve segment is (x_b, y_b, s_b), and the one that follows the blackout segment is $(x_{b+1}, y_{b+1}, s_{b+1})$, then any modification to the task parameters that enlarges the active interval to A_i^{new} within the limits of the blackout interval (i.e., $A_i^{new} \leq x_{b+1}$) will not increase the CPU time available to the task.

D.2.1.4 Real-Time Server Interfaces

Recall from section 4.1.1 in the report "Survey of Literature Related to Virtual Machines" (appendix A) that:

1. Reservations allow a software practitioner to ensure timing isolation (i.e., an overrun of one task in one reservation does not jeopardize timing guarantees of another task in another reservation).

2. There can be one or many tasks in a reservation. A task in a reservation is allowed to execute only when the higher level scheduler selects the reservation for execution.

3. It is common to associate a reservation with a parameter P and Q, where P is often called the period of the reservation, and Q is often called the budget of the reservation.

4. Each reserve maintains a dynamic budget q. A task in a reserve is allowed to execute only if this dynamic budget is strictly greater than zero. Typically, this dynamic budget of a reserve is decremented when there is a task in the reserve that executes. When this budget reaches zero, the reserve is depleted, and then no execution from software within the component is allowed. Later, the budget of the component may get replenished/recharged, and then execution of software within the component is allowed.

5. There are different reservation-based schemes; they differ in how reserves are scheduled and recharged, and whether a reserve can use budget from another reserve.

6. Typically, schedulability analysis is performed within a reserve (to make sure that if the reserve gets enough processing power, all tasks within the reserve meet their deadlines) and schedulability analysis is also performed of reserves (to make sure that the entire system has enough processing power that each reserve gets what it needs).

7. The reservation can be thought of as an interface in the sense that the tasks can be changed as long as they do not break the reserve; with such change, it is guaranteed that all deadlines are met.

Reservations have been very successful. They were originally conceived in the early days of Rate-Monotonic Scheduling in the 1980s, and significant subsequent work has been performed on improving, maturing, and extending reservations after that. Reservations are supported in many standards and are used in practice. However, they have one fundamental weakness, which will be reported in this section.

Consider a computer system with a single processor and K tasks. Assume that the tasks are constrained-deadline sporadic tasks; that is, a task is characterized by a minimum inter-arrival time (T), a relative deadline (D), and an execution time (C). Assume that these parameters are as follows: $T_i = \infty$, $D_i = i$, $C_i = 1$. It can be seen that each task has infinite minimum inter-arrival times, and this implies that a task generates a single job. It can also be seen that all tasks have the same execution times but different relative deadlines.

Figure D-7 shows the schedule generated when all tasks generate a job simultaneously at time 0, and they are scheduled with EDF.

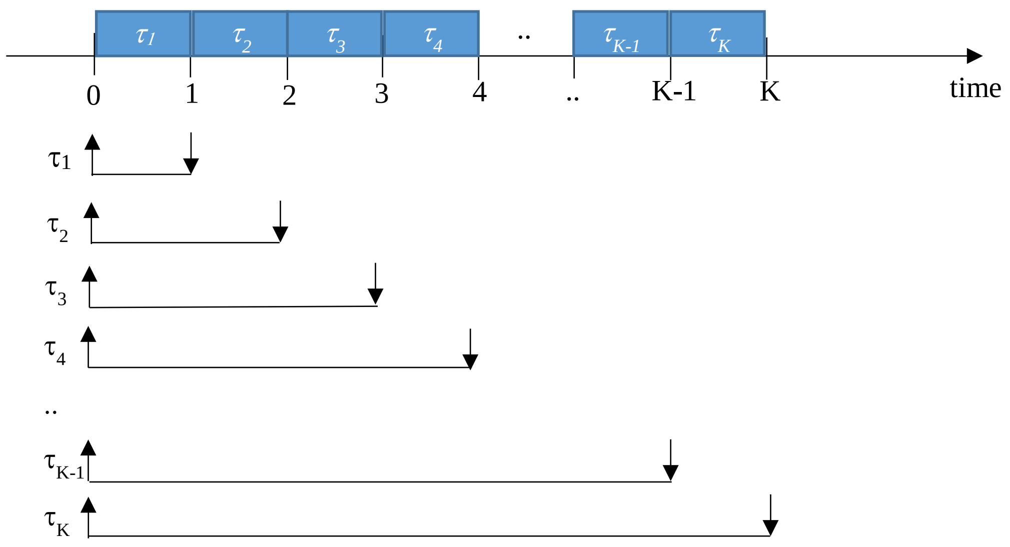


Figure D-7. An example of a task set that is schedulable but that performs poorly with reservation/bandwidth-like interfaces

It can be seen that all deadlines are met in this particular scenario. It can be shown that for all possible scenarios that are legal with respect to this task set, it holds for the generated schedule that all deadlines are met. However, suppose that reservations are used and that each task is in its own reservation. A reservation must be assigned a period (P) and a budget (Q). Let P_i denote the period of the i^{th} reservation and let Q_i denote the budget of the i^{th} reservation. Note that a task is described with three parameters, but a reservation is described with two parameters. Therefore, the global scheduler (which decides which reservation should execute at a given time) has less information about the tasks than the scheduler would have had if tasks were scheduled directly on the processor. The same applies to the global schedulability test (i.e., the schedulability test that takes the reservations as input). For this reason, hierarchical scheduling would be expected to lead to a performance loss. This will be shown in this section.

It is easy to show that for a task to meet its deadline, it is necessary that:

$$\frac{Q_i}{P_i} \geq \frac{C_i}{\min(D_i T_i)} \tag{D-18}$$

Plugging in the actual numbers from the example above yields:

$$\frac{Q_i}{P_i} \geq \frac{1}{i} \tag{D-19}$$

For all the reserves to be schedulable, it must hold that:

$$\sum_{i=1..K} \frac{Q_i}{P_i} \leq 1 \tag{D-20}$$

to meet deadlines on a unit-speed processor. Therefore, it must hold that:

$$\sum_{i=1..K} \frac{1}{i} \leq 1 \tag{D-21}$$

It can be seen that for $K \geq 2$, the above is false. However, a processor R times faster can be used. Therefore, it holds that it is necessary that:

$$\sum_{i=1..K} \frac{1}{i} \leq R \tag{D-22}$$

How large a value of R is enough? It is easy to see that for $K = \infty$, the left-hand side approaches infinity and, therefore, R must be infinite.

The above reasoning shows that there is a task set that can be scheduled on a unit-speed processor. If reservations are used, however, it is necessary to use a processor that is infinitely faster to make it schedulable.

Stated another way—reservations can cause an infinite loss of performance.

D.2.1.5 General-Purpose Virtualization Machine Interfaces

The Xen hypervisor will now be discussed. Recall section 2.2 in the report "Survey of Literature Related to Virtual Machines" (appendix A) that discussed Xen. The focus now will be on its scheduler, its application programming interface (API), and its ability to handle real-time requirements.

Xen uses a credit scheduler. It allows a software practitioner to implement a form of weighted fair queuing for scheduling VMs. For example, if Xen has two VMs, the API can be used to give them equal weight; then one VM would receive 50% of the processing capacity, and the other would receive 50% of the processing capacity. Another way would be to give the first VM a weight of 1 and the second VM a weight of 3. In this way, the first VM receives 25% of the processing capacity, and the second VM receives 75% of the processing capacity.

More processing capacity can be assigned to one VM over another, but this does not take deadlines into account. This can be problematic for certain real-time workloads in which the amount of processing capacity that each VM needs is the same, but they have different timing requirements. This can be seen by constructing an example analogous to the one shown in section 2.1.4, in which the weight of the credit scheduler plays the same role as the bandwidth (Q/P) mentioned in section 2.1.4.

D.2.1.6 Mixed-Criticality Interfaces

Mixed-criticality scheduling as initially presented by Vestal [D-19] presents layering interfaces in both priority and criticality. To see this, fixed-priority scheduling will be discussed followed by the modification to support criticality.

D.2.1.6.1 Priority Layer Interface

Given a task set $\{\tau_1, \ldots, \tau_n\}$ where each task is assumed to process events or pieces of data periodically (e.g., a frame from a movie) and where these tasks are assigned different priorities ($priority(\tau_i) \neq priority(\tau_j)$ if $i \neq j$)[10], a fixed-priority scheduler selects the higher priority task τ_i that is ready to execute. This task continues to execute until either it has finished its periodic execution (e.g., finished processing the current video frame and needs to wait for the arrival of the next frame) or until a higher priority task τ_j becomes ready to execute (in which case τ_j is selected for execution). The implication of these decisions is that a task τ_j is never preempted by a lower priority task (e.g., τ_i), effectively isolating τ_j from all lower priority tasks. In other words, if it is assumed that the tasks in the task set are ordered in decreasing order of priority ($priority(\tau_i) > priority(\tau_j)$ if $i < j$), fixed-priority scheduling defines a layered interface where a task τ_i is isolated from modification to the parameters of tasks $\{\tau_j | j > i\}$ if their priority remains lower than τ_is. This is reflected in the response time formula from [D-20]:

$$R_i = C_i + \sum_{j<i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad \text{(D-23)}$$

that takes into account the preemptions from the set of task $\{\tau_j | j < i\}$, that is considered the higher priority layer, but that ignores the task set $\{\tau_k | k > i\}$ that is considered the lower priority layer, given that the scheduler isolates τ_i from this set/layer.

An example of the priority layers is shown in figure D-8, in which, for example, τ_2 is able to ignore the priority layers below it where τ_3 and τ_4 reside but needs to take into account the priority layer where τ_1 resides.

10 Assigning the same priority to two tasks is possible but complicates the discussion.

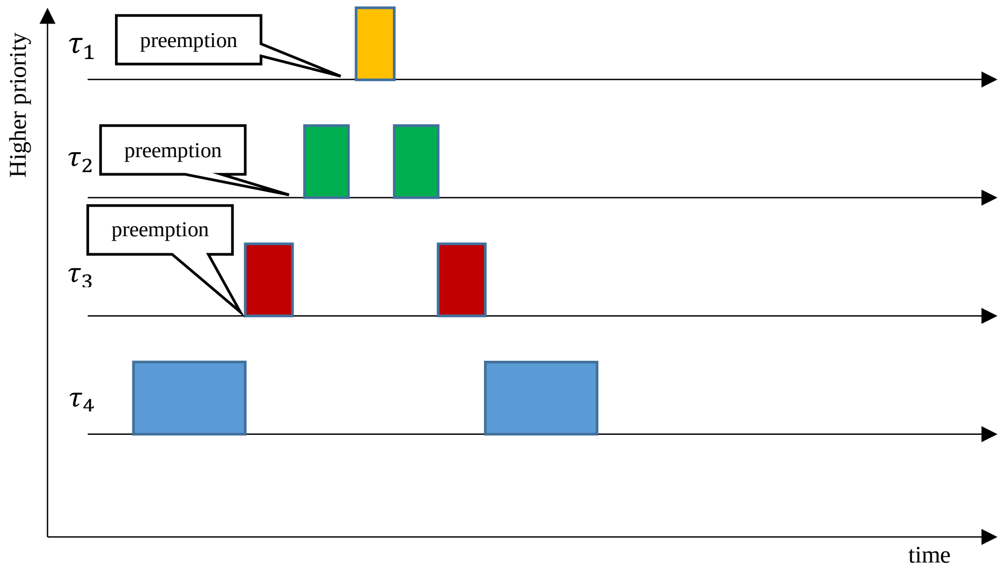


Figure D-8. Priority layers example

Some initial work in mixed criticality took advantage of the priority isolation to provide criticality isolation. For instance, Vestal [D-19] explored the use of using priority as criticality and [D-21] that defines a new priority assignment called Own-Criticality-Based Priority that searches all possible priority assignments to maximize schedulability while providing criticality protection. Unfortunately, these schemes suffer from a loss of schedulability given that a task τ_i with a short deadline may unnecessarily be delayed by a higher criticality task τ_j with a longer deadline (which could tolerate additional delay), making τ_i miss its deadline. As a result, alternative mechanisms were developed to improve the criticality protection while minimizing the schedulability penalty.

D.2.1.6.2 Criticality Layer Interface

A common approach to providing criticality protection while minimizing the schedulability penalty is to use priorities to maximize utilization and stop low-criticality tasks only when the deadline of a higher criticality task can be missed. This is the approach taken by the zero-slack rate-monotonic (ZSRM) scheduling [D-22][D-23]. In ZSRM, tasks are assigned priorities, ensuring that tasks with shorter periods have higher priorities than those with longer periods (known as rate-monotonic priority assignment), and stop lower criticality tasks only at the last instant possible to avoid a deadline miss of a high-critical task (known as zero-slack instant). Similar techniques are used by other schedulers, such as the Earliest-Deadline First Virtual Deadline [D-24], but the discussion will be limited to ZSRM for simplicity.

The low-criticality-stopping mechanism creates a layer similar to the priority layer. Specifically, this mechanism prevents the low-criticality task interference from leading to deadline misses of a high-criticality task. However, a key difference is that the mechanism does not isolate the high-criticality task from low-criticality task interference but only limits such interference to prevent deadline misses.

An example of the criticality layers is shown in figure D-9, which is a modification of figure D-8, where τ_3 has higher criticality but lower priority than τ_2. As a result, τ_3 is able to preempt τ_2 at time t_1, but at time t_2, τ_2 stops all lower criticality layers where tasks τ_2 and τ_4 reside, leaving higher criticality layers active where task τ_1 resides. This allows τ_1 to preempt τ_3 again. Once τ_3 finishes, the lower criticality layers are reactivated.

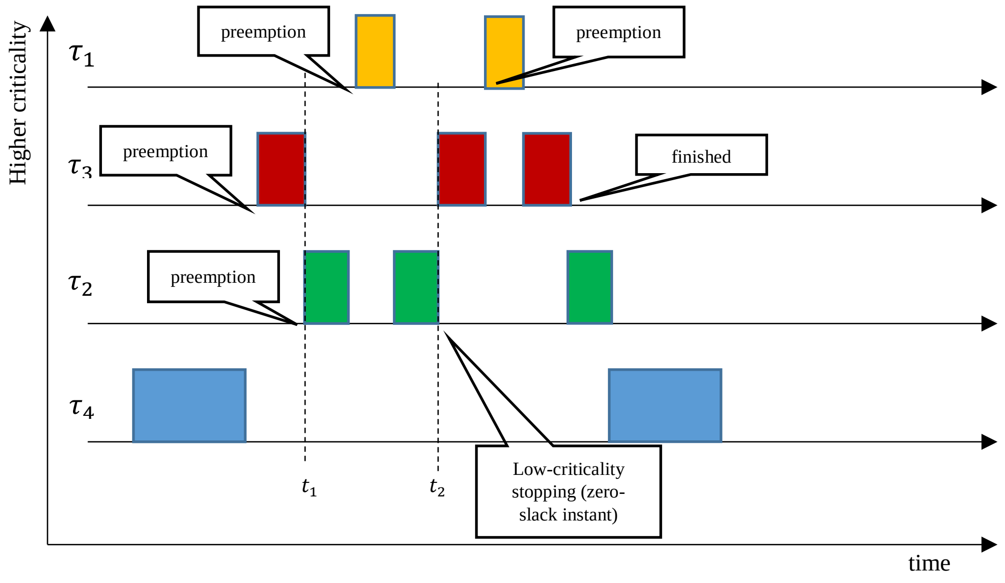


Figure D-9. Criticality layers

D.2.1.6.3 Assurance Level and Criticality

Mixed-criticality scheduling was created to meet the need to provide different degrees of assurance to tasks with different levels of criticality. More specifically, Vestal [D-19] observed that 1) standards like DO-178B/C [D-25] required a higher level of assurance and 2) in real-time scheduling, using a larger estimate of the worst-case execution time (WCET) of a task decreases the probability that such a task will exceed it. In mixed-criticality scheduling, this is used to assign one WCET per criticality level to each task and assign a criticality to each task so that it matches the assurance level required by the certification standard. As an example, consider a task set with three task $\{\tau_1, \tau_2, \tau_3\}$ with period (T), WCET ($C(criticality\_level)$) and criticality (ζ):

$$\tau_1 = (C_1(3) = 3, C_1(2) = 2, C_1(1) = 1, T_1 = 10, \zeta_1 = 3), \qquad \text{(D-24)}$$

$$\tau_2 = (C_2(3) = 2, C_2(2) = 2, C_2(1) = 1, T_2 = 20, \zeta_2 = 2), \qquad \text{(D-25)}$$

$$\tau_3 = (C_3(3) = 3, C_3(2) = 3, C_3(1) = 3, T_3 = 30, \zeta_3 = 1) \qquad \text{(D-26)}$$

Then, when the schedulability of a task τ_i is verified, the WCET is used for all tasks at the criticality level of τ_i (ζ_i). For example, to verify the schedulability of τ_2, $C_1(\zeta_2) = C_1(2) = 2, C_2(\zeta_2) = C_2(2) = 2, C_3(\zeta_2) = C_3(2) = 3$ would be used for the corresponding tasks in the task set.

ZSRM embraces the layering approach and uses only two execution times per task: one for its own criticality level and another for the lower criticality layers that are originally called overloaded execution time (C_i^o) and nominal execution time (C_i), respectively. This allows for providing a conditional schedulability that embeds some tolerance to uncertainty in the parameters. Specifically, the schedulability guarantee offered is:

A task τ_i is guaranteed to meet its deadline if it does not execute beyond its C_i^o, and no task τ_j with higher criticality ($\zeta_j > \zeta_i$) executes more than C_j.

In ZSRM, this tolerance to an overload was captured in the parameter names, and in mixed-criticality scheduling in general, it is embedded in the different execution-time parameters.

D.2.1.6.4 Brittleness of Mixed-Criticality Layer Interface

The brittleness of mixed-criticality layers had been investigated for ZSRM with a metric called ductility. More specifically, ductility is the opposite of brittleness, and the purpose of the metric was to investigate the consequence of overloads within the system. In particular, ductility measures the number of deadline misses and their importance (i.e., how critical). These deadline misses are combined in a weighted sum that are assigned an order of magnitude more weight to one deadline of one criticality level than to another of the next lower criticality level. This is reflected in a ductility table (see table D-1) presented in the following dual-criticality (high and low) system:

Table D-1. Ductility

Overloads		Deadline Misses	
High	Low	High-Criticality	Low-Criticality
Yes	Yes	No	Yes
Yes	No	No	Yes
No	Yes	No	Yes
No	No	No	No

This table reflects under the Overloads columns whether high-criticality tasks, low-criticality tasks, or both overload (runs for C^o), and under the Deadline Misses columns whether a deadline miss will be observed in high- or low-criticality tasks. This table is translated into a 0/1 matrix (yes = 0, no = 1), and the deadline misses are quantified, as shown in table D-2:

Table D-2. Deadline Misses for different overload states

Overloads		Deadline Misses	
High	Low	High Criticality	Low Criticality
Yes	Yes	1	0
Yes	No	1	0
No	Yes	1	0
No	No	1	1
Total		4	1

Then add the columns, and multiply the "High Criticality" column by the maximum number that can be achieved by adding all previous columns plus 1. In this case, if all the numbers in the "Low-Criticality" column were 1, then the sum would be 4. This means that the the second column needs to be multiplied by (4 + 1 = 5). This scales each column by one order of magnitude more than the previous one, as happens in the decimal system, in which tens are multiplied by the maximum number that a single digit can reach (9), plus 1. Then, for this example, the ductility obtained will be:

$$\text{Ductility} = 4\,(4 + 1) + 1 = 21 \qquad \text{(D-27)}$$

This metric was used to measure the improvements obtained by new deployment algorithms for mixed-criticality systems in multiple processors called "Compress-on-overload Packing" [D-26]. A plot of the normalized ductility obtained with this algorithm contrasting with a criticality-agnostic Worst-Fit Decreasing algorithm is shown in figure D-10.

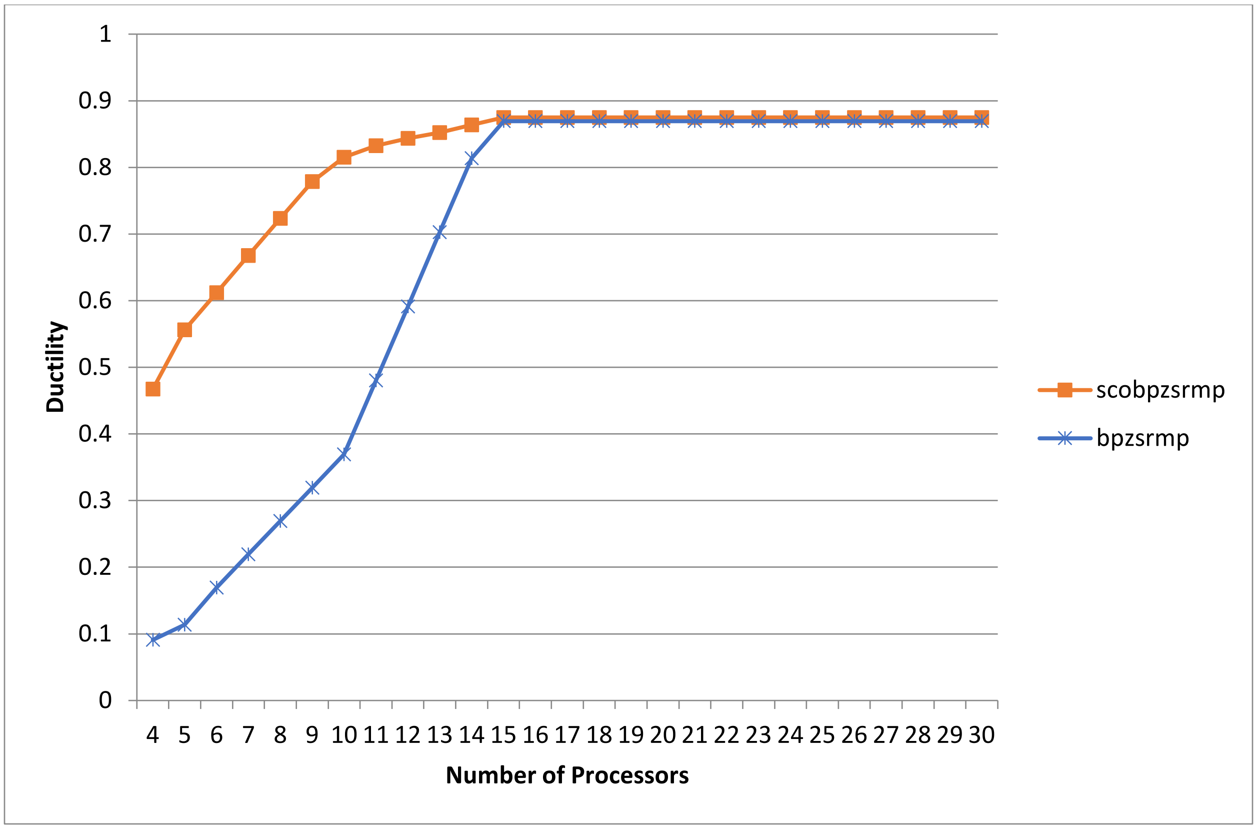


Figure D-10. Compress-on-overload packing for ZSRM

This ZSRM was extended to support multi-modal systems in which the criticality of a task can change in different modes (e.g., at low speed, an active suspension task is low criticality, but at high speed it is safety critical) with the corresponding deployment algorithm called “vector Mixed-Criticality Packing” (vMCP) [D-27]. Figure D-11 shows the normalized ductility obtained with the new deployment and scheduling algorithm compared to a criticality-agnostic vector packing (multidimensional packing). In this case, each mode is modeled as one dimension in the vector-packing algorithm.

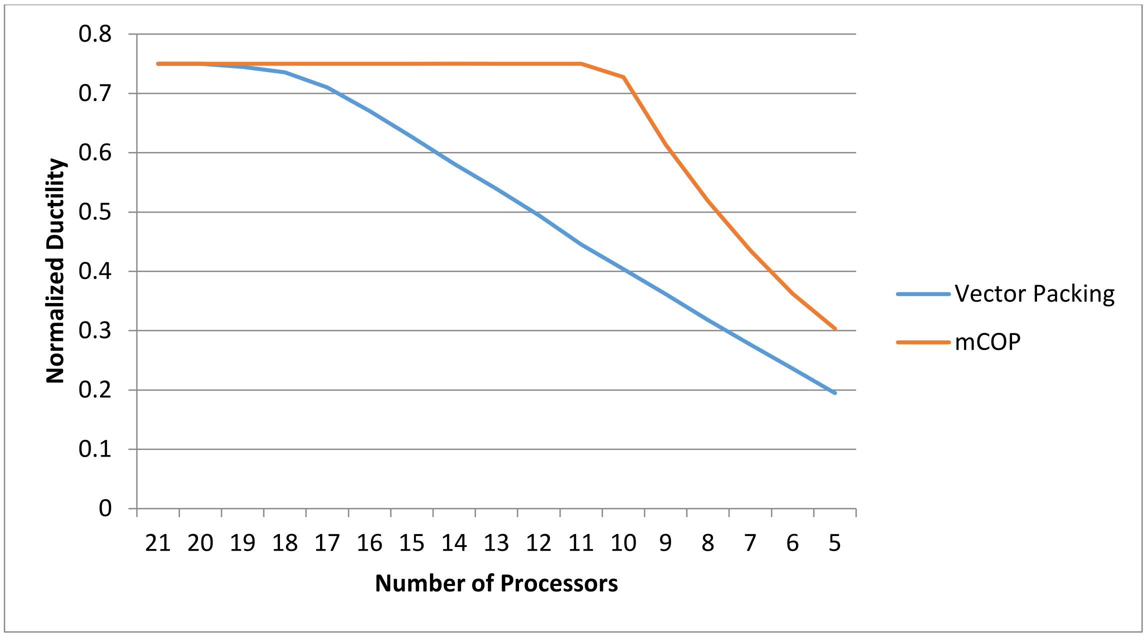
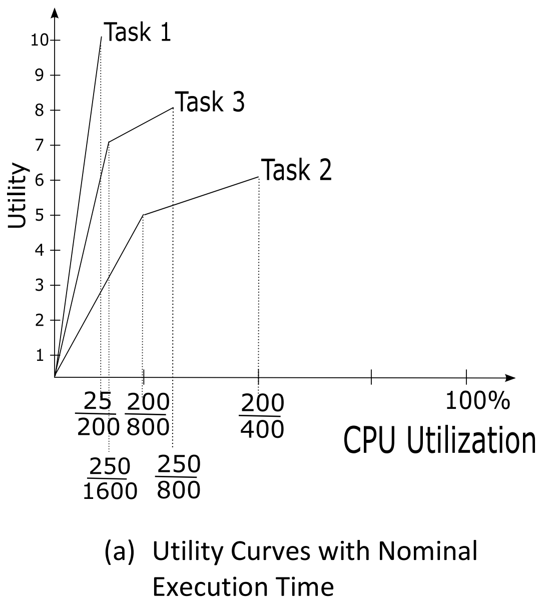


Figure D-11. vMCP ductility compared to vector packing

Clearly, not only can the execution time of a task vary, but also the periodicity. More importantly, changing the periodicity can be a way to compensate for overloads in the system. ZS-QRAM [D-28] is a variant of ZSRM that takes advantage of this. ZS-QRAM generalizes criticality into utility to capture the fact that the benefit of improving the quality of service (QoS) exhibits diminishing returns as QoS is increased. For example, increasing the frames per second (FPS) in a movie from 10 to 15 provides more utility to the user than increasing from 25 to 30, even though the number of FPS is the same. The diminishing-returns property allows us to model the gain in utility per unit of resource (CPU time) an application is given, creating what are called utility curves. This utility per unit or resource is known as the marginal utility. In this case, the task also has two execution times (C_i, C_i^o), but the amount of resources consumed is changed by varying the period (i.e., each task will have a set of periods and its corresponding marginal utility). This matches most common application-adaptation knobs, as is the case of the FPS, in which the period matches priority.

Figure D-12 shows the utility curves of a task set under nominal execution time (nominal curve) and overloaded execution time (overloaded curve).



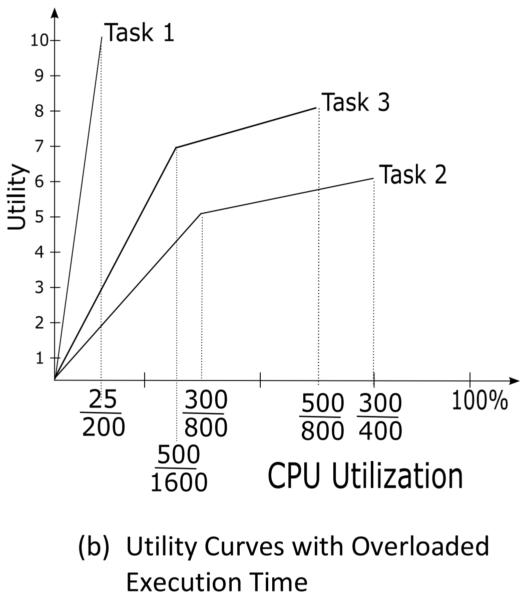


Figure D-12. Utility curves

The nominal curve is used offline to assign the initial period of the tasks and its initial priority following rate-monotonic scheduling. This initial selection is begun by using the longest period of all the tasks and performing ZSRM schedulability. If it succeeds, then the task with the steepest marginal utility (slope in the utility curve) in the nominal curve selected and its period are reduced by one step. The steepest slope is selected to ensure that the maximum amount of utility is obtained per unit or resource assigned. The schedulability is tested again, and this step is repeated until the next period reduction is not possible or until all the tasks are running with their smallest period.

At runtime, when a task τ_i executes beyond its C_i, a period degradation (a longer period is selected) on a task is performed, and its priorities are reassigned according to the new period. The selection of the task is performed using the overloaded curve by selecting the flattest curve to ensure that the least amount of utility is lost per unit of resource freed (to ameliorate the overload). Then the priorities are adjusted according to rate-monotonic scheduling.

For this scheduler, a variation of ductility called utility degradation resilience (UDR) captures the tolerance to preserve utility in the system on overloads. UDR is calculated in a similar manner to ductility, but using utility multiplying each of the "ones" in the matrix instead of the order-of-magnitude scaling factor.

Figure D-13 shows the UDR for an experiment in which different schedulers are evaluated as the overload ratio ($\frac{C_i^o}{C_i}$) changes. The figure shows UDR for rate-monotonic (RM) scheduling, RM with period transformation (e.g., instead of using a 10 ms period, use a 5 ms period for half the C_i), ZSRM, and finally, ZS-QRAM. Clearly, the adaptations of ZSQRAM allow it to preserve more utility (UDR) than the other schedulers.

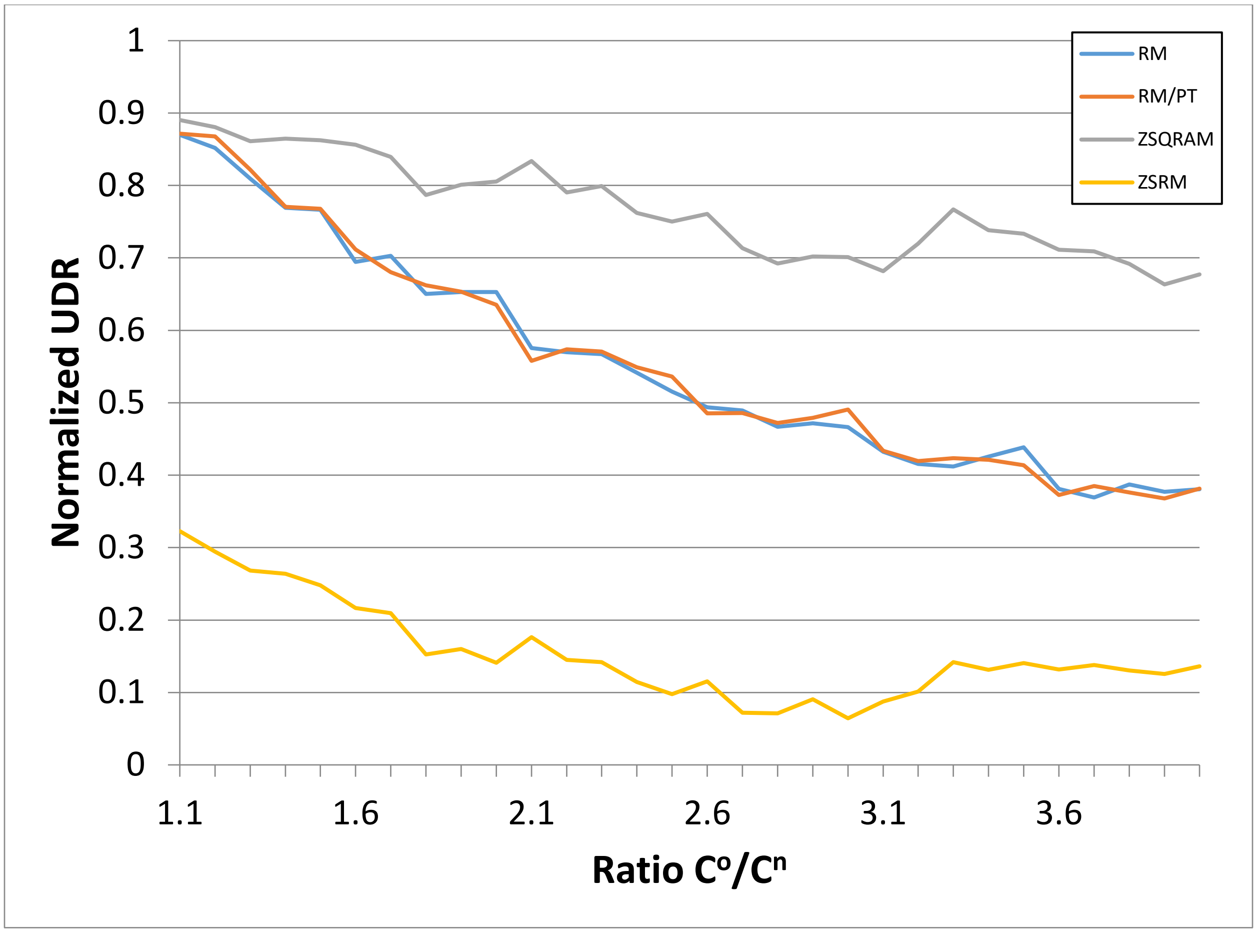


Figure D-13. Average UDR

The flexibility provided by mixed-criticality schedulers is offered on a per-task basis. This means that so far no concept of multi-task component has been developed. Clearly, this is an area of research that needs to be investigated.

D.2.1.7 Concepts Related to Brittleness

Recall that brittleness refers to the ability of an interface to withstand changes. A related concept will now be discussed: anomalies and sustainability. These concepts are not relevant for compositionality, but they are relevant for verification.

Scheduling anomalies refers to a situation in which a scheduler succeeds, but making a change expected to have positive effect causes the scheduler to fail. Scheduling anomalies were originally observed in 1968 by Graham [D-29] in non-preemptive scheduling of jobs on a multiprocessor. He considered a set of jobs that all arrive simultaneously in which each job is given a priority and wanted to find the earliest time so that, for all possible schedules that the jobs can generate, all jobs will finish (this time is called the makespan). Graham observed that for certain job sets, decreasing the execution time of one job could cause a larger makespan. This is unintuitive. Graham also found that for the slightly more complex problem in which jobs may have precedence constraints, there are situations where the makespan increases if a precedence constraint is removed.

Later, Andersson and Jonsson [D-30] saw that anomalies also exist in preemptive scheduling on multiprocessors. Consider a scheduling algorithm for tasks that do not migrate but use a so-called bin-packing algorithm to assign tasks to processors. For such a system, there are anomalies in the sense that there are task sets for which bin packing succeeds, but a decrease in the execution time of a task makes the bin-packing algorithm fail. Consider a scheduling algorithm for tasks that are allowed to migrate, for which tasks are stored in a ready queue shared between processors. For such migrative scheduling, consider the case that tasks are assigned fixed priorities. Then the lowest priority task will execute only if there is no higher-priority task that is ready for execution. In this case, if tasks are periodic, an increase in period (which makes the task intuitively use fewer resources) can create a new situation that generates a larger delay on a lower-priority task.

Anomalies also play an important role in WCET analysis, for which a cache hit may cause the WCET [D-31]. This is because although an instruction that experiences a cache hit executes faster than if it would have experienced a cache miss, the difference may change the scheduling of subsequent instructions within the processor. This is an issue in processors that can have multiple outstanding instructions, and instructions may be executed out of order.

Later, the notion of sustainability was conceived for scheduling tasks on a processor [D-32]. This refers to the schedulability test—not the scheduler. For a schedulability test that is sustainable with respect to execution times, if the task set is deemed schedulable by the schedulability test, and worst-case execution-time parameters of some tasks are then decreased, the task set is still deemed schedulable by the schedulability test.

One positive result is by Liu and Ha [D-33], who showed that scheduling jobs with fixed-priority preemptive scheduling has no anomalies on a single processor. This is fortunate because this scheduler is very common in practice.

Sensitivity analysis has been explored not for compositionality but for verification. This is important because response times are nonlinear with changes in task parameters. This can be seen as follows: Consider two tasks to be scheduled on a single processor. τ_1 has the highest priority, and τ_2 has the lowest priority. Their parameters are as follows:

$$\begin{array}{ll} T_1 = 1 & C_1 = 1 - \epsilon \\ T_2 = \frac{1}{\epsilon} & C_2 = 1 \end{array} \qquad \text{(D-28)}$$

If $\frac{1}{\epsilon}$ is an integer, then the response time of τ_2 is $\frac{1}{\epsilon}$ and, therefore, the task set is schedulable. Suppose the execution time of task τ_1 by $\epsilon \times (1 - \epsilon)$ is increased. Then the task set becomes:

$$\begin{array}{ll} T_1 = 1 & C_1 = 1 - \epsilon^2 \\ T_2 = \frac{1}{\epsilon} & C_2 = 1 \end{array} \qquad \text{(D-29)}$$

The response time of τ_2 is now $\frac{1}{\epsilon^2}$. Suppose that ϵ is chosen to be $\epsilon = 10^{-6}$. Increasing the execution time of τ_1 by approximately 10^{-6} can cause the response time of τ_2 to be 10^6 times larger. To put it differently, a small increase in the execution time of one task can cause a large increase in the response time of another.

It has been shown that a very small modification of one parameter of a task can cause a large difference in the response time of another task. Sensitivity analysis aims to determine the impact of such changes (and other changes). Three studies are noteworthy.

Vestal [D-34] considered fixed-priority preemptive scheduling on a single processor and used previous work on schedulability analysis for it to give answers to the following questions: 1) If a task set is schedulable given an index i, how much can the execution time of task τ_i be allowed to increase without making the task set unschedulable? 2) If a task set is schedulable, by how much is it possible to multiply all execution times of all tasks without making the task set unschedulable? The author of [D-34] presented closed-form expressions that answer these questions. The authors of [D-34] also extended these result to more complex systems: 1) a system in which a task may block because it waits for a semaphore held by a lower priority task, and 2) a system in which there is a set of modules and each task consists of a subset of these modules.

Punnekat et al. [D-35] pointed out that for more complex models, performing sensitivity analysis with closed-form expression is not possible. Therefore, the authors propose different types of binary-search procedures. One example of such a procedure is as follows:

1. Take a task set as input.
2. Store this task set as orig_taskset; later in the procedure, modify the task set, and then compare with the orig_taskset.
3. Check schedulability with some method.
4. When the task set is schedulable, multiply all execution times by some factor greater than 1.
5. When the task set is unschedulable, divide execution times by some factor greater than 1.
6. Now, there is a task set that is schedulable, but multiplying all execution times by some factor makes it unschedulable.
7. Repeat steps 4–5 with a sufficiently small factor.
8. Now, there is a task set that is schedulable but one in which multiplying all execution times by some small factor makes it unschedulable.
9. Choose some task in the task set obtained at line 8; let *i* denote the index of the task. Take the execution time of task *i* in the orig_taskset, and divide by the execution time of task *i* in the task set obtained after line 9. This factor tells how much the speed of the processor can change (or, alternatively, how much the execution times can be scaled) until schedulability changes.

In this procedure, a factor is obtained stating how much all execution times can be scaled without changing the result of schedulability testing. The authors of [D-35] also point out that a similar procedure can be used to determine how much a single task can have its execution time scaled without changing schedulability.

Bini et al. [D-36] studied sensitivity analysis from another perspective. They let certain variables of a taskset be free variables and expressed the space of assignments of these free variables that make the resulting task set schedulable. The authors show two examples of such spaces: 1) letting execution times of all tasks be the free variables yields the space of all assignments of values to execution times that makes the task set schedulable, and 2) letting periods of all tasks be the free variables yields the space of all assignments of values to periods that makes the task set schedulable. With such a space, sensitivity can be computed.

D.2.2 VIRTUALIZATION LOGICAL INTERFACES CHARACTERIZATION AND MODELING

Compositional verification has a long history in the domain of functional verification. For exhaustive verification techniques, such as model checking and theorem proving, it is widely known that compositional verification is crucial for scalable analysis. Some of the earliest work in this area is known as assume-guarantee or rely-guarantee reasoning. The assume-guarantee paradigm was proposed in various contexts in the early 1980s by Misra and Chandy [D-5], Jones [D-37], and Pnueli [D-38], and has since been explored (in manual/semi-automated forms) widely. The key idea is to represent each component as an automaton, with its interface being a set of actions (such as messages being sent or received) used to interact with other components. An assume-guarantee style proof rule is then developed and proved to be sound. This proof rule allows for verifying a property of the entire system by proving sub-properties about individual components. For example, one approach is to represent each component as a finite-state process, in which communication happens by synchronizing on shared actions, as in the Communicating Sequential Process formalism [D-39]. The property to be verified is also expressed as a finite state process, and $S \preccurlyeq P$ is written to mean that system S satisfies property P. Parallel composition of two processes M_1 and M_2 is denoted by $M_1 \parallel M_2$. In this setting the following "non-circular" assume-guarantee proof rule has been shown to be sound and complete [D-40]:

$$\frac{M_1 \preccurlyeq A \quad M_2 \parallel A \preccurlyeq P}{M_1 \parallel M_2 \preccurlyeq P} \tag{D-30}$$

Informally, this proof rule states that if component M_1 satisfies assumption A, and component M_2 satisfies property P under the assumption A, then the system composed of M_1 and M_2 satisfies property P. Note that the advantage of this proof rule is that the two premises (above the horizontal line) can be discharged without constructing the system $M_1 \parallel M_2$, which avoids state-space explosion. Of course, this means that an appropriate (and small) assumption A needs to be found that satisfies the two premises. The main difficulty is in doing this automatically. Several projects have explored the construction of such assumptions using learning algorithms, such as L* [D-41, D-42]. A number of other assume-guarantee proof rules exist. For example, consider the following "circular" proof rule:

$$\frac{M_1 \parallel A_2 \preccurlyeq A_1 \quad M_2 \parallel A_1 \preccurlyeq A_2 \quad A_1 \parallel A_2 \preccurlyeq P}{M_1 \parallel M_2 \preccurlyeq P} \tag{D-31}$$

This rule states that if there exist two assumptions A_1 and A_2 such that: 1) premise 1: component M_1 satisfies A_1 under an environment that satisfies A_2; 2) premise 2: component M_2 satisfies A_2 under an environment that satisfies A_1; and 3) premise 3: the system composed of A_1 and A_2 satisfies property P, then (conclusion) the system composed of M_1 and M_2 also satisfies P.

In another approach, originally proposed by Pnueli [D-38] for deductive program verification, components are program fragments, or environments under which they execute. Nevertheless, the proof rules have a similar structure irrespective of the underlying formalism used to represent components.

Relationship between premises and verification technology: The premises are discharged by a formal verification technique, such as model checking, SMT solving, or theorem proving. The specific verification technology used is tied to the formalism used to describe the components and assumptions. For example, one option is to express the components and assumptions as finite-state machines. In this case, the premises are typically discharged by a state-space-exploration technique, such as model checking. Another option is to represent the components and assumptions as logical formulas. In this case, theorem-proving-based techniques are more appropriate for discharging the premises. The application of theorem proving can range from being interactive (such as with PVS and HOL/Isabelle) to fully automated (such as with SMT solvers such as CVC4 and Z3).

Brittleness of premises: Given a specific system, property, and assume-guarantee proof rule, there may be multiple assumptions that are sufficient to verify the system against the property using the proof rule. For example, consider a specific system $M_1 \parallel M_2$ and property P. Suppose that the non-circular proof rule presented previously is being used and that the conclusion of proof rule holds (i.e., $M_1 \parallel M_2 \preccurlyeq P$). In this case, there may be several different assumptions A, each of which satisfy the two premises of the proof rule. One way to determine which of these assumptions is preferable is to choose the one that is least brittle. The concept of brittleness is inspired by the observation that, in practice, systems are often modified to fix errors and add new features. In this situation, it is desirable to have a verification approach that can be repeated with minimal additional effort if the target system and property undergo a small modification. In the context of assume-guarantee reasoning, an assumption (or set of assumptions) is said to be brittle if it (or they) can no longer satisfy the premises of the proof rule when one or more system components or the property is altered even slightly. In other words, brittle assumptions are "over-fitted" to the target system and property, and become unusable with even a small change.

In practice, the general observation is assumptions that are neither too restrictive nor too permissive are less brittle than those that are very restricted behaviorally. More specifically, suppose state machines are being used to represent assumptions. In this case, a state machine that accepts a moderately large language is less brittle and, therefore, preferable from a verification perspective compared to a state machine that either accepts too many traces or too few. To understand why this is the case, consider again the noncircular proof rule presented previously. If A's language is too small, it risks not satisfying the first premise if the language of M_1 increases. However, if A's language is too large, the risk is that the second premise will fail if the language of M_2 increases. A non-brittle assumption A should therefore balance between satisfying the two premises. This is also generally the case for other assume-guarantee proof rules. Similarly, if logical formulas are being used, those with a moderate number of satisfying assumptions are less brittle. The general guideline is that an assumption should be as general as possible (to accommodate for the larger possible range of behaviors of the components), whereas still being able to discharge the proof rules. Therefore, when constructing appropriate assumptions for assume-guarantee reasoning, whether manually or algorithmically, it pays to lean toward non-brittle ones.

Assume-Guarantee and Virtualization: In the context of virtualized systems, assume-guarantee reasoning can have several applications, including:

- Modeling and Verifying Spatial Isolation: Ensuring that applications executing in different partitions (or guests) are logically isolated, other than what is permitted by OS-level interfaces, such as shared filesystems and network devices.
- Modeling/Verifying Logical Interactions Between Partitions: Ensuring that applications in different partitions interact according to well-defined rules. For example, this might involve only certain partitions communicating with each other, and such interaction always using specific files and network protocols.
- Modeling/Verifying Logical Distributed Protocols: Ensuring that the correctness of protocols used by partitions are implemented properly. For example, this might involve verifying that correct synchronization mechanisms are used (to avoid deadlocks and race conditions), and message exchanges use appropriate formats and sequences.

From a certification perspective, it is advisable to have evidence that each of the above categories of correctness have been verified using appropriate techniques. Each of these classes of correctness will now be discussed in the context of various commonly used partitioning schemes.

D.2.2.1 Time-Division Partitioning and Interaction Protocol

D.2.2.1.1 ARINC 653

Partitions in ARINC 653 are referred to as "APEX" partitions. ARINC supports both inter-partition and intra-partition communication. Intra-partition interaction (i.e., between applications belonging to the same APEX partition) happen via standard OS-level inter-process (shared memory, pipes) and inter-thread (shared global variables) communication mechanisms. This can be logically verified. Moreover, applications that interact must use appropriate synchronization mechanisms (such as mutexes and semaphores) to avoid race conditions. These mechanisms must be implemented properly (e.g., using priority ceiling) to avoid priority inversions among tasks. Finally, the mechanisms must be used in the correct order to avoid deadlocks. Applications running in different partitions must communicate via ports and channels. Two types of ports are supported—polling and buffered. Polling ports have an effective slot size of 1. Newly sent or arrived data overwrite old data. In contrast, buffered ports can save several messages and deliver them in FIFO order. Both types of ports can be formally defined in terms of channels and verified using assume-guarantee reasoning for asynchronous communication processes. Verification can be of three varieties: 1) verifying that APEX partitions are logically isolated (i.e., there are no means of communication other than via ports); 2) verifying that applications use ports in the correct manner (e.g., that each polling port on a sender process side is paired up with another polling port on the receiver process side); and 3) verifying that the overall interaction leads to safe and secure behavior expressed by some system-level property.

D.2.2.1.2 TTA/TTP

In the TTA/TTP scheme, time is strictly divided among applications. It must be verified that applications cannot interact other than via standard interfaces. In particular, it must be ensured that information from one application does not inadvertently leak into another via a shared memory or device. In addition, as in the case of ARINC 653, correct use of synchronization mechanisms must be verified to avoid race conditions and deadlocks.

D.2.2.2 Rate-Monotonic-Based Partitioning Synchronization Protocols

In the rate-monotonic approach, time is allocated preemptively based on thread priorities. One major problem is to avoid priority inversion, which happens when a low-priority task holds a shared resource R (e.g., a mutex) and, therefore, blocks a higher priority task that is also trying to acquire R for an unbounded amount of time. To avoid this problem, special mutexes are used. Two such mutex schemes will now be discussed.

D.2.2.2.1 Reserve Inheritance

A reserve-inheritance mutex [D-4] is attached with a resource, and each resource is given the priority equal to that of the highest priority task that might access it. When a task acquires a mutex, its priority is bumped to that of the associated resource. This means that it will not be preempted by (and therefore cannot block) another task trying to acquire the mutex. One verification challenge is to prove that the mutexes have been implemented correctly (e.g., that priorities have been assigned to resources in the right way) and that thread priorities are updated correctly when they acquire a mutex. Another verification challenge is to prove the correctness of the scheme under all possible usage scenarios. Finally, it must be verified that tasks use mutexes properly. All these verification problems can be addressed in a compositional manner, and evidence to that effect will aid certification.

D.2.2.2.2 Priority and Criticality Inheritance

A priority-inheritance mutex [D-43] also involves updating thread priorities, but this happens more "lazily" than the reserve-inheritance scheme. When a low-priority thread T_L first acquires the mutex, its priority remains unchanged. However, if a high-priority thread T_H subsequently attempts to acquire the mutex, then the priority of T_L is bumped up to that of T_H. This allows T_L to be scheduled, and it can run until it releases the mutex, thereby avoiding priority inversion. Various variants of this protocol have been proposed in the literature. In a mixed-criticality setting, there is also the danger of criticality inversion, which can be avoided by using mutexes based on criticality inheritance protocols [D-44]. As in the case of reserve inheritance, correctness of these protocols at the algorithmic and implementation level, and their correct usage by threads, are challenges for verification. In each case, verification can be done in a compositional manner, and evidence of successful verification will increase confidence in predictable behavior at runtime.

D.2.2.3 General Purpose Virtualization Inter-Partition Coordination

In general, virtualized systems, in which the partitioning is done by a hypervisor and not based on time, there are also verification challenges that can be addressed by compositional logical analysis.

These fall into the categories mentioned above—ensuring that the coordination protocols among the partitions are correct at the algorithmic level, proving them correct at the implementation level, and finally verifying that the applications running within each partition use these protocols correctly. This approach has the hallmarks and advantages of compositional verification, because it allows for proving correctness of an entire system by first reasoning about each component (i.e., partition) and then verifying that the correct interaction between any group of components following the prescribed protocols leads to safe and predictable behavior.

D.2.2.4 Distributed Coordination and Virtualization

Finally, there are protocols and algorithms for distributed systems, such as leader election, consensus, and logical synchrony. Such protocols involve multiple components, which may execute in separate VMs, or on the same VM. In any case, there are still the verification challenges that have formed a common thread in this section. First, we need to prove that the protocols are correct at the algorithmic and implementation level. For example, the Physically Asynchronous Logically Synchronous (PALS) protocol has been subjected to rigorous verification both as an algorithm [D-45] and implementation [D-46]. Second, it needs to be proven that applications use the protocol in a correct manner.

D.3 MITIGATION STRATEGIES FOR COMPOSITIONAL VERIFICATION PROBLEMS IN VIRTUALIZED SYSTEMS

In this section, some of the techniques are discussed that can be used to mitigate shortcomings of virtualization with respect to compositional verification.

D.3.1 MITIGATION STRATEGIES FOR TIMING VIRTUALIZATION

From the timing point of view, compositional shortcomings come from both imperfect virtualization mechanisms and imprecise modeling. Some of the most important shortcomings and their mitigation strategies are discussed as follows.

D.3.1.1 Modeling Approximation for Scalable Verification and Enforcement

Some of the new challenges in virtualization are the correct identification and separation of the use of shared hardware. For example, in multicore processors, the shared-memory hierarchy is one of the big challenges for avionics systems. Whereas some virtualization/partitioning schemes had been offered for these resources [D-47], it is clear that at times it would be necessary to share partitions. The challenge in this case is absence of all the details. For such a case, it is possible to create safe model approximations and validate some of the parameters of the model through experiments. In [D-47], the authors used these concepts in two strategies. First, the authors developed a double-bounding model approximation that bounds the delay due to shared memory by taking the minimum between the worst case from each memory-access point of view and the worst case from the full-job execution. Second, the authors used experiments to obtain the value of the memory-access reordering queue inside the memory controller.

D.3.1.2 Hardware Scheduling Enforcement

Complementary to model approximations, it is also possible to restrict the behavior of the hardware to simplify the verification. Two examples are worth discussing. First, the enforcement of the concurrent memory accesses from DMA I/O devices can delay the memory access from the CPU (and the tasks running in it). In [D-48], the authors proposed a co-scheduler hardware that intercepts DMA requests and schedules them in a way that allows a predictable scheduling of CPU tasks.

In a second example, [D-49] Yun et al. describe a mechanism (called MemGuard) to police accesses to the shared memory bus in a multicore processor. This allows them to eliminate unpredictable accesses and simplify the timing verification, even if at the cost of performance.

D.3.2 MITIGATION STRATEGIES FOR LOGICAL VIRTUALIZATION

Compositional verification, whereas powerful in theory, is nontrivial in practice. For example, in the case of assume-guarantee reasoning, finding appropriate assumptions is quite complicated. Several strategies can help make this process practically feasible. First, the system should be properly architected so that its decomposition into components is clean and natural, and expressed rigorously. This means that components have well-defined interfaces, and such interfaces should be used in a relatively small number of ways. For example, the interface can be a set of input and output actions, and their legal use can be expressed as a finite state machine.

Second, the architecture can be multi-layered, in which components at each layer are further decomposed at the next layer. Key to this process is to represent each component at the right level of abstraction. The abstraction should eliminate details that are not important for verification, yet retain sufficient details so that target properties can be verified successfully. Formal notations, such as Architecture Analysis and Design Language and Statecharts, can be used to represent this hierarchical decomposition rigorously.

Third, the correspondence between abstract models of components and their concrete implementations should be demonstrable in a direct way. There are two ways of achieving this goal. One way is to generate the abstraction from the component's implementation, and use these abstractions for verification. There are several automated abstraction techniques, such as predicate abstraction [D-50], that can extract finite state machine (or push-down system) models from source code. Another way is to describe the models in an executable format, and then generate code directly. Of course, the correctness of the code generator must also be argued.

Finally, automated techniques can be used to construct the appropriate intermediate assumptions that can be used to discharge the premises of the assume-guarantee rules. For example, learning algorithms have been used successfully for this purpose.

D.4 DISCUSSION AND RECOMMENDATIONS

One of the key appeals of virtual machines (VMs) and related virtualization technologies is the ability to isolate components, thereby enabling the recertification of a component that has been modified without the need to recertify the rest of the components in the system. This is known as compositional verification.

In this report, the multiple issues related to compositional verification have been discussed from the point of view of virtualization mechanisms and analysis techniques. In this section, the main points of this report are summarized, and it ends with recommendations.

Perhaps the first observation was that, because their hypervisors lack a deterministic timing isolation between VMs, general-purpose VMs do not allow compositional verification. New efforts are currently being conducted to create real-time versions of these VMs. One of the main efforts is RT-Xen. Whereas RT-Xen is a step in the right direction, there are still some inherited mechanisms (like time quanta) that must be corrected to lead to hard real-time guarantees.

Second, generalized modeling techniques aimed at capturing any type of mechanisms, such as real-time calculus and timed automata, do not take full advantage of detailed knowledge of virtualization mechanisms because doing so can reduce their generality. As a result, they can lead to either intractable verification problems or very pessimistic models.

Mixed-criticality virtualization techniques that take into account the criticality levels present in standards such as DO-178C, are, at this time, focused on single-task isolation. Clearly, there is a need to extend these techniques to multi-task components that can support the challenges faced by the FAA.

Finally, assume-guarantee techniques for logical-verification compositionality still have a long way to go to make them more practical. Of particular importance is the development of architectures that simplify the assumptions, and the creation of interfaces.

This paper's recommendations can be summarized as follows:

1. Additional research is required to develop virtualization mechanisms and verification technologies that take into account the componentization required by the FAA (e.g., multi-task). This is because the current combination of virtualization mechanisms and compositional analysis do not provide true component independence (i.e., modifying one component impacts another).

2. New research in architectures and verification technologies for components with multiple levels of criticality, and with multiple tasks, is necessary. In this case, current research in mixed-criticality virtualization and verification either does not allow independent verification or can accommodate only one task per component.

3. New research in logical verification via the assume-guarantee style of interfaces and enforcement mechanisms is required. This is because the current technology to specify interfaces is either too complex (too many interacting options) or does not reflect the executable code.

4. There is a need to develop a protocol to allow the disclosure of details of commercial products that include hardware, operating systems, and verification tools to construct

evidence for certification with respect to the isolation claims that commercial real-time VMs offer.

For the certification engineer, these recommendations can be further elaborated as:

1. General-purpose virtual machines such as VMWare or VirtualBox should not be considered for avionics systems because there is no reliable technique to verify the timing isolation between VMs.

2. Real-time hypervisors that provide predictable temporal isolation should be paired with the verification technique that matches their mechanism. For instance, if time-division multiplexing is used (e.g., TTA), then exhaustive timeslot allocation algorithms must be used, and the arguments about why specific allocations satisfy the partition requirements must be presented. Similarly, if the technology is based on rate-monotonic scheduling and processing servers, the corresponding response time techniques must be used.

3. When considering recertification of partitions (or VMs) in isolation, the brittleness of the verification techniques must be considered. In particular, arguments to support why a modification to a partition does not affect other partitions must be presented. As elaborated in this appendix, different techniques have different sensitivities to modifications (brittleness). The brittleness discussion in this appendix can be used by a certification engineer to guide his/her evaluation of the isolation arguments.

4. When presenting logical arguments of separation, care should be taken to the assumptions of such claims. This is particularly important during recertification when the assumptions may change.

5. Arguments of isolation for VMs running in multicore processors must properly support the interference channels mentioned in CAST32A with the supporting details from the processor documentation.

Note: GR

The partitioned systems are never implemented with their connections, schedules, import/export channels. They are always designed as components that are integrated. This integration is directed by tools and a “language” that allows the composition to be expressed and then implemented through configuration files.

Of interest would be a description of these configuration files that help map applications onto the VM/Partitions/Task mechanisms together with the corresponding description of the communication mechanisms. This would allow for describing and simulating the behavior of composed systems and translating them to operational components.

D.5 REFERENCES

D-1. Kim, J., Abdelzaher, T.F., & Sha, L. (2015). Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 221–231.

D-2. Lipari, G. and Bini, E. (2011). On the problem of allocating multicore virtual resources to real-time task pipelines. *CRTS*.

D-3. Kopetz, H. (1998). The Time-Triggered Architecture. *Proceedings of the IEEE, 91*(1), 112–126.

D-4. Niz, D.D., Saewong, S., Rajkumar, R., Abeni, L. (2001). Resource Sharing in Reservation-Based Systems. *IEEE Real Time Technology and Applications Symposium*.

D-5. Misra, J., Chandy, K.M. (1981). Proofs of Networks of Processes. *IEEE Transactions on Software Engineering, SE-7*(4), 417–426.

D-6. Zhang, L., Goswami, D., Schneider, R., Chakraborty, S. (2014). Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC),* 119–124.

D-7. Easwaran, A., Anand, M., Lee, I. (2007). Compositional Analysis Framework Using EDP Resource Models. *28th IEEE International Real-Time Systems Symposium (RTSS 2007),* 129–138.

D-8. Feng, X.Y., Mok, A.K. (2002). A Model of Hierarchical Real-Time Virtual Resources. In *23rd IEEE Real-Time Systems Symposium*. 182–196.

D-9. Lipari, G., Bini, E. (2003). Resource Partitioning among Real-Time Applications. *ECRTS*.

D-10. Easwaran, A., Lee, I., Sokolsky, O., Vestal, S. (2009). A Compositional Scheduling Framework for Digital Avionics Systems. *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 371–380.

D-11. Asberg, M., Pettersson, P., Nolte, T. (2011). Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling. *2011 23rd Euromicro Conference on Real-Time Systems*, 172–181.

D-12. http://www.timestool.com.

D-13. Sun, Y., Lipari, G., Soulat, R., Fribourg, L., & Markey, N. (2014). Component-based analysis of hierarchical scheduling using linear hybrid automata. *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 1–10.

D-14. Boudjadar, A., David, A., Kim, J.H., Larsen, K.G., Mikucionis, M., Nyman, U., & Skou, A. (2013). Hierarchical Scheduling Framework Based on Compositional Analysis Using Uppaal. *FACS*.

D-15. David A., Larsen K.G., Legay A., Mikučionis M. (2012) Schedulability of Herschel-Planck Revisited Using Statistical Model Checking. In: Margaria T., Steffen B. (eds) Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. ISoLA 2012. Lecture Notes in Computer Science, vol 7610. Springer, Berlin, Heidelberg.

D-16. Boudjadar A. et al. (2015) Widening the Schedulability of Hierarchical Scheduling Systems. In: Lanese I., Madelaine E. (eds) Formal Aspects of Component Software. FACS 2014. Lecture Notes in Computer Science, vol 8997. Springer, Cham.

D-17. Boudjadar, A., David, A., Kim, J.H., Larsen, K.G., Mikucionis, M., Nyman, U., Skou, A. (2016). Statistical and exact schedulability analysis of hierarchical scheduling systems. *Science of Computer Programming, 127*, 103–130.

D-18. Andersson, B., Chaki, S., and de Niz, Dionisio. (2016). “FAA Virtual Machines Research Project: Evaluation of Verification Technologies for VMs.” SEI Special Report.

D-19. Vestal, S. (2007). Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 239–243.

D-20. Joseph, M., Pandaya, P. (1986). Finding response times in a real-time system. *The Computer Journal*, *29*(5), 390–395.

D-21. Baruah, S.K., Li, H., Stougie, L. (2010). Towards the Design of Certifiable Mixed-criticality Systems. *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 13–22.

D-22. Niz, D.D., Lakshmanan, K., Rajkumar, R. (2009). On the Scheduling of Mixed-Criticality Real-Time Task Sets. *2009 30th IEEE Real-Time Systems Symposium*, 291–300.

D-23. Andersson, B., Chaki, S., and de Niz, Dionisio. (2016). “FAA Virtual Machines Research Project: Survey of Literature Related to Virtual Machines.” SEI Special Report.

D-24. Baruah S.K., Bonifaci V., D’Angelo G., Marchetti-Spaccamela A., van der Ster S., Stougie L. (2011) Mixed-Criticality Scheduling of Sporadic Task Systems. In: Demetrescu C., Halldórsson M.M. (eds) Algorithms – ESA 2011. ESA 2011. Lecture Notes in Computer Science, vol 6942. Springer, Berlin, Heidelberg.

D-25. RTCA Inc. (2011). Software Considerations in Airborne Systems and Equipment Certification. (RTCA DO-178C).

D-26. Lakshmanan, K., Niz, D.D., Rajkumar, R., & Moreno, G.A. (2010). Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems. *2010 IEEE 30th International Conference on Distributed Computing Systems*, 169–178.

D-27. Niz, D.D., Phan, L.T. (2014). Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 111–122.

D-28. De Niz, D., Wrage, L., Rowe, A., and Rajkumar, R. (2014). Utility-Based Resource Overbooking for Cyber-Physical Systems. *ACM Transactions on Embedded Computing Systems, 13*(5s), 217–226.

D-29. Graham, R. L. (1969). Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics, 17*(2), 416–429.

D-30. Andersson, B. and Jonsson, J. (2002). *Preemptive Multiprocessor Scheduling Anomalies*. Proceedings from the IEEE International Parallel and Distributed Processing Symposium, Ft Lauderdale, FL.

D-31. Lundqvist, T., & Stenström, P. (1999). Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99).*

D-32. Baruah, S.K., & Burns, A. (2006). Sustainable Scheduling Analysis. *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 159–168.

D-33. Ha, R. and Liu, J. W. S. (1994). *Validating timing constraints in multiprocessor and distributed real-time systems.* 14th International Conference on Distributed Computing Systems, Pozman.

D-34. Vestal, S. (1994). Fixed-Priority Sensitivity Analysis for Linear Compute Time Models. IEEE Transactions on Software Engineering, 20(4), 308–317.

D-35. Punnekkat S., Davis R., Burns A. (1997) Sensitivity analysis of real-time task sets. In: Shyamasundar R.K., Ueda K. (eds) *Advances in Computing Science — ASIAN'97*. ASIAN 1997. Lecture Notes in Computer Science, vol 1345. Springer, Berlin, Heidelberg.

D-36. Bini, E., Natale, M.D., Buttazzo, G.C. (2008). Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems 39*(1–3), 5–30.

D-37. Jones, C.B. (1983). Specification and design of (parallel) programs. In: Mason REA (ed) *Proceedings of the 9th IFIP world congress*. Information Processing, vol 83, Paris, France, September 1983, pp 321–332.

D-38. Pnueli A. (1985) In Transition From Global to Modular Temporal Reasoning about Programs. In: Apt K.R. (eds) *Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol 13*. Springer, Berlin, Heidelberg.

D-39. Hoare, C. A. R. (1978). Communicating Sequential Processes. Commun. *ACM*, *21*(8), 666–677.

D-40. Barringer, H., Giannakopoulou, D., Pasareanu, C.S. (2003) Proof rules for automated compositional verification. In *Proceedings of the 2nd workshop on specification and verification of component based systems (SAVCBS '03)*, Helsinki, Finland, Iowa State University, Ames, (14–21).

D-41. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S. (2003). Learning Assumptions for Compositional Verification. *TACAS*. 331–346

D-42. Chaki, S., Strichman, O. (2008). Three optimizations for Assume-Guarantee reasoning with L*. *Formal Methods in System Design*, *32*(3), 267–284.

D-43. Sha, L., Rajkumar, R., Lehoczky, J.P. (1990). Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, *39*(9), 1175–1185.

D-44. Lakshmanan, K., Niz, D.D., Rajkumar, R. (2011). Mixed-Criticality Task Synchronization in Zero-Slack Scheduling. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 47–56.

D-45. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D. (2009). A Formal Architecture Pattern for Real-Time Distributed Systems. *2009 30th IEEE Real-Time Systems Symposium*, 161–170.

D-46. Nam, M., Sha, L., Chaki, S., & Kim, C. (2014). Applying software model checking to PALS systems. *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 5B4-1–5B4-14.

D-47. Kim, H., Niz, D.D., Andersson, B., Klein, M.H., Mutlu, O., Rajkumar, R. (2016). Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, *52*(3), 356–395.

D-48. Betti, E., Bak, S., Pellizzoni, R., Caccamo, M., Sha, L. (2013). Real-Time I/O Management System with COTS Peripherals. *IEEE Transactions on Computers*, *62*(1), 45–58.

D-49. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., & Sha, L. (2013). MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 55–64.

D-50. Graf, S., Saïdi, H. (1997). Construction of Abstract State Graphs with PVS. *CAV*. 72–83.

APPENDIX E—SINGLE-TO-MULTICORE PORTABILITY OF ASSURANCE DATA

E.1 INTRODUCTION

The notion of a multiprocessor implemented on a single chip is now more than 20 years old in academic circles [E-1, E-2]. A multiprocessor implemented on a single chip became known as a chip multiprocessor. With later commercialization, it become known as a multicore processor or multicore. Today, in many contexts, multicore processors are the norm; it is now more than 10 years since multicore processors became mainstream.

Given this context, the use of multicore processors in avionics has received increasing interest. There are two reasons for this: commercial availability and performance. With respect to commercial availability—single-core chips are simply not available from many chip vendors; if buying processor chips from such a vendor, then a multicore chip is the only option. With respect to performance—multicore processors offer advantages over single-core processors. These advantages include: 1) the potential for parallel execution of threads in multicore processors, and 2) lower power consumption and lower thermal dissipation, therefore reducing the need for advanced cooling and power generation. The potential for parallel execution is particularly helpful for software systems that are already multithreaded; this is typically the case for avionics. Reducing the need for advanced cooling and power generation is important for application domains in which size, weight, and power are important; this is also typically the case for avionics.

This increasing interest in using multicore processors in avionics raises the question of how to certify aircraft that use multicore processors. This document discusses this question—particularly what can go wrong when porting software that was originally developed for a single-core processor and now executing on a multicore processor.

E.1.1 SCOPE AND LIMITATIONS

Dealing with single-event upset (SEU) is important in avionics and this issue is more serious in multicore chips [E-3]. However, because this report focuses on software, SEU issues and Error Correcting Codes will not be discussed.

Hardware also needs to be approved. Often, an applicant gets credit for “service experience” on using a certain chip. Clearly, if a given software system has very long service experience on a given single processor system and now this software is ported to a multicore chip, it is natural to ask whether this multicore chip has the same service history with this software, and if the answer is no, it is natural to ask how much trust there can be in this new multicore chip. Because this is a hardware issue, this will not be discussed. For such issues, the reader may find [E-4] valuable.

The remainder of this document is organized as follows: Section 2 discusses timing correctness, section 3 discusses logical correctness, and section 4 presents conclusions.

E.2. TIMING

Section E 2.1 will revisit scheduling theory for a single processor. Similar material is mentioned in previous FAA reports, specifically “Fixed-Priority Scheduling” in “Assurance Issues on VMs in Avionics Systems” (appendix B, section 2.1.2.5) and “Reservations” in “Survey of Literature

Related to Virtual Machines" (appendix A, section 4.1.1). The scheduling theory is revisited for the reader to understand the remainder of this section without having to read the other reports.

Section E 2.2 shows that the independence of execution time in a multicore system is broken. Section E 2.3 shows the importance of the problem and ongoing efforts. Section 2.4 discusses one of the previously used ideas (time partitioning with ARINC 653) and how previously used synchronization mechanisms can fail/perform poorly when used with multicore processors. Tables E-1 through E-8 present the observations in the literature that in a multicore procecessor where execution time depend on other processors. Each table has a relevance to the section that follows.

Table E-1. Observations set 1 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-5]	Multiprocessor	"The copy-rate decreases to 50% of the value in the single CPU if only 2 memory banks are available."
		"We have clearly demonstrated the consequences of memory preemption on a SUN E3000 server with 4 CPUs, where the execution speed of a video-conferencing application running on a dedicated CPU drops from 25 to 20 frames per second if the remaining CPU demands a lot of megabytes per second (see FIG. 1.1.).
[E-6]	Single-core + I/O	"Summarizing Measurement Results: … We consider the slowdown of this application as the upper bound … For our machine, we determine an upper bound value of 1.49."
[E-7]	Processor and I/O	"…the interference between cache activity and I/O traffic generated by COTS peripherals can unpredictably slow down a real-time task by 44%."

E.2.1 SCHEDULING THEORY REVISITED

When describing and verifying real-time software, it is common to use the notion of a task. In this context, a task is an entity that generates a (potentially infinite) sequence of jobs in which each is described with an arrival time, a deadline, and its potential consumption of resources. A task can be used to model a thread in an operating system. However, a task can also be used to model other types of execution like periodic execution of an interrupt service routine.

On a system with a single-core processor, the resource consumption is typically described with a worst-case execution time (WCET). For example, it is common to describe a software system as a set of tasks for which task i is characterized by T_i, D_i, and C_i. Here, T_i denotes the minimum inter-arrival time of task i; D_i denotes the relative deadline of task i; and C_i denotes the WCET of task i. The interpretation of these parameters is that task i generates a sequence of jobs, the arrival time of these jobs is separated by at least T_i time units, and each job must have performed a certain number of units of execution (at most C_i) at most D_i time units after its arrival. This model has been very useful because it can describe a feedback-control system with periodic sampling (time-triggered arrivals), and it can also describe alarm systems (a job arrives because of a certain condition; e.g., the physical world is in a bad state). In this model, it is common to talk about the response time of a job—it is the time from when the job arrived until when it finished execution. It is also common to talk about the response time (sometimes called WCET) of task *i* and denote it R_i. This is a number such that 1) for all possible jobs that the system can generate, for each job of task *i*, the response time of this job is at most R_i, and 2) there is a schedule that the system can generate and for this schedule, there is a job of task *i* such that the response time of the job is equal to R_i. Clearly, if R_i could be computed for each task and then checked for all tasks, and it holds that $R_i \leq D_i$, it is sure that all deadlines are met. For this reason, researchers have developed runtime schedulers and analysis techniques that achieve this.

Researchers have given particular attention to systems in which each task is assigned a priority (a number) and then at runtime, among the tasks with a job that is ready for execution, the task that is selected to execute is the one with the highest priority. This type of scheduling is known as fixed-priority scheduling. The reason for its popularity is that it is supported in many real-time operating systems.

Two common ways of assigning priorities include: 1) rate monotonic (RM), and 2) deadline monotonic (DM). With RM, the priority of a task monotonically increases with its arrival rate (and its arrival rate is the inverse of its T-parameter). With DM, the priority of a task monotonically increases with the tightness of its deadline (the deadline is tighter if its *D*-parameter is smaller).

The research literature also offers methods for proving that deadlines will be met for all possible schedules that a system can generate. These methods are called schedulability tests. A schedulability test takes as input a set of parameters of the task set (e.g., *T*, *D*, *C* of tasks) and outputs a Boolean value. If it outputs true, then it is guaranteed that all deadlines will be met. For fixed-priority scheduling, R_i can be computed for each task τ_i and compared against D_i. Clearly, R_i depends on all tasks with higher and equal priority as task τ_i. For this reason, it is typical to let *hep*(*i*) denote the set of tasks with higher or equal priority as task τ_i. With this notation, R_i can be computed as the smallest R_i that satisfies the following equation:

$$R_i = \sum_{j \in hep(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{E-1}$$

Typically, this equation is solved with fixed-point iteration. It is done as follows: Let R_i^k denote the value of the *k*:th iteration. Then R_i^k is computed as follows:

$$R_i^1 = C_i \tag{E-2}$$

and

$$R_i^{k+1} = \sum_{j \in hep(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil * C_j \tag{E-3}$$

When convergence is obtained (i.e., $R_i^{k+1} = R_i^k$) then R_i has been found. If one of the R_i^k values exceeds D_i, then there is a deadline miss and the iterative procedure can be terminated. This termination condition is useful because there are task sets for which this iterative procedure does not converge; this happens for overloaded systems (e.g., $T_1 = 1$, $D_1 = 1$, $C_1 = 1$, $T_2 = 2$, $D_2 = 2$, $C_2 = 1$).

Table E-2. Observations set 2 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-8]	Single processor	“the utilization increment [because of cache eviction] can be as high as 13%”
[E-9]	dual core + I/O	“…measured a WCET increase 2.96 times for the task.”
[E-3]	multicore	About cache sharing: “If the data set is smaller than the L2 cache visible to a core and the L2 cache is shared (Intel Processor), the worst case performance loss through the second core depends on the data set size and it between 30% and 95% for write operations and 19% and 92% for read accesses.”
		About cache coherency: “On the AMD processor, the performance loss is 99% on small data sets and it moves to 50% for large data sets.”
		About data buses: “If the cores operate on a data set which is so large that the caches have no effect, the performance drops down to 50% if both cores are active.”

E.2.2 THE INDEPENDENCE OF WORST-CASE EXECUTION TIME IS BROKEN ON A MULTICORE PROCESSOR

Unfortunately, the results described so far apply only to single-core processors. On a multicore processor, the execution time of a job depends on jobs executing on other processors. Specifically, a job can have longer execution time because of execution of jobs on other processors, and this can cause a deadline miss. This will be shown with a simple example.

Example 1: Consider three tasks and two processors. Task 1 is assigned to processor 1. Task 2 and Task 3 are assigned to processor 2. The tasks are characterized with the parameters $T_1 = 1$, $D_1 = 1$, $C_1 = 0.5$, $T_2 = 1$, $D_2 = 1$, $C_2 = 0.5$, $T_3 = 1$, $D_3 = 1$, $C_3 = 0.5$ (i.e., they all have the same *T, D* and *C* parameter). Priorities are assigned so that task 2 has higher priority than task 3. Then the response times of these tasks can be computed; this yields $R_1 = 0.5$, $R_2 = 0.5$, $R_3 = 1$, and therefore all deadlines are met. This is shown in figure E-1. However, this reasoning assumed that the system can be analyzed as two single-core systems. In reality, the execution times may be larger. For example, it may happen that $C_1 = 0.6$ and $C_2 = 0.6$ because execution times may increase in a multicore processor; then task 3 misses its deadline.

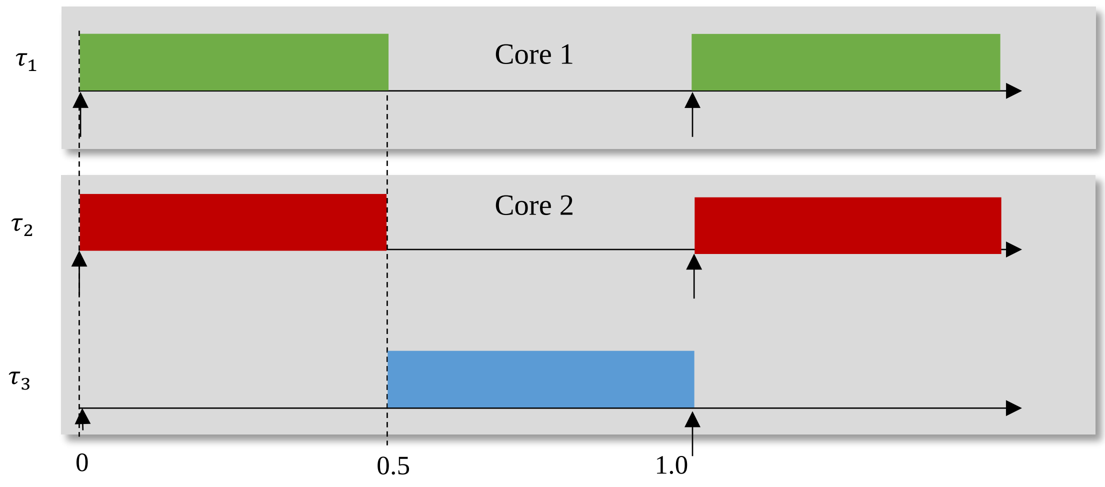


Figure E-1. Example of a task set that meets its deadline if $C_1 = 0.5$ and $C_2 = 0.5$ but, because of multicore, the execution time can become larger ($C_1 = 0.6$, $C_2 = 0.6$), and then task τ_3 would miss its deadline

Therefore, when verifying timing of software on a multicore processor, it is important to be aware that the execution time of one job may depend on execution of other jobs on other processors. The independence of WCET is broken. This increase in execution time has multiple causes; one is that one job can evict a cache block that another job has fetched.

Table E-3. Observations set 3 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-10]	Multithreaded processor	"We also observe that, in general, the detected slowdown is quite high (up to 15.3x)"
[E-11]	Multicore	"…the worst-case execution time (WCET) can be multiple times slower than the same application running on a single-core…"
		"A major result demonstrated by the measurements is the substantial impact that concurrently active devices may have on a single devices' performance, in terms of storage type instructions. The influence could be of a factor from approximately 1.6 for L3 SRAM and 5.1 when accessing DDR memory."

E.2.3 IMPORTANCE OF THE PROBLEM

It is well established through the scientific literature that in a multicore processor, the execution time of one job may depend on jobs on other processors. Tables E-1–E-8 show evidence from the research literature on this. These tables show results from researchers' measurements on execution time and how they are affected by execution of jobs on other processors. These tables also show the experimental results when using techniques to reduce this effect.

The degree to which the execution time of one job depends on other jobs on other processors depends on the context, on the processor architecture, and on the application behavior. For example, if jobs are less memory intensive, contention on the memory system is not too severe, and this may cause the execution time of one job to be only marginally dependent on other jobs on other processors. From the tables, it can be seen, however, that for certain settings, the dependence of the execution time on other jobs can be severe. [E-12] (mentioned in table E-8) observed that the execution time could be 103 times larger.

As already mentioned, in a multicore processor, the execution time of one program can be impacted by execution of a program on another processor core because the programs may share resources in the memory system. This is often referred to as memory interference channels. Examples of memory interference channels are:

1. One program fetches a cache block to the cache and later accesses could operate faster if accessing the same memory address; but with multicores, another processor core could fetch some other data that maps to the same cache set and, therefore, evict the cache block.

2. The execution of one program is delayed because it needs to use the memory bus—the bus between the memory controller and the main memory (DRAM). The delay is caused by another program using the memory bus.

3. One program opens a memory row in a memory bank (in DRAM) to read a word from it. If additional accesses are to the same row, these additional accesses are performed faster. However, it could happen that another program on another processor core requests to access another row in the same memory bank, and then it is necessary to close the current row and open a new row. This could cause an extra delay to the former program.

4. A memory controller can reorder memory requests. This reordering is typically implemented in hardware, and it aims to speed up execution of programs in general. Unfortunately, this reordering does not account for the deadlines of respective programs, and it is hard to analyze this delay. Typically, this reordering is done by 1) prioritizing read operations over write operations (because if a read operation is delayed then a processor may stall), 2) prioritizing operations that perform data movement in the same direction as previous operations (because there is an extra overhead in changing the directionality of the memory bus), and 3) prioritizing memory operations to memory addresses that are in a bank with a currently open row.

5. There is contention on memory ports (e.g., the memory port of on-chip SRAM used in cache memories).

6. There are additional memory operations caused by execution of the program that are not visible in a program's executable code. For example, consider a load instruction that is already in memory and experiences a cache miss when fetching data; if this instruction generates a TLB miss, this instruction may require four memory accesses. Many processors use (stride) prefetching units that speculatively fetch data ahead of the need for the data.

7. There are additional memory operations that are not caused by execution of the program. These include memory operations from DMA operations generated by I/O devices.

Table E-4. Observations set 4 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-13]	Multicore	"Experimental results show that, in the considered benchmarks, eliminating the interference of the last level cache can lead up to a 250% improvement in the execution time."
[E-14]	Multicore	"proper shared cache management can enable significant WCET reductions; on our test platform, observed WCETs were reduced up to almost five-fold."
[E-15]	Multicore	"For instance, the execution time of PS.streamcluster is increased by 60% under the no-bank-protection approach, but the increase is only 12% under our combined cache and bank coloring approach."

Table E-5. Observations set 5 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-16]	Multicore	"Figure 5(b) illustrates the response times when all cores share the same bank partition. With bank sharing, we observed up to 12x of response time increase in the target platform. "
[E-17]	Multicore	"As can be seen in Figure 7 (H and H+P), the cache sensitive workload *gobmk* experienced a performance gain of as much as 13% under the interference of a heavy background workload with page coloring."

Table E-6. Observations set 6 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-18]	Multicore	"The memory latency can increase more than 15-fold on 8-cores for a e500mc processor as witnessed by the statement: Table 1 shows the memory access latencies for read and write operations with increasing number of interfering cores. TABLE 1. P4080 MEMORY ACCESS LATENCIES FOR INCREASING NUMBER OF CONCURRENT CORES. LATENCIES USE FOR EVALUATION ARE MARKED BOLD Latency (cycles)"

Cores	1	2	3	4	5	6	7	8
Read	**41**	75	171	269	296	439	460	604
Write	39	**164**	**245**	**463**	**517**	**737**	**784**	**1007**

There are additional reasons why the execution time of one job depends on execution of other jobs on other processors.

8. Consider two jobs that execute on different virtual cores (as part of hyperthreading) but that share functional units (for example ALUs) or a floating-point unit. If both jobs request use of the same type of resource, only one job might be granted the resource, and the other job would have to wait. This sharing of resources within a processor core could cause the execution time of one job to depend on another job.

9. Consider two jobs that execute on different processor cores in a system that uses dynamic thermal management. In such a system, a processor can execute at high speed if all other processors are idle. If one job executes first, and this job is the only job in the system, it can execute at high speed. However, if the other job arrives and now executes on the other processor, the processor speed would be reduced (so as not to overheat the processor). Therefore, thermal management can cause the execution time of one job to depend on another job.

These examples show that if a job executes in a multicore system, it can experience larger—sometimes much larger—execution time than it would have experienced if executing in a single-core system. Tables E-1–E-8 show significant documentary evidence for this. Therefore, researchers have developed techniques (see reference list at the end of this document) and certification bodies have produced a position paper [E-19] because DO-178C does not explicitly offer guidance on multicores. Researchers have also collaborated to provide recommendations [E-20]. These two documents and their context will now be discussed.

As reported by [E-21], the aviation community has 1) formed a "Multi-Core for Avionics" working group, 2) organized a first workshop in January 2013 at EASA, 3) organized a second workshop in Cologne in July 2014, 4) produced a "Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems" (FAA), 5) started producing a document, "Identification of Issues with Multi-core Processors" (FAA), and 6) produced a report "MULCORS–Use of

Multicore Processors in Airborne Systems” (EASA). Some of these interactions led to the CAST-32 position paper that later (in November 2016) led to the CAST-32A position paper coordinated among certification authorities in North and South America, Europe, and Asia. The CAST-32A position paper will now be discussed.

The CAST-32A position paper [E-19] discusses topics that can impact safety in avionics. It also gives rationale for why these are topics of concern and discusses objectives to address the concerns. The CAST-32A position paper states that interference channels should be identified, does not recommend schedulers that allow tasks to migrate arbitrarily between processors at runtime (called global scheduling in the real-time systems research literature), and does not recommend the use of hyperthreading. It is noteworthy that some hypervisors (e.g., Xen and RT-Xen, one of the most popular hypervisors) use global scheduling. The paper recommends the use of a safety net. There are many reasons for this; one is that multicore processors may have unintended functionality and produce unexpected behavior. The paper does not give any example of this. It is noteworthy that it has been reported that the Intel Skylake processor can freeze[11] because of a defect in the hardware. The paper points out the risks in using simulation for timing verification. Specifically, it states that: “Because interference between applications occurs via the proprietary internal mechanisms of an MCP, any simulation of those mechanisms is less likely to be representative in terms of functionality or execution time than testing conducted on the target MCP.”

The position paper “Minimal Multicore Avionics Certification Guidance” [E-20] is written by academics and industry participants within the area of real-time systems. It came out of the First TCRTS Workshop on Certifiable Multicore Avionics Systems (CMAS) in 2015. The paper’s objective is to “identify necessary requirements on multicore avionics certification process that will maximize the freedom to develop innovative solutions without sacrificing quality and consistency of the certification process.” It introduces the notion of a core group and argues that certification of software within a core group should not depend on software in other core groups. Further, the paper argues that certification of multicore avionics should address interference channels and that “developers need to investigate the processor architecture and identify all the interference channels.” The paper points out that solutions to mitigate and analyze interference channels rely on data provided by chip manufacturers and that this information may be incomplete, contain errors, or have ambiguities.

[11] See https://arstechnica.com/gadgets/2016/01/intel-skylake-bug-causes-pcs-to-freeze-during-complex-workloads/ and https://communities.intel.com/thread/100321.

Table E-7. Observations set 7 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-22]	Multicore	"The difference of slowdown factors between the two tasks could be as large as factor of two (2.2x against 1.2x)"
		"As we increase 470.lbm's assigned memory bandwidth, however, performance of 462.libquantum gradually decreases; when the reserved bandwidth for 470.lbm is 2.0GB/s (i.e., 3.0GB/s aggregate bandwidth reservation), more than 40% IPC reduction is observed due to increased memory contention."
		"Note first that MemGuard-RO does not guarantee performance isolation anymore as 462.libquantum is 17% slower than the baseline. It is because the 2.4GB/s bandwidth cannot be guaranteed by the given memory system, causing additional queuing delay to the 462.libquantum."

Table E-8. Observations set 8 in the literature that in a multicore processor, execution time may depend on other processors

Paper	Context	Statement/Testimony
[E-12]	Multicore	"Without cache partitioning, a task can suffer up to 103X slowdown due to interference at the shared LLC."
[E-23]	Multicore	"Measurements we performed on a commercial multicore platform (Freescale P4080) revealed that a task's WCET can increase by as much as 600 percent when a task on one core runs with logically independent tasks in other cores."
[E-24]	Multicore	"When validating real-time constraints on an m-core platform, excessive analysis pessimism can effectively negate the processing capacity of the additional m-1 cores so that only "one core's worth" of capacity is available."
		"Obs. 1. Providing LLC isolation reduced WCETs by up to 277% for the uB task and by up to 242% for the Matrix program."

E.2.4 WHY PORTING SOFTWARE TO A MULTICORE PROCESSOR IS CHALLENGING

It is tempting to believe that every task set that is schedulable on a single processor is also schedulable on a multiprocessor. If this were true, porting one software system to a multicore would be trivial. Unfortunately, as shown by the example, this is not true.

Example 2: Consider two tasks and a single processor. Assume that the tasks are characterized as $T_1 = \infty$, $D_1 = 1$, $C_1 = 0.6$, $T_2 = \infty$, $D_2 = 2$, $C_2 = 1$. Priorities can be assigned using deadline monotonic, and this shows that task 1 has higher priority and task 2 has lower priority. The worst-case response time can then be computed, and this computes the worst-case response time of task 1 as 0.6 and the worst-case response time of task 2 as 1.6. Port this system to a multicore system with two processor cores, assign task 1 to processor 1, and assign task 2 to processor 2. If there is no co-runner interference because of sharing of resources in the memory system, the response times are as follows: the response time of task 1 is 0.6, and the response time of task 2 is 1; therefore, all deadlines are met. However, with co-runner interference, it may happen that a task executes with the speed 1/2 when it has a co-runner, and a task executes with speed 1 when it has no co-runner. In that case, if both tasks arrive at time zero, the schedule shown in figure E-2 is generated. In this schedule, during the time interval [0,1], task 1 and task 2 execute both with speed 1/2. At time 1, the deadline of task 1 expires without finishing. During the time interval [1,1.2], task 1 and task 2 both execute with speed1/2. At time 1.2, task 1 finishes. During the time interval [1.2,1.6], task 2 executes with speed 1. It can be seen that task 1 misses its deadline because its deadline is at time 1, but it finishes at time 1.2. This example shows that a software system that meets all its deadlines when executing on a single-core system cannot be assumed to fulfill all timing requirements when this software is ported to a multicore processor.

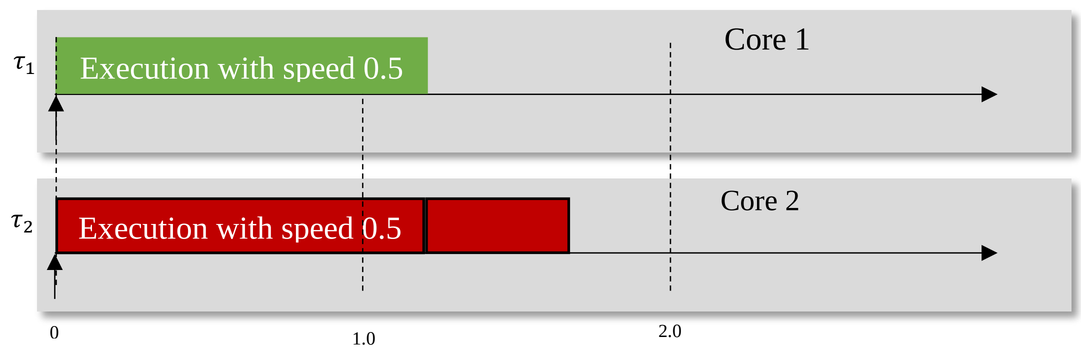


Figure E-2. Example of a task set that meets its deadline on a single-core system, but executed here on a multicore system and misses its deadline

E.2.5 TEMPORAL PARTITIONING

In single-core processors, temporal partitioning prevents a task from delaying another task in an unpredictable manner. More specifically, a temporal partition is defined with a set of temporal parameters, also known as a temporal interface.

The temporal interface parameters typically vary depending on the underlying runtime infrastructure but are always based on the worst-case time a task needs to execute to finish before the next (periodic) activation of the task (e.g., due to the arrival of another frame in a video-processing application). More specifically, it has two types of parameters: 1) the amount of CPU time the task needs to finish its periodic execution, and 2) the interval within which such CPU time must be delivered to meet its deadline.

Example 3: A video player displaying 20 frames per second (FPS) needs to process one frame every 50 ms. If the frame processing takes 10 ms in the worst case, the interface should define how 10 ms will be delivered within an interval of 50 ms, assuming the deadline is equal to the end of the period (i.e., it needs to finish processing a frame before it is time to process the next one).

The interface defined by a time-slot-based partitioning consists of an interval known as a major frame and a set of time slots (subintervals) within this major frame[12]:

$$(M, \langle s_1, \dots, s_n \rangle) \tag{E-4}$$

where M is the size of the major frame and s_i is the size of the ith interval assuming that $M = \sum_{i=1}^{n} s_i$. Then partitions are given a set of time slots in which their task(s) execute.

Partition System 1: For instance, a system with five time slots and three time slot-based partitions may be defined (in milliseconds) as:

$$(M = 100, \langle s_1 = 20, s_2 = 10, s_3 = 30, s_4 = 20, s_5 = 20 \rangle) \tag{E-5}$$

[12] Different variations of this definition are used by different vendors and standards.

with three partitions, $P_1 = \{1,5\}$, $P_2 = \{4\}$, and $P_3 = \{2,3\}$. Then, the video player in example 3 can be guaranteed to execute at least 10 ms every 50 ms in partitions[13] P_1or P_3, but not in P_2. This is because even though P_2 has enough execution time allocated (20 ms) to support two executions of the video player, this allocation is not spread into two subintervals of 50 ms as required by the application.

Processing servers and resource reservations [E-25, E-20, E-21, E-26, E-27] allow the definition of a simpler interface with only two parameters—the WCET C_i and the period of execution T_i. For the video player, this interface would be as simple as $C_1 = 10ms$, $T_1 = 50ms$. For completeness, this interface also includes a deadline that can be shorter than the period leading to the three-parameter interface: (C_i, T_i, D_i).

E.2.5.1 WCET Dependence

Timing interfaces assume that the WCET of a task does not change if other tasks running in other temporal partitions are run in the system. This assumption is preserved in a single-core processor where the partitions take turns to the only core. However, in multicore processors, multiple partitions can be running in real concurrency in different cores. Because the cores of the processor share hardware resources, the execution of one task in one core can delay the execution of another task in another core. This delay invalidates the WCET independence assumption of the timing interfaces, as we have seen in section 2.2.

A number of research projects [E-9, E-12, E-23, E-16, E-15] have presented experiments that show the WCET dependence on tasks running on other cores (i.e., co-runners). For example, figure E-3 shows an experiment from [E-16] in which it is possible to observe the sensitivity of different application benchmarks [E-28] to the execution of co-runner tasks. Four key observations are worth highlighting:

1. Different applications have different sensitivity to the interference from other cores. In particular, in figure E-3, the swaptions benchmark shows no sensitivity to co-runners, with an execution time that remains unchanged no matter how many tasks run in the other cores. In contrast, the streamcluster benchmark exhibits extreme sensitivity to the interference, increasing its execution time five-, eight-, and 12-fold when the benchmark is run with one, two, and three co-runners.

   The intuition behind this difference is that less sensitive applications access memory less frequently (execute instructions that mostly perform computations), or the memory that they access is small enough to fit in the private cache (as opposed to the cache shared between cores) such that they rarely access main memory. In contract, highly sensitive applications have frequent accesses to memory from a region that cannot fit in the private cache.

2. The execution time of a task increases with the number of co-runners.

[13] The starting of the period of the task must be appropriately synchronized with the start of the major frame.

3. The increase in execution time depends on the code of the co-runner. This is not obvious in figure E-3, in which all the co-runners are memory-intensive tasks, but it is worth observing that if a sensitive task has only co-runners with very few memory accesses, the task will not be affected.

4. As a consequence of the previous point, the increase in execution time will depend on: 1) whether a task is restricted to run always on the same core (partitioned scheduling) or allow it to run on any core (global scheduling), and 2) the cores on which the tasks are run (if partitioned scheduling is used).

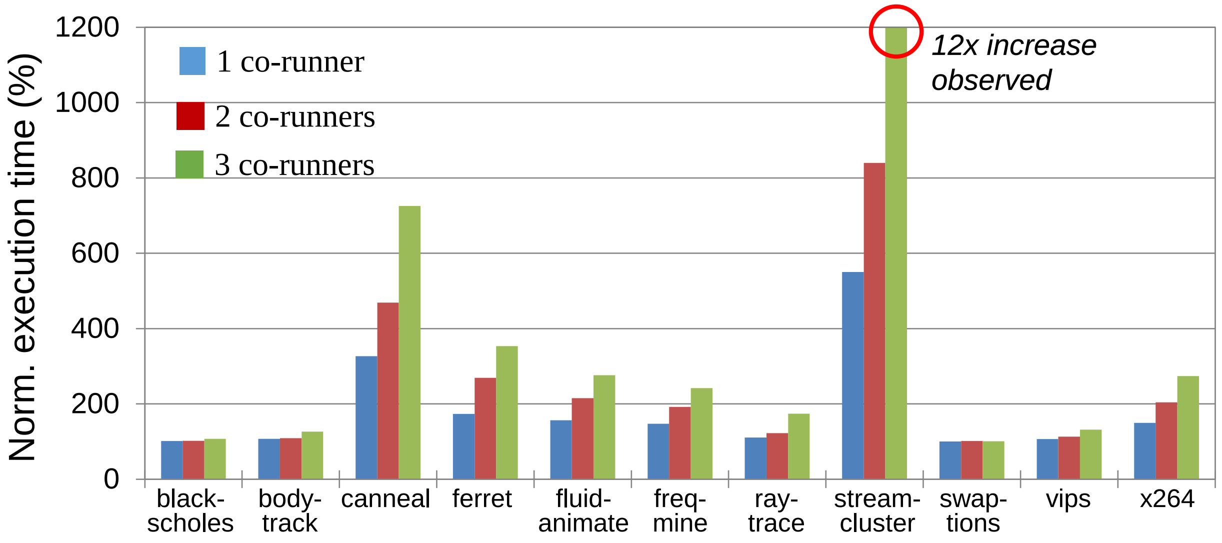


Figure E-3. WCET dependence

The observations about figure E-3 clearly highlight WCET dependence on co-runners. This breaks the temporal interfaces that rely on the WCET independence. As a result, virtualization technologies that aim at providing temporal isolation and rely on independent WCET interfaces are affected by multicore co-runner interference. This means that the timing parameters of their interface are not portable from single-core to multicore processors.

For example, consider again the video player from example 3 with the Partitioned System 1. For the sake of the example, assume that the video player has the same sensitivity as the streamcluster benchmark and that the Partitioned System 1 is used in core 1 of a quad-core processor. Now assume that the same co-runners from the experiment in figure E-3 are run. This means that with one co-runner, the WCET will go to (at least) 50 ms, reaching 100% utilization ($\frac{50}{50}$), and it will not fit in any of the partitions from Partitioned System 1. The only possibility is to run it in the three partitions using 100% of the processor. Clearly, with two and three co-runners, the WCET jumps to 80 ms and 120 ms with utilizations of 160% and 240%, respectively, which cannot fit in one core.

E.2.5.2 Enhanced Partitioning Techniques for Multicore Processors

New partitioning techniques for multicore processors have been developed in the research community. These include techniques for cache partitioning [E-8, E-12, E-29, E-30, E-31], memory-bank partitioning [E-8, E-16, E-32, E-33], and the memory bus [E-22, E-31, E-34]. However, in this case, the interface must include new parameters. For example, when the cache is partitioned, a task is restricted to a smaller region of the cache. As a result, its WCET will increase because of the additional accesses to memory necessary to fetch the data that were not able to fit in the cache. The implication of this increase is that our interface would need to incorporate additional parameters that describe memory accesses, cache and memory partition sizes, and their interrelationships.

An example of an enhanced timing interface is presented in [E-16], in which the number of memory accesses H_i is included leading to the interface: (C_i, T_i, D_i, H_i). This interface is intended only to enable memory-bank partitioning. Another example that takes into account cache partitions is presented in [E-15], in which a timing interface is defined as: $(T_i, \{C_i^1, C_i^2, \dots, C_i^k\}, M_i)$. In this case, each of the C_i^j specifies the execution time when j cache partitions (i.e., colors) have been assigned to the task, and M_i specifies memory-size requirement. Clearly, each C_i^j represents one parameter that combines two: the execution time and the amount of cache.

So far, the timing interfaces have been discussed as an interface to a single task and not necessarily a collection of tasks as it happens when using virtual machines (VMs). Clearly, because the individual task interface is not portable, the combined interface (for a task set) is also not portable.

E.2.6 SYNCHRONIZATION

When tasks synchronize to implement critical sections and prevent race conditions, they delay each other. This delay creates three problems when tasks are scheduled under fixed-priority scheduling. First, a high-priority task may be delayed waiting for a mutex that is locked by a lower priority task, creating what is known as priority inversion. Second, the lower priority task may be preempted by medium-priority tasks multiple times, enlarging the delay (e.g., blocking) that the high-priority task suffers. This problem is known as unbounded priority inversion. Finally, this blocking may modify the schedulability equations necessary to verify that a task meets its deadline.

Figure E-4 shows a Gantt chart with an unbounded priority inversion example. In this figure, the low-priority task arrives at time 0 and enters the critical section (black rectangle) at time 2, locking the mutex shared with the high-priority task (not shown). Then, the high-priority task preempts it at time 5 and executes until time 10 when it tries to lock the mutex. Because the mutex is locked, the high-priority task blocks waiting for the mutex to be released. At this time, the low-priority task resumes, but at time 15 is preempted by the medium-priority task. Then the medium-priority task runs up to time 32 when the low-priority task resumes inside its critical section, and it continues to execute until time 37 when it unlocks the mutex. At this time, the high-priority task is unblocked and resumes, locking the mutex, runs in the critical section up to time 40 and continues to run up to time 42. However, at time 40, its period, which is also its deadline, elapses, missing its deadline.

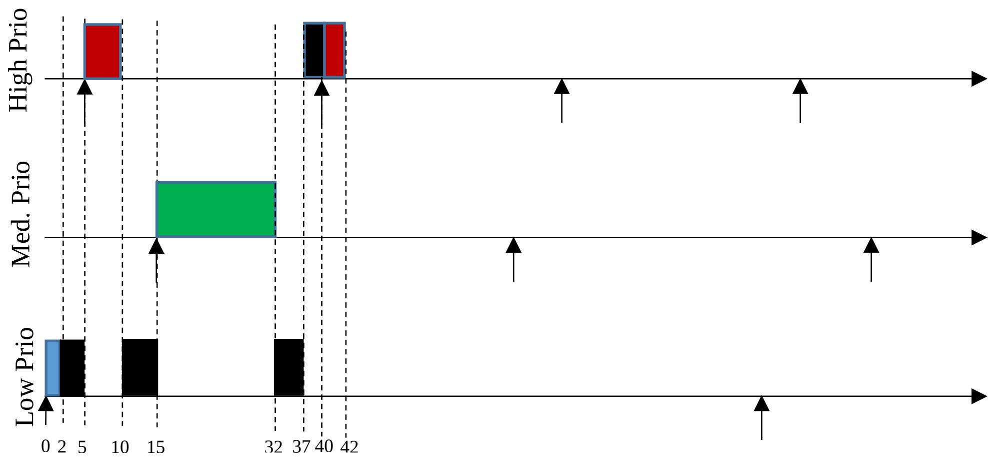


Figure E-4. Unbounded priority inversion

In [E-35], Sha et al. presented techniques to prevent the unbounded delays and calculate the delays imposed by the priority inversion to consider this delay in the schedulability equations. In particular, two protocols known as the Basic Priority Inheritance (PI) Protocol and the Priority Ceiling Protocol (PCP) were presented in [E-35]. These protocols are specifically designed to address the priority inversion in real-time systems; therefore, they will be referred to as real-time synchronization protocols. With the PI protocol, when a high-priority task tries to lock a mutex that is being held by a lower priority task, the lower priority task inherits the priority of the high-priority task. As a result, if a medium-priority task arrives, it will not be able to preempt the low-priority task that now is running with the priority of the high-priority task. Avoiding the medium-priority tasks preemptions limits the delay suffered by the high-priority task. With this protocol, it is possible to calculate the worst-case delay that a task may suffer when it locks a set of mutexes. In general terms, this calculation is the sum of the largest critical sections among all the lockers with lower priority for each of the mutexes[14].

Using PCP, the lower priority task that holds the mutex acquires the highest priority among the potential lockers as soon as a higher priority task tries to lock the mutex. This simplifies the calculation of the maximum delay as just the largest possible critical section among all the critical sections of the potential lockers with lower priority:

$$B_i = \max_{\forall k \wedge j \in lockers}(CS_j^k) \tag{E-6}$$

where CS_j^k is the WCET of the k critical section of locker j and $lockers$ is the set of potential lockers.

An additional property of PCP is that it prevents deadlocks given that, once one of the potential lockers locks the first mutex, no other locker can lock any other mutex because the first one cannot be preempted by any other locker because of the inherited priority.

[14] See [E-37] for a method to calculate this term.

The effect of the priority inheritance approach (both PI and PCP) is shown in figure E-5. In this case, as soon as the high-priority task tries to lock the mutex at time 10, the low-priority task inherits the high-priority task priority. This prevents the medium-priority task from preventing the low-priority task when it arrives at time 15. As a result, the low-priority task is able to release the mutex at time 25, allowing the high-priority task to resume and finish at time 29. It is worth noting that, in this case, both PI and PCP produce the same execution scenario, but they can produce different scenarios in another situation.

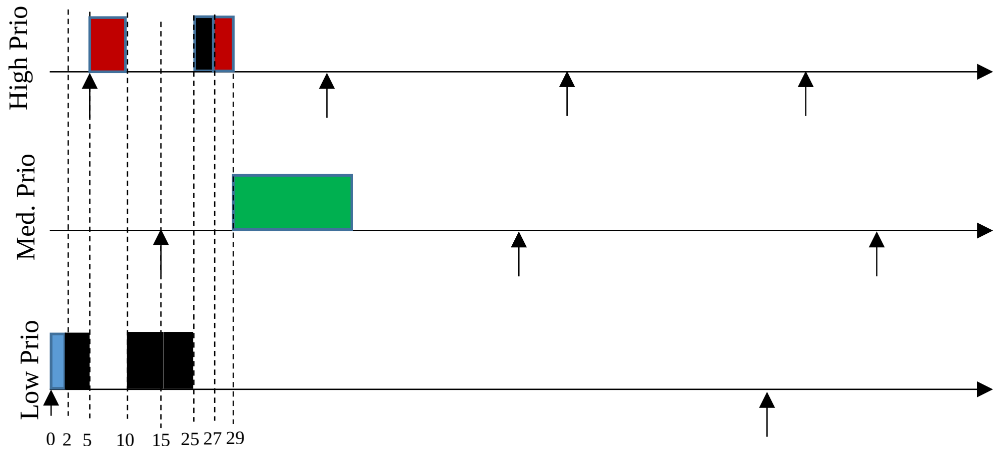


Figure E-5. Bounded priority inheritance with PI and PCP

E.2.6.1 Synchronization in Multicore Processors

In multicore processors, the synchronization problem has more dimensions to consider. The problems and current solutions will be discussed.

The concept of remote blocking will first be discussed. Remote blocking happens when a task τ_{trying} tries to lock a mutex that is being held by another task $\tau_{holding}$ in another core. In this case, using priority inheritance to prevent unbounded blocking does not always work because the task $\tau_{blocking}$ (originally the medium-priority task) may have a higher priority than task τ_{trying}. This is shown in figure E-6.

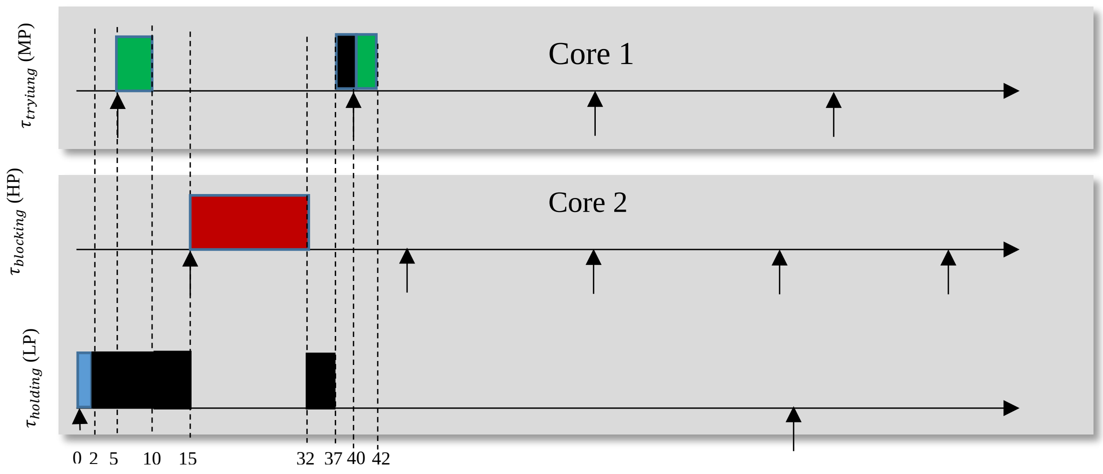


Figure E-6. Ineffective priority inheritance in remote blocking

To bound the remote blocking with priority inheritance, Rajkumar [E-36, E-37] introduced the Multiprocessor Priority Ceiling Protocol (MPCP) that uses the concept of global mutex. Global mutexes are assigned a priority ceiling from a special priority band for global priority inheritance that goes above all local priorities. This allows a task to obtain this global priority on locking the mutex and avoid being preempted by any task with local priorities that does not participate in global locking. Figure E-7 shows how the unbounded blocking in figure E-6 is solved.

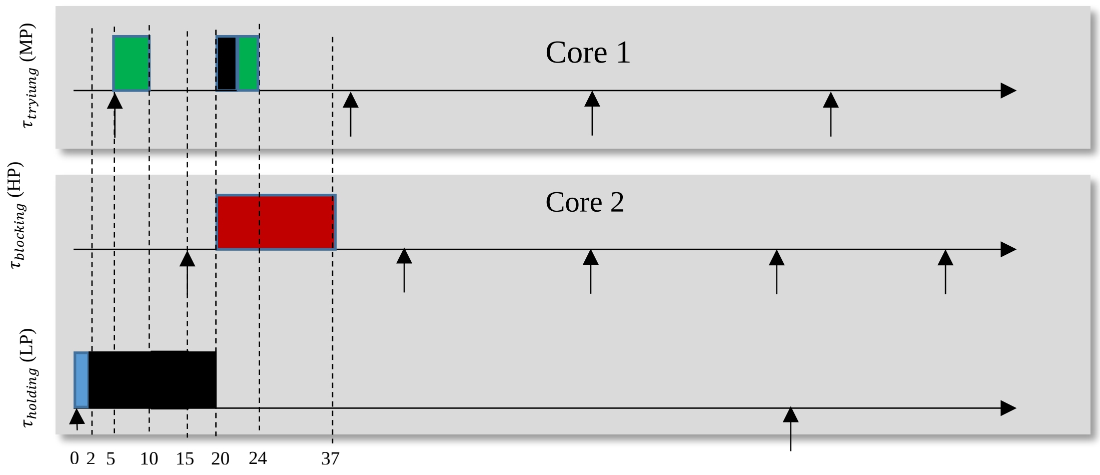


Figure E-7. Global priority inheritance under MPCP

However, not all problems are solved. Two problems, or perhaps drawbacks, remain. First, MPCP can no longer prevent deadlocks. To see this, consider two tasks τ_1 and τ_2 running in two different cores that lock two mutexes $m_1$and m_2. In this case, τ_1 first locks mutex $m_1$and obtains the global priority ceiling. However, because τ_2 is running in another core, it can continue to run. Now τ_2 locks m_2 and after that tries to lock m_1. It then suspends itself waiting for m_1 to be released. Meanwhile τ_1 continues to execute in the other core and then tries to lock m_2, suspending itself waiting for m_2 to be released. There is now a circular-wait characteristic of a deadlock.

Second, remote blocking can cause a processor to remain idle for some period. This can be seen in figure E-7 in core 1 between times 10 and 20 when τ_{trying} waits for $\tau_{holding}$ to release the lock, but because there is nothing else to run, the processor remains idle. This was explored in [E-38] in which new task-to-core assignment algorithms based on bin-packing were developed to reduce this idle time (utilization loss). The experiments in the paper showed a reduction of up to 50% in the wasted utilization.

The calculation of the blocking term also changes. In particular, it has to take into account the self-suspension behavior that remote blocking induces. See [E-37, E-38] for a further discussion on the topic.

With respect to synchronization, this analysis shows a lack of portability from a single-core to a multicore processor.

E.2.6.2 Synchronization Among Partitions

Ideally, synchronization between partitions should not be allowed. However, on occasion, it cannot be fully avoided. In some cases, two tasks do not need to achieve mutual exclusion but only need to share data in a manner that does not cause corruption. For such situations, lock-free data sharing can be used; see, for example, [E-39]. However, in other cases, it is important for a task to hold a resource under mutual exclusion. This is the case, for instance, when two video players are trying to use the screen or a screen manager like the X server in a Unix-based system. In this case, it would be possible to assign a different partition (or reservation) to each of the players to prevent them from interfering with each other. However, they would either need to lock the shared screen every time they would use it or request another task (e.g., the X server) to perform an action (modify a part of the screen) on the shared resource on their behalf. For resource reservations, one would think that it would be possible to use priority inheritance because they use rate-monotonic scheduling. However, this will create another problem because of the use of enforcement. Specifically, if a task $\tau_{holding}$ that is holding the mutex is enforced while it is running in its critical section and while another task τ_{trying} is trying to lock the mutex, the τ_{trying}'s blocking time will be enlarged because of the enforcement. Figure E-8 shows an enforcement at time 15.

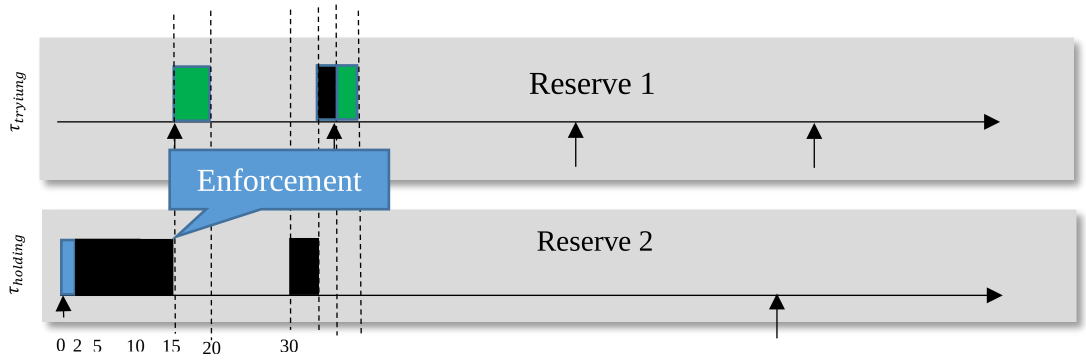


Figure E-8. Additional blocking due to reservation enforcement

To solve this issue, de Niz et al. [E-40] developed the basic reserve inheritance and the reserve-ceiling protocols that allow a task locking a mutex to inherit a reserve to prevent it from being enforced because of the exhaustion of its own reserve. More specifically, in the reserve-ceiling protocol, a reserve ceiling is assigned to the shared resource to be used only when such a resource is used (e.g., the X server in the example). Unfortunately, the budget of the single ceiling reserve that supports multiple lockers requires the budget of such a reserve to be the sum of all the critical sections using the shared resource in addition to the original reserve for each locker in which the critical sections were already accounted. Clearly, this leads to very pessimistic (low) utilization. To remove this pessimism, de Niz et al. developed the multi-reserve ceiling protocol that creates one reserve per locker, but the budget of the ceiling reserve is considered part of the original locker's reserve budget.

Once a resource is shared across partitions, the isolation property is lost. Instead, when a task overruns its budget, it is possible to contain the consequences of this overrun only to the group of tasks with which it shares resources.

It is clear that the assurance data coming from the analysis of synchronization protocols in single-core processors is not portable to multicore processors. This is the case not only for timing calculations but also for logical properties such as deadlock prevention that some synchronization protocols are able to offer (e.g., PCP) in single-core but cannot offer in multicore.

E.3. FUNCTIONAL CORRECTNESS

Porting assurance data for functional correctness is also nontrivial when software is migrated from single-core to multicore platforms. In this section, the main challenges that must be addressed during such a porting process are discussed. By functional correctness, it is meant that all software procedures eventually complete execution and produce the correct result. Therefore, violations of functional correctness typically take one of two forms—either some procedure fails to terminate, or it terminates but produces an incorrect result. Note that the focus is on procedures, because each individual procedure is still expected to terminate even though the overall software may not. For example, in the context of real-time software, although the overall task may not terminate, each periodic job must do so.

E.3.1 VIOLATION OF FUNCTIONAL CORRECTNESS IN SEQUENTIAL SETTING

Generally, violations of functional correctness can have different underlying reasons. For example, consider a procedure P that is purely sequential. In other words, P executes in a single thread that does not interact with any other thread. In such a situation, failure to terminate could be due to an infinite loop. Similarly, an incorrect result could be due to a logical error. Figure E-9 shows a C procedure (with a bug) that is supposed to compute the factorial of its argument, but has both kinds of problems:

```
int factorial (int n) {
 if (n == 0) return 1;
 else return n * factorial(n-2);
}
```

Figure E-9. Example program that violates functional correctness

Specifically, a call to factorial (3) will never terminate, and a call to factorial (4) will return an incorrect result of 8 instead of the correct result of 24. However, these types of bugs are typically not introduced by the migration to multicore because the semantics of a sequential procedure do not depend on the number of cores of the execution platform.

E.3.2 VIOLATION OF FUNCTIONAL CORRECTNESS IN CONCURRENT SETTING

The focus in this report, however, is on safety-critical avionics software. Such software is typically concurrent, and therefore its semantics do depend on the number of cores. In particular, this difference in semantics can lead to the two types of violations of functional correctness mentioned previously. In the rest of this section, without loss of generality, attention will be restricted to multi-thread software in which threads communicate via shared variables and synchronize via mutexes. However, results will apply to general concurrent software with other units of concurrency (e.g., processes), communication (e.g., message passing), and synchronization (e.g., monitors and semaphores). Specifically, functional correctness violations can occur in the following ways:

- Deadlocks can cause procedures not to terminate. Broadly speaking, a deadlock occurs when one or more threads are blocked when trying to acquire a mutex. This can happen, for example, when two threads acquire different locks and then try to acquire the lock that is already held by the other. Consider the code fragment shown in figure E-10, in which one thread executes procedure p1() and another thread executes procedure p2(). Here, the functions lock(m) and unlock(m) are used to acquire and release mutex m, respectively.

```
void p1() {
 lock(mutex1);
 lock(mutex2);
}
```

```
void p2() {
 lock(mutex2);
 lock(mutex1);
}
```

Figure E-10. Example program with deadlock

  There is an execution in which both threads succeed on their first lock statement, but then deadlock on their second lock statement.

- Livelocks can also cause procedures not to terminate. A livelock happens when one or more threads constantly change their state (i.e., they are not blocked), but do not make any progress. For example, consider the code fragment shown in figure E-11, in which one thread executes procedure p1() and another thread executes procedure p2(). Here, function trylock(m) attempts to acquire mutex m. On success, it returns 1. Otherwise, it returns 0.

```
void p1() {
 for(;;) {
 lock(mutex1);
 if(trylock(mutex2)) break;
 unlock(mutex1);
 }
}
```

```
void p2() {
 for(;;) {
 lock(mutex2);
 if(trylock(mutex1)) break;
 unlock(mutex2);
 }
}
```

Figure E-11. Example program with livelock

There is an execution in which both threads livelock, continuously succeeding to acquire their first lock, but failing on their second lock.

- Race conditions can cause procedures to compute incorrect results. In some sense, race conditions are the dual of deadlocks and livelocks. Whereas deadlocks and livelocks are caused by too much synchronization, race conditions are caused by inadequate synchronization, leading to unforeseen executions. For example, consider the code fragment shown in figure E-12, in which one thread executes procedure p1() and another thread executes procedure p2(). Both threads access a global variable g, which is initialized to 0.

```
void p1() {
 g = g + 1;
}
```

```
void p2() {
 g = g + 1;
}
```

Figure E-12. Example program with race condition

It is natural to expect that after the two procedures have terminated, the final value of *g* must be 2. This is true if there was adequate synchronization to ensure that each procedure executes atomically. However, note that each procedure performs a read and a write, and there is no synchronization. Therefore, the following execution leads to a final value of *g* being 1. First, both threads read the initial value 0 of *g* into some local variable. Next, each thread increments this local variable to obtain 1. Finally, the new value 1 of the local variable is written back to *g*.

There is a wide body of literature on deadlocks, livelocks, and race conditions, their underlying causes, and various automated techniques [E-41, E-42, E-43] to detect them in concurrent software. Such techniques involve both static and dynamic verification, heuristics (such as partial-order reduction [E-44]), compositional verification (such as assume-guarantee reasoning [E-45]), and abstraction. Many of these techniques have been implemented in tools and applied on industrial systems. Before the emergence of multicore processors, these tools and techniques did a good job of detecting and helping to eliminate many serious functional flaws in programs, including avionics software. From a certification standpoint, as long as only single-core CPUs are used for execution, the evidence of the application of at least some of these tools to check functional correctness of a piece of code could be used to argue that the code in question is acceptably bug-free.

E.3.3 VIOLATIONS OF FUNCTIONAL CORRECTNESS IN MULTICORE PLATFORMS

However, multicore processors reduce the effectiveness of these tools in critical ways. Even if a piece of code was found to be completely correct (e.g., free of deadlocks, livelocks, and race conditions) by existing verification tools, that code can still demonstrate functionally incorrect behavior when executed on a multicore platform. The main culprit is that the vast majority of functional-verification tools were developed under an assumption of software memory access semantics (known as sequential consistency) that work for single-core CPUs but do not hold for multicore CPUs. Therefore, verification tools that assume sequential consistency are unsound with respect to multicore platforms. In the rest of this section, this issue is discussed in more detail.

E.3.3.1 Sequential Consistency

Every computing platform has to implement some memory consistency [E-46] semantics to reconcile between the memory operations performed by various threads. In single-core platforms, and even on shared-memory multi-processor platforms, the de facto standard is known as sequential consistency [E-47] (SC). These semantics were defined originally by Lamport as follows:

Definition: [A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Consider the example program shown in figure E-13, in which one thread executes procedure p1() and another thread executes procedure p2(). The two threads execute on different cores. Both threads access global variables *x* and *y*, which are initialized to 0. In addition, p1() writes to variable r1 and p2() writes to variable r2. Both r1 and r2 are also initialized to 0. This example is taken from a presentation on “Multicore Semantics and Programming” [E-48], and its correct behavior is at the heart of mutual-exclusion algorithms (e.g., Dekker’s).

```
void p1() {
 L1: x = 1;
 L2: r1 = y;
}
```

```
void p2() {
 L3: y = 1;
 L4: r2 = x;
}
```

Figure E-13. Example program to demonstrate sequential consistency

Under sequential consistency, the final values of *r*1 and *r*2 cannot both be 0 after the two threads terminate. For brevity, statements in the example are referred to by their labels L1, …, L4. $L_i \rightarrow L_j$ is written to mean that the statement at label L_i executed before the statement at label L_j. For example, by SC, L1 → L2 and L3 → L4 is known. Note that the relation → is transitive (i.e., if $L_i \rightarrow L_j$ and $L_j \rightarrow L_k$ then it is known that $L_i \rightarrow L_k$).

Claim 1: After procedures p1() and p2() terminate, the final values of variables *r*1 and *r*2 cannot both be 0 if the memory accesses are sequentially consistent.

Proof: By contradiction. Suppose that the values of r1 and r2 are both 0. Because r1 is 0, it is known that L2 → L3. Therefore, by SC and transitivity of →, it is known that L1 → L3. Similarly, because r2 is 0, it is known that L4 → L1. Again, by SC and transitivity of →, it is known that L3 → L1. However, it is impossible to have any sequentially ordered execution in which both L1 → L3 and L3 → L1 are true.

Unfortunately, modern multicore processors do not enforce sequential consistency. If the above example is implemented (e.g., using C++) and executed on a multicore machine, rare but regular violations of claim 1 would be observed. This is problematic for two reasons. First, because SC is at the heart of many critical algorithms (such as Dekker's mutual exclusion), violation of SC can lead to serious faults. Second, because the violations of SC are rare, such faults may not show up, even after a considerable amount of testing. They must be detected by more rigorous methods that provide better coverage. Unfortunately, many such rigorous verification tools assume sequential consistency semantics and, therefore, their results cannot be ported in a direct way from single-core to multicore platforms.

Sequential consistency is not enforced primarily because of performance considerations. Modern multicore platforms have complex memory systems, involving multiple levels of caches and memory banks. Enforcing sequential consistency would require some ultimate memory arbiter that would determine the order in which reads and writes occur. Such an arbiter would be a severe performance bottleneck. Modern multicore processors therefore implement consistency semantics that are more relaxed than SC. These semantics result from the fact that each core maintains its own write buffer to prevent blocking on write operations and also reorders reads and writes (to different memory locations) to improve performance. In the rest of this section, three such semantics are considered—Total Store Order (TSO), Processor Consistency (PC), and Partial Store Order (PSO).

E.3.4 RELAXED MEMORY CONSISTENCY MODELS

As mentioned before, relaxed memory-consistency models result from write buffers and reorderings of reads and writes that violate sequential consistency. Therefore, it is useful to consider various orderings of read and write operations on a thread[15]. Because there are two basic operations—read and write—there are four possible orderings:

- W→R: write must complete before subsequent read
- R→R: read must complete before subsequent read
- R→W: read must complete before subsequent write
- W→W: write must complete before subsequent write

Sequential consistency requires that all four orderings must be enforced in each thread. TSO and PC both allow the first ordering (W→R) to be violated. This means that a read R can happen before a write W even if R occurs after W in the program. Recall again the program from figure E-13. Under both TSO and PC, the following orders are allowed: L2→L1 (because the read of *y* can be moved before the write to *x*) and L4→L3 (because the read of *x* can be moved before the write

[15] The material in this section is influenced by a set of online lecture notes on "Relaxed Memory Consistency," available at http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/14_relaxedReview.pdf

to *y*). Consequently, as was observed in practice, the values of *r*1 and *r*2 can both be 0 after the two procedures terminate.

The key difference between TSO and PC is how writes by one core are visible to other cores. TSO requires that, at any point in time, each write by a core is either visible to all other cores or to none. In other words, it is not possible for a write by core C_1 to be visible to core C_2 but not to core C_3 under TSO. In contrast, this type of different visibility of writes by different cores is allowed by PC. Consider the program fragment shown in figure E-14, in which procedures p1(), p2(), and p3() are executed by different threads on different cores. Global variables *x*, *y*, and *r* are initialized to 0.

```
void p1() {
 x = 1;
}
```

```
void p2() {
 while (x == 0);
 y = 1;
}
```

```
void p3() {
 while (y == 0);
 r = x;
}
```

Figure E-14. Example showing difference between TSO and PC

Under TSO, the final value of *r* must be 1. This is because p3() can read *x* only after it breaks out of the while loop. However, this means that the value of *y* has been set to 1 by *p*2(), which implies that *p*2() observed the value 1 written to x by *p*1(). Under TSO, therefore, *p*3() must also now observe this new value 1 of *x*. However, under PC, it is possible for the write to *x* by *p*1() to be observed by *p*2() but not by *p*3(). Therefore, the final value of *r* under PC can be 0.

The PSO semantics differ from both TSO and PC in that they also allow the W→W ordering to be violated, in addition to W→R. Consider the program fragment shown in figure E-15, in which procedures *p*1() and *p*2() are executed by different threads on different cores. Global variables *x* and *y* are initialized to 0.

```
void p1() {
 L1: x = 1;
 L2: y = 1;
}
```

```
void p2() {
 L3: while (y == 0);
 L4: r = x;
}
```

Figure E-15. Example showing the difference between TSO, PC, and PSO

Under both TSO and PC, the final value of *r* must be 1:

1. Because W→W, there is L1→L2.
2. Because L3 observes L2, there is L2→L3.
3. Because R→W, there is L3→L4.
4. By transitivity of →, there is L1→L4, which means the final value of *r* must be 1.

However, under PSO, it is also possible for the final value of *r* to be 0. Essentially, because PSO does not guarantee W→W, there is no longer L1→L2. Specifically, consider the following execution:

$$L2 \rightarrow L3 \rightarrow L4 \rightarrow L1 \tag{E-7}$$

This execution is possible under PSO and results in a final value of *r* being 0. In summary, this shows that relaxed memory-consistency models are subtly different from sequential consistency. Therefore, assurance data for functional correctness generated by the majority of existing functional verification tools (such as static analyzers and software model checkers) are not directly portable to multicore platforms because these tools are based on the assumption of sequential consistency.

E.3.5 PRIORITY-BASED LOCKING FAILURES

The issues related to multicore memory-consistency semantics discussed so far are relevant to both real-time and non-real-time software. There is, however, another issue that affects real-time software specifically: the failure of priority-based locking protocols on multicore CPUs. In particular, the focus is on real-time software consisting of multiple threads scheduled based on their priorities. This is a common paradigm used in avionics software (e.g., fixed-priority scheduling with rate-monotonic priority assignment). In such software, a lock is often acquired by changing thread priorities at runtime. For example, a thread can be given a lock by raising its priority to be higher than all other threads. The lock can be released by reverting the thread back to its normal priority. There are many variants of this idea, including priority ceiling and priority inheritance [E-35] protocols. However, all such schemes were designed originally under the assumption of a single CPU capable of executing only one thread at a time, as was discussed in section 2.6. These schemes do not work when multiple cores are available. For example, raising a thread's priority to be higher than all other threads does not give it exclusive access to a shared resource because another thread can always execute in parallel on a different core and access the resource simultaneously. There has been considerable work on synchronization mechanisms for multiprocessor systems [E-49], and many such mechanisms may be applicable for multicore platforms. However, the assurance data for logical correctness must be examined carefully to ensure that the synchronization mechanisms used by the avionics software are appropriate for the target platform to which the software is being ported.

E.4 SUMMARY AND RECOMMENDATIONS

This increasing interest in using multicore processors in avionics raises the question of how to certify aircraft whose avionics include multicore processors. This report discusses this question, particularly what can go wrong when porting software that was originally developed for a single-core processor and now executes on a multicore processor. Moreover, as seen in section 2.5, the isolation techniques used to allow independent certification of different partitions do not work immediately when moving these techniques from single-core to multicore processors. Obviously, this includes virtualization technologies for both general-purpose computing and real-time computing. New techniques have appeared in the academic community to create new forms of partitioning. However, the number of partitions that is possible to create and the utilization that is possible to achieve are still limited and cannot scale to large avionics systems (e.g., with 200-plus tasks).

From the logical point of view, the goal so far in this section has been to demonstrate that memory-consistency semantics on multicore processors are quite complicated and differ substantially from sequential consistency, which is the de facto standard for single-core processors. These differences can lead to violations of functional correctness when a program is executed on a multicore platform, even when the program behaves correctly on single-core CPUs. Moreover, the majority of current formal verification tools assume sequential consistency, and are therefore unsound for multicore platforms. One issue in developing formal verification tools for multicore processors is that the actual memory-consistency semantics implemented by hardware vendors for multicore processors is not rigorously documented. There have been some recent efforts to remedy this situation by the academic community. For example, TSO-like semantics for x86 processors [E-50] have been proposed by Owens et al. and incorporated into the CompCert compiler [E-51]. However, there has been limited acceptance of these semantics by hardware vendors and verification-tool developers. Nevertheless, the bottom line for certification authorities remains that the memory-consistency semantics of the target hardware platform, and how it matches the memory consistency assumed by applied verification tools, must be carefully considered while porting certification data for functional correctness of avionics software from single-core to multicore platforms.

It has been seen that from both the perspective of timing and logical correctness, there are significant risks when migrating software to multicore processors. More research is needed in the following areas:

- Timing-analysis methods that deal with undocumented hardware
- How to properly configure methods that reduce the execution-time variation caused by hardware
- How to develop scalable partitioning techniques and configuration methods for multicore processors
- How to achieve mutual exclusion among tasks in the context of a real-time scheduler
- New logical-verification methods that target the pitfalls of the multicore memory-consistency models
- New synchronization algorithms for the new memory-consistency models

For the certification engineer, these recommendation can be further elaborated as:

- When considering the timing portability arguments, it is important to verify the proper documentation is provided. In particular, different arguments may need different levels of detail. However, all arguments must include some level of inter-core interference, in which the timing behavior of a task in one core is evaluated when other tasks with different a type of code is running in other cores.

- For timing portability arguments for software that uses mutual exclusions, the proper argument that explicitly mentions the mechanisms and the proper verification techniques must be presented.

- For logical correctness portability arguments, it is important to consider the consistency model implemented by the multicore processor and whether this model matches the presented arguments.

E.5 REFERENCES

E-1. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K.G., & Chang, K. (1996). The Case for a Single-Chip Multiprocessor. *ASPLOS*.

E-2. Hammond, L., Nayfeh, B.A., & Olukotun, K. (1997). A Single-Chip Multiprocessor. *IEEE Computer, 30*(9), 79–85.

E-3. Fuchsen, R. (2010). How to address certification for multi-core based IMA platforms: Current status and potential solutions. *29th Digital Avionics Systems Conference,* 5.E.3-1–5.E.3-11.

E-4. FAA Report. (2011). Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems. (DOT/FAA/AR-11/2).

E-5. Bellosa, F. (1997). Process Cruise Control Throttling Memory Access in a Soft Real-Time Environment. Technical Report, University of Erlangen-Nürnberg.

E-6. Schönberg, S. (2003). Impact of PCI-Bus Load on Applications in a PC Architecture. *RTSS*.

E-7. Pellizzoni, R., Bui, B.D., Cacao, M., Sha, L. (2008). Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. *2008 Real-Time Systems Symposium,* 221–231.

E-8. Bui, B.D., Cacao, M., Sha, L., Martinez, J. (2008). Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 101–110.

E-9. Pellizzoni, R., Schranzhofer, A., Chen, J., Cacao, M., Thiele, L. (2010). Worst case delay analysis for memory interference in multicore systems. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, 741–746.

E-10. Radojkovic, P., Girbal, S., Grasset, A., Quiñones, E., Yehia, S., & Cazorla, F.J. (2012). On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO), 8*(4), 34:1–34:25.

E-11. Nowotsch, J., & Paulitsch, M. (2012). Leveraging Multi-core Computing Architectures in Avionics. *2012 Ninth European Dependable Computing Conference*, 132–143.

E-12. Yun, H. (2015). Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms.

E-13. Mancuso, R., Dudko, R., Betti, E., Cesati, M., Cacao, M., Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 45–54.

E-14. Kenna, C.J., Herman, J.L., Ward, B.C., Anderson, J.H. (2012). Making Shared Caches More Predictable on Multicore Platforms. In *Proceedings - Euromicro Conference on Real-Time Systems.*

E-15. Suzuki, N., Kim, H., Niz, D.D., Andersson, B., Wrage, L., Klein, M.H., Rajkumar, R. (2013). Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. *2013 IEEE 16th International Conference on Computational Science and Engineering*, 685–692.

E-16. Kim, H., Niz, D.D., Andersson, B., Klein, M.H., Mutlu, O., Rajkumar, R. (2014). Bounding memory interference delay in COTS-based multi-core systems. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 145–154.

E-17. Ye, Y., West, R., Cheng, Z., Li, Y. (2014). COLORIS: A dynamic cache partitioning system using page coloring. *2014 23rd International Conference on Parallel Architecture and Compilation (PACT)*, 381–392.

E-18. Nowotsch, J., Paulitsch, M., Buhler, D., Theiling, H., Wegener, S., & Schmidt, M. (2014). Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. *2014 26th Euromicro Conference on Real-Time Systems*, 109–118.

E-19. Certification Authorities Software Team (CAST), Position Paper, CAST-32A, Multi-core Processors, COMPLETED November 2016 (Rev 0), Available at https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

E-20. Sha, L., Cacao, M., Shelton, G., Nuessen, M., Smith, J. P., Miller, D., … Pak, E. (2016). Position Paper on Minimal Multicore Avionics Certification Guidance.

E-21. Strasburger, J. (2014). "FAA Status on Multicore Processors," slides from Keynote at First TCRTS Workshop on Certifiable Multicore Avionics Systems (CMAS). Slides available at http://rtsl-edge.cs.illinois.edu/CMAS/media/00-John-Keynote.pdf

E-22. Yun, H., Yao, G., Pellizzoni, R., Cacao, M., & Sha, L. (2016). Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers, 65*(2), 562–576.

E-23. Sha, L., Cacao, M., Mancuso, R., Kim, J., Yoon, M., Pellizzoni, R., Yun, H., Kegley, R., Perlman, D.R., Arundale, G., & Bradford, R.M. (2016). Real-Time Computing on Multicore Processors. *Computer, 49*(9), 69–77.

E-24. Kim, N., Ward, B.C., Chisholm, M., Fu, C., Anderson, J.H., Smith, F.D. (2016). Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1–12.

E-25. Strosnider, J.K.; Lehoczky, J.P.; Sha, L. (1995). The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers, 44*(1), 73–91.

E-26. Mercer, C.W., Savage, S., Tokuda, H. (1994). Processor Capacity Reserves: Operating System Support for Multimedia Applications. *ICMCS*.

E-27. Oikawa, S., Rajkumar, R. (1999). Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. *IEEE Real Time Technology and Applications Symposium*.

E-28. Bienia, C., & Li, K. (2010). Fidelity and scaling of the PARSEC benchmark inputs. *IEEE International Symposium on Workload Characterization (IISWC'10)*, 1–10.

E-29. Kim, H., & Rajkumar, R. (2016). Real-time cache management for multi-core virtualization. *2016 International Conference on Embedded Software (EMSOFT)*, 1–10.

E-30. Liedtke, J., Härtig, H., Hohmuth, M. (1997). OS-Controlled Cache Predictability for Real-Time Systems. *IEEE Real Time Technology and Applications Symposium*.

E-31. Chattopadhyay, S., Roychoudhury, A., Mitra, T. (2010). Modeling shared cache and bus in multi-cores for timing analysis. *SCOPES*.

E-32. Yun, H., Pellizzoni, R., Valsan, P.K. (2015). Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. *2015 27th Euromicro Conference on Real-Time Systems*, 184–195.

E-33. Wu, Z.P., Krish, Y., Pellizzoni, R. (2013). Worst Case Analysis of DRAM Latency in Multi-requestor Systems. *2013 IEEE 34th Real-Time Systems Symposium*, 372–383.

E-34. Dasari, D., Andersson, B., Nélis, V., Petters, S.M., Easwaran, A., Lee, J. (2011). Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 1068–1075.

E-35. Sha, L., Rajkumar, R., Lehoczky, J.P. (1990). Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers, 39*(9), 1175–1185.

E-36. Rajkumar, R. (1991). *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* AA Dordrecht, the Netherlands: Kluwer Academic Publishers. (ISBN 0-7923-9211-6).

E-37. Rajkumar, R. (1990). Real-Time Synchronization Protocols for Shared Memory Multiprocessors. *ICDCS*.

E-38. Lakshmanan, K., Niz, D.D., Rajkumar, R. (2009). Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors. *2009 30th IEEE Real-Time Systems Symposium*, 469–478.

E-39. Kopetz, H. and Reisinger, J. (1993). The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem. *RTSS*.

E-40. Niz, D.D., Saewong, S., Rajkumar, R., Abeni, L. (2001). Resource Sharing in Reservation-Based Systems. *IEEE Real Time Technology and Applications Symposium*.

E-41. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L. (2010). Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development, 54*(5), 3.

E-42. Wilcox, J.R., Finch, P., Flanagan, C., Freund, S.N. (2015). Array Shadow State Compression for Precise Dynamic Race Detection (T). *30th IEEE/ACM International Conference on Automated Software Engineering (ASE),* 155–165.

E-43. Wolf, K., Stahl, C., Ott, J., Danitz, R. (2009). Verifying Deadlock- and Livelock Freedom in an SOA Scenario. *2009 Ninth International Conference on Application of Concurrency to System Design,* 168–177.

E-44. Godefroid, P., Wolper, P. (1991). Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design, 2*(2), 149–164.

E-45. Chaki, S., Sinha, N. (2006). Assume-Guarantee Reasoning for Deadlock. *2006 Formal Methods in Computer Aided Design,* 134–144.

E-46. Adve, S.V., Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. *IEEE Computer, 29*(12), 66–76.

E-47. Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers, C-28*(9), 690–691.

E-48. Sewell, P., Harris, T: (2011). Multicore Semantics and Programming. Retrieved from https://www.cl.cam.ac.uk/teaching/1112/R204/slides-acs-2011.pdf

E-49. Rajkumar, R., Sha, L., & Lehoczky, J.P. (1988). Real-Time Synchronization Protocols for Multiprocessors. *RTSS*. 259–269.

E-50. Owens, S., Sarkar, S., & Sewell, P. (2009). A Better x86 Memory Model: x86-TSO. *TPHOLs*. 391–407.

E-51. Sevcík, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., & Sewell, P. (2013). CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, *60*(3), 22:1–22:50.

APPENDIX F—IDENTIFICATION OF ASSURANCE ISSUES ON EMULATION OF CERTIFIED HARDWARE

F.1 TECHNOLOGY FOR HARDWARE EMULATION

Hardware emulation, often referred to as "emulation" in this report, is a widely prevalent technique for combating the problem of hardware obsolescence. The hardware industry innovates at a rapid pace to meet the growing and morphing computational needs of society. Therefore, old hardware standards, instruction sets, and design principles are quickly supplanted by newer ones. In many cases, necessitated by the need to balance considerations of cost, efficiency, and time to market, the newer hardware is not even backward compatible with its older counterparts. Further, once the newer hardware grabs a sufficiently large market share, it is no longer cost effective to even manufacture the older hardware because of the loss of the economy of scale. Consequently, software vendors repeatedly face the following dilemma: should they update their software to be compatible with the newer hardware, or should they develop the software from scratch? In the non-safety-critical software world, a vendor typically makes this decision after considering a host of technical and economic factors.

In the safety-critical avionics domain, however, it is often cost prohibitive to change the software, because that triggers an expensive recertification process. This makes hardware emulators a particularly attractive option. In essence, a hardware emulator imitates the "complete hardware and software" stack of a "target/guest" system on top of a "host" hardware and software stack. This means that software for obsolete hardware can be run seamlessly on the latest hardware without any changes.

Note that emulation and virtualization are related, but subtly different. In virtualization, the guest operating system (OS) is often targeted at the same hardware as the host OS. The main motivation for virtualization is to allow multiple guests to share the same hardware, even if each guest could have executed directly on the hardware. The main concern in virtualization from a safety perspective is to ensure proper "logical and timing isolation" between guests. In contrast, the main motivation for emulation is to allow software for one hardware platform to execute on different hardware, on which it could not execute directly. The main concern in emulation from a safety perspective is to ensure "logical and timing equivalence" between the behaviors of the emulator and old hardware. This is particularly important when the hardware has been certified for specific avionics software. In the rest of this section, emulation will be presented in more detail and assurance issues involved in the use of emulators for certified software will be discussed.

F.1.1 THREADED CODE

Emulators are often implemented with threaded code [F-1], a concept from the 1970s that is still useful. For this reason, a short discussion on this topic is warranted. In normal programs executing on a normal processor, the processor has an instruction set and a program counter, and the program executes instructions from that instruction set. This offers the advantage that each instruction can execute quickly because it is interpreted directly by the processor. Unfortunately, a drawback is that the program requires lots of memory. It can be said that its code density is low. An alternative approach is to write the program with instructions that are not necessarily the ones that the processor can execute. For each instruction, there is a subroutine that executes this instruction and this subroutine is implemented with instructions from the processor. For example, a single byte with the hexadecimal number 3F might mean "pop two words from the stack and then add them and store the result on the stack." If this operation occurs frequently in the program, this approach (of writing the executable code with an instruction set that differs from the processor it executes on) can offer significant space savings.

This alternative approach, in which a program may have instructions that differ from those of the process and the program also contains an interpreter of the instructions, is known as threaded code [F-1]. The article [F-1] reports results on a PDP-11 and points out that threaded code is slower than inline code but is faster than function calls (PDP-11 has an instruction that is well suited for threaded code). The article [F-1] also points out that one can write a program in which some parts use threaded code and other parts do not.

Threaded code is used in Java virtual machines (VMs), and it is also used in an early version of SimICS[16] [F-2]. There are many different ways of implementing threaded code. Some methods implement the program as a set of subroutine calls in which each subroutine ends with a return instruction. Another method stores the program as a sequence of pointers to subroutines so that the interpreter calls these subroutine in that order. Typically, there is a variable that keeps track of a virtual program counter.

The concepts of hardware emulation and one of the key implementation examples (threaded code) will be referred to in the rest of this section.

F.2 LOGICAL EQUIVALENCE/DIFFERENCES EMULATION VS REAL-HARDWARE

The main concern from a safety perspective when using an emulator is to ensure that it has the same logical and timing behavior as the target hardware. In this section, the assurance issues involved in ensuring logical equivalence are discussed. Logical equivalence is important to ensure that the target software reaches the same execution states on the emulator as on the original hardware. This is critical to ensure that safety properties, such as absence of deadlocks, race conditions, and other application-specific invariants, continue to be preserved on the emulator.

When the guest hardware is single-core, the emulator must imitate detailed behavior of each instruction in the instruction set architecture of the guest. This is often complicated by the fact that instructions have esoteric side effects. In many cases, the side effects should probably be classified as hardware bugs. However, the target software could have been functioning properly despite these side effects. Even worse, in many cases, the software could have been relying on the bugs inadvertently. Therefore, it is important to identify these side effects and reproduce them accurately as part of the emulator. This process is complicated by the fact that such side effects, especially bugs, are often undocumented and must be discovered by a painstaking trial-and-error process. In addition, the emulator must also imitate the behavior of peripheral devices and the software stack that were present on the old system.

[16] SimICS is a full-systems simulator originally intended to simulate server processors, but it now also targets avionics systems. See, for example, https://www.windriver.com/markets/aerospace-defense/face.html.

In the case in which the old system was multicore or multi-processor, the assurance issue is further exacerbated by the need to implement appropriate memory-consistency semantics. Memory consistency can be a complicated issue on multicore and multi-processor systems. In theory, there are several choices to select. However, in practice, the actual memory consistency implemented by specific hardware is often undocumented and can be recovered only by tedious reverse-engineering.

In both the single-core and multicore cases, it is equally important that the emulator imitate each instruction within an appropriate amount of time that is commensurate to the amount of time required to execute that instruction on the original hardware. This is referred to as temporal equivalence, and it is discussed next.

F.3 TEMPORAL EQUIVALENCE/DIFFERENCES EMULATION VERSUS REAL-HARDWARE

Consider an old hardware platform and a software system to be used on a new hardware platform to emulate the old platform. Unless certain precautions are taken, it may happen that the timing of the execution of the software on the new platform is different from what it was when executing on the old platform. There is typically not an interest in having exactly the same timing; instead, there is an interest in having exactly the same timing between I/O events. The paper [F-3] provides important results for this situation and is therefore discussed in this section.

The paper [F-3] considers a software system that executes on a source platform. The goal is to run this program on a target platform so that the timing is the same. The software system is described with a control flow graph. An example of such a graph is shown in figure F-1.

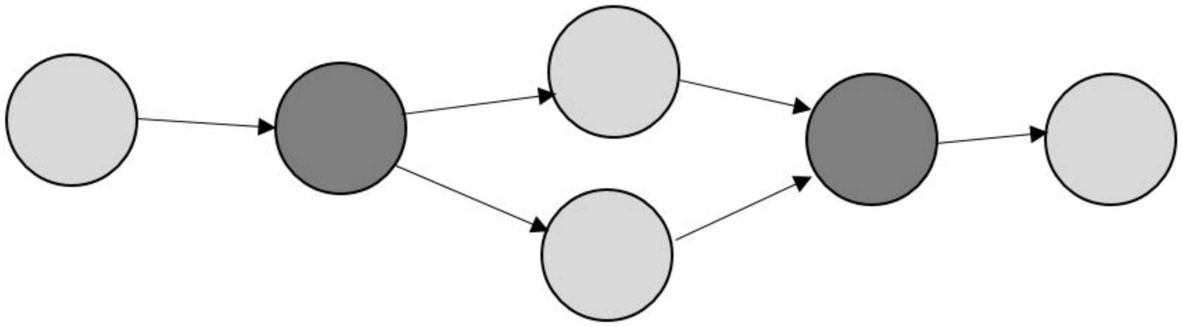


Figure F-1. An example of a control flow graph (no cycle)

A node in a control flow graph represents a basic block. A basic block is a piece of code in the program that has a single entry point and a single exit point. The edges in a control flow graph represent control flow. For example, consider the leftmost node in figure F-1. There is an edge to the second-leftmost node. This means that when the basic block of the leftmost node in the graph has finished, the program executes the basic block of the second-leftmost node. Note that to the right, there are two nodes. There are edges to these two nodes. This means that if the basic block of the second-leftmost node finishes, the program has two possibilities for the next basic block to execute. Therefore, a trace of execution of a program can be thought of as a path in the control graph.

Some of the nodes are basic blocks that perform no I/O. Other nodes are basic blocks that perform I/O. This report is interested in making sure that for a path from one node that performs I/O to another node that performs I/O, the execution time on the new hardware platform is the same as the execution on the old hardware platform. For this reason, in figure F-1, the nodes without I/O are lightly shaded, and the nodes with I/O are darkly shaded.

In the example in figure F-1, the graph is directed but it has no cycles. In general, a control flow graph can have cycles. Figure F-2 shows an example of that.

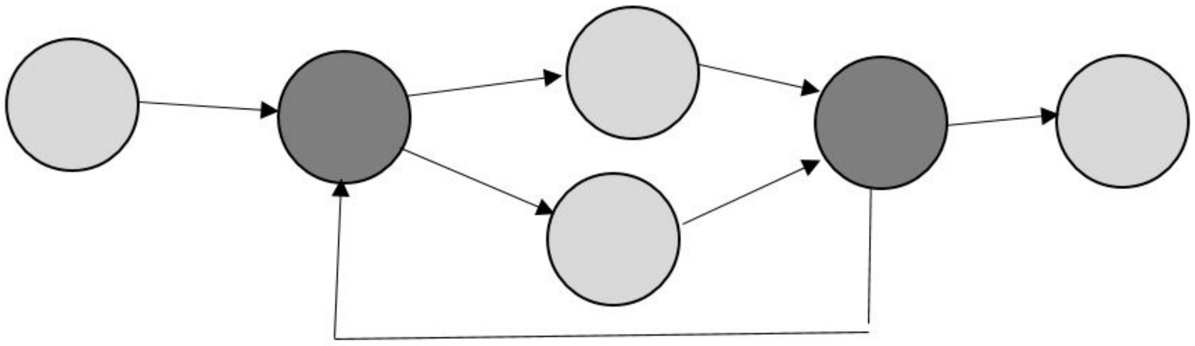


Figure F-2. Another example of a control flow graph (with a cycle)

Note that a control flow graph with cycles has an infinite number of paths. Therefore, making sure that, for the given software system, the new platform has the same timing as on the old platform is challenging. The reason it is challenging is that the property must be ensured for an infinite number of paths.

The paper [F-3] annotates each node in the control flow graph with two numbers: 1) execution time of the basic block in the old platform and 2) execution time of the basic block in the new platform. Figure F-3 shows an example of this.

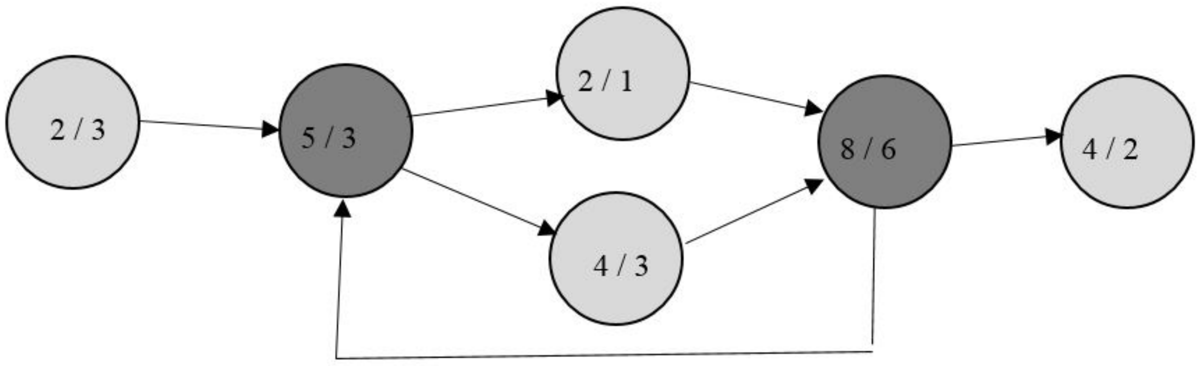


Figure F-3. Another example of a control flow graph in which each node is described with the execution time of the old platform and the execution time of a new platform

In figure F-3, the first number in a node is the execution time of the basic block corresponding to the node for the old platform. The second number is for the new platform. It can be seen in this figure that the leftmost node executes faster on the old platform than on the new platform. Therefore, if the program would execute only the leftmost node and if the leftmost node were an I/O node, the new platform would have incorrect timing. Fortunately, the leftmost node is not an I/O node. This report is interested in proving that for each path from source to destination, timing is correct for each pair of nodes that perform I/O.

The paper [F-3] defines three timing properties related to correctness. It introduces a path as complete if the source in the path is the same as the entry point in the control flow graph and the destination of the path is the same as an exit point of the control flow graph. For a given complete path, *cp*, the paper defines Δ(cp) as the maximum difference between the timing of the old and new platform between each pair of I/O nodes. The paper defines a complete path in the execution on the new platform as timing equivalent with respect to the execution on the old platform if $\Delta(\text{cp}) = 0$. A complete path *cp* is executable with timing equivalence if there exist synchronization methods that guarantee the timing equivalence of *cp*.

The paper introduces another slightly weaker notion, timing invariance, meaning that the new platform is slower by a certain bound; specifically, timing invariance is achieved if there is a constant *C* such that $\Delta(\text{cp}) \leq C$. If no such number exists, then it is divergent.

One of the main results in the paper [F-3] is that a set of subpaths can be identified between I/O nodes such that the subpath has no cycle, and then it is needed only to ensure that the timing between the I/O nodes for the target platform is faster than for the source platform. This result is important because it allows consideration of control flow graphs with an infinite number of paths yet to be analyzed.

F.4 DISCUSSION AND RECOMMENDATIONS

Hardware emulation introduces two important problems to certification—timing and logical equivalence. From the logical equivalence point of view, one of the key challenges is to identify and account for the side effects of an implementation of the native hardware that software can actually rely on. The issue here is that hardware emulation may not reproduce the native hardware faithfully given that such side effects may not be part of the documented behavior of the hardware.

From the timing perspective, the main challenge is to ensure timing equivalence between inputs and outputs because the timing effects are visible only to the external world at the input and output points. This is especially important for real-time systems such as avionics systems that are sensitive to when the software senses the environment (input) and produces the actuation (output) to keep the aircraft under control.

Clearly, hardware emulation is a significant challenge for the certification of avionics systems. In general, there are no definitive solutions for all the issues that may be faced when using emulation, but in this section two key perspectives about how to frame the problem were presented.

For the certification engineer, these recommendations can be further elaborated as:

1. The arguments for timing equivalence can be presented as an input-to-output argument as soon as the equivalence is presented in a way that include all the paths.
2. Arguments for logical portability must include native hardware side effects used by the software.

F.5 REFERENCES

F-1. Bell, J. (1973). Threaded Code. *Communications of the ACM, 16*(6), 370–372.

F-2. Magnusson, P. and Werner, B. (1995). Efficient Memory Simulation in SimICS. *Proceedings of the 28th Annual Simulation Symposium*.

F-3. Kim, I. and Segall, Z. (2000). A Formal Method for Providing Temporal Equivalence in Binary-to-Binary Translation of Real-Time Applications. *IEEE Real-time Systems Symposium*, 185–194.

APPENDIX G—FUTURE WORK AND RECOMMENDATIONS

The gaps identified in this project are first described; recommendations are then provided to cover these gaps.

G.1 CURRENT GAPS

G.1.1 TEMPORAL VIRTUALIZATION

Current temporal-virtualization technologies do not fully cover the application requirements from aircraft manufacturers or certification needs from the FAA. Three aspects are worth discussing: development, application requirements, and the verification of temporal-virtualization implementations.

G.1.1.1 Development

The most popular virtualization products in the market are designed for general-purpose computing and do not offer strict timing guarantees. This type of virtualization will not be discussed in this section. However, the real-time operating systems that offer real-time guarantees provide virtualization schemes with a number of drawbacks that complicate development and hinder maintenance and certification and proper isolation in multicore processors. These issues will be discussed next.

There are a number of aspects of development that current real-time virtualization (RTV) approaches do not support properly. First, commercial RTV is based on the ARINC 653 temporal-partitioning standard. This standard, although it can provide strong isolation properties, requires the calculation of time slots that satisfy all the timing requirements of the different applications. Unfortunately, calculating such partitions requires techniques of high computational complexity, such as mixed-integer linear programming or satisfiability modulo theory (SMT). Therefore, this complicates development scenarios in which different teams define the timing parameters of their modules independently from other teams. Such development scenarios are becoming increasingly common with increasing application complexity. There are other products based on rate-monotonic scheduling (RMS), which favors modularity. Such products are, unfortunately, not mature enough. This is perhaps in part due to the lack of a standard that supports the commercialization of products based on RMS.

Second, maintenance can be hindered by the lack of modularity of current RTV. Specifically, changes in one module may require changes in timing parameters (e.g., execution time, periods, deadlines) that may require changes in the time slots of the module. In this case, the new requirements for the time slots in one module may then trigger the need to change the time slots in other modules, which can potentially require recertification of the other modules.

Third, as already mentioned before, when changes in one module require changes in another, the modularity of the recertification may be affected. If that is the case, the affected modules may need to be recertified.

Finally, multicore processors present new challenges to virtualization. In particular, the hardware resources shared across cores, such as cache and memory banks, can create significant delays in a task running in one core because of the access to these resources by another task in another core. Today, a large amount of RTV has been ported to multicore processors, but it is not clear that all these potential delays had been properly validated.

G.1.1.2 Application Requirements

With respect to application requirements, it is worth noting that avionics applications are distributed in nature. Specifically, an application may need to gather sensing data from one computer node, merge the data and perform some computation in another node, and perform actuation or display information in yet another node. As a result, the timing behavior needs to match these end-to-end requirements, and the virtualization needs to support this distributed nature. Unfortunately, current RTV techniques are mostly focused on a single node and do not take into account distributed applications. In practice, virtualization (or partitioning) support for these applications is provided as a collection of node-bound partitions. Unfortunately, this collection can create more modularity problems in which the modification of an application can impact multiple applications in multiple nodes. Some temporal verification technologies for distributed algorithms have been explored in the past, such as the real-time pipeline model [G-1]. More recently, such a model was used to create mixed-criticality temporal isolation mechanisms and the corresponding analysis in [G-2]. However, much more needs to be investigated to match distributed applications, which are more complex than pipelines.

G.1.1.3 Verification of Temporal-Virtualization Implementations

As formal verification evolves and becomes easier to apply to real applications, the FAA has been allowing its use for certification purposes. However, one of the challenges in this area is the formal verification of the temporal properties of the temporal-virtualization implementations. In general, this is a hard problem, and new models of time that can be verified are needed. The work to verify the temporal properties of ZSRM [G-3] is a start. However, the demonstration of this work is limited to the assumptions made about the underlying Linux kernel. More research is needed to understand how to verify such assumptions or verify other implementations.

G.1.2 SPATIAL VIRTUALIZATION

Spatial virtualization has been, in general, more stable and more generic than temporal virtualization. In particular, spatial virtualization from general-purpose virtualization technology can be used for real-time systems. However, when formal proofs on spatial partitioning are needed, traditional spatial-virtualization implementations have proven to be too complex. This complexity prevents the application of logical verification techniques in a practical manner. As a result, a new approach to minimizing the size of a hypervisor to enable full verification of its implementation is taking place. Two examples of this approach are the XMHF micro-hypervisor [G-4] and the seL4 [G-5] micro-kernel. However, separation proofs from these micro-hypervisors to prove application properties can be used only if all the code in the specific partition can be verified. Therefore, such a partition needs to be simple, ruling out complex virtual machines (VMs) that host complex operating systems, such as Linux. Moreover, combining the temporal-protection proofs from [G-3] and the spatial-protection proofs [G-4] is another challenge that must be addressed.

Multicore processors can actually create interaction patterns between temporal and spatial partitioning. In particular, temporal partitioning in multicore processors involves partitioning memory regions in a way that eliminates (or minimizes) the delays that one task running in one core may experience because of the execution of another in another core. In this case, temporal and spatial partitioning must be coordinated.

Finally, supporting distributed applications implies that a spatial partition to support an application can actually be composed of an aggregation of partitions in different nodes. Such an aggregation may be enough for some applications, but the verification of what code is authorized to access what partitions in an end-to-end manner needs to be investigated.

G.1.3 VIRTUALIZATION-AWARE TEMPORAL VERIFICATION

G.1.3.1 Temporal Predictability of I/O Virtualization

In current implementations, the virtualization of I/O is typically performed in a server-based manner. This means that one separate core is dedicated to the I/O services that the hypervisor provides to all VMs. However, current timing-verification techniques do consider this. New verification techniques to consider this need to be developed.

G.1.3.2 Temporal Predictability of Memory Accesses in Virtualization

Processors today use a variety of techniques to bridge the difference in speed between the processor(s) and the memory. As a result, there are many shared resources in the memory system (e.g., caches, memory bus, row buffer in memory banks, and port(s) to cache memories). Some of these shared resources can be dealt with by introducing mechanisms (e.g., cache coloring) that ensure timing isolation with respect to that resource. However, there are other resources for which such mechanisms do not exist. If high assurance is needed, then it may be necessary to model those shared resources in timing verification and to develop verification methods that can allow such models. It should be noted that many of these resources are very complex and depend on information that vendors do not disclose. Therefore, it is not obvious whether an explicit model (modeling the arbitration protocol for the resource and the possible accesses from software) is practical or even possible for most resources. There may be a need for more indirect approaches.

G.1.3.3 Temporal Predictability of Accesses to Data Structures in Virtualization

When using virtualization, there are (at least) two types of operating system kernels, the hypervisor and the guest operating system. Each has data structures. For example, each has schedulers and each needs data structures to keep track of the scheduled entities (a hypervisor schedules VMs; a guest operation system schedules processes). However, computers today do not write to an entire data structure; computers today write a word (e.g., 64-bit word). As a result, there are cases when an operating system performs an update on a data structure such that during this update, there is an instant where the data structure has been partially updated but not fully updated. Because of that, the operating system must not allow reading that data structure. There are different ways of dealing with this issue, such as mutual-exclusion mechanisms (e.g., spinning or semaphores) or lock-free data-sharing protocols. What they have in common, however, is that they influence timing. There are known techniques for verifying timing of software in the presence of such mechanisms. However, they require that the data structures/mechanisms be modeled. In many cases, the source code for the hypervisor and the guest operating system is not available and, therefore, it is very difficult to obtain such a model.

G.1.4 VIRTUALIZATION-AWARE LOGICAL VERIFICATION

As discussed in previous sections of this report, logical verification of virtualized systems is further complicated by additional interaction between the hypervisor and guests, and between the guests themselves. Such interactions can occur through interfaces that are not present in non-virtualized systems. For example, unless proper care is taken, a guest can access the hypervisor's memory and the memory of other guests directly. Similarly, a guest can interact with other guests and the hypervisor via shared devices. In general, such unwanted interactions are prevented via logical isolation mechanisms, such as locks and mutexes, and hardware-supported privilege-separation schemes (e.g., the hypervisor can set its own memory to be inaccessible by software such as the guest OS running at lower privilege levels). However, the correct usage and correct implementation of these isolation and privilege-based mechanisms must be verified. Whereas software analysis and verification techniques have made rapid progress in recent years, the presence of low-level code and sheer code size put realistic hypervisors still beyond the reach of fully automated and scalable verification. New verification techniques, such as focusing on analysis of low-level code and compositional reasoning, must be developed to overcome these challenges.

G.2 RECOMMENDATIONS

G.2.1 FUTURE RESEARCH

New research efforts to improve modularity must be fostered. These efforts should proceed in two directions. First, improve modularity of current partitioning standards such as ARINC 653. Second, develop new standards acceptable to the practitioner based on other more analytic foundations such as RMS.

The challenge presented by multicore processors to virtualization technology needs to be addressed. In particular, efforts to identify the evidence necessary to validate virtualization technology must continue. This is because the research in real-time multicore scheduling is not yet complete, and therefore the impact of new research results to virtualization will need to be investigated as these results become available.

Clearly, virtualization must address the distributed nature of avionics applications. This must be done while preserving modularity supporting modular certification and recertification. The exploration of RTV to support complex distribution models is necessary.

With respect to timing verification for virtualization, there is a need to address the three gaps. Specifically, there is a need for research to develop methods to analyze timing of software considering that I/O is performed by a server. There is also a need for research to develop methods to analyze the timing of software executing on hardware in which the hardware is undocumented (mentioned as one of the gaps above). In addition, there is a need for research to develop methods to analyze the timing of software executing on virtualization platforms (hypervisor and guest operating system)

From the spatial-virtualization point of view, continued research is proposed in three areas. First, verification schemes need to be investigated for partitioning implementation. This may start with hypervisors, but it must be ensured that enough support is provided to the application. Second, the interactions between temporal and spatial partitions in multicore processors must be investigated and certifiable solutions provided to such interactions. Finally, the support of distributed applications must be investigated to support today's application requirements. This will require, in particular, an increased focus on verifying low-level system code, and compositional reasoning and abstraction to handle the state-space explosion problem.

G.2.2 CURRENT PRACTICE

As discussed before, current standards for temporal virtualization are based on techniques that are not always sufficiently scalable or modular. In particular, the ARINC 653 style of partitioning may impose severe complexity and brittleness to software modifications. Exploring the development of new standards based on more flexible technologies such as rate-monotonic scheduling (RMS) is recommended.

With respect to timing requirements, it is recommended that the operating system vendor provide evidence that the use of shared data structures inside its operating system is based on a data-sharing protocol (e.g., mutual exclusion) with predictable timing behavior. It is also recommended that the application developer list all known resources that impact timing. In addition, it is recommended that the application developer provide evidence of the worst-case impact on delay on an execution path for the resources. These include the three mentioned: 1) I/O server, 2) data structures in operating systems, and 3) shared hardware resources (particularly in the memory system). Because more research is needed on these topics, it is probably not realistic to require that a software practitioner formally prove correct timing with respect to these resources. It is recommended, however, that the practitioners list these resources and estimate (sometimes called engineering judgment) the worst-case delay it can cause for respective timing requirements paths.

With respect to logical/spatial requirements, hardware vendors, verification researchers, and tool developers must agree on a formalization of low-level software semantics. Whereas hardware vendors typically document their instruction and architecture in a lot of detail, such documentation is informal (e.g., in English prose) and not amenable to formal reasoning. As a result, different verification groups have attempted to develop their own formalizations of hardware semantics. These efforts, though worthy of praise, are not coordinated and are also incomplete. Each group formalizes only a subset of the hardware necessary for their current project. This means that tools are not only incomparable, but also that the semantics formalized by the research groups (which always involve some level of abstraction) are not vetted by the hardware industry. This situation must change if the challenge system software is to be verified in a scalable and sound manner. Second, the state of the art in compositional reasoning of system software must be advanced. This requires formalizing the architecture of low-level systems, such as hypervisors, and new compositional techniques, such as assume-guarantee proof rules, derived from the architecture.

G.2.3 CERTIFICATION ENGINEER PERSPECTIVE

From the certification engineer perspective, recommendations can be summarized as follows:

- General-purpose virtual machines (VMs), such as VMWare or VirtualBox, should not be considered for avionics systems because there is no reliable technique to verify the timing isolation between VMs.

- Real-time hypervisors that provide predictable temporal isolation should be paired with the verification technique that matches their mechanism. For instance, if time-division multiplexing is used (e.g., time-triggered architecture), then exhaustive timeslot allocation algorithms must be used and the arguments regarding why specific allocations satisfy the partition requirements must be presented. Similarly, if the technology is based on rate-monotonic scheduling and processing servers, the corresponding response time techniques must be used.

- When considering recertification of partitions (or VMs) in isolation, the brittleness of the verification techniques must be considered. In particular, arguments to support why a modification to a partition does not affect other partitions must be presented. As elaborated in a previous appendix, different techniques have different sensitivity to modifications (brittleness). The brittleness discussion can be use by a certification engineer to guide evaluation of the isolation arguments.

- For applications with end-to-end timing requirements (e.g., end-to-end deadlines), the arguments must include how individual node deadlines are combined to satisfy end-to-end deadlines and how they are affected by the partitioning mechanisms and verification techniques.

- When presenting logical arguments of separation, care should be taken with the assumptions of such claims. This is particularly important during recertification when the assumptions may change.

- The interactions between logical correctness of spatial separation and timing separation must be presented. This is particularly important if these arguments are formalized with exhaustive verification techniques.

- Arguments of isolation for VMs running in multicore processors must properly support the interference channels mentioned in CAST32A with the supporting details from the processor documentation.

- Arguments to support the correctness of the interactions between spatial and temporal partitions must be included when using multicore processors. This is because the incorrect use of shared-hardware partitions can affect timing guarantees and partitioning assumptions.

- When considering the single-to-multicore timing portability arguments, it is important to verify that proper documentation is provided. In particular, different arguments may need different levels of detail. However, all arguments must include some level of inter-core interference, in which the timing behavior of a task in one core is evaluated when other tasks with different types of application code are running in other cores.

- For single-to-multicore timing portability arguments for software that use mutual exclusions, the proper argument that explicitly mentions the mutual exclusion mechanisms and the proper verification techniques must be presented.

- For logical correctness, single-to-multicore portability arguments, it is important to consider the consistency model implemented by the multicore processor and whether this model matches the presented arguments.

- The arguments for timing equivalence of emulated hardware can be presented as an input-to-output argument as soon as the equivalence is presented in a way that include all execution paths.

- Arguments for logical portability of emulated hardware must include native hardware side effects used by the software.

G.3 REFERENCES

G-1. Jayachandran, P., Abdelzaher, T.F. (2007). A Delay Composition Theorem for Real-Time Pipelines. *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, 29–38.

G-2. Niz, D.D., Andersson, B., Kim, H., Klein, M.H., Phan, L.T., Rajkumar, R. (2017). Mixed-criticality processing pipelines. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 1372–1375.

G-3. Chaki, S., de Niz, D. (2017). *Proceedings of the International Conference on Embedded Software (EMSOFT), October 15-20, 2017, Seoul, South Korea*.

G-4. Vasudevan, A., Chaki, S., Jia, L., McCune, J.M., Newsome, J., Datta, A. (2013). Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. *2013 IEEE Symposium on Security and Privacy*, 430–444.

G-5. https://sel4.systems/