DESIGN OF FAULT-TOLERANT VIRTUAL EXECUTION ENVIRONMENTS FOR CYBER-PHYSICAL SYSTEMS

Dissertation

zur Erlangung des Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

BOGUSŁAW JABŁKOWSKI

Dortmund
2019

Ort:	TU Dortmund
Fakultät:	Informatik
Tag der mündlichen Prüfung:	23.07.2019
Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr. Peter Marwedel

ABSTRACT

The last decade revealed the vast economical and societal potential of Cyber-Physical Systems (CPS) which integrate computation with physical processes. In order to better exploit this potential, designers of CPS are trying to take advantage of novel technological opportunities provided by the unprecedented efficiency of today's hardware. There are, however, considerable challenges to this endeavor.

First, there is a strong trend towards softwarization. Functions that were originally implemented in hardware are now being increasingly realized in software. This fact, together with the ever growing functionality of modern CPS, translates to unrestrained code generation which, in turn, directly influences their safety and security. Second, the spreading adaptation of multi-core and manycore architectures, due to their considerable increase in computation power, additionally generates issues related to timing properties, resource partitioning, task mapping and scalability.

In order to overcome these challenges, this thesis investigates the idea of adopting virtualization technology to the domain of CPS. Several research questions originate from this idea and the following work aims at answering those questions. It addresses both technological and methodological issues. With respect to the technological aspects, it investigates problems and proposes solutions related to timing properties of a virtualized execution platform as well as the thereon based high availability technique. Regarding the methodological aspects, it discusses models and methods for the planing of safe and efficient virtualized CPS compute and control clusters, proposes architectures for the development and verification of virtualized CPS applications as well as for the testing of non-functional characteristics of the underlying software and hardware infrastructure. Further, through a set of experiments, this thesis thoroughly evaluates the proposed solutions.

Finally, based upon the provided results and some new considerations regarding the requirements of future CPS applications, it gives an outlook towards a generic virtualized execution platform architecture for emerging CPS.

PUBLICATIONS

The ideas and findings presented in this dissertation have partly been published in the following peer-reviewed journals and proceedings of international conferences and workshops. The list is sorted in chronological ascending order:

1. B. Jablkowski, M. Mueller, and O. Spinczyk. High availability in cyber-physical systems by self-determined virtual machine replication. In Proceedings of the 13th IEEE International Symposium on Industrial Embedded Systems (SIES 2018), June 2018. [62]

2. B. Jablkowski et al. vGridLab – a testbed for virtualized smart grids. Journal of Computer Science - Research and Development, Aug. 2017. Extended abstract. [65]

3. B. Jablkowski, U. T. Gabor, and O. Spinczyk. Evolutionary planning of virtualized cyber-physical compute and control clusters. Journal of Systems Architecture, Feb. 2017. [63]

4. B. Jablkowski and O. Spinczyk. CPS-Remus: Eine Hochverfügbarkeitslösung für virtualisierte cyber-physische Anwendungen. In W. A. Halang and O. Spinczyk, editors, Tagungsband zum Thema Betriebssysteme und Echtzeit (Echtzeit 2015). Springer, Nov. 2015. [68]

5. B. Jablkowski and O. Spinczyk. CPS-Xen: A virtual execution environment for cyber-physical applications. In 28th International Conference on Architecture of Computing Systems (ARCS '15), pages 108–119, Porto, Portugal, Mar. 2015. Springer. [69]

6. A. Kubis, L. Robitzky, M. Küch, S.-C. Müller, B. Jablkowski, H. Georg, N. Dorsch, S. Krey, C. Langesberg, D. Surmann, D. Mayorga, C. Rehtanz, U. Häger, O. Spinczyk, C. Wietfeld, C. Weihs, U. Ligges, J. Myrzik, and J. Götze. Validation of ICT-based protection and control applications in electric power systems. In PowerTech Conference (POWERTECH), Eindhoven, Netherlands, June 2015. IEEE Press. [75]

7. N. Dorsch, B. Jablkowski, H. Georg, O. Spinczyk, and C. Wietfeld. Analysis of communication networks for smart substations using a virtualized execution platform. In Proceedings of the International Conference on Communications (ICC '14). IEEE Press, 2014. [37]

8. B. Jablkowski, M. Küch, O. Spinczyk, and C. Rehtanz. A hardware-in-the-loop co-simulation architecture for power system applications in virtual execution environments. In Proceedings of the Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES 2014), pages 1–6. IEEE Press, 2014. [64]

9. B. Jablkowski and O. Spinczyk. Continuous performance analysis of fault-tolerant virtual machines. In Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12), Lecture Notes in Informatics, pages 494–505. German Society of Informatics, Sept. 2012. [67]

ACKNOWLEDGMENTS

First and foremost I want to thank my advisor Prof. Dr.-Ing. Olaf Spinczyk, who provided me with the opportunity to start working in research and continuously supported me with advice and funding along the path leading to this thesis. I would also like to thank Prof. Dr. Peter Marwedel for his engagement and time to review this work.

Further, I would like to thank for the help and advise of all those, who I meet and had the pleasure to collaborate with over the years while working as a researcher. This includes people I collaborated with in the context of the DFG research group 1511, among them Nils Dorsch, Andreas Kubis, Markus Küch and Sven-Christian Müller as well as colleagues from my department: Hendrik Borghorst, Christoph Borchert, Daniel Friesel, Ulrich Gabor, Claudia Graute, Andreas Grosche, Alexander Lochmann, Matthias Meier, Michael Müller, Horst Schirmeier and Jochen Streicher

I would like to express special thanks to Markus Buschhoff – with whom I shared the office since the first day at work – and Alexander Munteanu for their support, the countless discussions and their continuous willingness to help.

Last but not least, I wish to thank my family. My parents, who for my entire live provided me with the freedom and support to pursue my interests, my wife, who assisted and supported me along the way as well as my little son, from whose very fact of existence I draw joy and motivation.

CONTENTS

ACRONYMS

ABI *Application Binary Interface*

API *Application Programming Interface*

CPS *Cyber-Physical Systems*

CPES *Cyber-Physical Energy Systems*

CP40/CMS *Control Program/Conversational Monitor Systems*

CTSS *Compatible Time-Sharing System*

DM *Deadline-Monotonic Scheduling Policy*

DMA *Direct Memory Accesses*

EA *Evolutionary Algorithms*

ECU *Electronic Control Units*

ET *Execution Time*

FOM *Federation Object Model*

FP *Fixed Priority Scheduling Policy*

GOOSE *Generic Object Oriented Substation Event*

HA *High Availability*

HiL *Hardware-in-the-Loop*

HLA *High Level Architecture*

HVM *Hardware Virtual Machine*

ICT *Information and Communication Technologies*

IED *Intelligent Electronic Devices*

ILP *Integer Linear Programming*

IOMMU *Input/Output Memory Management Unit*

ISA *Instruction Set Architecture*

LibOS *Library Operating Systems*

MAST *Modeling and Analysis Suit for Real-Time Applications*

MMS *Manufacturing Message Specification*

MOP *Multiobjective Optimization Problems*

NAPI *New API Packet Reception Mechanism*

NSGA-II *Non-Dominated Sorting Generic Algorithm II*

OLTC *On Load Tap Changer*

OMT *Object Model Template*

OPC DA *OLE for Process Control Data Access*

OS *Operating Systems*

PMU *phasor measurement unit*

PV *Paravirtualized*

RM *Rate-Monotonic Scheduling Policy*

RTI *Runtime Infrastructure*

RTT *Round-Trip Time*

RTC *Real-Time Calculus*

RTDS *Real-Time-Deferrable-Server*

QDisc *Queuing Discipline*

QEMU *Quick Emulator*

SDN *Software-defined networking*

SEDF *Simple Earliest Deadline First*

SLAT *Second Level Address Translation*

SOME/IP *Scalable service-Oriented MiddlewarE over IP*

SR-IOV *Single Root I/O Virtualization*

SRT *System Response Time*

SV *Sampled Values*

UDP *User Datagram Protocol*

VIF *virtual interfaces*

VM *Virtual Machine*

VMM *Virtual Machine Monitor*

WCET *Worst-Case Execution Times*

1 INTRODUCTION

At the turn of the century, technological advances in computing and engineering initiated a new phase of the digital revolution. An ubiquitous and profound transition from analog to digital information storage, processing and communication took place. The shift affected all aspects of life as well as all branches of industry. In early 90's, high-end car models had on average less then ten *Electronic Control Units* (ECU) installed, while in 2005 cars from the same price category had about one hundred build-in ECUs [39]. In 1993 approximately only 3% of worlds technologically stored information was in digital format, in 2007 already more than 94% [53]. Mobile cellular subscriptions reached 0.61% of the worlds population in 1993, in 2010 they were amounted to 76.5% [100].

To a great extent, the technical foundation of this paradigm shift was formed by embedded systems, which can be defined as "information processing systems embedded into enclosing products" [95]. It was, however, not solely enabled by optimizing already known solutions. It was rather due to the fact that researchers and developers, driven by the novel technological opportunities, kept opening up new areas of application for embedded systems. As a consequence, the process of conjoining computation and communication with physical processes accelerated. By functionally interconnecting tasks, areas and domains that were previously operationally independent, embedded systems started to expand massively. Once functionally dedicated, single-purposed, closed, simple and manageable solutions now became multi-layered, eclectic, interconnected complex systems with a strong trend towards openness, evidently defying their original definition. As a consequence, established design and analysis approaches for embedded systems were rendered partially inadequate. A new abstraction was needed.

In order to better reflect the complex and heterogeneous character of the newly emerged systems, Helen Gill (National Science Foundation in the U.S.) introduced, in 2006, the term *Cyber-Physical Systems* (CPS) [77], a concept denoting the "integration of computation with physical processes". Of course recognizing this dependability at that time was not a new finding, at least not in the embedded system community. Yet, the incorporation of this view allowed, in contrast to classical embedded system design and analysis approaches, for the modeling and analysis techniques to concentrate less on the computational aspect and, instead, to focus on the interaction between the computational and physical worlds.

The intensification of research in this direction yielded numerous findings. Prominent book examples that summarizes many of these are: Marwedel [95], focusing on the fundamental bases of hardware and software models, Alur [2], describing formal methods for modeling and verification of CPS, and Lee [77], describing the realization of CPS by studying the interplay of the involved models (software, hardware, physical environment) and their temporal dynamics.

On the one hand, rising abstraction levels facilitates interdisciplinary design, analysis and verification of CPS. On the other hand, as fruitful and necessary the approach proves to be, it only solves a specific subset of problems. It is often not sufficient in respect of practical realization of CPS, as many of the problems and challenges – especially in the implementation and as a consequence also in the design of execution environments – are either not being addressed by such an approach or the problems are being hidden behind different abstraction layers. This is one of the reasons why CPS designers and engineers, despite using state-of-the-art modeling and implementation techniques, are still struggling with issues related to system complexity. In fact, the situation is becoming even more complicated as the complexity of CPS is increasing rapidly. In this regard, two factors can be differentiated: the software- and hardware-related sources of complexity.

Concerning the former, a strong trend can be observed of shifting from hardware-driven designs to software-driven designs. Functions that were originally implemented in hardware are increasingly being realized in software. Further, the technologically-driven growing functionality of modern CPS translates to unrestrained code generation. This process is being inevitably reflected in the exploding size of their software stacks. For example, contemporary premium-class automobiles contain about 100 million lines of software code.

Regarding the latter, they mainly stem from the spreading adaptation of multi-core and manycore architectures. These architectures offer a considerable increase in computation power, yet in turn they generate issues and challenges related to efficient parallelization, resource partitioning, task mapping, scalability or communication. The heterogeneous landscape of offered hardware products additionally intensifies the problem.

All of these aspects render modern CPS already today heavily complex as well as software-reliant systems and it is likely that the above described direction of CPS evolution will gather momentum. This will further increase the role of software in these systems and, as a consequence, the complexity of CPS. Getting an answer to the possibly hardest question that CPS developers face today will become even more complicated, namely: How to design and implement a safe and efficient system that fulfills all of the functional and non-functional requirements and – at the same time – remains manageable? The following dissertation aims at providing an answer to this question.

1.1 MOTIVATION

The complexity of a system originates from it's functional and non-functional requirements. In the case of modern CPS, those are numerous and multifaceted. Depending on the application, the functional requirements range from simple sensor readings to complex algorithms conducting image analysis or solving optimization problems. In other words, the functional requirements of CPS cover a large spectrum of computation. Regarding the non-functional requirements, for many CPS, the most important one is meeting the timing-constraints from a given domain of application. Depending on the situation, failing at this task can result in a lower service quality or in a system failure with potential catastrophic outcome. However, there are also other non-functional requirements that CPS have to comply with. Among them are reliability, maintainability, availability, safety, security or efficiency [95]. The amount of constraints adds to the complexity of CPS.

Fortunately, the issue of complex software stacks, efficient utilization of manycore architectures as well as the optimization of multiple orthogonal requirements is not exclusive to the field of modern CPS. There exist other domains where researchers and developers are struggling with a wide range of similar challenges. A good example are large data centers, where comparable issues and challenges have been successfully tackled by the technique of virtualization – or more specific – by platform virtualization. The success of this approach is a consequence of the following features of this technology:

- It allows for the integration and consolidation of systems and system components on homogeneous and scalable hardware. This significantly reduces system complexity while increasing flexibility. Further, it allows for a considerably better resource utilization.

- Its excellent isolation properties provide a high degree of fault-containment – both in time and space dimension. This significantly impedes the propagation of errors or unwanted withholding of crucial system resources. Additionally, this also makes it difficult for malware to compromise a system.

- It enables efficient high availability solutions which significantly improve dependability by allowing systems to survive hardware failures and guarantee service continuity.

- The techniques of *Virtual Machine* (VM) migration and live migration enable proactive maintenance and ease management. In addition, they facilitate efficient load balancing.

- Finally, as a consequence of the above listened features, virtualization allows for a significant reduction of the procurement, operation and maintenance costs.

Considering the characteristics of modern CPS and their development trend, the above listed features of virtualization are of clear relevance for CPS, as they address many of the current and future challenges posed by these systems. In fact, the success stories from data centers and cloud environments have motivated a process of rethinking of the currently used system architectures. Due to the believe that virtualization technology has the potential to notably advance the automation process, many domains have started to explore the possibilities provided by this technology. This is especially true for the embedded system and CPS communities. Today, in the field of energy grids, logistic, automotive or avionic systems designers and engineers are struggling to develop novel, virtualization-based execution platforms. Unfortunately, the integration of CPS by means of virtualization is not a straightforward task as virtualization technology was initially not designed to cope with strict timing constraints. These, however, are omnipresent in CPS. Despite this fact, until recently, research has put the most emphasis on topics of fault-tolerance and hardware utilization. There is little literature on how to improve the real-time capabilities of this technology or how to model and integrate a CPS by means of virtualization. This is a relatively new and open research field.

1.1.1 *Can It Work?*

As discussed above, virtualization technology comes with many attractive benefits of which several are of high interest to the embedded and cyber-physical system communities. However, also reservations regarding the adoption of virtualization technology for embedded systems exist and have been expressed in literature. A prominent example is the article [51] by Gernot Heiser. The main points of criticism are that virtualization can not address the scheduling needs of embedded systems, as it is characterized by a hierarchical scheduling model which is, in fact, inherent to this technology, and that the virtual-machine model does not support an efficient way to share data between the components of a system. Finally, VMs are heavyweight and thus not suitable for embedded hardware and systems.

As far as the first objection is concerned, in Section 3.3.2.1, we describe how the concept of unikernels can help in overcoming the problem of hierarchical scheduling. Unikernels are lightweight enough to encapsulate individual threads into VMs and, as a consequence, allow for imposing global scheduling policies and facilitating timing analysis. Moreover, they also provide an answer to the reproach of resource inefficiency (see Section 2.1.6). In recent literature [93],

unikernels are shown to be as efficient as modern lightweight virtualization or container technologies, such as Docker [34] or LXC [84], yet without their substantial security issues. A trade-off between isolation (VMs) and efficiency (container) does not have to exist with unikernels. When optimally tailored, unikernels can have as small footprints as 480KB and run in as little as 3.6MB of RAM. Further, unikernels can be booted as fast as in a few milliseconds and thus comparable to process startup times – note that Docker containers need about 200ms.

Other techniques that can help to mitigate the discussed scheduling problem are partitioning approaches. For example, while using the Xen hypervisor [10], *cpupools* can be used for this purpose. This approach allows to partition the physical cores on a machine into different pools. Each of these pools can have a separate CPU scheduler, which can be set to meet different scheduling requirements. This is a flexible way to separate scheduling concerns, if needed, VMs can still be moved on-the-fly from pool to pool and the pools itself reconfigured, in order to meet new constraints.

Also with regard to the possibility of efficient data sharing between components of a virtualized system new possibilities emerged. The problem of efficient information exchange between VMs can be tackled by shared memory systems which are based upon granting mechanisms (see Section 2.1.5.2). This way, a VM can explicitly offer parts of its memory (typically at page granularity) to other VMs and set the operation flags to the desired access level.

Yet the main point of criticism addresses I/O operations. Traditionally, virtualized systems maintain isolation properties at the cost of at least one extra copy operation. From the perspective of embedded systems using single address space shared buffers with simple synchronization primitives, this seems inefficient. We want to point out two approaches that help to mitigate the problem for virtualized systems without sacrificing isolation properties.

The first is the hardware extension IOMMU, which provides secure virtual machine guest operating system access to selected I/O devices by safely remapping guest-physical addresses to host-physical addresses. This extension enables the socalled pass-through technology for passing or dedicating devices to VMs.

The second approach is called *Single Root I/O Virtualization* (SR-IOV) [36, 108]. SR-IOV provides additional definitions to the PCI Express specification in order to enable I/O virtualization methods based on PCI Express native hardware. This technology allows a PCIe device to appear as multiple separate physical PCIe devices. The natively virtualized devices are directly accessible to the guests without the involvement of the hypervisor. SR-IOV together with IOMMU can allow for a efficient, secure and scalable device access.

In the end, however, there is no free lunch. At some point, a system designer dealing with these kind of questions will be confronted

with the decision to either partition/dedicate hardware or to adopt some form of multiplexing. Whether classical shared buffers coupled together with simple low level synchronization primitives are better suited than paravirtualization, or the above discussed approaches, has to be decided depending on the given use case and the safety and timing requirements it imposes on the system.

We argue that none of the discussed reservations regarding the adoption of virtualization technology seems to constitute an insurmountable problem. We claim that from the technical point of view, virtualization in embedded and cyber-physical systems is possible and feasible. The next chapters of the following thesis will scientifically substantiate this claim.

1.2 RESEARCH QUESTIONS

Several research questions originate from the idea of integrating CPS by means of virtualization. Those can be roughly divided into two categories: the technological- and methodological issues. Considering the latter, two aspects can be additionally differentiated focusing either on the functional- or on the non-functional properties. Corresponding to these categories, this thesis addresses the following ones:

- Technological Issues
  - Can real-time capabilities of virtual execution environments be improved, so that they meet the strict timing-constrains of CPS?
  - Are contemporary high availability solutions via virtual machine replication suitable for improving the dependability of CPS?
- Methodological Issues
  - How to adequately model a CPS and include the different non-functional requirements?
  - Does a holistic – including the execution platform, communication infrastructure and fault-tolerance aspects – model exist that allows for the planning of safe and efficient virtual execution environments?
  - How to test virtualized CPS applications during development?
  - How to consider the complex and interwoven dependencies between the execution platform, communication infrastructure and the physical world in the process of verifying the functional correctness of the CPS application?

1.3 SCIENTIFIC CONTRIBUTIONS

The scientific contributions of this thesis address the research questions stated in the previous section. In detail, this work advances the state-of-the-art by the following contributions:

- Technological Issues
  - In order to enable the execution of time-constrained virtualized CPS applications, we analyze the architecture of Xen [10] – a popular, performant and open-source *Virtual Machine Monitor* (VMM) – identify its shortcomings with respect to real-time capabilities and extend it with a suite of real-time schedulers [69].
  - To solve an issue of priority inversion during network packet processing, we introduce an architecture that synergizes the work of the VMM-scheduler with I/O-scheduling [69].
  - To facilitate scheduling [69] and analysis [63] as well as an efficient high availability solution [62], we propose and evaluate the utilization of *unikernels* as guest *Operating Systems* (OS).
  - In order to increase the dependability of virtualized CPS, we propose a novel approach to high availability and present a *self-determined* virtual machine replication model that reduces latency costs by an order of magnitude when compared with state-of-the-art techniques [62, 68].
- Methodological Issues
  - We propose a methodology for the planning of safe and efficient virtualized execution environments which aim at hosting time-constrained virtual machines [63]. To this end, we combine evolutionary algorithms with formal system performance analysis – in particular algorithms considered in classical scheduling theory. We show that such an approach allows to optimally dimension the execution environment and – at the same time – provides strict guarantees regarding the timing predictability of an integrated CPS.
  - We present a *hardware-in-the-loop* (HiL) *Hardware-in-the-Loop* (HiL) co-simulation architecture for the development and testing of virtualized CPS applications [64].
  - Based upon application requirements from the smart grid domain, we describe how the complex dependencies between the *Information and Communication Technologies* (ICT) and the electric power system can be taken into account during application development and verification [64, 65].

- Software Contributions: CPS-Xen and CPS-Remus
  - We developed CPS-Xen with CPS-Remus. A virtual execution environment for a dependable and efficient hosting of virtualized CPS applications. It is based upon the Xen-Hypervisor and implements the above listed scientific contributions related to the architectural aspects. CPS-Xen was used as the evaluation platform for obtaining the results presented in this thesis. For reasons of openness and reproducibility of the presented results, the entire source code is available for download on our GitHub project page[1] [27]. The software stack also includes all of the extensions made by us to the MiniOS-Unikernel.

1.4 CONTRIBUTIONS OBTAINED IN COOPERATION

In accordance with §10(2) of the PhD regulations of the department of computer science, TU Dortmund, 2011, for all results presented in a thesis, which were obtained in cooperation, an additional list has to be provided that separates the author's own contributions. A separate acknowledgment has to be given to my advisor Prof. Dr.-Ing. Olaf Spinczyk who contributed advice, ideas and technical comments to all publications presented in this thesis.

- Chapter 4: This chapter concerns the methodology of planning safe and optimally dimensioned virtual execution environments for CPS applications. My contribution is the idea and the design of the proposed methodology as well as the concept for evaluation. I am also the principal author of the publication that presents the obtained results [63]. This contribution was achieved in cooperation with Ulrich Gabor and is based upon his master thesis, which I supervised.

- Chapter 5: The fifth chapter describes an architecture for developing and testing of CPS applications. This architecture was developed in cooperation with Markus Küch and is partially based upon his master thesis. We published the results – with me as the principal author – in [64]. My contribution to these results is the overall concept of the presented architecture which conjoins a co-simulation approach from the master thesis with my virtual execution platform. Furthermore, I contributed the implementation and evaluation parts that are related to virtualization technology.

1 https://github.com/cpsxen/cps-xen

1.5 THESIS STRUCTURE

This thesis is structured as follows:

- Chapter 2: *Background* (pages 11-26)
  The second chapter provides background information that facilitates the understanding of the scientific contributions presented in this thesis. It introduces basic notions from the area of virtualization technology and briefly discusses different approaches to system performance analysis. Further, it also provides related work for the topics presented in this chapter. In-depth literature that is directly related to our contributions is being discussed in corresponding chapters and sections.

- Chapter 3: *Virtualized CPS-Architecture* (pages 27 - 70)
  This chapter describes the architecture of our virtual execution environment, it discusses the technological aspects related to its infrastructure software and presents our scientific and software contributions in this area.

- Chapter 4: *Planning Virtualized CPS* (pages 71 - 92)
  The fourth chapter presents a methodology for the construction and integration of safe and efficient cyber-physical systems by means of virtualization. It describes the necessary techniques, the corresponding models and discusses how system designers and administrators can benefit from this approach.

- Chapter 5: *Testing of Virtualized CPS* (pages 93 - 104)
  The fifth chapter proposes architectures for the verification and testing of virtualized cyber-physical systems, including the applications as well as the underlying computer hardware and field devices.

- Chapter 6: *Discussion and Outlook* (pages 105 - 110)
  This chapter discusses the topic of safety certifications as well as anaylzes the potential requirements of future CPS applications. Based on this analysis, it provides an outlook in form of a generic execution platform architecture for emerging CPS applications.

- Chapter 7: *Summary and Conclusion* (pages 111 - 114)
  The last chapter summarizes and concludes this thesis.

- Appendix a: *Numerical Evaluation Results* (pages 115 - 116)
  The seven chapters are succeeded by an appendix presenting the full results of the experiments comparing the CPS-Xen RM and RT-Xen RDTS schedulers in Section 3.3.4.

2 BACKGROUND

This chapter provides background information with respect to virtualization technology and system performance analysis. We refrain from an exhaustive and in-depth description of those fields, as this would go beyond the scope of this thesis. Therefore, in the following sections, the aforementioned research areas are being presented to an extend that is necessary for the understanding of the scientific contributions described in this work.

2.1 VIRTUALIZATION TECHNOLOGY

This section begins with a brief historical perspective and definitions with respect to virtualization. Next, in order to provide a context for our findings, we discuss different types of execution environments. The discussion is being followed by the introduction of the technique of *platform virtualization*. After presentingplatform virtualization, the architecture of the Xen hypervisor – including its high availability solution Remus – are being discussed. The section concludes with information regarding *unikernels*.

2.1.1 *A Brief Historical Overview*

The idea of virtual machines has its origins in the 1960s, in the so-called mainframe era. At that time, mainframe computers were expensive machines. Institutions and companies were searching for means that would allow multiple users to simultaneously share a common hardware platform. A pioneer system that met these requirements, by implementing the concept of a time-sharing operating system, was presented in 1962 [26]. It was developed at Massachusetts Institute of Technology's Computation Center and called the *Compatible Time-Sharing System* (CTSS) . Few years later, the IBM company took the time-sharing concept to a new level by developing a revolutionary architecture that could simultaneously run multiple instances of an operating system by using virtualization. The *Control Program/Conversational Monitor Systems* (CP40/CMS) [1, 28] was able to create fourteen virtual IBM System/360 environments – virtual duplicates of real System/360 machines. The CP was responsible for the creation and management of virtual machines, while CMS was a lightweight single-user operating system that was developed before CP existed and originally aimed at the evaluation of modularization concepts in system evolution [28]. In contrast to conventional time-sharing systems,

which divided the computer resources between multiple users, the CP provided each user with an emulated stand-alone computer. Several releases followed and soon the CP/CMS were able to host dozens of OS's and time-sharing users. The CP/CMS architecture was revolutionary as it established a novel technological paradigm. All contemporary virtualization solutions are descendants of this architecture.

Even though virtualization technology started with the idea of abstracting mainframe resources, it became relatively quickly clear that beyond resource partitioning virtualization also provides the means to solve a variety of problems when it comes to compatibility in a broad sense. As a consequence, the concept of virtualization was also adopted by other areas, among them to compilers, operating systems and programming languages.

To date, 50 years after the invention of this technology, plenty of virtualization types emerged. The most important ones are: application-, network-, desktop-, operating system-, and platform virtualization. Even if not always fully aware, most of us use one or multiple forms of virtualization every day.

2.1.2 *What Is Virtualization Technology?*

A good way to explain the concept of virtualization is by using another concept, the emulation. Emulation and virtualization are to some degree similar. Virtualization can even be interpreted as an extension of emulation. In fact, for many virtualization techniques emulation is an enabling technology. What they have in common is that both create logical representations of hardware. The main difference lies in the scalability. While with emulation one system can imitate a behavior of another system, virtualization is able to simultaneously support, on a single machine, an arbitrary number of different systems.

Another way to describe virtualization is by means of interfacing [124]. When a component is being virtualized, an additional layer of indirection is being added for the accessing of its resources. In the process, this interface, and the resources visible through this interface, are being mapped onto the interface and resources of the real component. By this, the real component (or an entire system) can appear as one or multiple different ones.

Finally, virtualization can also be defined as an isomorphic relation between a *guest* and a *host*, and the process of virtualization as the construction of such an isomorphism [111]. More formally, it is a mapping which satisfies the condition that for a sequence of operations, which modify the state of a guest, a corresponding sequence of operations exists on the host, which perform an equivalent transformation of its state. Readers interested in a more detailed description of formal definitions with respect to virtualization, are referred to the publication of Popek and Goldberg [111].

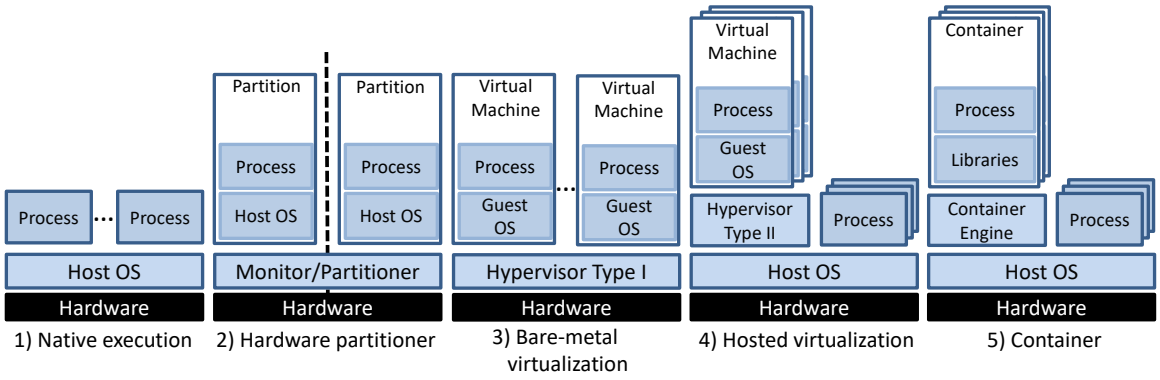


Figure 2.1: Types of execution environments.

In computer science, a common practice for solving complexity issues is to introduce abstraction layers that are separated by interfaces. It helps to ignore implementation details and to concentrate on higher level functionalities. In a way virtualization embeds this approach, yet besides facilitating design and management, it also provides the means for a relaxation of resource constraints and the solving of compatibility issues. This is why there exist virtualization solutions for any level of a system architecture, including both hardware- and software components.

In order to focus the discussion and define a context for the findings presented in this dissertation, before we proceed with the discussion of platform virtualization, the next section provides basic notions with respect to types of execution environments. For an exhaustive description of the virtualization technology as such, please refer to Smith and Nair [124].

2.1.3 *Types of Execution Environments*

Figure 2.1 depicts architectures for different types of execution platforms, which are being discussed in contemporary literature and press articles in the context of virtualization and efficient service or application development and deployment.

Native execution (1) represents platforms that run software – more or less – directly on hardware for which this software was specifically implemented for. In other words, this architecture represents non-virtualized execution environments.

Hardware or system partitioners (2) provide system-wide hardware isolation for software. They are being used to host bare-metal applications or operating systems directly on the hardware without any additional abstraction layer. The possibility of resource or device sharing is limited and mostly only hardware-based. Examples of partitioners are the *ARM Trusted Firmware* (ATF) [1] and *Jailhouse* [115]. The former is

1 https://github.com/ARM-software/arm-trusted-firmware

a reference implementation of the *ARMs TrustZone* technology [2]. It divides the hardware of the system in two environments, the so-called secure world, aiming at hosting safety and security sensitive workloads, and the normal or less trusted world. The latter – *Jailhouse* – is a lightweight real-time static partitioner for multicore platforms based on Linux.

The software for creating logical abstractions of hardware as well as instantiating and running virtual machines is called *Virtual Machine Monitor* (VMM) or *hypervisor*. Traditionally, hypervisors running directly on the hardware are being referred to as *bare-metal* or *type-1* VMMs (3) [46]. Beside hardware partitioners, hypervisors provide the strongest isolation properties by leveraging dedicated hardware virtualization extensions. They also provide an advanced and rich set of features with respect to resource and device sharing. Most prominent examples of bare-metal hypervisors are *Xen* [10], *VMware ESXi* or *Microsoft Hyper-V*. A special case represents the *L4Re Microkernel*, a microkernel-based operating system framework with support for type-1 virtualization yet with a split functionality between the hypervisor and the user-space based VMM. It is based on the Fiasco.OC microkernel [110] from the L4 microkernel family [52].

VMMs running on top of a host operating system are being referred to as *type-2* or *hosted* VMMs (4), as they coexist with other programs executed under the host operating system. In this architecture, the created virtual machines run as processes on the host operating system. Two popular type-2 hypervisors are *Virtualbox* and *VMware Workstation Player*. A special case constitutes the *Kernel-based Virtual Machine* (KVM) [74] hypervisor. There is a discussion on how to classify KVM. On the one hand, VMs under KVM run as regular Linux processes and thus rely on the host OS. On the other hand, the KVM module turns the Linux kernel into a hypervisor that closely resembles a type-1 VMM.

Finally, an execution environment that is currently gaining enormous traction are *containers* (5). Containers are instances of user-space software packages isolated and run by a single kernel. They hold assets like libraries, files and other dependencies that are necessary for the encapsulated applications or system to run. The *containerization* approach is also being referred to as *operating-system-level virtualization* or *lightweight virtualization*. Containers facilitate the development and deployment of services by abstracting code from infrastructure, simplifying configuration as well as enabling scalability and portability. Popular examples of container engines are LXC [84] – for system containerization, and Docker [34] – for application containerization.

In the following section, we will discuss the concept of platform virtualization – as well as the enabling techniques – in more detail as it forms the basis for the results presented in this dissertation.

2 https://developer.arm.com/technologies/trustzone

2.1.4 *Platform Virtualization*

The form or type of virtualization can be determined based on where in a system architecture the virtualization software is located. In the case of platform virtualization, the virtualization interface is placed between the *host* and the *guest*. With regard to terminology, the underlying hardware of a machine is usually being referred to as a *host* and the executing software – the operating system including the encapsulated applications – as a *guest*. Both bare-metal as well as hosted virtualization enable platform virtualization. However, as type-1 VMMs reduce one layer of indirection and give the hypervisor an exclusive control over the hardware, they are – in respect of efficiency and security – the preferred technology for implementing platform virtualization.

The major feature of platform virtualization is the ability to support the execution of multiple different OS simultaneously on a single hardware. To this end, the virtualization layer has to provide a complete system environment (CPU, memory, I/O) to all of its guests. In order to facilitate this, the virtualization software has to multiplex the existing physical resources among the running guests. This is done by the VMM. The VMM manages all hardware resources of the underlying host platform and traps privileged accesses in order to verify their correctness. Normally, the guest software is unaware of this indirection. Note that not all of the virtual hardware presented to the guests has to be available on the physical host. The virtualization software can embed emulation in order to provide the desired resource.

In literature, *platform virtualization* is sometimes being referred to as *system virtual machines* [124] – as it aims at providing system replication. Another term is *server virtualization*. This term originates from the great popularity of this virtualization form in the server domain. Connected to this popularity is the computer architecture that is most commonly being virtualized, the x86 architecture. Since decades, this architecture is widespread and very popular. Ironically, it is one of the more difficult architectures to virtualize. The issue resides in the concept of privilege levels, known as rings, of the x86 architecture and a set of seventeen nonvirtualizable instructions [117]. Nonetheless, its popularity made it a very attractive target and a lot of effort has been put into overcoming the limitations of this architecture in respect of virtualization. In the following, a description of the popular solutions enabling platform virtualization on x86 is provided.

2.1.4.1 *Binary Translation*

Binary translation [98, 123] or *binary rewriting* is a technique for detecting and patching the set of problematic instructions – the non-trapping sensitive instructions – on the x86 architecture. As those instructions have to be emulated, the VMM dynamically analyzes the instruction

stream, identifies the ones that have to be substituted and rewrites them so that they produce the intended effect. As a consequence, most of the code of the guest can be executed in user space. The obvious downside of this approach is the additional indirection that imposes a performance penalty, especially in the case of intensive I/O operations. Binary translation was popularized by VMware. VMware refers to its platform virtualization solution that combines binary translation and direct execution of instructions as full virtualization. However, this seems disputable as in a traditional sense full virtualization translates to providing the guest with an interface to the full system architecture – the *Instruction Set Architecture* (ISA) . The virtual hardware exposed to the guest is typically functionally identical to the underlying hardware. For example, this was the case with the IBM System/370 which had a total commonality between the ISA of the virtual machine and the real machine [121].

2.1.4.2 *Paravirtualization*

The *paravirtualization* technique [10, 132] takes a different approach. Instead of dealing with the emulation of the problematic instructions, it bypasses them, respectively, delegates their execution. This is done by exposing a virtual machine interface to the guest that is similar but not identical to the underlying hardware. In the case of the most prominent embodiment of the paravirtualization technique, the Xen hypervisor [10], the guests are being presented with a *hypercall Application Programming Interface* (API). *Hypercalls* are conceptually similar to system calls. Through this interface the OS can delegate the execution of privilege instructions to the VMM and thus significantly reduce the virtualization overhead. The downside of this approach is that using the hypercall-API requires a modification of the guest OS kernel. Although no changes to the *Application Binary Interface* (ABI) are necessary and therefore no modifications the guests application code are needed. A significant part of the evaluation results presented in this thesis are linked to this form of platform virtualization.

2.1.4.3 *Hardware-assisted Virtualization*

The notion *hardware-assisted virtualization* denotes a class of hardware enhancements that aim at improving the execution performance of virtual machine environments. The first to implement such extensions was IBM. IBM introduced additional hardware to their System/370 machines, in order to assist the VMM in instruction emulation. In respect of the x86 platform, in 2005 and 2006, Intel and AMD added a set of new instructions that facilitates virtualization on x86. These technologies are called Intel VT-x and AMD-V, respectively. In 2010, also ARM announced architectural support for virtualization and released, in 2011, first CPUs with this feature. Depending on the solution the

implementation details may vary, though conceptually they are similar. All three extensions introduce an additional mode of operation. This allows the hypervisors to be executed in an extra privileged mode, while the guest OS is able to execute with its traditional OS privileges (within ring 0 on x86 or supervisor mode on ARM). In the context of Xen, this virtualization form is called *Hardware Virtual Machine* (HVM). The advantages of HVM are faster transitions for system calls and – when compared with paravirtualization – the support for unmodified operating systems. Introducing this mode theoretically removes the need for either paravirtualization or binary translation. However, there are also some trade-offs. HVM innately does not provide virtual devices. Those have to be emulated, along with BIOS, timers and interrupts. The cost of this is performance. At least this was the case with the first generation of hardware-assisted virtualization extensions where only features regarding CPU virtualization were provided.

2.1.4.4 *Hybrid Virtualization*

Since 2006, manufacturers extended hardware assistance for virtualization by iteratively adding new features. The introduction of nested paging or *Second Level Address Translation* (SLAT) allows the hypervisor to take advantage of hardware supported translation from *pseudo-physical* or *real-memory* addresses to machine addresses. Prior to that, this mechanism had to be implemented in software – for example using the *shadow page tables* [124] concept. Beyond memory virtualization, another important extension concerns with I/O virtualization. Through the remapping of *Direct Memory Accesses* (DMA) and interrupts, the *Input/Output Memory Management Unit* (IOMMU) allows guests to use devices directly – without the intervention of the hypervisor. In order to take advantage of hardware assistance, hypervisors are trying to incorporate guest support for those features. An approach that combines both the software- and hardware solutions is often being called *hybrid*. Such an approach seems only natural, as it combines the best of both worlds. Eventually, hardware assistance will replace all need for software assistance. However, in respect of Xen, compatibility issues still exist and a comprehensive and stable software support is still being in development.

2.1.5 *The Xen-Hypervisor*

The results presented in this dissertation are linked to the Xen-Hypervisor [10]. There are several reasons why we have originally chosen Xen as the target platform to evaluate our research:

First, Xen is an efficient platform virtualization solution. Due to the technique of paravirtualization, Xen provides low-latency I/O processing, which is of high importance for CPS. Next, even though Xen does not include device drivers itself, it instantiates a privilege guest

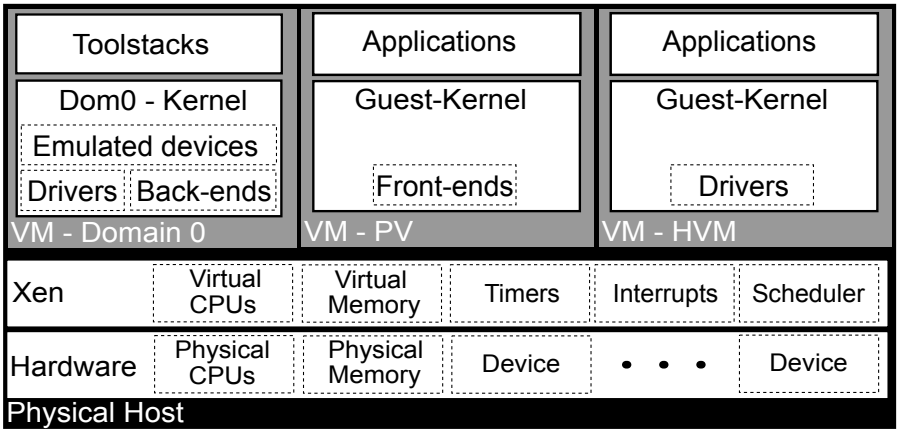


Figure 2.2: The Xen architecture with a simple configuration of one paravirtualized (PV) and one fully virtualized guest (HVM).

(called Domain 0 or Dom0) that facilitates device utilization for guests. Domain 0 typically runs the Linux operating system – although Solaris or NetBSD are also supported. By leveraging Domain 0, Xen indirectly supports all drivers available for the Linux operating system. Moreover, when porting a new operating system to Xen, it is not necessary to implement a myriad of drivers and repeat work that has already been done by others. For this purpose, Xen provides an abstract and simplified interface to devices – the *split device driver model*, which will be described later in this chapter. Besides x86, Xen also supports the ARM architecture, a predominant platform for embedded- and cyber-physical systems – two domains where virtualization technology is gaining significant interest. Furthermore, at the time of decision, Xen was the only VMM to provide a relatively efficient *High Availability* (HA) solution [30]. Finally, the Xen hypervisor is an open-source project. This significantly facilitates research, as all parts of its software architecture are open to adaptation and extension.

2.1.5.1 *The Architecture of Xen*

The physical host, illustrated in Figure 2.2, represents a server which deploys the Xen hypervisor. Xen runs directly on the host's hardware and is the first software to execute after the system leaves the bootloader. The hypervisor is responsible for managing hardware resources, including the CPUs, memory and interrupts. Further, it also handles timers as well as the scheduling of VMs.

After setting up all of the required software structures, Xen boots a privileged guest – Domain 0 (*Dom*0). Dom0 is always the first domain to load under Xen, as it holds the drivers for the underlying hardware devices. Beside some xen-specific infrastructural software, this is also were Xen's management API resides. The Xen API (not to be confused with the hypercall-API used by the guest's kernels) is being utilized

by different toolstacks (most popular being the *xl-toolstack*) in order to provide user space VM management functionalities for administrators.

Figure 2.2 also depicts two additional domains: one *Paravirtualized* (PV) guest and one fully virtualized HVM guest. In contrast to Dom0, these domains are restricted (e.g. are not allowed to access hardware directly) and are therefore also called *unprivileged domains* (DomU) in Xen. Both guests encapsulate an operating system and some applications. However, due to the different virtualization form, they differ in way they access hardware devices. The PV guest makes use of the Xen's split device driver – to be more precise it implements the *front-end* part of this model. In contrast, the HVM guests uses conventional drivers provided by its operating system while Xen emulates the corresponding devices. To this end, Xen utilizes the well-known open-source *Quick Emulator* (QEMU).

In the following, we will discuss the split device driver model in more detail as it provides a good illustration of Xen's overall paravirtualization approach.

2.1.5.2 *The Split Device Driver Model*

The motivation for the split device driver model is – besides omitting the x86 virtualization issues – to make use of the already existing drivers in Dom0. However, as not all hardware devices are designed to cope with multiple access, some form of multiplexing has to be provided. In this regard, Xen mimics the approach encountered in operating systems. Operating systems provide different software abstractions, in order to facilitate hardware access for processes. For VMs, an analogous service is being provided by the split device driver.

The split device driver is composed of components which are the backbone of Xen's paravirtualization approach: the *grant tables* – a generic mechanism for sharing memory between domains, *events* – a concept of software signals and interrupts, and the *XenStore* – a hierarchical, tree-like transactional key-value data structure for exchanging configuration information between domains. The keys denote a path in the tree and the values store domain information. The driver itself consists of two parts: the *front-end* and the *back-end*. Both parts are isolated, the back-end typically resides in Dom0, whereas the front-end is located in DomU. Now, how do those elements all fit together?

Figure 2.3 exemplifies the architecture of the split device driver for the network driver. Suppose a guest domain wishes to send out a network packet. In order to do that it has to fulfill the following requirements.

First, the guest domain needs a virtual device – in this case one that represents a network card. Analogues to the driver, this device is split into two parts. One resides in DomU and is bound to the front-end driver (*netfront*), the other is typically located in Dom0 and connects to the back-end part (*netback*).

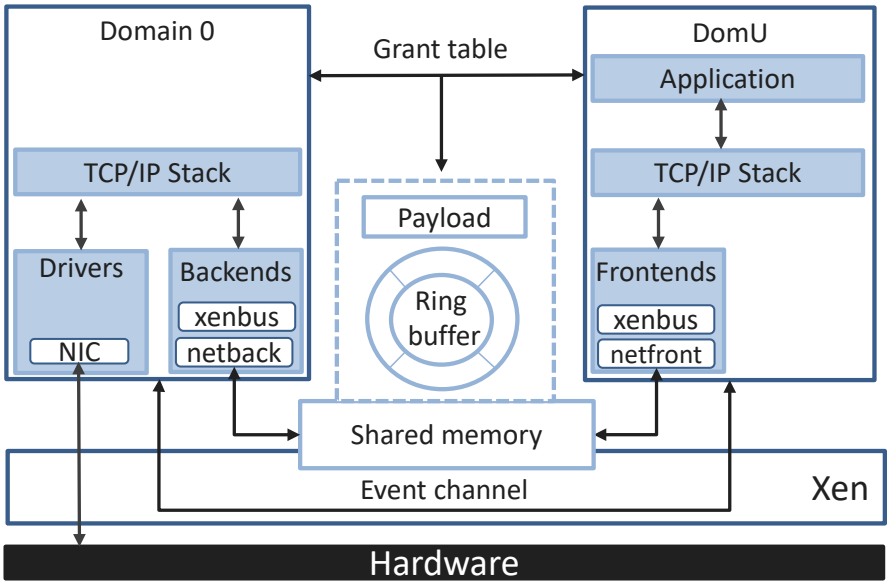


Figure 2.3: The Xen split device driver model exemplified for networking.

Next, using the grant table mechanism, DomU has to dedicate memory for inter-domain communication. On top of the shared memory segments it has to instantiate a ring buffer – a data structure for the realization of the producer-consumer communication model. The ring buffer is being filled with requests and responses that contain instructions, e.g. a *grant reference* to the granted page. Data (*payload*) is transmitted elsewhere, and for this purpose, DomU has to share additional pages which then are to be referenced within the instructions. Finally, in order to deliver the asynchronous request and response notifications, a communication channel has to be established between the two domains. To this end, both domains bind to a dedicated *event channel*. In contrast to interrupts, these channels are bidirectional and connection-oriented. The IDs of the channels are being announced through the XenStore.

After setting up the required infrastructure, DomU can insert data into the granted shared memory pages, enqueue a corresponding request in the ring buffer and notify – through the dedicated event channel – Dom0 that there is a packet pending. Dom0 is then able to read the packet and pass it to the appropriate components of the operating system (e.g iptables), in order to deliver it to the real driver.

Readers interested in more details regarding the architecture of Xen are referred to [20].

2.1.5.3 *Remus - High Availability in Xen*

High availability solutions aim at ensuring a continuous operation of a system. A crucial aspect of their efforts relates to the ability of surviving hardware failures. Classical, hardware-based solutions,

which involve physically redundant components, are fast and reliable, yet expensive.

Remus [30] – an extension to the Xen hypervisor – takes a different approach. By capitalizing the techniques of VM replication and live migration [23], Remus provides a high availability solution implemented solely in software. In face of hardware failure, it allows a running system to transparently continue execution on an alternate physical host. This process of switching to a redundant source of computation is known as *failover*.

To this end, Remus periodically suspends the execution of the protected guest, captures its state and asynchronously transmits it to a backup host. The captured state is also being referred to as *snapshot* or *checkpoint* while the process of capturing is being called *checkpointing*. Each checkpoint comprises of *dirtied pages*, that is, memory that has been altered since the previous round. Due to the periodic character of the replication process, we refer to this approach as to the *periodic checkpointing model*. In order to minimize the amount of time in which the protected guest has to remain suspended, Remus allows the guest to resume its execution ahead of synchronization points, that is, without waiting for the backup to acknowledge the receipt of the last checkpoint. This technique is known as *speculative execution* [104]. The combination of the asynchronous replication and speculative execution allows for an efficient guest protection, especially when compared to the performance penalties induced by the classical approaches to VM replication, e.g. the lock-step [16] technique. However, in contrast to the deterministic lock-stepping approach, Remus cannot guarantee that a restored guest will produce the same output as before a failover.

In order to prevent the impression of inconsistent states between the protected guest and its backup, Remus provides the possibility to buffer output until the state of both VMs has been synchronized. This is, after the backup has acknowledged the receipt of the last checkpoint. Remus refers to this option as *network buffering*. An enabled network buffering option guarantees that an external view on the system will remain coherent – also in the event of a failure. The cost of coherency preservation is latency.

For the purpose of monitoring the availability of a platform, Remus utilizes a heartbeat mechanism. A signal in form of a network packet is being periodically exchanged between the protected guest and the backup guest. In face of failure, the backup identifies the absence of the heartbeat and initiates a failover process. As a result, the service resumes its execution on the alternate host. Note that in the current implementation, there is no mechanism that identifies the failure of a backup host. Only a failure of the master can be detected.

Finally, in contrast to other software-based high availability solutions [35, 116], Remus does not rely on parallel execution. As long as the protected guest is alive, there is only one VM instance of a

service actively running. The backup VM resides in the memory of the backup host, yet its execution is suspended and therefore, it does not consume CPU resources. However, in contrast to solutions adapting redundant execution, Remus is highly demanding in respect of network bandwidth.

Unfortunately, for several reasons Remus is not suitable for protecting CPS applications. In fact, to the best of our knowledge, there exists no virtualization-based approach to high availability that fits the needs of the CPS domain. The current techniques are either inadequate or inefficient. The further elaboration of this assessment is provided in Chapter 3 together with the proposal and discussion of a novel model for high availability that avoids the drawbacks of the established techniques.

2.1.6 *Unikernels*

In the context of platform virtualization, it is common practice to employ general-purpose operating systems for guests. Those provide a lot of functionality, ease the deployment of common services or legacy software and are accessible not only to experts. However, fully functional operating system are demanding in terms of resources. In fact, in most cases, to perform a task of interest they consume significantly more resources than actually necessary. Nevertheless, in data-center or cloud environment domains, such an approach is being accepted as it does not interfere with the fulfillment of the functional requirements of commonly deployed services. It does, however, translate into cost issues and efforts are being made to reduce the generated overhead.

In contrast, in the domain of CPS, where functional requirements are often as critical as non-functional requirements, such an inefficiency will not only result in a waste of resources but it can also lead to a system failure. This is due to the fact that the generated overhead negatively impacts crucial system properties like timeliness or availability.

Regarding the former, in order to guarantee a wide spectrum of functionalities, general-purpose OS's instantiate a lot of additional processes of which most of them are not related to the service of interest. The instantiated processes – even if properly isolated by means provided for this purpose by the OS – still interfere with the execution of the service in question. This negatively impacts the service timing predictability.

Considering the latter, the memory overhead generated by processes that are not service related also induces an adverse impact on the efficiency of high availability solutions. During VM replication all of the modified pages have to be transferred to the backup host. This also includes the pages that were altered by processes that are not service related. Even in the case of lightweight Linux distributions aiming at

virtual appliances, due to this phenomena, the unnecessary increase in checkpoint size can be in average amounted to 2 MB [68].

Finally, the deployment of dedicated services embedded in general-purpose operating systems also poses serious security risks – both in the case of CPS as well as cloud environments. For the aforementioned reasons, a rationalization of the deployment process seems in order.

In fact, in recent years successful efforts have been made to increase the resource efficiency of service deployment in virtual environments. These efforts are based upon the concept of *Library Operating Systems* (LibOS) [40, 42, 86]. LibOS's allow to optimally adapt the required OS code base to the particular needs of an application. For each application only those parts of the OS API are being implemented, and later on compiled into a VM image, on which the application actually depends. This way, sealed against modifications, single-purpose appliances [90, 91, 94, 112] – also called *unikernels* – can be constructed. Such an approach to application deployment has several advantages. Unikernels have a significantly reduced images size, this minimizes the attack surface for malicious code injection. The fact that no unnecessary services are being executed inside the unikernels additionally increases their security properties. Further, the tailored character of unikernels allows for a substantial reduction of the overall system resource usage. This also translates into increased system performance and, as a consequence, facilitates the fulfillment of non-functional requirements. Finally, due to the fact that unikernels allow for the encapsulation of individual threads into VMs, the hierarchical scheduling model that is inherent to virtualization technology can be flattened, facilitating global scheduling and timing analysis (see Section 3.3.2.1).

Considering these aspects of unikernels, as well as the fact that most CPS applications are specialized and functionally dedicated tasks and can therefore be implemented as single-purpose appliances, CPS applications provide an excellent target for unikernels. This is why, in this thesis, we leverage unikernels for the construction and evaluation of fault-tolerance CPS applications. In particular, we employ and extend MiniOS which is a tiny paravirtualized OS kernel – originally distributed with the Xen project – that serves as the basis for most of the currently available unikernels.

2.2 SYSTEM PERFORMANCE ANALYSIS

Designing virtual execution environments for cyber-physical systems is a daunting task. The problems originate from the intrinsically heterogeneous and distributed character of CPS, the strict non-functional constraints imposed on those systems by the physical environment as well as the technological challenges when virtualizing time-critical applications. Therefore, a successful process of designing, planning or testing virtualized CPS has to rely on methods that are able to

Empirical methods (Simulating, test, measurement)	*State-based verification (Model checking)*	*Analytical methods (Mathematical system abstractions)*
+ Large modeling scope	*+ Exhaustive*	*+ Exhaustive*
- Not exhaustive	*+ Accurate (exact)*	*+ Fast*
(-) Slow	*-Slow (state-space explosion)*	*- Limited modeling scope*
		- Limited accuracy
		- Pessimistic results

Table 2.1: Approaches to system performance analysis.

quantify relevant system characteristics in a holistic manner, this is, by taking all aspects of the system – as well as its environment – into account. Only by this, a robust evaluation of a virtualized execution platform for CPS can be guaranteed.

In general, the task behind the objective of collecting data that reflects the performance of a system is being referred to as *system performance analysis*. There exist different approaches to system performance analysis, all having their advantages and disadvantages. The choice of an appropriate method always has to depend on the given requirements. In the case of CPS, a fundamental property that an analysis technique has to exhibit is exhaustiveness, that is, the ability to include corner cases, in particular the worst-case scenarios. Otherwise, no guarantees regarding critical system characteristics can be provided.

2.2.1 *Categories of System Performance Analysis*

System performance analysis methods can be roughly divided into the three categories: *empirical-*, *analytical-* and *state-based verification methods*. Table 2.1 summarizes the properties of the three different classes. The classification is derived from [109].

Simulation-based approaches form the first class. These are established evaluation techniques characterized by a mature tool base. The fact that they offer a large modeling scope, provide accurate results and are relatively easy to adopt renders them popular. However, these methods tend to become slow with increasing model complexity and – what is important in respect of analyzing CPS – they exhibit the inability to satisfactorily cover corner cases. Due to the latter, they fail at providing hard guarantees for lower and upper performance bounds of a system. A mature and popular simulation interface for architectural exploration, performance and system-level modeling is SystemC [45, 81] which has been approved by the IEEE Standards Association as IEEE 1666-2011.

The state-based verification approaches form another category of the performance analysis methods. As the name implies, those depend on a state-based system representation, e.g. timed automaton, as well as a model checker which is being employed in order to verify whether a system model meets a given system property. Similar to the simulation-based approaches, the state-based verification methods have rich modeling capabilities and are being able to model – in detail – any state-related behavior of a system-under-study. As their models comprise complete systems behaviors, the provided evaluation results are exhaustive. Moreover, those are also exact, meaning, the computed performance bounds are not only correct but also precisely accurate. Unfortunately, this class of approaches struggles with state-space explosion which renders it impracticable for the verification of larger systems. A prominent example for these techniques is UPPAAL [12].

The third category of performance analysis techniques is being represented by the analytical methods. These are based on mathematical formalisms intended to abstract the system in a way that facilitates a quantification of the system's performance characteristics. As a consequence, analytical methods are fast and exhaustive. Moreover, the computed performance bounds can be – in a mathematical sense – proven to be correct. On the downside, analytical methods exhibit a trade-off between the accuracy of the computed results and the modeling scope. In some cases, the analysis of a system is only feasible after the constraints of a mathematical model – the system is supposed to fit in – have been relaxed. Yet, this leads to overly conservative approximations of the performance bounds. Two prominent examples for the analytic approaches are: the *Modeling and Analysis Suit for Real-Time Applications* (MAST) [49, 50] and the *Real-Time Calculus* (RTC) [128].

For the modeling and verification of the virtualized execution environments for CPS, we have chosen to rely on the analytic methods. There are two reasons for this decision. First, as indicated earlier, those methods allow for the computation of reliable performance bounds – a crucial aspect for the validation of virtualized CPS. Secondly, designers of virtualized CPS will inevitably be confronted with large design spaces. Yet, an efficient exploration of a large design space is only realizable with a fast assessment technique. Analytical approaches to performance evaluation fulfill this requirement. Therefore, besides the ability to provide strict guarantees regarding the non-functional characteristics of a system-under-study, they also represent a suitable driver for planning and optimizing virtualized CPS. The adaptation of the analytic methods for the purpose of planning safe and efficient virtualized CPS is being presented in more detail in Chapter 4.

3 VIRTUALIZED CPS-ARCHITECTURE

The research on virtual execution environments for CPS falls broadly into two categories: the technological aspects involving the efficiency of the execution and communication infrastructure and the formal or methodological aspects. The former concerns with technological issues related to the hardware and software foundations of the virtualized execution environment, the latter aims at providing formal methods and methodologies for the planning and verification of virtualized CPS. This chapter deals with the former category, specifically with infrastructure software. The findings presented in the following sections have been published in [62, 68, 69].

The first section of this chapter introduces the reader to the domain of *Cyber-Physical Energy Systems* (CPES). We use CPES as a background for our research, in particular, from real-life applications encountered in this domain we derive timing constrains and used them for the evaluation of the virtual execution environment CPS-Xen, which is being described in the third section of this chapter. Before that however, a discussion concerning architectural challenges related to timing characteristics of virtualized CPS is being provided. The fourth and last section concerns with the efficiency of high availability solutions that employ virtual machine replication. It discusses the state-of-the-art approaches and presents our contributions in this field.

3.1 CYBER-PHYSICAL ENERGY SYSTEMS

This section briefly describes the domain of modern power systems. Power systems are a good example of large, distrusted and complex CPS. This renders them well suited to serve as an exemplary target for our approach as well as the background for our evaluation. In this passage also the non-functional requirements encountered in this domain are being presented.

In recent years, due to the liberalization of the electricity markets, the ongoing integration of renewable energy sources and the introduction of new technologies (e.g. electrical mobility) substantially changed the circumstances in operation of the electric power grid. All these affected the expectations regarding the IT infrastructure and the currently applied software solutions. As a consequence, the requirements imposed in this domain on the infrastructure also changed. Yet, the current system architectures were not designed to meet the multifaceted requirements of future power systems. Therefore, there is a need for new architectural solutions that adequately reflect the novel

Protocol	*Inter-Arrival Time*	*Transfer Time Limit*
SV	0.250 ms	3 ms
GOOSE	5 ms	3-20 ms
MMS	50 ms	100 ms

Table 3.1: Timing constraints derived form the IEC 61850 specification for the three main protocols.

expectations [136]. Our CPS-Xen architecture [62, 68, 69], described later on in this chapter, aims at the fulfillment of those expectations.

In respect of our research, an interesting area of modern power systems is the substation automation field. Recently, in this area, the concept of interconnected and intelligent microprocessor-based controllers – the *Intelligent Electronic Devices* (IED) – is being tested. The hope is that those devices will allow to meet the requirements of the novel monitoring, control and protection applications. To this end, the control or protection applications are being commonly implemented on dedicated IEDs. This, together with the strict safety requirements, which enforce policies dictating redundant device deployment, translates to a considerable amount of IEDs that have to be installed and managed in each substation. That, in turn, leads to high procurement, operation and maintenance costs. Fortunately, in most cases, the computation logic executed on these devices can be abstracted. This opens the possibility to employ virtualization. The aforementioned characteristics render IEDs an excellent target for CPS-Xen with its unikernel-based approach.

The non-functional requirements encountered in this field are being defined by specific regulations and specifications – as the IEC 61850 Standard [58, 60]. The IEC 61850 is a bundle of standards for power system automation and transmission grid protection. It is mainly composed of three protocols: the *Sampled Values* (SV), the *Generic Object Oriented Substation Event* (GOOSE) and the *Manufacturing Message Specification* (MMS) protocol. The SV protocol is used for transmitting measurements values, the GOOSE messages are responsible for carrying state changing commands, and finally, the MMS protocol is being utilized for exchanging general purpose data between substation applications. The first two protocols implement the second – and the MMS the third – layer of the OSI model.

The timing requirements imposed on a system by the IEC 61850 specification are summarized in Table 3.1. The *transfer time limit* comprises of the network communication delay and the communication processing time both at the sender and receiver. The *inter-arrival time* denotes the frequency at which the devices communicate with each other (e.g. an IED with a circuit breaker). As the timing constraints specified in IEC 61850 only refer to the end-to-end communication for

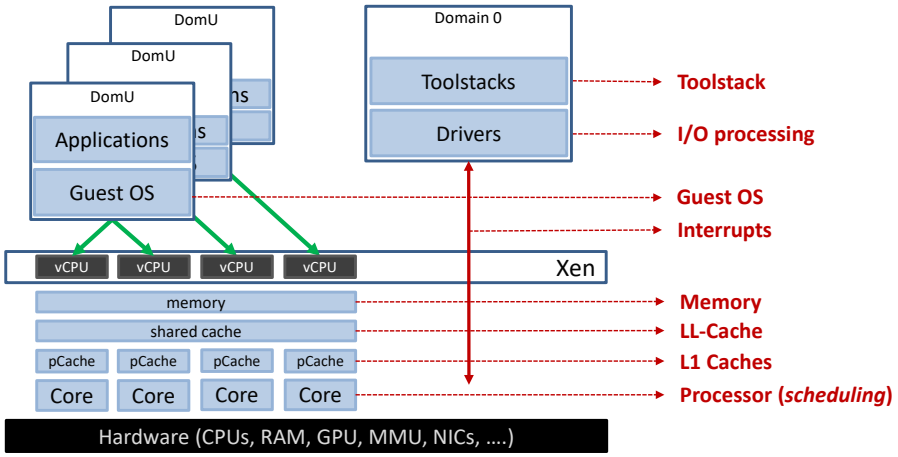


Figure 3.1: Major potential sources of indeterministic latencies in the Xen architecture.

the purpose of the experiments, described later in this chapter, we assume those to also include the computational delays of the algorithms abstracted from the IEDs and encapsulated in VMs. Even though this assumption tightens up the overall timing constraints imposed on our system, the computational latencies – when put in relation with the communication delays – only constitute a small fraction of the overall latency. For example, the computational delay for our implementation of a distance protection function – used for overcurrent protection in power lines and representing one of the most critical function regarding timing constraints in this domain – measured in realistic load scenarios never exceeded 40 µs.

3.2 SOURCES OF INDETERMINISM

Before proceeding with the analysis of selected issues and shortcomings concerning the timing properties of the Xen architecture and the description of the proposed solutions, this section provides an overview over the potential sources of indeterminism encountered on this platform. As the following analysis focuses on the architectural aspects, we abstract from computational delays induced by application workload.

When analyzing the Xen architecture, several sources of latencies that may unpredictably influence the timeliness of a system can be identified. However note that those are mostly not specific to Xen and have also their equivalents in other virtualization platforms. Figure 3.1 depicts the major potential sources of indeterministic latencies in the Xen architecture.

PROCESSOR AND I/O PROCESSING The way VMs are being scheduled by the VMM is of fundamental importance for the timing proper-

ties of a system. However, as will be shown later, in order to obtain system wide characteristics that are not limited to a single part of the architecture, the examination should include the interplay between the VM scheduler and the I/O processing subsystem. In the next sections of this chapter, we identified, discussed as well as proposed solutions for the issue of indeterministic latencies related to processor scheduling and I/O processing exhibited by the Xen architecture. The adaptation of real-time scheduling policies from classical scheduling theory for the Xen architecture as well as subsequently adjusting and synchronizing them with the I/O processing subsystems, allow for a substantial improvement of the timeliness properties of the system. As a result, our CPS-Xen architecture is characterized by significantly lower system response times as well as a minimal latency dispersion.

MEMORY AND CACHES Depending on the hardware architecture and due to the use of shared resources, there can be additional sources of indeterministic latencies while accessing data – mainly stemming from memory controllers, interconnects and bus contention.

On platforms mostly dedicated for servers and high-performance centers, the *Non-Uniform Memory Access* (NUMA) design exemplifies such a source. Even though that NUMA was devolved to enable scalability in multi-processor environments – by reducing the number of processors competing for access to a shared memory bus – a suboptimal mapping of processes or VMs to cores can induce significant fluctuations in memory access times. In order to tackle this issue, NUMA-aware scheduling algorithms have been proposed [14]. Since version 4.3, also Xen supports NUMA-aware placement of VMs[1].

On a lower level of memory architectures and far more common in memory design, another way to deal with latencies attributed to controllers and bus contention are caches. Caches aim at keeping the necessary data local to the processors. However, the starting-point of this technology was the optimization of memory access times in average. Therefore, in their standard form, caches contribute to a better overall performance, yet do not guarantee deterministic timings. In literature, different approaches to cache management have been proposed in order to increase system predictability [82, 102, 131]. A current approach [15] that utilizes hardware extensions for cache partitioning, e.g. the ARM cache controller PL310 or the *Intel Cache Allocation Technology* (CAT), implements cache management as an OS feature. The published results show a significant reduction in the dispersion of access latencies, rendering the systems under study predictable. Further studies are needed in order to determine the potential of these approaches for real-time virtualization.

1 https://wiki.xen.org/wiki/Xen_4.3_Feature_List

Currently, in the Xen implementation, both CAT and its latest extension, the *Code/Data Prioritization* (CDP), can be used to control cache allocation on VM basis.

INTERRUPTS An important feature of virtual execution environments, aiming at hosting time-sensitive CPS applications, is the ability of the hypervisor to minimize the latencies for interrupt delivery. This is not an easy task as virtualization adds an extra indirection layer that comes at a price. When virtualizing interrupts, at least what the hypervisor has to do is to handle the physical interrupts and inject the corresponding virtual interrupts to the appropriate virtual CPUs. This induces additional delays. Interrupt delivery latencies are not only software but also highly hardware specific, therefore there is no general factor that represents the cost of interrupt virtualization. Yet a specific hardware can be tested in this regard. An evaluation of the interrupt processing delays in Xen was done by Stefano Stabellini[2] on Xilinx Zynq Ultrascale+ MPSoC[3], an ARM Cortex A53 based development board. While the native interrupt latency on this board amounts to ca. 300ns, the maximum delay under Xen – after the cache warm up phase – reached 4980ns in the experiments. Even if a worst-case delay of 5 µs fulfills many – if not most – real-time systems timing requirements, it differs from the native delay by a factor of 16. This overhead has to be kept in mind while designing and testing virtualized CPS.

GUEST OS A guest OS can influence the timeliness of a system in several ways: due to the need of hardware emulation, unnecessarily long boot times, execution of non-essential processes and encapsulation of superfluous libraries that are not related to the application of interest, or other forms of inefficient resource consumption. Therefore, in this thesis, we leverage unikernels (see Section 2.1.6) for the construction of virtualized CPS. In particular, we employ, adjust and extend the MiniOS kernel. In Section 3.4.4, we compare MiniOS and Linux openSuse Leap with respect to resource efficiency and the implications for timing properties.

TOOLSTACK In the case of Xen, the current default xl toolstack for managing VMs as well as the tools and infrastructure behind its API were not designed with hard – or even – soft real-time requirements in mind. This has negative implications for the timing properties of employed tools.

In the final sections of this chapter, we describe our high-availability solution CPS-Remus. CPS-Remus is an exemplification of the idea that

2 https://xenproject.org/2017/03/20/xen-on-arm-interrupt-latency/
3 https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

the design space of virtualization-based fault-tolerance approaches can be extended to support time-sensitive VMs. Other fault-tolerance approaches, like *N-Version Programming* (NVP) [18], are also possible. However, the proper design of high level functionality is not enough to guarantee deterministic timing behavior. Tools are commonly depending on lower parts of system software. The Xen toolstack includes the libxl and libxc libraries, needed to carry out commands, as well as the XenStore infrastructure for storing domain related information. These are known for causing unnecessary overheads [92, 93]. The bottomline is that regardless the hypervisor, the design of virtualization-based tools and solutions for time sensitive applications has to involve a thorough analysis of the entire toolstack, including infrastructure functionalities.

3.3 CPS-XEN FOR REAL TIME

This section begins with an analysis of the shortcomings of Xen with respect to scheduling time-critical VMs and timely I/O processing, specifically the processing of network packets. After discussing the deficiencies, CPS-Xen is being introduced, an architecture for hosting virtualized CPS applications that extends Xen with a suite of real-time schedulers and addresses the issue of efficient and timeliness scheduling of VMs and their network packets. Finally, the proposed solutions are being evaluated through an extensive set of experiments.

3.3.1 *Real-Time Scheduling in Xen and CPS-Xen*

The experiments, described later in this chapter, were conducted with the Xen hypervisor version 4.1.4. In this version Xen shipped with two default schedulers: the *credit* and the *Simple Earliest Deadline First* (SEDF) as well as additionally supported *ARINC653*.

ARINC653 is a real-time scheduler from the aviation domain. It is a non-premeptive cyclic executive scheduler which, due to its limited functionality, strongly resembles a dispatcher. It defines a major frame of fixed duration and divides it into minor frames which can then be allocated to domains. The predetermined domains are repeatedly scheduled with a fixed periodicity. The scheduler is designed with strong focus on temporal isolation and a statically predictable schedule. Therefore, the ARINC653 scheduler has a well defined yet very limited field of application.

Due to the described features, a significantly reduced performance and lack of support for multicore systems, the scheduler fails at fulfilling our needs and it is not a suitable choice for efficient integration of CPS. However, in some specific scenarios it could be applied to a partition of a system which requires strong timing isolation or time transparent static schedules for security reasons.

The credit scheduler [29] is a general purpose, proportional share scheduler and currently the default scheduler in Xen. While using this scheduler each domain, including domain 0, is assigned a *weight* and a *cap*. The weight parameter defines an order on the VM set. This order determines proportions in which CPU time should be allocated. For example, a VM with a weight of 128 will get twice as much CPU time as a domain with a weight of 64. The cap parameter, whereas, defines an upper boundary for CPU time that can be assigned to a VM.

The scheduler is *quantum-based* with a default time slice of 30 ms. This means that in this time period a scheduled VM cannot be preempted. The high default value aims at CPU intensive workloads. Since Xen verison 4.2, however, this parameter can be adjusted by means of the xl-toolstack.

The credit scheduler also implements load balancing. The method is rather simple. Every CPU manages its local scheduling queue. If there are no runnable VMs in the queue, the CPU can change its state into idle. However, before that it has to check whether there are runnable VMs in the queues of the other processors. If this is the case, those have to be scheduled. Such an approach guarantees a system-wide fair sharing of the CPU resources.

Due to the described features, the credit scheduler delivers solid performance for most workloads types. However, for the same reasons it is not an optimal choice for the scheduling of latency-sensitive VMs.

The second scheduler shipped with Xen 4.1.4 is the SEDF scheduler. This one also provides a weighted CPU sharing, yet at the same time enforces time guarantees based upon algorithms from the domain of real-time scheduling [79]. With SEDF the CPU requirements of each domain are being specified by three parameters: a *time slice*, a *period* and a boolean flag indicating whether a domain should receive extra CPU time. The slice is a time interval of CPU time which is being allocated to a domain in each period. Together, the time slice and the period parameter represent the CPU share requested by a domain. For example, a domain wishing to receive 20% of the CPU could set its slice to 2 ms and its period to 10 ms. Finally, the boolean parameter allows to turn the scheduler into a semi-work conserving type. A domain with an enabled extra CPU time option makes use of the so-called slack time which is the remaining CPU time after all runnable domains received their CPU share.

The SEDF scheduler also keeps tracks of two additional values for each of its domains: the wall-clock time at which a domain period ends – called deadline, and the remaining CPU time of a domain in a given period. Conform to the EDF policy, the scheduler always schedules – as the next VM – a runnable domain which has the earliest deadline of all runnable domains in the queue. The SEDF is a preemptive scheduler implemented with per CPU queues.

Unfortunately, in the tested implementation the SEDF scheduler does not handle increased load situations well and fails at holding real-time constraints of time sensitive applications. The corresponding results are shown in Section 3.3. Due to this finding, we extended the Xen hypervisor with a suit of real-time schedulers which we will discuss in the next section. Regarding the SEDF scheduler, its shortcomings led later (Xen version 4.6) to its removal from the list of supported schedulers in Xen.

Since version 4.5, Xen ships with a new scheduler called the *Real-Time-Deferrable-Server* (RTDS). The RTDS is a real-time CPU scheduler built to provide guaranteed CPU capacity to guest VMs on *Symmetric Multiprocessing* (SMP) hosts. The scheduler applies the preemptive global EDF real-time scheduling algorithm to schedule VCPUs in the system. On the Xen project website[4], the scheduler is being advertised as aiming at the scheduling of soft and firm real-time embedded, mobile and automotive graphics and gaming in the cloud workloads. The RTDS scheduler bases its predictability guarantees on results from the domain of hierarchical scheduling theory [31, 38] and was originally developed in the RT-Xen project [133, 134]. In Xen 4.10, the status of the RDTS scheduler remains an experimental feature. As at the time when we conducted our studies the Xen hypervisor was still in version 4.1.4, in the following series of experiments we did not compare CPS-Xen with the RTDS scheduler. Nonetheless, in Section 3.3.4, we provide a rudimentary evaluation and comparison of the CPS-Xen RM and the RT-Xen RTDS schedulers based upon Xen Version 4.10 on an embedded development board. Now, before we proceed with the evaluation of the schedulers, we would like to briefly address two points.

First, in our research we aim at a different scheduling model than hierarchical scheduling does. In contrast to hierarchical scheduling, we explicit flatten the scheduling hierarchy in order to facilitate both the scheduling as well as its analysis (see Section 3.3.2.1).

Secondly, there is another shortcoming of Xen in respect of scheduling real-time workloads. Irrespective of the performance and efficiency of a Xen VMM scheduler, even while running at normal utilization levels, VMs can still miss their deadlines. The problem is related to the way Xen processes I/Os. The architecture has limitations that may introduce priority inversion issues to the system and result in unpredictable network traffic latencies. In the evaluation section (see Section 3.3.3) we quantify the impact of this architectural deficiency for network packet processing. The results reveal that the negative impact on VMs delays is substantial. Despite this, works concerned with virtualization, Xen and real-time systems either do no address this issue, e.g. RT-Xen [133, 134] or [97], restrict their solution to local inter-domain communication [135] or only alleviate, yet do not

4 https://wiki.xenproject.org/wiki/Xen_Project_Schedulers

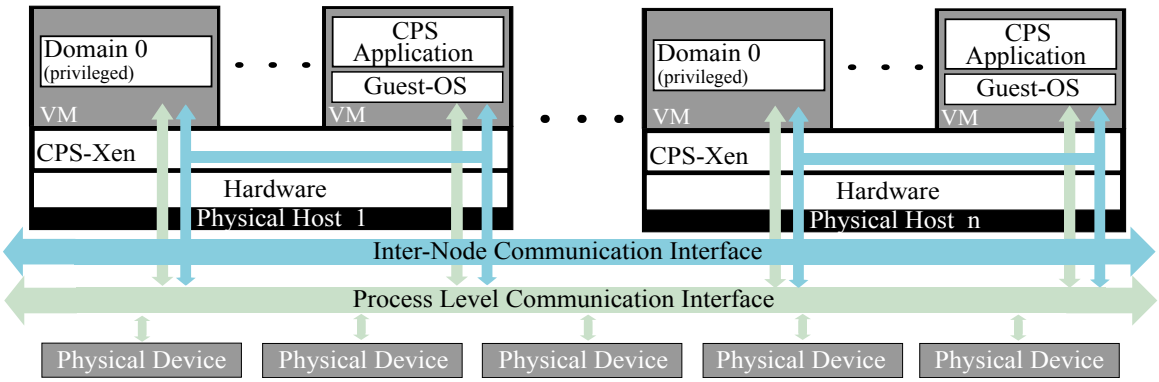


Figure 3.2: Architecture overview of an integrated CPS by means of CPS-Xen.

solve the problem [47]. To the best of our knowledge the only work that successfully addresses this problem is [80]. There, the authors propose an traffic control architecture for Xen that enables network streams prioritization. To this end, they extend the Domain 0 network card driver with additional netback devices which can be assigned to dedicated VMs. The network devices can then be prioritized with a one-thread-per-priority approach and their number is set to the amount of needed priority levels.

Our own solution to this problem was proposed in [69]. In contrast to the aforementioned work, our approach is transparent and does not require any source code modifications. However, both approaches rely on the Linux real-time scheduling policies and the concept of dedicated packet processing threads. Our approach is being described later in this chapter, but first of all, the CPS-Xen architecture and its scheduling model are being discussed.

3.3.2 *Architecture of CPS-Xen*

In this section we discuss the architecture of CPS-Xen as well as its real-time scheduler suit. For a detailed description of the Xen architecture please refer to Section 2.1.5 .

The generic architecture for CPS-Xen is presented in Figure 3.2. It depicts how the information and communication infrastructure of CPS-Xen interweaves with the physical world – represented in the figure by the physical devices. Further, it shows how we assume a CPS – that was integrated by means of CPS-Xen – to look like. Every physical host represents a computational node, which is a server running CPS-Xen. Each of the servers hosts a privileged domain 0 as well as multiple guests encapsulating CPS applications. In order to prevent or minimize traffic interferences during the transmission of time-critical data, the architecture assumes two communication interfaces. The inter-node interface is responsible for handling the VM management traffic. For example, this includes data that has to be

transmitted during checkpoint operations or the migration of VMs. The process level interface, whereas, connects the computational part of the architecture with its physical counterpart. This includes the exchange of data between VMs and the sensors and actuators.

3.3.2.1 *CPS-Xen Scheduling Model*

In platform virtualization the scheduling problem forms a two- (or n, in the case of nested virtualization) level hierarchy. The first level is constituted by the hypervisor scheduler which assigns each VM its share of resources. The second layer is being introduced by the guest OS scheduler which, in turn, assigns its tasks to the virtual CPUs. Such a model poses significant challenges both in respect of scheduling analysis as well as scheduling decisions. The scheduling policies (also called server policies) provided by hierarchical scheduling models, in practice, introduce an additional computational overhead, and in theory, generate overly pessimistic projections for upper bounds. This is fine and acceptable if a system designer can take advantage of the features provided by the hierarchical scheduling approach, meaning, the underlying platform architecture corresponds with such a model. However, this is not the case for our model. A project that aims at enabling hierarchical scheduling studies for VMs is the aforementioned RT-Xen [133].

We avoid the complexity of a hierarchical scheduling model and the attendant difficulties by flattening the scheduling hierarchy. We achieve this by employing unikernels (see Section 2.1.6) for the construction of specialized standalone kernels. This allows us to embed each application into a dedicated VM and therefore to reduce the scheduling level to a single layer. From this perspective, missing a deadline within a VM is equivalent to missing a deadline by the hypervisor. As a consequence, scheduling algorithms and analysis methods from the field of classical scheduling theory can be applied to our architecture, respectively, our scheduling model.

Due to the single level scheduling hierarchy as well as motivated by the shortcomings of Xen in respect of scheduling real-time workloads (see Section 3.3.1), we extended Xen with a suit of classical real-time schedulers and implemented it as a part of our CPS-Xen project. In the current version 1.3, CPS-Xen allows to switch – on the fly – between the following three fixed-priority preemptive scheduling policies:

- *Fixed Priority Scheduling Policy* (FP)
- *Rate-Monotonic Scheduling Policy* (RM)
- *Deadline-Monotonic Scheduling Policy* (DM)

All three schedulers ensure that each time a scheduling decision is being made by the scheduler, the processor executes the runnable domain which has the highest priority of all other and ready to execute

domains. The policies differ, however, in the way how priorities are being assigned and handled.

In the case of the FP scheduler, the priority parameter is being assigned statically and explicitly, meaning, it represents an absolute value that can only be changed by the system administrator. The priority parameter is the only parameter required by the FP policy.

In the case of the RM policy, the scheduler computes the priorities of the domains dynamically. Each time a new domain is being added to the queue, e.g. due to instantiation or migration, or some of the domain parameter change, the scheduler recomputes the assigned priorities. Accordingly to the RM algorithm, the highest priority is being assigned to the domain with the shortest period [78, 85]. Note that our CPS-Xen implementation differs from the classical definition of the RM policy. Besides taking the *period* parameter, it also expects the *slice* argument to be defined for each of the scheduled domains. The slice denotes the amount of CPU time a domain may receive during a period. It can be used to enforce strict timing isolation – e.g. in case of malicious VMs. If set equal to the value of the period argument, the slice parameter does not influence the policy of the RM scheduler. The introduction of the slice argument also enables weighted CPU sharing. The fairness, however, depends on the values of the periods. Finally, the period parameter determines the priority of the domain and the time at which the used CPU time is being cyclically reset, or in other words, the budged for the next period refreshed.

The third implemented policy is the DM scheduler [4–6]. In our implementation it takes three parameters. Additionally to the two RM scheduler parameters also a *deadline* parameter has to be specified. The deadline value has to be less or equal of the period value. The priorities are being computed analogous to the RM approach, yet based upon the deadline values of the domains.

At this time, CPS-Xen only supports a partitioned scheduling model where each CPU holds and manages its own dedicated scheduling queues. Therefore, it lacks global load balancing on multiprocessors and all of the available scheduling policies work only in the non work-conserving mode.

3.3.2.2 *Real-Time Networking in Xen and CPS-Xen*

Another factor – besides the VMM-scheduler – that significantly influences the timing properties of virtualized CPS applications is I/O scheduling. As CPS applications commonly assume distributed architectures, the I/O processing of network packets becomes a matter of particular importance.

Recall that in Xen I/O processing is done by Domain 0 (or driver domains) on behalf of the guests and is based on the split device driver model (see Section 2.1.5.2). By default, CPS-Xen delegates the

processing of networks packets to Domain 0, which runs a Linux kernel. The standard mainline Linux kernel provides mechanisms for supporting latency-sensitive applications. The software layer responsible for this functionality is called *Queuing Discipline* (QDisc) and implements policies for network traffic shaping, including prioritization and classification. Xen inherits the Linux network stack, including QDisc. However, in order to be in accordance with the idea of virtualization and guarantee isolation, Xen introduces additional virtualization-related components to the stack, the virtual network devices. Unfortunately, the way these interact with the Linux kernel may lead to priority inversion issues.

Before the Linux kernel version 3.12, in the xen-related part of the stack, the network packets were processed by a single kernel thread. All of the guests virtual network devices (*netif*) where managed by the back-end of the split network device driver (*netback*) and the dedicated kernel thread serviced all tasks related to the netback, including the processing of the shared *tx* and *rx* queues. In terms of real-time performance this approach exhibited several limitations [135]. The network packets were scheduled or processed regardless of the priority of the destined VMs which led to priority inversion and indeterministic latencies.

Since Linux kernel version 3.12, there is a new netback model. This model is build around the concept of *virtual interfaces* (VIF) which are back-end devices that merge and replace the netifs and netbacks. For packet processing the new model utilizes the *New API Packet Reception Mechanism* (NAPI) [119] as well as multiple kernel threads. The model is also called 1:1 since for every booted VM a dedicated VIF kernel thread is being instantiated. The concept of dedicated threads for handling guests traffic removes one of the potential sources of priority inversion which was related – before Linux 3.12 – to the processing of the shared *rx* and *tx* queues. In the new model, each VIF device handles its queues independently and does not need to coordinate – as before – with the netback device. However, this does not remove all sources of indeterminism in the virtualization-related parts of the network stack. The VIFs still may process packets independently of the priority of the destined VM. In order to address this issue, we propose a solution [69] that utilizes the new 1:1 model together with a Linux feature that allows for configuring scheduling policies for threads (including kernel threads) and setting their priorities. An example of such a Linux scheduling policy is the preemptive fixed-priority scheduling policy SCHED_FIFO.

The idea behind our solution is to synergize the work of the VMM-scheduler with the processing of the network packets in Domain 0. This is done by aligning the priorities of the VMs with the priorities of corresponding VIF threads. Our approach does not require any kernel modification and is transparent as it utilizes the POSIX interface (*chrt*

command) in Domain 0 in order to manipulate the real-time attributes of the packet processing threads. The results for this approach are being presented in the next section.

3.3.3 *Evaluation*

This section provides evaluation results for CPS-Xen. In particular, in a set of experiments the CPS-Xen execution environment is being analyzed in respect of the VMM-scheduling quality, the impact of the scheduling synergy – between the VM and I/O scheduling – on the reactiveness of the system, and finally, platform scalability. Before that however, the setup of the experiments as well as the benchmark used for the purpose of evaluation are being described.

3.3.3.1 *Experimental Setup*

The experiments were conducted on a Dell PowerEdge R620 machine consisting of two 8-core Intel Xeon E5-2650v2 processors running at a constant speed of 2.6 GHz and an integrated Intel I350 1Gbit Ethernet network card. All power management features as well as Turbo Mode were disabled. Domain 0 ran on a 64-Bit version of Ubuntu 14 Server with a para-virtualized kernel 3.13 on an exclusively dedicated core. The used VMM was CPS-Xen based upon the Xen version 4.1.4. Due to the fact that the safety-critical applications from the CPES domain strictly depend on periodic sensor values and that in respect to such assumptions the rate-monotonic algorithm provides the optimal priority assignment [78, 85], in the following experiments, the RM policy was chosen for VM-scheduling. The workloads representing the power system applications were all embedded into the para-virtualized Mini-OS guests. The generation of the request network packets for the VMs under test was conducted on additional computers. Those were connected to the server via a gigabit Ethernet switch.

3.3.3.2 *Benchmark and Latency Types*

For the purpose of this evaluation a *User Datagram Protocol* (UDP)-based client-server benchmark has been implemented in the programming language C. The benchmark servers represent CPS services (e.g. protection algorithms) and are embedded and executed inside VMs. The servers can be configured for variable CPU loads. All VMs run on the PowerEdge R620 host. In turn, the benchmark clients represent sensors or actuators and are instantiated on separate computers. The clients are responsible for triggering the computation inside the VMs by generating request for the CPS services. During benchmarking latencies are being recorded for every single request/response pair.

The benchmark allows to quantify three different latency types across the platform: the algorithm/service *Execution Time* (ET), the

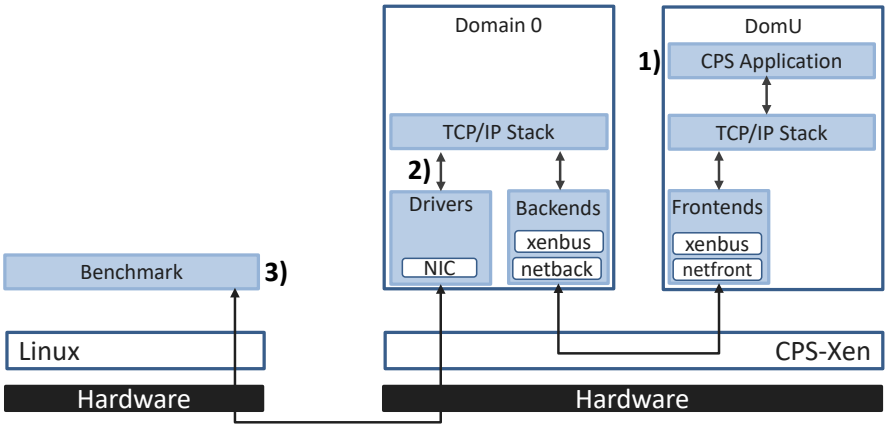


Figure 3.3: Latency measurements locations.

System Response Time (SRT) and the *Round-Trip Time* (RTT). For estimating execution times of the algorithms encapsulated in VMs, we implemented a clock cycle precise measurement technique [107] that provides a maximum measurement deviation of four clock cycles. The system response latencies – defined as the time interval between the moment when the network packet destined for a given VM arrives at the bottom of the Linux TCP/IP network stack and the time-stamp at which it is being delegated to the network adapter for a response transmission – are being collected in Domain 0. To this end, we hook into the TCP/IP stack layer-2 kernel functions using systemtap[5] [114] and log the appropriate time-stamps. Finally, on the machines generating the network packets representing sensor values, we collect RTT for every single packet. We use these values to additionally validate the plausibility of the measured execution and response time latencies. In the subsequent experiments for each measurement in each of the presented figures - if not explicitly stated otherwise - a total number of 10,000 packets were sent.

Figure 3.3 depicts the benchmark architecture and the measurements locations. Table 3.2 summarizes the latency types and the measurement locations together with the corresponding collection methods.

3.3.3.3 *VM Types and Parameters*

In our experiments three types of VMs are being used. Each type represents a different class of workload with different timing constraints. The parameters for the VMs were derived from the IEC 61850 standard, which is described in Section 3.1. Table 3.3 summarizes the VM types and their parameters including their *Worst-Case Execution Times* (WCET) and the request frequencies. The VM types are as follows: the MMS-VMs – representing the soft or non-real-time workloads, the GOOSE-VMs – representing services with hard real-time requirements

5 https://sourceware.org/systemtap

Latency type	*Measuring point*	*Measuring method*
1) Execution time	*Guest domain*	*Clock cycles (RDTSC instruction)*
2) System response time	*Domain 0*	*Systemtap (TCP/IP-stack layer 2)*
3) Round-trip time	*Benchmark client*	*POSIX's clock_gettime() function*

Table 3.2: The three different latency types with the corresponding measurement locations and measuring techniques.

yet with relaxed request frequency, and the SV-VMs –again representing workloads with hard real-time constraints but also with high request frequency. Note that in the case of the SV-VMs, the IEC 61850 standard assumes a sending frequency of 250 µs. In our experiments, however, we assume a window of four SV-values and set the period for triggering SV-based computation to 1 ms. We do this for the following reason. When implementing SV-based protection functions it is common practice to bundle high frequent sensor data into arrays of values. This is due to the fact that many algorithms require a specific amount of values in order to be able to successfully determine a significant change in the physical environment. Sending each and every value separately does not influence the quality of the algorithm's decision, however, it significantly increases the communication load. In the scalability experiments, a total number of 140 thousand packets has to be send in less than a minute. In order to facilitate the evaluation, we bundle four values into one packet.

Before we proceed with the description of the VM parameters, we wish to briefly address the way the WCET notion is being used throughout the next sections. Considering the complexity of our architecture, the use of classical (analytical) approaches to WCET-analysis, which assume a relatively simple and precise hardware model and combine it with a static software analysis, is in practice not feasible. Thus, in the following experiments the notion of WCET is being used in an empirical sense, that is, for denoting an empirically based approximation of the analytical WCET. The WCET values were determined through a campaign of thoroughly conducted latency measurement experiments. For example, for the SV-VM type the measurements were obtained using an implementation of a real CPES application – a distance protection function. In turn, the genesis of the WCETs for the GOOSE and MMS-VMs is of synthetic nature, though the presented WCET timings were empirically verified. The values for MMS-VMs are used in the scalability experiments for imposing a variety of CPU loads on the processor.

Recall (see Section 3.3.2.1) that in our scheduling model we use the slice parameter to define the amount of CPU time a VM can be given in a period of time. In our experiments this parameter is set to the WCET value – while the period parameter is being set to

VM type	*Request frequency*	*WCET*
SV-VM	1 ms	40 µs
GOOSE-VM	5 ms	375 µs
MMS-VM	50 ms	2.4 - 33.6 ms

Table 3.3: VM types and their parameters.

the request frequency. Unfortunately, such a simple mapping which applies the WCET values to the slices has proved insufficient. During long term experiments (10^6 requests) and in high CPU load situations, we witnessed rare outliers in our results which violated VM deadlines (in case of the RM policy the deadline is equal to the period). As the reason for the seldom outliers we identified the context switch function inside the hypervisor. During the conduction of our tests, the measured WCETs for the context switch function amounted to 180 ns in average while the highest value reached 2122 ns. Taking the used scheduling quantum of 10 µs into account, the worst-case overhead for the context switching function of 2122 ns translates to a 21% resource loss. Therefore, we decided to calculate the worst-case context switch latency into the value of the slice parameter. This solved the outlier issue. However, note that such pessimistic parametrization assumptions may lead to over dimensioned systems.

3.3.3.4 *Results*

This section provides results for experiments concerning with the VMM-scheduling quality, the impact of VMM- and I/O-scheduler parameter alignment and platform scalability.

SCHEDULING The first series of experiments focuses on analyzing the interdependency between the VMM-scheduler and network packet scheduling with respect to system response times. Further, also the scheduling suitability for latency-sensitive VMs of the CPS-Xen RM scheduler and the Xen SEDF scheduler is being analyzed.

Figure 3.4 depicts the combined results of three separate experiments in form of six boxplots. We use boxplots, instead of e.g. the commonly used cumulative distribution function, as we are interested in each and every obtained response latency – including outliers. In each of the experiments, ten VMs have been instantiated and executed concurrently on a single core – this induces a realistic load on the schedulers. All of the VMs were running an echo server (no workload) for which reaction latencies have been measured. For each experiment and each of the VMs the requests were send over a network from a sensor node with a constant period of 1 ms for a total number of 20,000 packets. The VMs were prioritized with VM_1 having the highest

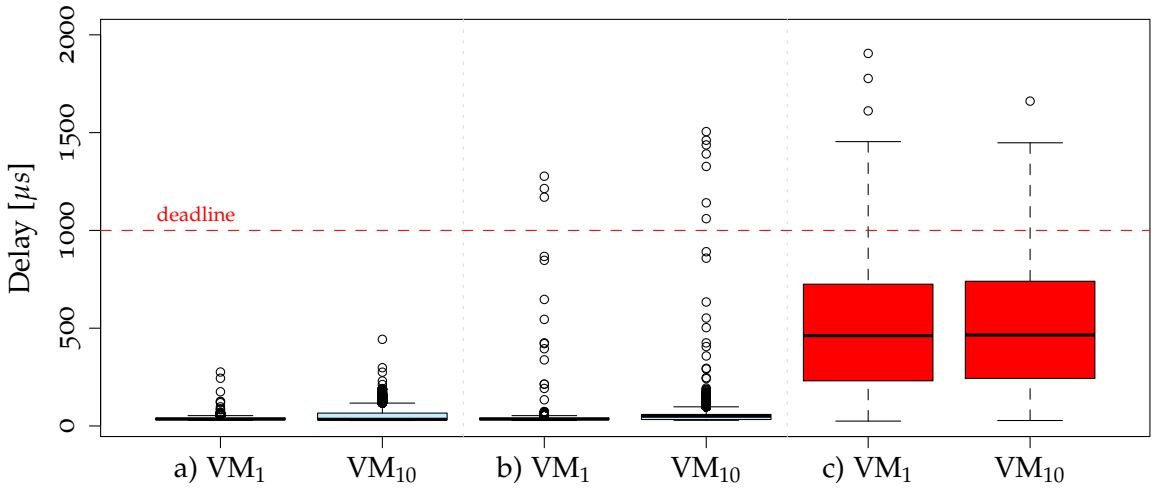


Figure 3.4: Differences in the response times of time sensitive VMs under a) CPS-Xen RM scheduling with synergized network packet scheduling, b) standard CPS-Xen RM scheduling and c) standard Xen SEDF scheduling.

priority and VM_{10} the lowest one. To improve readability, only the results for each of the two most significant VMs, VM_1 and VM_{10}, have been included in the figure.

The three experiments differ in the way VMs and packets are being scheduled by the system. The VMs depicted under a) represent a system running the CPS-Xen RM scheduler with aligned scheduling priorities of the VIF kernel threads in Domian 0. The second VM pair b) represents system response times for VMs also scheduled under the CPS-Xen RM scheduling policy, yet in this case the VIF threads priorities were left unaligned. Finally, pair c) depicts response times for VMs that were scheduled with the standard Xen SEDF scheduler. For the purpose of analysis, a synthetic deadline that is in accordance with the request period of 1 ms is being assumed.

The comparison of the pairs b) and c) shows that even though the CPS-Xen RM scheduler performs better than the Xen SEDF scheduler, which exhibits hundreds of deadline violations, it still misses several deadlines. The fact that this is also true for the highest prioritized VM_1 confirms that optimizing the VMM-scheduler alone is not sufficient. A comparison of the VM pairs b) and a) reveals the degree of influence that the alignment of both schedulers (the VMM- and I/O-scheduler) has on the response latencies. In a) not a single deadline is being missed – for the total number of 200,000 requests. Further, also the dispersion of the latency values is significantly lower. The numerical details for each of the VMs are supplied in Table 3.4.

The obtained results show that our approach of synergizing the VMM-scheduler with the scheduling of the network packets notably improves the timing properties of the system and enables the hosting of latency-sensitive CPS applications. Moreover, the minimized latency

VM	*Maximum*	$\overline{x}$	σ	*Minimum*
a) VM_1	276 µs	37.33 µs	6.49 µs	29 µs
a) VM_{10}	443 µs	51.63 µs	28.22 µs	28 µs
b) VM_1	1277 µs	37.77 µs	19.41 µs	28 µs
b) VM_{10}	1505 µs	51.68 µs	34.94 µs	29 µs
c) VM_1	1905 µs	482 µs	291 µs	28 µs
c) VM_{10}	1661 µs	493 µs	290 µs	28 µs

Table 3.4: The maximum, arithmetic mean, standard deviation and minimum values of the latency measurements for the different VMs.

dispersion significantly facilitates the planning of such virtualized compute and control clusters.

SCALABILITY The following series of experiments investigates CPS-Xen in respect of its scalability characteristics. The first series concerns timing characteristics of the platform under increasing CPU load and the second analyzes timing properties with respect to an increasing number of VM instances.

CPU LOAD Figure 3.5 shows the result of a series of scalability experiments concerning CPU load. Depicted are the response latencies of VMs scheduled under the CPS-Xen RM policy and the Xen SEDF policy. In each of the experiments three VMs – including one GOOSE-VM, one SV-VM and one MMS-VM – were instantiated and executed concurrently on a single core. As before, the priorities are distributed inversely to the lengths of the periods – with SV-VM having the highest and MMS-VM the lowest priority. The MMS-VM was used solely to iteratively increase the processor utilization and is therefore not shown in the results. The CPU load in the experiments ranges from 20% up to 90%. Each boxplot for each of the load levels represents values obtained in separate experiments. Note that the indicated load level is computed based on the WCET parameters of the VMs, meaning, for a given setup the specified loads represents worst-case load bounds. While conducting the experiments the actually measured loads were on average about 15% lower than the specified worst-case bounds. A deadline equal to the request period has been assumed for these experiments.

Figures 3.5 a) and b) depict the response latencies under the RM scheduling policy for the SV-VM_1 and GOOSE-VM_2. We observe that the response times for those VMs are situated – irrespectively of the CPU load – significantly below the defined deadlines (1 and 5 ms) and are characterized by a relatively small variance. For all load situations,

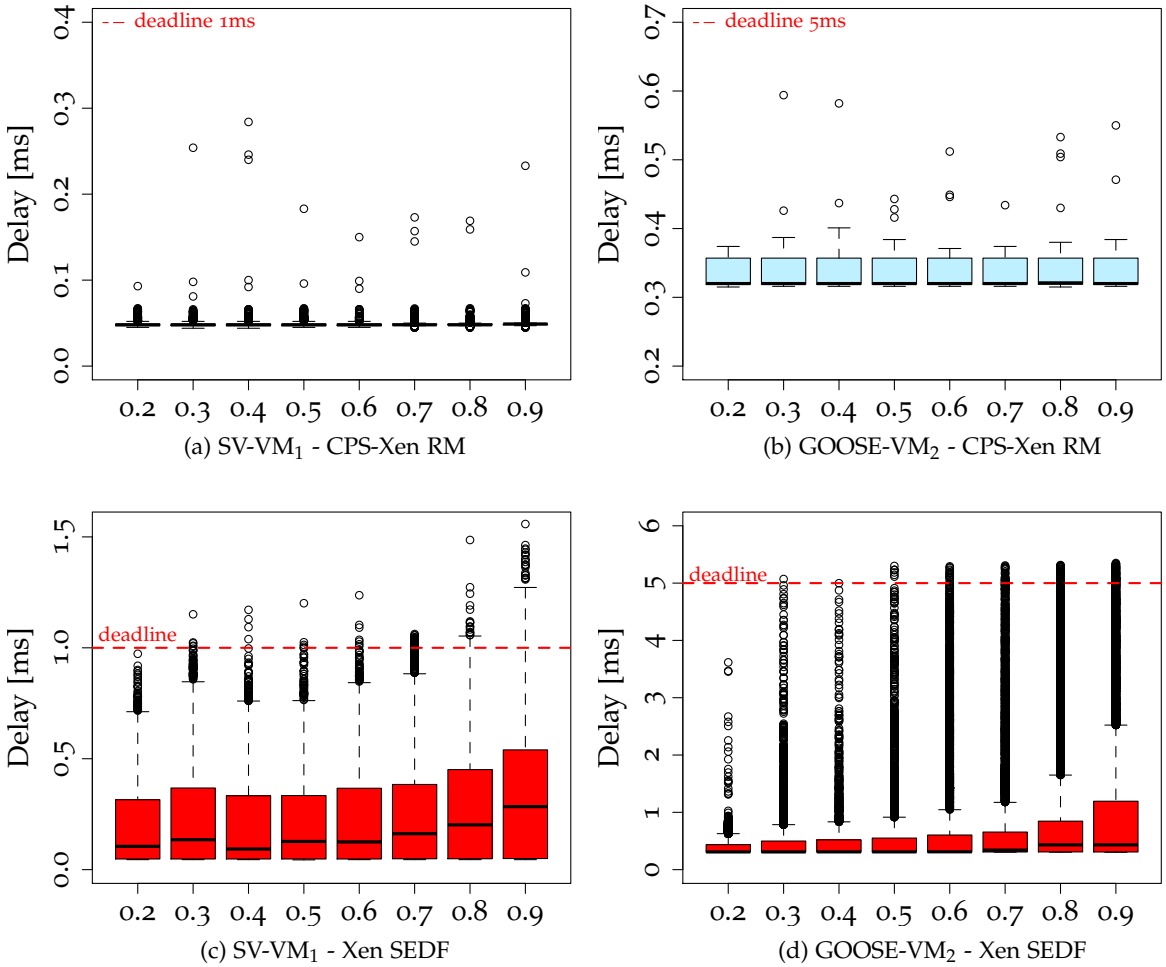
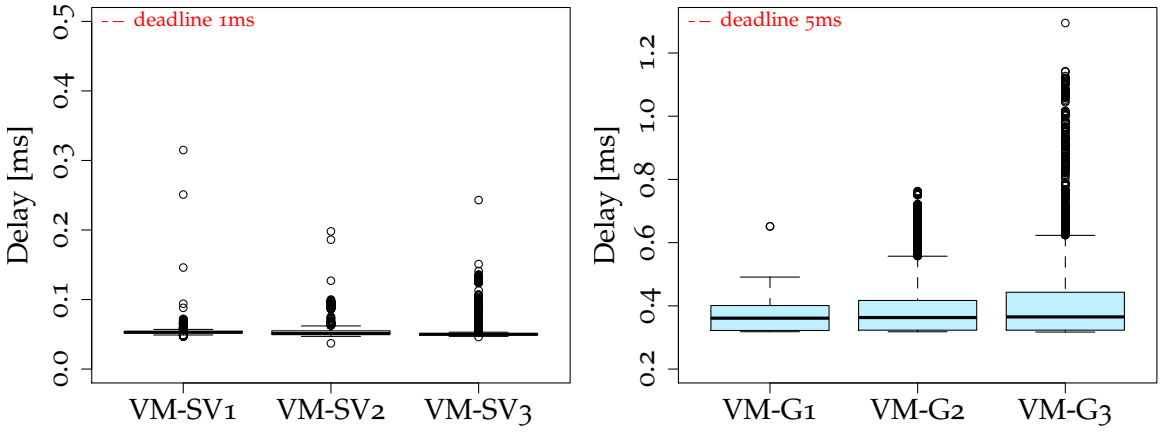


Figure 3.5: Response times of VMs in relation to CPU load under RM and SEDF.

the highest prioritized VM (SV-VM_1) exhibits a standard deviation of no more than 2.5 µs and an average response latency of 48 µs. Also the performance of the GOOSE-VM_2 is stable across all load situations. The standard deviation for the GOOSE-VM_2 does not exceed 18 µs and the VM response time in average amounts to 331 µs. Figure 3.5 c) and d) depict the response latencies for SV-VM_1 and GOOSE-VM_2 under the SEDF scheduler. The results reveal that the SEDF scheduler can not handle load situations and starts to miss deadlines even under a low CPU utilization, starting at a load level of 30%.

VM INSTANCES This series of experiments addresses the scalability potential of CPS-Xen with respect to the number of instantiated VMs. The first experiment investigates the scale up characteristic of the platform for the single core case while the second experiment concerns the multi-core case.



(a) SV-VMs

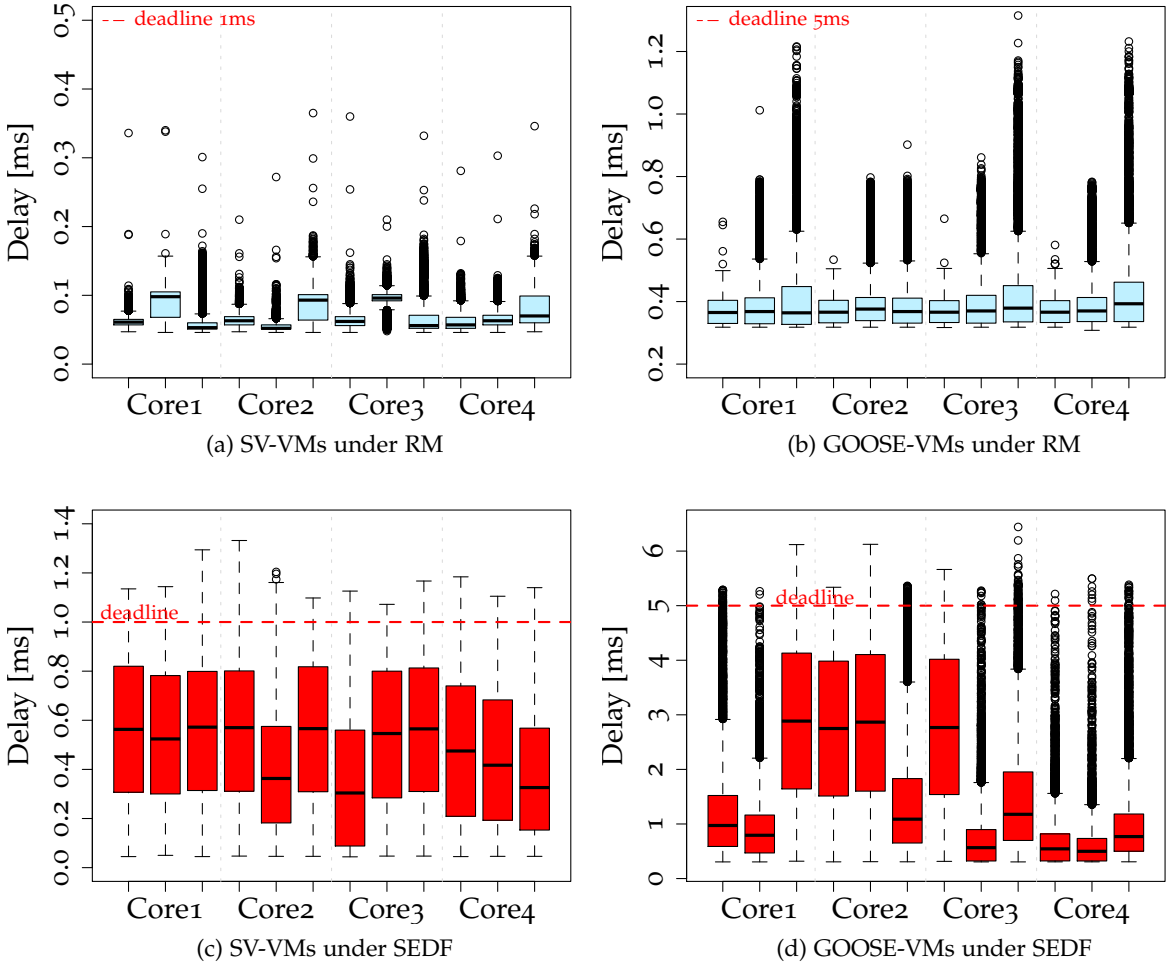
(b) GOOSE-VMs

Figure 3.6: Response times of six VMs with real-time constraints running on a single core under the CPS-Xen RM scheduler.

In the first experiment, nine VMs – three of each VM type – are being instantiated and executed concurrently on a single core. In order to stress the influence of the scheduler, the VMs are also being prioritized within each of the classes – with VM_1 having the highest and VM_3 the lowest priority. Figures 3.6 a) and b) summarize the results for the SV-VMs and the GOOSE-VMs. As can be observed, in none of the cases a deadline is ever missed. Further, the highest measured response delays as well as the average response times correlate with the results obtained in the CPU load experiments. Solely the standard deviation values diverge, yet this is to be expected due to the additional prioritization within the VM classes. The σ values for the SV-VMs and the GOOSE-VMs, starting with VM_1, are: 4.22 µs, 6.08 µs, 11.28 µs and 42 µs, 87 µs, 196 µs, respectively. Note that also all of the non real-time VMs completed their execution on time.

The second series of experiments aims at assessing whether the platform maintains its deterministic behavior in situations where load is being induced on multiple cores. In order to investigate this aspect, a total number of 36 VMs is being instantiated and concurrently executed on four separate cores. Each core hosts nine VMs. The sets of VMs comprise three SV-VMs, three GOOSE-VM and three MMS-VMs. The priorities of the VMs are set as in the previous experiment. For the sake of completeness we rerun the experiments for the SEDF scheduler. Figure 3.7 presents the measured response time values for the SV-VMs and the GOOSE-VMs under both schedulers and for all four cores. To improve readability, the results for the non real-time MMS-VMs are left out.

Figures 3.7 a) and b) show the latencies under the CPS-Xen RM scheduler. In none of the depicted cases a deadline is being missed.



(a) SV-VMs under RM

(b) GOOSE-VMs under RM

(c) SV-VMs under SEDF

(d) GOOSE-VMs under SEDF

Figure 3.7: Response times of 24 real-time VMs from a total of 36 VMs running on 4 cores under RM and SEDF.

Unfortunately, in contrast to the previous experiment, the latencies of the SV-VMs exhibit some irregularities that are partially inconsistent with the assumed scheduling model. For example, on core 3 the highest prioritized VM_1 has a greater worst-case delay than the other two lower prioritized VMs. Another example provides the VM_2 which exhibits a slightly different timing behavior on each of the cores. Considering the CPS-Xen architecture and the results obtained in the previous experiments, in all likelihood, the divergences in timings are to be attributed to the efficiency of the network stack as the pointed out inconsistencies seem to disappear in the case of the GOOSE-VMs.

Finally, Figures 3.7 c) and d) show that in load situations the Xen SEDF scheduler in not able to fulfill the timing-constraints of the VMs and misses each of the deadlines.

VM	*Period*	*Workload*
VM_1	1 ms	300 µs
VM_2	3 ms	400 µs
VM_3	6 ms	500 µs
VM_4	12 ms	600, 1700 and 3400 µs

Table 3.5: VM types and their parameters.

3.3.4 *CPS-Xen and RT-Xen on Embedded Hardware*

This section rudimentarily compares the CPS-Xen RM scheduler with the RT-Xen RTDS scheduler. The presented experiments aim at an empirical evaluation of the schedulability guarantees of the schedulers in increasing load situations. As both schedulers aspire to be adopted in embedded systems, the experiments are being conducted on a *Commercially available Off-The-Shelf* (COTS) development board called UP. For a detailed description of the schedulers, please refer to Sections 3.3.1 and 3.3.2.1.

3.3.4.1 *Experimental Setup*

The UP board is an inexpensive credit card size development board equipped with a Intel® Atom™ x5 Z8350 Processors (1.44 GHz), 4GB DDR3L RAM and a 1x Gb Ethernet with a RJ-45 connector. The experiments are conducted with CPS-Xen and RT-Xen using version 4.10 of the Xen hypervisor. Domain 0 runs on a dedicate core using one VCPU as well as Ubuntu 16.04.3 LTS with a para-virtualized kernel 4.10.0-42. During the experiments, all power management as well as acceleration features of the Atom processor are disabled. The CPU load generating computations are all being executed on the same core and embedded in MiniOS guests.

For the purpose of evaluation, we use a modified version of our UDP-based client-server benchmark (see Section 3.3.3.2) to trigger computation and measure processing times, defined as time intervals between the starting and completion of workloads, using the clock cycle precise measurements technique.

The evaluation comprises three experiments which differ in respect of CPU loads, specifically it includes scenarios with 55%, 65% and 80% CPU load levels. The experiments are being repeated for both schedulers and each load level. Table 3.5 describes the used VM types and their parameters, including the workload triggering period and the computation demands. These workloads represent the highest measured execution times for each of the VMs. The values have been rounded up to the nearest hundred, yet never more then a few milliseconds. In other words, in practice, the load levels are in average a

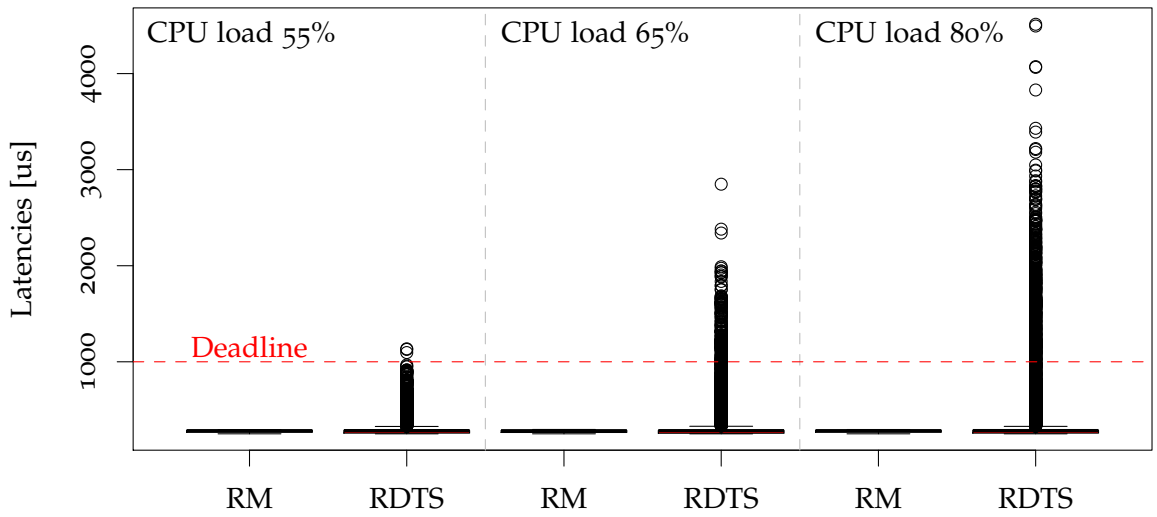


Figure 3.8: VM_1 workload processing wall clock times for different CPU loads under the CPS-Xen RM and RT-Xen RTDS schedulers.

few percents lower than assumed. In order to estimate these execution delays, for every VM at least 10^4 measurements were conducted. The computation demands of VM_1 to VM_3 remain constant across all test. The workload of VM_4 is being adjusted depending on the scenario in order to increase the CPU load.

The schedulers are being parametrized adequately to the requirements of the VMs. The *period* parameters of the schedulers, used for CPU time budget replenishment, are set to the corresponding computation triggering periods and are to be interpreted as deadlines. The *slice* parameters – denoting the amount of time that the VCPU will be allowed to run every period – are set to the workload demands of the VMs. In case of the RTDS scheduler, the *extratime* flag is set to enable. If needed, this binary flag is suppose to allow the scheduler to provide additional CPU time to VCPUs from unreserved system resources.

The results of the experiments are depicted in form of boxplots in Figure 3.8. Due to the high amount of experiments, in the following we only depict the results for VM_1, which has the most strict timing requirements of all VMs. For the full numerical results of all conducted experiments, please refer to Table a.1 in Apendix a.

Last but not least, before interpreting the results, note that the measure of dispersion of latencies alone, can not be seen as an adequate comparison criterion between the two schedulers. The higher variance values exhibited by the RTDS scheduler are inherent to the EDF policy, which is a dynamic priority scheduling algorithm. However, the standard deviation values are of interest for the assessment of the RM scheduler and are therefore being provided by Table 3.6, which presents the numerical values for the experiments.

CPU Load	*Scheduler*	*VM*	*Maximum*	*σ*	*#Missed Deadlines*
55%	RM	VM_1	293 µs	13.4 µs	0
	RTDS	VM_1	1135 µs	59.6 µs	3 (0.15%)
65%	RM	VM_1	295 µs	13.6 µs	0
	RTDS	VM_1	2848 µs	126 µs	174 (0.87%)
80%	RM	VM_1	295 µs	13.4 µs	0
	RTDS	VM_1	4513 µs	290 µs	814 (4.07%)

Table 3.6: Numerical results of conducted experiments for VM_1 under the RM and RTDS schedulers, including the amount of missed deadlines, the maxima of measured latencies and standard deviation values.

As can bee seen in Figure 3.8, the RTDS scheduler already starts to miss deadlines at moderate CPU load conditions of 55%, and with an increasing load the situation deteriorates further. In case of VM_1, the RTDS scheduler misses 3 deadlines at a load level of 55%, 174 at load level of 65% and 814 at a load level of 80%. Respectively, this amounts to 0.15%, 0.87%, and 4.07% of the $2 * 10^4$ recorded periods in each of the experiments. Unfortunately, this phenomena also affects the other VMs. For details, please see Appendix a.

In the case of the RM scheduler, irrespective of CPU load and the VM type, no deadlines are ever missed. Further, as can be seen in Table 3.6, the RM scheduler performs deterministically. In all load situations, the maximum execution time for VM_1, as well as the standard deviation of the measured latencies values, are nearly equal.

Two conclusions can be distilled from the results presented in this section. First, our in previous sections thoroughly evaluated RM scheduler also performs well on inexpensive COTS platforms with significantly scarcer resources then server hardware. Second, the RTDS scheduler does not live up to the promise of delivering a reliable scheduler for soft and firm real-time workloads.

3.4 CPS-REMUS FOR EFFICIENT HIGH AVAILABILITY

The second part of this chapter concerns the efficiency of high availability solutions that employ virtual machine replication. To this end, Section 3.4 starts with an examination of the state-of-the-art techniques. Next, it evaluates the performance of Remus and discusses its suitability for the protection of virtualized CPS applications. Further, this section introduces the *self-determined* virtual machine replication model, a novel and efficient approach to high availability that avoids

the drawbacks of the established methods. Finally, also the evaluation results for the proposed approach are being presented.

3.4.1 *High Availability - State of the Art*

This section provides an overview of the existing high availability approaches using VM replication and the existing optimizations to VM replication.

At the lowest level highly available systems can be implemented using redundant hardware components. This approach is robust, yet very expensive. Therefore, efforts have been made to achieve this goal in software instead. The first architectures that methodically explored the idea of software-based fault-tolerance using replication were investigated in the 1980's. The Delta-4 project [19, 113, 125] coined many of the concepts relating to replication schemes and checkpointing, including event-based as well as periodical approaches. Most of those concepts can be found in later works concerning VM-based replication. Our self-determined approach is also partially rooted in the Delta-4 concept of *systematic checkpointing*.

A pioneer concept of hypervisor-based high availability through lock-stepped VM replication was proposed by Bressoud and Schneider [16, 120]. In this concept, the input events from the primary host are being forwarded to the backup host and deterministically replayed. Unfortunately, this approach suffers from high performance costs – especially in multiprocessor environments.

The overhead related to lock-stepping and replaying can be reduced if replication is conducted through state copying. In such an approach, instead of deterministically replaying VM inputs, the entire state of a VM is being captured and transferred to a backup host. The first to present and implement this idea was Cully et al. [30]. The implementation is based upon the Xen hypervisor and its feature of live migration [23]. However, in contrast to the Bressoud and Schneider approach, Remus is highly demanding in respect of network bandwidth.

There exist approaches that aim at mitigating the network overhead induced during VM replication by leveraging parallel execution. In [116] the authors present a hypervisor-based fault tolerant system that tolerates non-benign random faults and software faults by utilizing redundant execution of a service and making use of N-version programming [9].

Another approach relying on parallel execution is COLO [35]. In COLO a deployed service executes on a primary VM as well as simultaneously on a number of backup VMs. Upon service response the outputs of the primary VM and its replicas are being analyzed. In the case where responses diverge, a synchronization process is being triggered before the final response is sent to the clients. If this is not the case, the response is being sent directly to the requesting

party. Due to this mechanism, the authors refer to their approach as coarse-grained VM lock-stepping.

A different technique that tries to combine lock-stepping with state copying is Kemari [127]. Kemari synchronizes the states of the primary and backup VMs upon event issuing. Each time the primary VM tries to read or write to storage or to send a packet, the operation is being trapped and resumed only after synchronization. The authors claim that such an approach allows for transparency, yet doesn't require to buffer the responses.

Several optimizations to the process of VM replication have been proposed. In [88] the authors present three different optimization techniques for memory state synchronization: fine-grained dirty region identification, speculative state transfer, and synchronization traffic reduction using active slave. In [138, 139] the behavior of memory accesses among checkpointing epochs is being analyzed, in order to improve the memory tracking mechanism during checkpoint capturing. Further, a technique for efficient mapping of large memory regions between VMs is presented. Finally, [99] describes optimizations concerning with checkpoint sizes and checkpointing latencies in respect of Remus. The authors utilize data compression mechanism to reduce checkpoint sizes and leverage paravirtualization features to reduce latencies. Note that all of the described optimizations can – or partially have been – adapted for our approach. Our approach is transparent in this regard, as it does not directly interact with the way checkpoints are being constructed but focuses on determining the checkpoint capturing moment.

High-availability via VM replication is a promising technology that significantly improves service availability. Unfortunately, none of the presented approaches fully fits the needs in respect of protecting CPS services. The current HA techniques either utilize a periodic replication model and transfer state modification over a network to an inactive backup VM or have to rely on parallel/redundant execution with a voting mechanism. Concerning the requirements of CPS services both approaches are partially inadequate or unnecessary inefficient. This aspect is being further elucidated in the next sections.

3.4.1.1 *Shortcomings of Remus*

This section discusses the performance of Remus and its shortcoming with respect to protecting CPS applications. We have chosen to discuss Remus, as it represents a prominent example of the current state-of-the-art techniques for high availability via VM replication as well as due to the fact that we have used Remus for the implementation of our self-determined replication model, which is being described later in this chapter. For further details regarding Remus see Section 2.1.5.3 or [30].

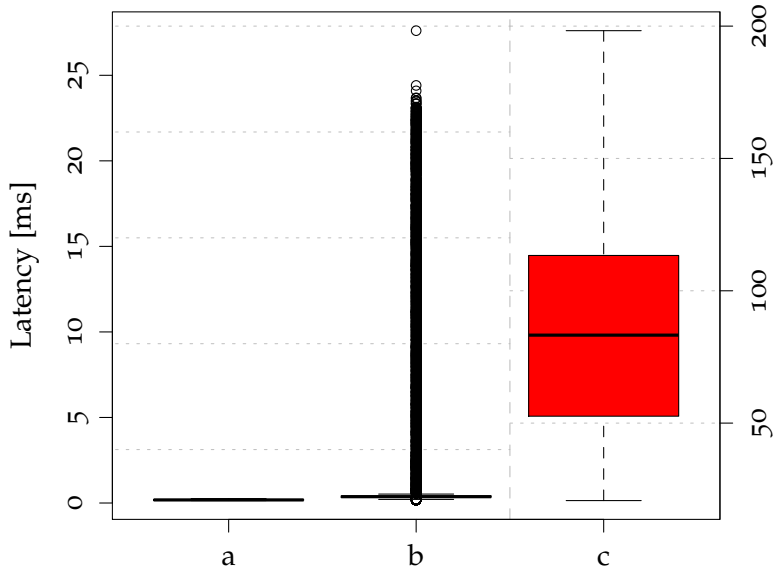


Figure 3.9: Effect of Remus on response times of an echo server. (a) classically deployed VM (no protection), (b) protected with network buffering disabled, (c) protected with network buffering enabled.

The following analysis is based on the 4.7 version of Xen and Remus, the latter is part of the toolset for VM management distributed with Xen. The analysis focuses on the latency overhead generated by Remus. The latencies were quantified by the means of the same UDP-based client-server benchmark that was used to evaluate CPS-Xen (for details refer to Section 3.3.3). In order to collect data, an echo-server was embedded in a VM and deployed as a fault-tolerant service, that is, the VM was being protected by Remus. Remus was set to perform checkpointing with an interval of 100 ms. This checkpointing period was equal to the request frequency of the benchmark client. Figure 3.9 summarizes the results of three experiments in form of three boxplots, each comprising the round-trip times of 10,000 packets acquired on the client side.

For the purpose of a reference the first experiment (a) measures the delays of a non-protected echo server. The highest measured latency is 0.25 ms while the mean response time of the server amounts to 0.18 ms. The second experiment (b) collects data of a protected echo server. Network buffering is disabled, that is, the primary host does not wait for the backup to acknowledge the receipts of checkpoints before sending responses. Here, the mean response value is 2.48 ms and the highest measured delay reaches 27.06 ms. Finally, the experiment (c) is being repeated with enabled network buffering. The mean response latency amounts to 83.14 ms and a highest latency of 198.27 ms can be observed.

The results reveal that in its current form Remus induces an unacceptable – with respect to the requirements of most CPS applications – overhead on the response times of protected services. Some of the sources for the additional delays are to be attributed to the current implementation of Remus, which deviates from the approach described in the original Remus paper [30]. For instance, in its current implementation and in contrast to the Remus article, Xen resumes a checkpointed VM only after all of the dirty pages have already been transmitted to the backup. Now, if a request is to arrive during the execution of the function responsible for this transmission (*suspend_and_send_dirty()* in *xc_sr_save.c*), the processing of the request will be delayed. In worst case for the time it takes to capture and send the current checkpoint and – additionally – by the duration of the guest suspend and resume processes. However, the significant delays introduced to service response times by Remus are not to be attributed to implementation issues alone. We argue that the cause is inherent to periodic checkpointing and that the presented results motivate a rethinking of this replication model and its assumptions in respect of protecting CPS applications.

3.4.2 *High Availability Assumptions*

In this section, we discuss some of the assumptions made by current VM replication-based HA solutions [30, 35, 87] and the resulting checkpointing models. Further, we assess the adequacy of the models in respect of protecting CPS applications.

While analyzing the current virtualization-based high-availability approaches, the following three assumptions can be distinguished:

1. Though not restricted to it, the VM replication-based HA solutions assume server hardware and thus a powerful and scalable infrastructure.
2. Transparency - neither the protected applications nor the OS should require code modification.
3. External state consistency has to be guaranteed.

How can these assumptions be explained? Their genesis is rooted in the fact that until recently the scope of application for replication-based high availability was – in practice – restricted to data-center and cloud computing environments. Thus, present HA approaches assume homogeneous hardware and scalable resources, applications that have to be deployed from unmodified binaries and, finally, workloads such as databases or web services. Such assumptions favor the use of the periodic checkpointing model or the redundant execution techniques.

Although data-center and cloud environments still remain the main area of application for HA systems, in recent years hardware man-

ufacturers have started to add virtualization support into products aimed at the embedded domain and thereby significantly increased the interest in CPS virtualization. The added hardware support facilitates the adaptation of HA techniques for the CPS domain. However, there exist substantial differences both in respect of the assumptions as well as the constraints encountered in cloud environments and the CPS domain. Those have to be taken into account while adopting or designing HA solutions for CPS:

1. Though not restricted to it, the HA solutions for CPS services have to assume heterogeneous hardware and often significantly scarcer resources.

2. Transparency is not an issue. Most CPS applications are tailored for a specific task and infrastructure. For efficiency reasons also the OS code has to be – at least partially – paravirtualized.

3. Especially in the case of the monitoring and control applications, external state consistency is not required. Control applications have to possess the intelligence to cope with missing sensor values – typically sensors do not implement the TCP protocol and actuators do not request retransmissions of control commands.

4. Finally, CPS services impose considerably stricter timing constraints on HA approaches that the one encountered in cloud environments.

These constraints reveal the gap between the requirements of CPS applications and the present HA design space comprising of the periodic checkpointing model and the redundant execution solutions. The latter assume a scalable and homogeneous hardware, something that can not be fully satisfied by the hardware platforms from the embedded systems domain. Concerning the former aspect, the issue is twofold. The first problem relates to the significant latency overhead generated by the periodic checkpointing model. The second issue is associated with task activation or service requests patterns. Not all processes or requests are being triggered in an exact and constant time interval and can therefore be adequately approximated by a period. There exist phenomena like delays and jitters, burst behaviors or sporadic and aperiodic requests patterns [126] common to the CPS domain. In order to be able to protect heterogeneous CPS services with the periodic replication model, it has to be configured to include worst-case scenarios. This in turn enforces a very conservative parametrization that – once again – generates and adds a considerable and unreasonable overhead.

In summary, the cost of adopting current HA solutions in their present form for the CPS domain are unnecessary high. Therefore, we argue that the current HA design space needs to be extended by an

adequate replication model that allows for an efficient protection of emerging CPS applications.

3.4.3 *Self-determined Replication*

This section introduces the idea of self-determined VM replication, compares the two models of periodic and self-determined VM replication by discussing their basic stages of operation as well as describes the design and implementation details of our approach.

The main idea behind self-determined replication is to enable a semantic-based determination of the checkpoint capturing moment. This implies that the control over synchronization has to be transferred from the virtual machine monitor (VMM) toolstack to the service encapsulated in the VM. It also means that the transparency property has to be sacrificed as the service has to be aware of the fact that it is being virtualized in order to trigger synchronization. We argue that waiving transparency is an acceptable price as in most cases the transparency condition is anyway inadequate in respect of CPS services (for discussion please refer to Section 3.4.2). In practice, sacrificing transparency offers numerous possibilities in respect of potential synchronization points. Using self-determined replication a CPS service developer could trigger checkpointing, for example, at the following points:

- depending on input or output semantic
- after executing a specific code path
- after an arbitrary number of iterations

The simple idea of giving control over synchronization to the service not only provides the developer with these options but it also has important consequences in respect of service response latencies. In order to facilitate the understanding of the origin of these delays, in the following, we provide a model for the basic stage operations for both of the synchronization approaches.

3.4.3.1 *Stages of Operation*

Figure 3.10 depicts the basic stages of operation during VM replication for both the self-determined and the periodic approach as well as each of the network buffering options.

In the case where network buffering is enabled, the main difference between the two approaches – in respect of the way the response latencies are composed – is to be attributed to the idle phase of the periodic technique. Here, depending on the timing, the idle phase, between issuing a response (2) and a checkpoint (3), can be in the worst case equal to the period of the checkpointing frequency. However,

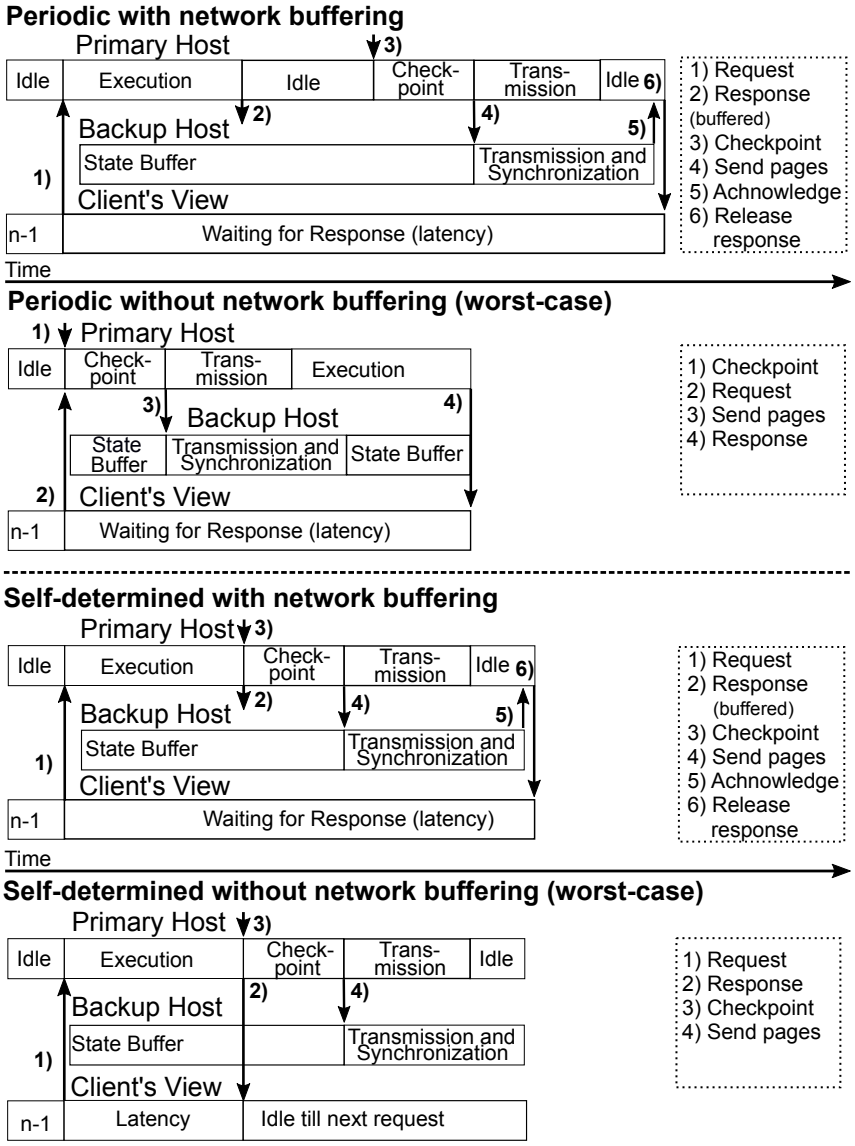


Figure 3.10: Basic stages of operation during checkpointing.

suspending service execution at an arbitrary moment in time may also lead to an additional delay of two checkpoints as well as the transmission and synchronization (4) phases. In the case of self-determined checkpointing, there is no idle phase between issuing a response (2) and a checkpoint (3) as the VM chooses to synchronize its state only after completing its task. Therefore, in the worst case, the latency is bounded by the execution time of the service and the delay of the checkpoint as well as the transmission and synchronization phases (4).

In the case where network buffering is disabled, the difference in latency composition between the two approaches is defined by the checkpoint and transmission delays. In terms of the periodic approach, if a request (2) is to arrive right after a checkpointing process has been triggered (1) then the client will have to wait for the duration of the

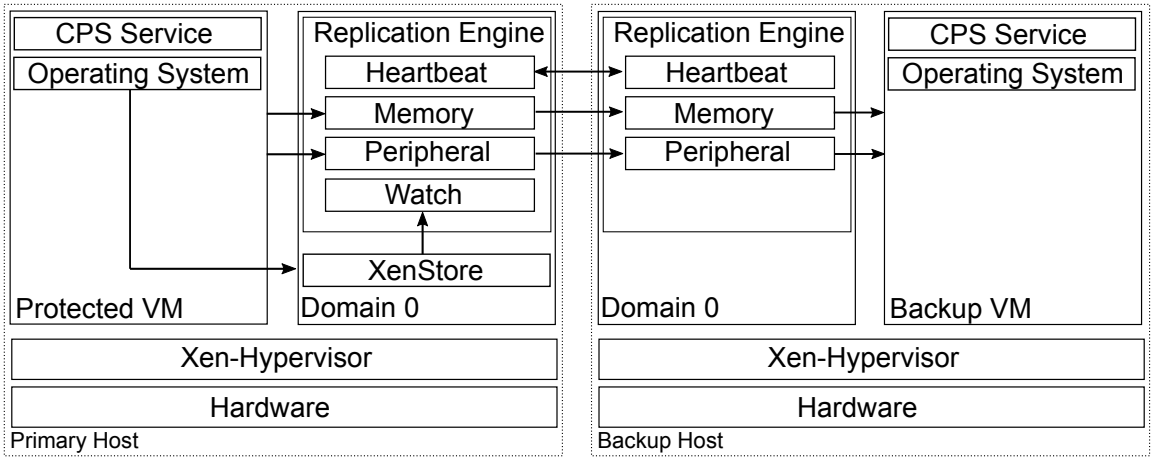


Figure 3.11: Overview of CPS-Remus architecture for self-determined VM replication.

checkpointing, transmission and execution stages. In the case of the self-determining approach, the worst case response latency is only bounded by the execution time of the protected service.

3.4.3.2 *Design and Implementation*

We have based the implementation of the self-determined replication model on Remus [30] and our CPS-Xen project [27, 69], which extends Xen [10] by implementing a suite of established real-time schedulers (see Section 3.3). The solution was named CPS-Remus. CPS-Remus extends the original Remus functionality by enabling the deployment of fault-tolerant CPS services using the self-determined checkpointing approach. In our implementation, all of the original features and options provided by Remus remain available and can be used in combination with our approach.

In the following description, we focus on novel aspects of the architecture that directly relate to self-determined replication. For details regarding the original Remus architecture and its functionality, please refer to Section 2.1.5.3 or [30, 99].

Figure 3.11 depicts the high-level architecture of CPS-Remus. The heart of the architecture is formed by the Remus replication engine. It has been extended to introduce an additional communication path between the protected VM and the replication mechanism. For this purpose, *XenStore* is being utilized. *XenStore* is a data structure that provides a filesystem-like hierarchy and can be used to exchange information between Xen domains. Xen tools can utilize this infrastructure in order to configure and control virtual devices. We, in turn, utilize it to signalize checkpointing requests of the protected VMs.

SELF-DETERMINED CHECKPOINTING In order to trigger checkpointing, the protected VM has to change the value of a key in *XenStore*. First, however, it has to create a specific path "data/ha", right

below its domain node, in the *XenStore* hierarchy. Depending on the execution context, this can be achieved either by utilizing *libxenstore*, a library that exports *XenStore* functions to userspace, or from kernel space by using *XenBus* – an in-kernel API provided for guests to interact with the database. This way, both the services deployed as unikernels as well as the one instantiated inside a general purpose OS, like Linux, can easily access the required *XenStore* infrastructure. Now, invoking CPS-Remus with the parameter "-E", using Xen's standard toolstack *xl*, initializes the self-determined checkpointing process in Domain 0. First, this checks whether the appropriate path exists in the *XenStore* database. If not, an error message is displayed, otherwise, CPS-Remus registers a *watch* on the above mentioned path together with a callback function. *Watches* are a mechanism that allows to react to changes in the database. When a given path is being "watched over" then any change, at that point in hierarchy, will result in invoking a corresponding callback function. In the discussed case, this function triggers the checkpointing process.

HEARTBEAT AND FAILOVER CPS-Remus also reworks the heartbeat mechanism of Remus. This was necessary due to the fact that in the current implementation Remus utilizes the timeout property of the secure shell (SSH) connection to mimic the heartbeat mechanism. In order to change the heartbeat frequency, the SSH server has to be reconfigured and restarted. This renders the deployment of services with different timing requirements impractical.

CPS-Remus implements an active heartbeat mechanism. On startup, the user is supposed to supply a heartbeat frequency by setting the "-t" option with an appropriate value. This allows for a fine-grained and application-specific parametrization of the heartbeat and therefore affects the reaction time of the failover procedure. With CPS-Remus, two processes are being instantiated: a sender process at the primary server side and a receiver process on the backup host. After the instantiation, the sender and receiver start to exchange – at the preselected frequency – a heartbeat signal. Failing to receive this signal, on the backup side and within the specified time interval, triggers the failover procedure. Technically this is being realized by identifying the PID of the underlying migration process – utilized by Remus – through a named pipe and issuing a UNIX signal to an appropriate handler. The handler is responsible for the triggering of the original Remus failover procedure. Finally, in order to provide security, our heartbeat implementation also relies on the SSH technology.

3.4.4 *Evaluation*

In this section we examine the performance of CPS-Remus. The first part describes the experimental settings. The second part addresses

the efficiency of the self-determined VM replication by quantifying its latency overhead and comparing it with the one generated by the periodic approach. Next, after identifying and analyzing additional sources of latency overhead during checkpointing, we cover performance improvements gained by leveraging unikernels for the construction of fault-tolerant CPS services. Finally, we analyze the service recovery latencies and evaluate the real-world applicability of our approach.

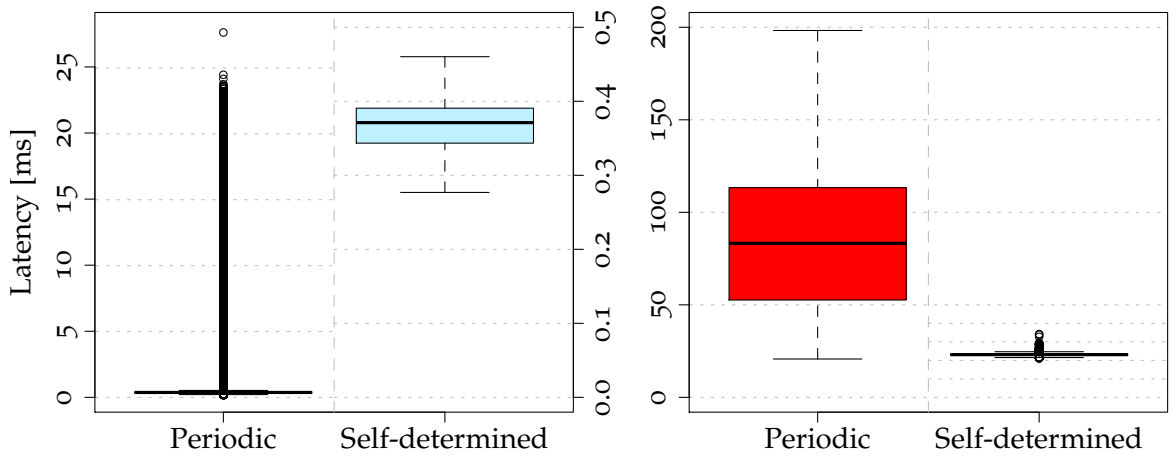
3.4.4.1 *Experimental Setup*

As before, for the purpose of evaluation, we use the UDP-based client-server benchmark described in Section 3.3.3.2. In each of the experiments, we deploy the server as a protected service and generate requests on clients instantiated on separate computers. We use two different communication interfaces: one for sending requests and another for the VM management traffic (heartbeats and checkpoint transmissions). On computers generating requests, we record the round-trip times for every single request/response pair. Again, the results of the experiments are being presented in form of boxplots as in the case of protecting time sensitive services we are interested in each and every obtained response latency – including outliers.

We use two different OS's for deploying our protected services. To evaluate the efficiency of self-determined checkpointing and compare its latency overhead with the one generated by the periodic replication, we use the minimalistic openSuse Leap 42.1 Linux distribution which is tailored for the purpose of virtual appliances. The provided OS image ran the kernel version 4.1.36. In order to quantify the improvement achieved by leveraging unikernels for the deployment of CPS services, we use MiniOS. Unfortunately, MiniOS innately does not support a VM suspend/resume mechanism that is necessary for para-virtualized OS's in order to be protected by Remus. In fact, none of the unikernels that we have considered did in practice. Unlike VMs using hardware virtualization (e.g. HVM), where device states are being secured during suspension using the QEMU emulator, in case of para-virtualized VMs the OS has to take care of this operation by itself. Otherwise, it won't survive the process of checkpointing. Therefore, based partially on the sources of Mirage OS [91], we extended MiniOS to support this functionality. The full source code is available on our GitHub project site [6].

The experiments were conducted on two Dell PowerEdge R620 machines consisting each of two 8-core Intel Xeon E5-2650v2 processors running at a constant speed of 2.6 GHz and two integrated Intel I350 1Gbit Ethernet network cards. All power management features as well as Turbo Mode were disabled. Domain 0 ran on a 64-Bit version of

6 https://github.com/cpsxen/cps-xen



(a) Network buffering disabled.

(b) Network buffering enabled.

Figure 3.12: Response times of a protected echo server using the periodic and self-determined replication approach with both network buffering disabled a) and enabled b).

Ubuntu 14 Server with a para-virtualized kernel 3.13 on a dedicated core. As the VMM, we used CPS-Xen based upon Xen version 4.7. All workloads embedded into openSuse Leap were executed with real-time priorities (set through the POSIX interface *chrt* command), in order to minimize the influence of background processes on the benchmark. Also the guests corresponding kernel threads (VIFs) in Domain 0, dedicated for packet processing, were set to run under real-time priorities.

3.4.4.2 *Results*

Our first experiments measure the latency overhead of the periodic and self-determined checkpointing approach. In the case of periodic checkpointing, Remus was configured to perform synchronization with an interval of 100 ms, equal to the clients requests frequency. For the self-determined approach, due to the event-based character of this method, the checkpoint triggering frequency also equals the request frequency which additionally facilitates the comparison. In the subsequent experiments, each of the presented latency distributions was generated from 10,000 observations.

Figure 3.12 depicts response delays of the protected benchmark server for both the periodic and the self-determined approach. The service was configured to serve as an echo server with no workload and without memory dirtying. The experiments were conducted twice, once with disabled and once with enabled network buffering.

Protecting the server with disabled network buffering, in case of the periodic approach, yields a mean response latency of 2.48 ms, a standard deviation of 5.24 ms and a worst case delay of 27.06 ms.

	Network buffering disabled		*Network buffering enabled*	
Stat. values	*Periodic*	*Self-determined*	*Periodic*	*Self-determined*
Max.	27.06 ms	0.46 ms	198.27 ms	34.23 ms
$\overline{x}$	2.48 ms	0.36 ms	83.14 ms	23.11 ms
σ	5.24 ms	0.02 ms	35.18 ms	0.69 ms

Table 3.7: The maximum, arithmetic mean and standard deviation values for latency overheads generated by the different VM replication approaches.

In the case of the self-determining approach, the values are, respectively, 0.36 ms, 0.02 ms and 0.46 ms. The contrasting results are to be attributed to the fact that in the case of periodic checkpointing, Remus resumes the service only after all of its dirty pages have been transmitted to the backup. In the instance of the self-determined approach, as the protected VM is not being arbitrary suspended by the VMM, this functionality is being triggered by the VM itself and at a more convenient time (for operation details see Section 3.4.3.1 *Stages of Operation*). This reduces the latency overhead significantly. A comparison of the worst case delays of both approaches reveals a difference in latency by a factor of 58.

Next, we repeated our experiment, but this time with enabled network buffering (b). In case of the periodic approach, a worst case latency of 198.27 ms was observed. From the collected data, we computed a mean latency of 83.14 ms and a standard deviation of 35.18 ms. In turn, the self-determined version delivered a worst case delay of 34.23 ms, a mean latency of 23.11 ms and a standard deviation of only 0.69 ms. This discrepancy in the results is – again – caused by the fact that synchronization is being enforced on the service by the periodic approach at a arbitrary moment in time. This leads to additional latencies composed of the checkpointing period and delays of the replication stages (compare Section 3.4.3.1 *Stages of Operation*). In the worst case, the latency between these two approaches varies by a factor of 5.7. The numerical details for each of the experiments depicted in Figure 3.12 are summarized in Table 3.7.

The analysis of the different response delays clearly shows that periodic checkpointing induces – irrespective of the network buffering option – a significant latency overhead when compared with the self-determining approach. Still, some of the higher latencies cannot be fully explained based only on the model of checkpointing stage operations presented in Section 3.4.3.1. Therefore, in the following we analyze the source of these latencies in more detail.

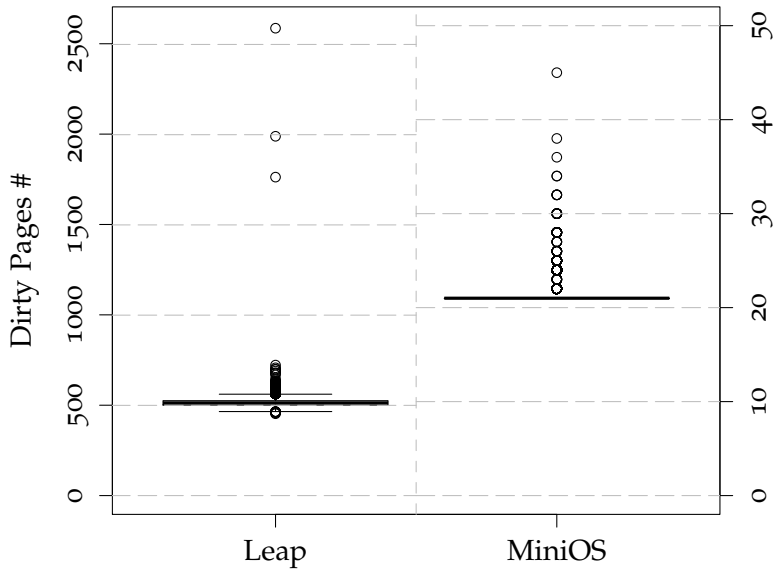


Figure 3.13: Amount of dirty pages identified during replication of idle guests for both Leap and MiniOS.

CHECKPOINT FUNCTION DELAYS For this experiment we augmented the Xen libraries involved into checkpointing to provide timing data. Especially two latencies were of interest to us: the delay for retrieving modified pages (dirty pages) and the time for which a protected domain remains suspended. Table 3.8 shows the collected data.

Delay	*Retrieval*	*Suspension*
Max.	222 µs	116.168 ms
$\overline{x}$	122 µs	20.194 ms
Min.	112 µs	17.707 ms

Table 3.8: Delays associated with checkpointing functions

The results reveal an anomaly in form of a worst case suspension delay that is significantly larger than any other of the 10,000 logged delays. As it seems to correlate with the worst case delay of the dirty page retrieval function, in the next experiment, we analyze guests memory usage.

IDLE GUESTS MEMORY OVERHEAD In order to identify and analyze the source of the worst case delay during the dirty page retrieval function, we evaluate the memory overhead generated by protected guests during replication and compare the results. To this end, in the subsequent experiment, we instantiate idle guests and identify for each OS the amount of modified pages per checkpoint iteration. We

expect the non-service related activities in Leap to be responsible for a significant part of the measured delays. Figure 3.13 presents the collected data from 10,000 iterations for both Leap and MiniOS.

The obtained results confirm our assumption. On average, an idle Leap guest modifies 515 pages per checkpoint iteration. In the same time interval, MiniOS dirties 21 pages on average. Mapping these pages to memory size, on our x86 architecture, yields an average checkpoint size of 2.1 MB for Leap and 0.08 MB in the case of MiniOS. Based on the utilized 1 Gigabit Ethernet interface, the computed theoretical network transmission delays are approximately 16.8 ms for Leap and 0.68 ms for MiniOS, respectively. The difference equals a factor of 24.

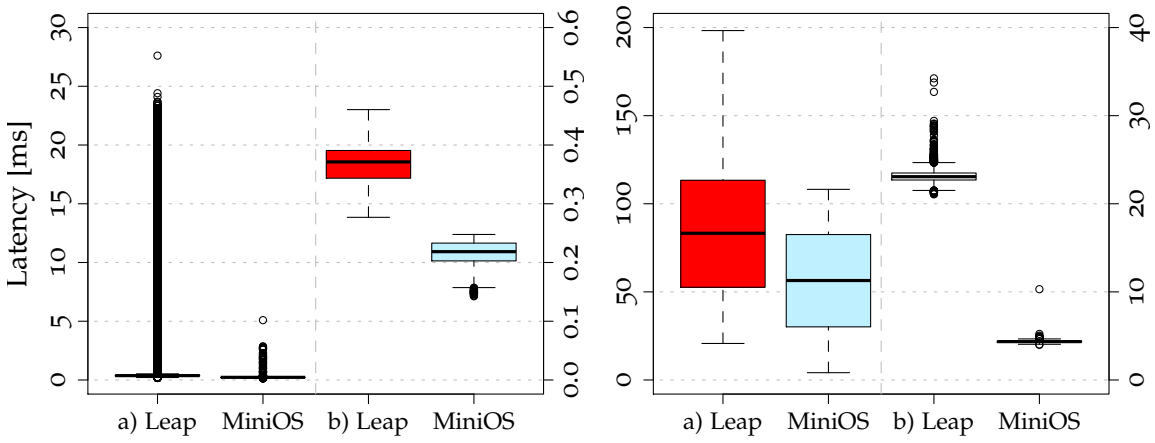
Further, the results also show an issue with memory usage that explains the occasional outliers in the measured delays of the checkpoinitng functions and thus the response timings. During this experiment, in seldom cases (approximately one per 3,000 checkpoints), an idle Leap guest modifies up to 2000 pages and in worst case even as much as 2586 pages. This gives a checkpoint size of 10.59 MB and takes, based on our network property assumptions, 84.73 ms to transmit. In the case of MiniOS, a worst case of 45 modified pages was recorded. This translates to a checkpoint size of 0.18 MB and a transmission delay of 1.47 ms.

Two conclusions can be drawn from this experiment. First, there is a substantial difference in memory usage between the two idle guests – on average by 2670% and, when comparing the worst cases, even as high as 5883%. Second, Leap generates non-deterministic page modification spikes approximately five times greater than in average, whereas MiniOS approximately by a factor of two. Moreover, comparing the complexity of the OS's and their lines of codes, we expect any further optimization – in this respect – to be easier to accomplish with MiniOS than Leap.

The presented results clearly motivate the deployment of protected CPS services as unikernels. Therefore, the next section evaluates this approach in more detail.

LATENCIES OF UNIKERNEL-BASED CPS SERVICES The following experiments evaluate the effect of leveraging unikernels for the construction of protected CPS services in respect of their timing properties and compare the results with the standard approach. To this end, we repeat the latency experiments with services deployed as unikernels and contrast them with the previously obtained results.

Figure 3.14 depicts the latency distributions of the differently (periodic and self-determined) protected echo servers, both with disabled (1) and enabled (2) network buffering. The response times with disabled network buffering (1) under periodic checkpointing (a) for the server deployed under Leap are: 27.60 ms in worst case, 2.48 in average



(1) Network buffering disabled.

(2) Network buffering enabled

Figure 3.14: Effect of classical and unikernel-based CPS service deployment on latency. a) response times under periodic checkpointing and b) response times under self-determined checkpointing. Both for network buffering disabled (1) and enabled (2).

and the computed standard deviation is 5.24 ms. The results obtained with MiniOS are respectively: 5.09 ms, 0.28 ms and 0.35 ms. The values measured from the server under self-determined checkpointing b) when deployed with Leap are: in worst case 0.46 ms, in average 0.36 ms and 0.029 ms for standard deviation. In the case of MiniOS 0.24 ms, 0.21 ms and 0.021 ms, respectively.

The obtained results confirm our previous findings. In the case of periodic checkpointing, due to less memory overhead generated by MiniOS, the protected unikernel provides significantly better response timings. Moreover, the experiment with self-determined replication reveals another downside of encapsulating services in a general purpose OS's. Despite the minimal computational workload of the echo server and its real-time prioritization Leaps non-service related background activities still interfere with its execution and induce additional delays. A comparison between Leap and MiniOS of the worst case response latencies shows a difference of 191%.

In the case with network buffering enabled (2), the collected response times for the periodic checkpointing (a) under Leap are 198.27 ms in the worst case and amount to 83.14 ms in average, compared to a worst case of 108.17 ms and an average of 56.31 ms in the case of MiniOS. Once again, the results validate and comply with the checkpointing model discussed in Section 3.4.3.1 as well as with the previous findings with respect to the overhead generated by Leap. While the worst case latency exhibited by MiniOS is to be explained by the 100 ms period and the delay of the *transmit and synchronization stage*, the additional latency of the service encapsulated in Leap is to be accounted to the previously measured and discussed non-

deterministic memory usage and the interfering effect of the execution of the non-service related threads.

Finally, in the last experiment conducted under self-determined replication, in the case of Leap, a worst case delay of 34.23 ms was measured and an average response time of 23.11 ms was computed. For MiniOS, we obtained a worst case latency of 10.30 ms and computed a mean response value of 4.36 ms. The results show that also in the case where network buffering is enabled, a significant performance improvement can be achieved by utilizing unikernels. And this both with respect to the worst case delay as well as the deterministic system behavior. Note than the computed latency dispersion of the service encapsulated in MiniOS exhibits a standard deviation of only 230 µs.

3.4.5 *Real-world Applicability and Service Recovery Latencies*

One of the areas where virtualization is gaining significant interest is the automotive domain. In this domain, the system software is moving towards an integrated and service-oriented architecture. Therefore, in order to test the real-world applicability of our approach, we implemented the AUTOSAR protocol *Scalable service-Oriented MiddlewarE over IP* (SOME/IP) [7, 8] and integrated it as a library into our unikernel. SOME/IP is an automotive and embedded communication protocol which supports – among others – remote procedure calls, event notifications and service discovery. In the following experiments, we instantiate (on separate physical hosts) one protected SOME/IP server-service and three unprotected SOME/IP client-services. The server-service represents an ECU offering a service instance that provides sensor values which are being periodically refreshed with a frequency of 20 ms. This is also the frequency at which checkpointing is being conducted. The heartbeat frequency is set to the half of the sending frequency, which is 10 ms. All services are being deployed on dedicated cores as unikernels and each is granted 16 MB of memory. In this scenario, the protection focus is on the server-service as it holds the subscriptions for the offered services. After a successful service-discovery procedure, which involves the announcement, detecting and finding of services, the clients subscribe to the server-service for notifications. In the following, we analyze the CPU overhead and the checkpoint sizes during replication as well as the server-service recovery times in the face of a primary host failure. Table 3.9 summarizes the replication costs. The results are the same for both replication approaches.

The CPU overhead was measured in Domain 0, which was running on a single core and where the replication engine threads are being executed. The measured value is adjusted for the CPU costs of the server-service, yet, it still comprises the logging costs. During the 10,000 measured checkpoints, the checkpoint sizes varied between

Stat. values	*CPU overhead*	*Checkpoint size in pages*	*Checkpoint size in kB*
$\overline{x}$	24%	28.19	115.4
Max.	-	49	200.7

Table 3.9: CPU overhead induced by server-service replication and checkpoint sizes.

28 and 49 pages and amounts to 28.10 pages in average. Based on the utilized 1 Gigabit Ethernet interface, these values translate to a theoretical network transmission delay of 0.92 ms in the average case and 1.6 ms in the worst case.

	Net. buff. off		*Net. buff. on*	
Client	*Periodic*	*Self-det.*	*Periodic*	*Self-det.*
1	38 ms	34 ms	40 ms	39 ms
2	38 ms	33 ms	40 ms	39 ms
3	38 ms	34 ms	40 ms	39 ms

Table 3.10: Server-service recovery delays from the client-services point of view in the presence of a primary host failure.

The next table, Table 3.10, depicts the server-service recovery times from the point of view of the clients. The latencies represent the delta between the point in time when the last packet was received from the primary host and the first one received from the backup server after failover. Note that during normal operation these delays amount to 20 ms, as this is the frequency at which the server notifies the clients due to the sensor values update. Therefore, from the point of view of the clients, the actual additional delay, induced by the primary host failure and the service recovery, ranges from 14 to 20 ms – depending on the method and settings. While network buffering was enabled all packets arrived in the correct order. With network buffering disabled, one packet was resend and was therefore duplicated from the point of view of the clients. The fact that after the failure of the primary host all clients continued to receive packets shows that the whole state of the server-service, including active network connections, sensor states and the subscriber list, was correctly preserved.

Finally, please note that in order to provide a comparable and therefore fair evaluation environment of the two replication methods, we have chosen a setting where our approach mimics the periodic replication model. Recall that the server-service is determining its sensor values in a strictly periodical fashion. In an additional experiment, where the sensor values were refreshed based on a sporadic activation

pattern, this is randomly determined (using uniform distribution) over an interval of 20 - 100 ms, the CPU overhead amounted only to 11% – compared to the 24% while using the periodic replication approach. Consequently, an analogue improvement of factor 2 was also observed for the network utilization.

In summary, we argue that in respect of real-world applicability the provided results are promising and our method is only a few steps away from use in a production environment.

3.5 CHAPTER SUMMARY

In this chapter we introduced the reader to the technological issues related to the architectural foundations of the virtualized execution environment Xen with respect to the deployment of fault-tolerant CPS applications. We identified and discussed the shortcomings of Xen and state-of-the-art high availability approaches, in particular Remus, and proposed solutions that address those deficiencies.

The first part of the chapter concerns with the scheduling models and characteristics of Xen and CPS-Xen. In this part we thoroughly evaluated the properties of the default Xen SEDF scheduler as well as the CPS-Xen RM scheduler and assessed their suitability for scheduling latency-sensitive CPS applications. The suitability of the schedulers was being validated by using timing constraints – encountered in real-life applications – that were derived from the domain of cyber-physical energy systems. The results of the conducted series of experiments shed light on several aspects of the analyzed architecture. First, the only scheduler of Xen 4.1.4 that aims at the fulfillment of timing constraints fails at this task even at low CPU utilization levels. Secondly, and more important, our studies of the CPS-Xen RM scheduler revealed that optimizing the VMM scheduler alone is not sufficient. It turns out that in the standard Xen architecture I/O scheduling may introduce priority inversion into the system, in particular, occasionally handling network packet of lower prioritized VMs before packets of higher prioritized VMs have been processed. The proposed solution of synergyzing the work of both schedulers by aligning the priorities of the VMM scheduler and the I/O scheduler significantly improves the response times of the virtualized CPS applications as well as minimizes latency dispersion. The latter translates to a more deterministic system behavior and therefore facilitates the planning of virtualized execution environments for CPS. Finally, we compared our CPS-Xen RM scheduler with the RT-Xen RTDS scheduler on an inexpensive COTS development board. The results show that in contrast to the RTDS scheduler, our CPS-Xen RM scheduler performs well and – even in high load situations – never misses a deadline.

The second part of this chapter concerns with the efficiency of high availability solutions that employ virtual machine replication. In this

part we analyzed the state-of-the-art techniques and discussed their suitability for the construction of fault-tolerance CPS applications. To this end, we preliminarily evaluated Remus a prominent HA solution that employs the periodic checkpointing model. The preliminary results revealed substantial efficiency deficiencies which – despite some implementation issues – mainly result form the underlying and inadequate – with respect to the CPS domain – assumptions of the periodic replication model. Therefore, as the next step we introduced the self-determined replication model, which is tailored for the requirements encountered in the CPS domain. Next, by incorporating the unikernel-based approach for the deployment of CPS applications and combining it with the proposed novel replication model, in a series of experiments, we showed that our combined concept fulfills the timing constraints encountered in the CPS domain – and this in a highly efficient manner. Finally, the results of the real-world application evaluation are promising and demonstrate the feasibility of our approach in production environments.

In summary, this chapter describes our contributions to the technological foundations of virtual execution environments for CPS.

4

PLANNING VIRTUALIZED CPS

Despite the growing popularity of virtualization in the embedded system domain, there is little literature on how to integrate a CPS into one functional whole on a homogeneous platform by means of this technology. Instead, till recently, research has put the most emphasis on the topics of hardware utilization, fault-tolerance and security. The possibility of integrating and consolidating systems – one of the most important features of virtualization technology and one of the main reasons for its popularity – remains an open research topic in the embedded systems community. This chapter aims at reducing that gap by presenting a methodology for the integration of virtualized CPS. More specifically, the proposed methodology addresses the issue of planing safe and efficient virtualized CPS compute and control clusters which host CPS applications embedded in VMs. In contrast to typical resource allocation solutions from other domains that employ virtualization (e.g. cloud computing), our solution not only optimally dimensions the compute and control clusters, but also provides strict guarantees regarding the timing predictability of the integrated CPS. Furthermore, it facilitates the modeling of such systems, making it more accessible to system designers. The findings presented in the following sections are based on [63].

The first section of this chapter discusses the challenges to the approach and the related work. Next, an overview of the methodology is being provided. It is being followed by an detailed description of the used models and the employed techniques. The fourth section presents the evaluation results and is being succeeded by a discussion which shows how our solution can help in answering practical questions that a system designer or administrator of a virtualized CPS cluster could encounter. Finally, the chapter concludes with a summary.

4.1 CHALLENGES

The integration of CPS by means of virtualization is a complex and challenging undertaking. In order to guarantee a safe and efficient outcome, numerous requirements have to be fulfilled. The planning of virtualized CPS compute and control clusters has to take the following aspects into account:

- CPS are mostly real-time systems. This implies that computational delays of their critical functions are as important as their functional correctness. The violation of a real-time property of a critical task can lead to damage or even a catastrophe. In order

to avoid such problems, guarantees regarding timing properties have to be provided.

- Due to the distributed character of CPS, a local – single core or host – timing analysis is not sufficient. A global interference analysis of the system components is required and therefore metrics like end-to-end latencies have to be obtained. This implies that the analysis also has to take communication delays as well as data dependency into account.

- CPS control or protection functions have to be characterized by high dependability and availability. Therefore, the analysis should allow to incorporation VM migration into its model. This is due to at least three reasons: Migration forms the technological foundation for high availability solutions, is of advantage when resolving maintenance issues and even becomes a necessity, when deploying additional workload to an already running system that is subject to certain quality-of-service conditions, like minimal service downtimes or response times.

- CPS vary significantly in their size. Therefore, the planning process should remain efficient and scale with the increasing size of a CPS.

- To obtain efficiency, the planning and integration process has to allow for the optimization of several non-functional requirements, such as latencies, VM migration overhead or the amount of needed hardware.

Considering the above listed requirements with respect to integrating CPS by the means of virtualization, our research goal can be expressed by the following question: How to find a suitable mapping between the VMs encapsulating the CPS applications and an appropriately dimensioned hardware as well as – at the same time – fulfill the strict timing requirements of CPS?

Even though this question has not been addressed by literature directly, there exist two research areas where related problems are being discussed and are of importance for our work.

The first related issue is the application placement problem which is being discussed in the area of embedded systems and considered as one of the most urgent issues [96]. The importance of this topic results from the technologically motivated shift of chip manufactures from improving single-core processors towards designing and constructing many- and multiple-cores processors. As a consequence, plenty of methodologies have been proposed to tackle this problem. They aim at finding an efficient mapping by optimizing metrics like execution times, energy consumption, throughput or memory usage. For solving the mapping problem most of them adopt one or more of the well established search approaches, among them: Genetic Algorithms [21,

24, 55], Simulated Annealing [105, 140] or *Integer Linear Programming* (ILP) [3, 11, 140]. For a more extensive survey see [122].

Cloud computing, especially the aspect of efficient allocation of cloud infrastructure resources for VMs, constitutes the second relevant area. Until recently, most works from this domain focused on allocating resources in Infrastructure as a Service (IaaS) clouds [70]. Since lately, concepts like SDN and Network Function Virtualization (NFV) extended the problems definition by incorporating the requirements for network service deployment in cloud infrastructures [56, 89]. The efficient deployment of flexible and scalable softwarized network functions is considered to be an enabler for future cloud networking solutions including carried clouds based upon the fifth generation (5G) of mobile networks [101, 137].

Despite strong structural similarities with our problem, all of these approaches fail at fulfilling our needs. The reason is twofold. Firstly, none of the approaches provides actual guarantees regarding the response times of the deployed functions, tasks, VMs or services. And although some of them include the minimization of latencies into their solutions, the optimization or minimization of performance metrics does not automatically translate into providing strict guarantees regarding these quantities. Secondly, there is the modeling issue. Most of the approaches – especially from the cloud computing domain – operate on a higher abstraction level than the one that is required in order to adequately model our problem (e.g. CPU scheduling is not being modeled). Therefore, they are not suitable for modeling real-time or cyber-physical systems. On the other hand, the few approaches that potentially or explicitly enable the modeling of time-critical systems, when applied to our constraints and objectives, render the problem – due to its complexity – practically intractable.

In the context of our studies, an interesting work is provided by Thiele et al. in [130]. The authors propose a framework called Distributed Operation Layer (DOL) for alleviating early design stage system performance analysis and optimizing task mapping by a multi-objective algorithm. Unfortunately, this approach could not be adopted due to the fact that for performance analysis DOL embeds real-time calculus [129] which does not handle well the modeling of state-based systems. As our problem requires to differentiate system states (e.g. event-based high availability), we were unable to adequately model our architecture with DOL.

4.2 CONCEPT

In literature, computational problems displaying requirements and constraints similar to those that have to be fulfilled while planning efficient virtualized CPS, are being referred to as *Multiobjective Optimization Problems* (MOP) [25, 106]. However, in our case, the application

of methods dedicated for solving these kind of problems is being complicated by the fact that the criteria for providing guarantees regarding the timeliness of the planned system cannot be represented as a simple function. In order to determine a feasible solution, an exhaustive system performance analysis of each postulated VM to hardware mapping has to be conducted. To solve these challenges, we propose an approach that combines multiobjective optimization with formal system performance analysis. In particular, we use *Evolutionary Algorithms* (EA) to tackle the problem of exponential search spaces and adopt classical scheduling theory formulas in order to analyze the timing properties of the candidates in question and thereby to asses their feasibility. The contributions of the following sections can be summarized as follows:

- We propose a holistic approach for the planning and integration of virtualized CPS.
- We describe the methodology behind the approach, define the used methods, the corresponding models, as well as the assumed system architecture.
- We combine EAs with algorithms considered in classical scheduling theory. This allows not only for an efficient search of the solution space but also for a detailed modeling of the problem – including CPU-scheduling – which in turn enables to provide guarantees regarding the timing predictability of the system and, at the same time, to optimize additional non-functional requirements.
- Finally, our approach also provides answers to numerous practical questions that arise when integrating or managing a CPS: How to fulfill the strict requirements without an overdimensioning or overprovisioning of the system? Can additional functions be safely deployed in an already running system? Or, how to efficiently plan maintenance?

4.3 METHODOLOGY, ARCHITECTURE, MODELS AND TECHNIQUES

An overview of our entire approach is given in Figure 4.1. From a high-level perspective the core of our approach comprises two methods: an evolutionary algorithm and a system performance analysis technique. These methods are combined in order to compute an efficient mapping of VMs to a distributed execution platform, provide guarantees regarding the timing predictability of the system and – at the same time – enable the optimization of additional non-functional requirements. The process starts with a specification of the CPS. This input is being read and transformed into parameters for the EA. Every time the EA computes a new allocation it is being transformed into

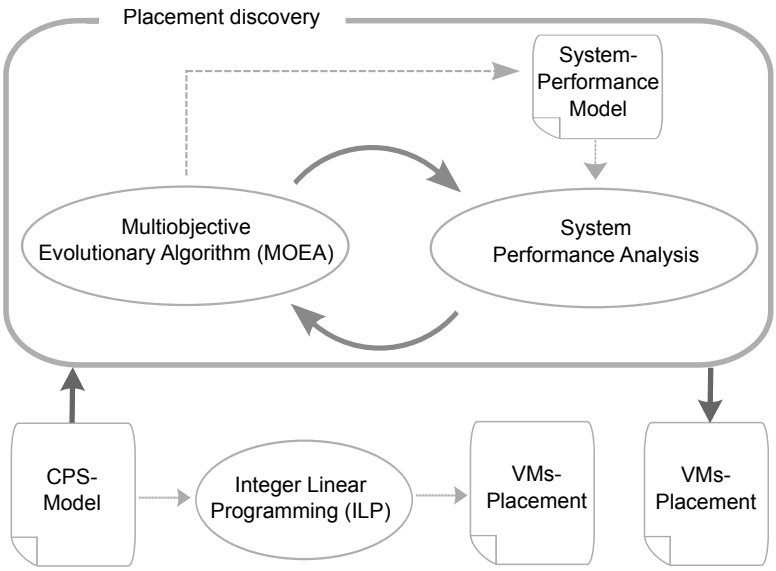


Figure 4.1: Approach Overview

a system model that serves as an input for the performance analysis. After the evaluation of the candidate in question, the analysis output is being used to update the EA. This process terminates when pre-defined conditions are satisfied. The result is a placement for the VMs.

In order to facilitate the evaluation of our EA and to provide a holistic modeling view on the problem, we have extended our framework with ILP. The EA approach faces the following problem: Even though the solutions found by our EA fulfill the a priori defined constraints and the strict timing requirements of the applications, the approach can not make any statements regarding the quality of the solutions. In particular, this means that for a given request it is unable to quantify the distance between the results of the EA and an optimal solution. In turn, ILP equips us with the means for computing an optimal placement, yet the approach has to struggle with scalability issues. Therefore, a complementary analysis including both the ILP and EA approach is only feasible for relatively small instances of our placement problem. For most real life tasks we have to depend on the EA approach. Nevertheless, the ILP results can be used as a metric to derive statistical insight regarding both the quality of the EA as well as the impact of the adjustments done to the EA operators.

4.3.1 *Architecture and Scheduling Model*

The starting point for our approach forms the virtualized execution platform architecture described in Chapter 3 and depicted in Figure 3.2. The execution platform is able to host multiple time sensitive applications, encapsulated in VMs, and comprises two communication

interfaces, one for the virtualization management traffic and another for the communication between the physical part of the CPS (sensors and actuators) and the servers.

As described in Section 3.3.2.1, platform virtualization inherently generates a two- (or n, in case of nested virtualization) level scheduling hierarchy. On the first level, the hypervisor schedules its VMs, assigning to each the designated share of resources. The second layer is constituted of the VMs' schedulers, which in turn schedule their own tasks.

We avoid this complexity by employing unikernels (see Section 2.1.6). This allows us to flatten the scheduling hierarchy. We embed application threads into VMs and instantiate as many dedicated VMs as needed for the integration of the CPS. From this perspective, missing a deadline of a thread executing within a VM is equivalent to missing a deadline by the hypervisor.

Regarding the scheduling policies, in CPS most of the applications strictly depend on periodic sensor data. Scheduling policies that perform well under such conditions are rate- and deadline monotonic algorithms. These are dynamic preemptive scheduling algorithms based on static priorities where the highest priority is assigned to the task with the shortest period or, respectively, to the one with the shortest deadline. Therefore, we have incorporated these algorithms into our models.

4.3.2 *CPS-Model*

In order to compute a placement for the VMs a model is required which unifies information about several aspects of the CPS. This includes detailed information about the applications encapsulated in the VMs, the underlying execution platform (hardware and the scheduling policies), the communication infrastructure as well as the interactions with the physical environment. Let us begin with the definition of a virtualized CPS application (vCPSA) representing the VM and the encapsulated application.

Definition 1. A *vCPSA* is a virtualized CPS application defined by a tuple:

$$\tau_i = (C_i, T_i, B_i^{\text{in}}, B_i^{\text{out}}, D_i, H_i^{\text{init}}, H_i^{\text{regular}}), \tag{4.1}$$

where C_i is the execution demand, T_i denotes the period triggering the computation, B_i^{in} and B_i^{out} the ingoing respectively outgoing message size, D_i the deadline, H_i^{init} the size of the migration overhead and H_i^{regular} the maximal checkpoint size.

This compact definition already includes most of the required characteristics. Still, this abstraction level does not allow to express data dependency between the vCPSA. Therefore, we need another definition:

Definition 2. A *vCPS* is a virtualized cyber-physical system defined by a tuple

$$\tau = (\{\tau_1, \ldots, \tau_n\}, \phi) \tag{4.2}$$

of *vCPSA* and an additional function ϕ:

$$\phi(\tau_j) = \begin{cases} \tau_k & \text{in case } \tau_k \text{ is data dependend on } \tau_j \\ \perp & \text{otherwise.} \end{cases} \tag{4.3}$$

Finally, we need a definition of a placement:

Definition 3. Given a *vCPS* $\tau = (\{\tau_1, \ldots, \tau_n\}, \phi)$, a *placement* is a disjunct partitioning $P_1 \sqcup \cdots \sqcup P_k$ of the virtualized CPS applications $\{\tau_1, \ldots, \tau_n\}$ on k hosts.

As already mentioned, in order to compute a placement we also have to provide characteristics about the capacities of the computation and communication hardware. This information is required for the generation of the system models for the performance analysis and has to be identified separately. Some parts of the information are of objective nature and can be determined by appropriate standards and associations from the given field of application. For example, if we were to integrate a subsystem from the domain of smart grids and would like to virtualize some substation control or protection functions then, in order to find out how often functions are being invoked or what their timing requirements are, we would have to refer to the IEC 61850 Standard [58].

The computation or communication capacities of the underlying hardware infrastructure, like the amount of available cores or network bandwidth, can be obtained from reference manuals and data sheets as well as benchmarking or profiling of the hardware. Regarding scheduling, the assumed scheduling policy has to map the one that is actually being used on the execution platform. An adequate representation of a scheduling policy – both in the EA and for the performance analysis technique – that still remains computationally efficient is not as straightforward as it may seem at first glance.

Another issue that is frequently rising the most controversies is the estimation of the execution demands. In the context of our work, we have to distinguish two aspects of this problem. There exists a rich literature on how to determine the WCET of a piece of software. Classical approaches assume a relatively simple yet precise hardware model and combine it with a static software analysis. Unfortunately, in our case, considering the complexity of the underlying execution platform that is needed for virtualization purposes, the application of established WCET-analyzers is – at least for today – not possible. Therefore, for the purpose of empirical studies that aim at testifying to our approach, the best we can hope for is an approximation of the WCET through a campaign of very thoroughly conducted latency

measurement experiments. However, regarding the core of this work, our approach is of formal nature and therefore independent of the quality of the WCET-parametrization in the input model. Still, we find it important to make this distinction between the analytical and empirical meaning of the WCET notion used in this context.

In summary, our CPS-Model is a matrix that comprises a number of characteristics, like the number of VMs to integrate, their periods, deadlines, WCETs and other attributes required to fully specify the placement problem. Which of these are actually needed and used depends on the objective function and the given constraints.

4.3.3 *The Evolutionary Algorithm*

At the heart of our approach lies an evolutionary algorithm. Generally speaking, an EA is a metaheuristic optimization algorithm. It transfers the mechanisms encountered in biological evolution, like selection, recombination or mutation into the language of mathematics and computer science and tries to find a solution to a problem through the iterative usage of such mechanisms. In most implementations, after generating an initial population, the EA repeatedly traverses through the following steps:

- evaluate individuals
- select candidates for variation based on their fitness
- reproduce

The EA terminates after satisfying some a priori defined constraints and optimization objectives (in other terms, achieving a sufficient fitness) or simple by exceeding a time limit.

As mentioned before, several conflicting objectives render our discussed placement problem a multiobjective optimization problem. Fortunately, there exists a large number of frameworks that facilitate such optimization. The majority of the available multiobjective optimization evolutionary algorithms (MOEA) use a derivative of the *Non-Dominated Sorting Generic Algorithm II* (NSGA-II), which promotes the discovery of a diverse subset of the Pareto-frontier (non-dominated solutions) [32]. In this work, we use the Open BEAGLE [43] framework, which provides an implementation of the proven NSGA-II algorithm, yet additionally supports features, like parallel fitness evaluation or checkpointing, and is free.

4.3.3.1 *Encoding and Operators*

The choice of problem representation is crucial to the performance of an evolutionary algorithm. In order to reduce the computational complexity, which is determined mainly by fitness evaluation, we

decided to choose a fixed length representation for the encoding of our individuals.

Definition 4. Given a vCPS $\tau = (\{\tau_1, \ldots, \tau_n\}, \phi)$ then a genotype space of fixed length is defined as:

$$\mathcal{G} = \{((1, i_1), \ldots, (n, i_l)) \mid 1 \leq i_l \leq n, \quad \forall 1 \leq l \leq n\}, \qquad (4.4)$$

where each tuple represents a vCPSA, whereby the first component denotes the vCPSA index and the second the allocated host index.

Definition 4 ensures that every generated candidate already encodes all of the CPS applications that need to be mapped. Therefore, every candidate is formally valid, yet not necessarily feasible in regard to the given constraints.

The second factor that has a decisive influence on how well the EA performs are its operators. For the exploration of the search space we use the two standard operators: crossover and mutation. As is known, the operators have to be appropriately tailored and adjusted to the structure of the problem, otherwise, the EA may not perform better than a random search over the solution space. Therefore, we have fine-tuned our operators by repetitive EA execution and comparison with the ILP solution metric.

4.3.3.2 *Fitness*

The fitness of a candidate in question gives evidence to the quality of the solution this candidate is encoding. We have chosen to model the fitness of an individual as a layered vector of fitness values representing both the constraints as well as the optimization objectives:

1. i) average violated resource consumption
   (average resource consumption $> 100\%$)
   ii) average violated deadline
   (finish time of vCPSA $i > D_i$)
2. i) migration traffic
   ii) number of used hosts
   iii) average response time

The first layer contains criteria for the estimation of the distance to a feasible placement. The values are provided by system performance analysis. Only when both values are zero, meaning (i) the utilization of all resources (hosts and networks) is less or equal than 100% and (ii) the needed time for any vCPSA is less or equal than its deadline, a placement is feasible. We found out that using the average – instead of the worst-case – values has an positive influence on guiding the EA in its search.

The second layer contains optimization criteria. Those are only being evaluated in the case of an already existing feasible placement. The migration traffic denotes the overhead necessary to reach a postulated placement when starting from a currently given one. The other two are self-explanatory: ii) minimizes the needed amount of hardware and iii) minimizes the response times of the applications in the system. Note that these criteria are potentially conflicting. For example, shorter response times can likely be achieved by providing more hardware to the system and optimal solutions for management issues, like replacing CPUs, might require extensive migrating of virtual machines.

The list of the second-layer metrics is not exhaustive, also other aspects of the system can be adopted as an optimization criteria – given they can be satisfactory quantified.

The fitness evaluation proceeds hierarchically. Each of the candidates can be in one of the two states: Either his layer 1 is still being evaluated, then the placement is not (yet) schedulable, or layer 2 is being optimized and the candidate is schedulable with the actually given characteristics. This modeling approach was chosen due to the two aspects: Feasible (scheduable) solutions always dominate not scheduable solutions and the model of the system can easily be extended by additional constraints, objectives or states and still the fitness vectors remain comparable across all generations.

4.3.4 *Performance Analysis*

A feasibility assessment of a postulated placement depends on several interdependent constraints. The analysis has to include characteristics like the utilization level of the computation and communication infrastructure, data dependency and timing requirements of the virtualized CPS applications. Moreover, the analysis has to be exhaustive, otherwise no guarantees can be given regarding the safety-critical characteristics of the system and has to be fast, or else the computation times for the fitness evaluation become not practically feasible. For this reason, we have decided to rely on analytical methods for system performance analysis, specifically, conducting this work we utilize the MAST framework. For a more detailed motivation regarding the choice of performance analysis method, please refer to Section 2.2.

The results of the analytical methods are only as good as their underlying models. Therefore, the system model has to be well parametrized. For this, we transform the information encoded in the CPS model as well as the placement candidate in question into a MAST system model. This takes place on each invocation of the performance analysis function. The following elements need to be specified in conformity with the MAST syntax: the *processing resources*, representing the computational and communication capacities, the *schedulers*, which define the scheduling policies, the *scheduling servers*, which represent the

$$\text{minimize} \sum_{i=1}^{n} \sum_{j=1}^{n} j \cdot X_{i,j} \tag{4.5}$$

$$\text{minimize} \sum_{i=1}^{n} H_i^{\text{init}} \cdot (1 - X_{i,\text{preset}(i)}) \tag{4.6}$$

Subject to:

$$\sum_{j=1}^{n} X_{i,j} = 1 \qquad (\forall i,\, 1 \leq i \leq n) \tag{4.7}$$

$$t_0 \geq X_{i,j} \left(C_i + \sum_{k=1}^{i-1} X_{k,j} \left\lceil \frac{t_0}{T_k} \right\rceil C_k \right) \qquad (\forall i, j,\, 1 \leq i, j \leq n, \forall t_0 \in P_{i,j}) \tag{4.8}$$

$$t_0 \geq X_{i,j} C_i + \sum_{k=1}^{i-1} \left(X_{k,j} \left\lceil \frac{t_0}{T_k} \right\rceil C_k - (1 - X_{i,j}) \left\lceil \frac{t_0}{T_k} \right\rceil C_k \right) \qquad (\forall i, j,\, 1 \leq i, j \leq n, \forall t_0 \in P_{i,j}) \tag{4.9}$$

$$X_{i,j} = \{0, 1\} \qquad (\forall i, j,\, 1 \leq i, j \leq n) \tag{4.10}$$

schedulable entities in the scheduler and finally, the *operations*, which define the actual computation or communication demands.

We then use the "Holistic Analysis" implemented in MAST [48] to compute resource consumption and to check if all deadlines are met.

4.3.5 *Integer Linear Programming*

As already mentioned, we have extended our framework with an ILP formulation of the placement problem (Equations 4.5 to 4.9). The formulation is based on the approach presented in [11]. Our formulation assumes the same system model as the EA, both are being generated from the shared CPS-Model. In its current form, the formalism allows for finding an optimal allocation for the vCPS applications in respect to the number of used hosts (Equation 4.5) or the VM migration overhead (Equation 4.6) – for example in case when we want to add a new vCPSA to a already deployed system. In order to guarantee real-time capabilities, we incorporated the exact schedulability test for the rate monotonic scheduling policy as constraints into our ILP formulation.

There exist other formalization approaches [3] that rely only on CPU utilization. However, as utilization based tests can classify schedulable task set as not schedulable, we chose the more precise, sufficient and

Notation	Definition
$X_{i,j}$	The ith VM runs on the jth host (boolean value)
C_i	Computational demand of the ith VM
T_k	Period of the kth VM
H_i^{init}	Migration overhead for ith VM
$X_{i,\text{preset}(i)}$	The ith VM resides on the same host as before (boolean value)
t_0	Point in time, see [11, 13]

Table 4.1: Variable definitions in the ILP formulation.

necessary schedulability test and based the constraint formulation upon the well-known worst-case response time analysis formula [72].

The schedulability constraint checks if the needed computation time until timestamp t_0 is less or equal than the timespan till t_0 (Equation 4.8) for one timestamp in the range $[0 : D_i)$ for a vCPS τ_i. Since there are exponentially many points, which may fulfill this equation, the set of timestamps must be reduced. Bini and Buttazzo devised a method to identify a subset $P_{i,j}$ of timestamps, which may be checked [13]. Before we can implement this constraint in an ILP, the formula has to be linearized (Equation 4.9). Additionally for every combination of vCPS τ_i and host j, only one of the $|P_{i,j}|$ formulas must be fulfilled. One method to achieve this in an ILP formulation can be found in the appendix of [11].

Finally, the first constraint (Equation 4.7) ensures that every VM is deployed exactly once. The variables used in the ILP formulation are described in Table 4.1.

4.4 ANALYZING VIRTUALIZED CPS

In this section, we describe the obtained findings from experiments conducted with the proposed algorithm. First, we present some general results showing the plausibility and performance of our algorithm and the computed solutions. Next, we show how good the computed solutions are compared to the optimal solutions we received through evaluating the ILP formulation. Finally, we examine how our algorithm can help answering practical questions that system designers encounter in their daily work and aid in decision making.

4.4.1 *Scenario Generation*

For the purpose of evaluation, the models of the CPS to be analyzed are being generated randomly. The parameters of the virtualized CPS

applications are inferred from probability density functions (PDF). As the focus of our evaluation lies on the testing of the quality of our algorithm, the following parametrization of the PDFs is chosen freely, in other words, it is not tailored for any specific CPS domain or real life scenario, although the PDFs can easily be adapted to fit the requirements of any specific domain. The parameters C_i, T_i, B_i^{in}, B_i^{out}, D_i, H_i^{init}, H_i^{regular} of the vCPSA are being inferred as follows:

$$\begin{aligned} D_i = T_i &\sim 0.8 \cdot \mathcal{N}(6,2) + 0.2 \cdot \mathcal{N}(60,20)\ [ms] \\ C_i &\sim \mathcal{N}\left(\frac{T_i}{3}, \frac{T_i}{8}\right)\ [ms] \\ B_i^{\text{in}} = B_i^{\text{out}} &\sim \mathcal{LN}(0, 0.125)\ [KB] \\ H_i^{\text{init}} &\sim 0.8 \cdot \mathcal{N}(256,64) + 0.2 \cdot \mathcal{N}(1024,256)\ [KB] \\ H_i^{\text{regular}} &\sim \frac{H_i^{\text{init}}}{2}\ [KB], \end{aligned}$$

where $\mathcal{N}$ denotes the normal and $\mathcal{LN}$ is the log-normal distribution. T_i and H_i^{regular} are mixtures of distribution functions and consist of two weighted parts, in order to model some computationally intensive and multiple lightweight components. Our specification of H_i^{regular} is based on H_i^{init} and stems from [23] where the authors thoroughly analyze various workloads in the context of live migration. Their results show that even for problematic workloads only less then 50% of the pages of a VM are being modified during execution. For the purpose of illustration, we interpret evaluated random variables with the given units in squared brackets.

4.4.2 *Settings*

We conducted the experiments with an initial population size of 100. Per iteration the NSGA-II generates additional individuals – from the remaining ones – and extends the population to 200 in order to reduced it again using its selection algorithm. In each iteration, we apply the one point crossover operator with the probability of 0.7 and the mutation operator with the probability of 0.3. If mutation is being applied, the flip probability is 0.02. Mutation is being conducted on randomly selected individuals. The parameters and operators are changeable and might be adjusted to specific models.

For statistical purpose, we repeat each experiment 15 times. Therefore, the following box plots represent the values of 15 independent runs on the same vCPS model. Since the multiple optimization objectives can not be combined into one scalar value (otherwise one could simply use a single-objective EA), we compare the different criteria separately.

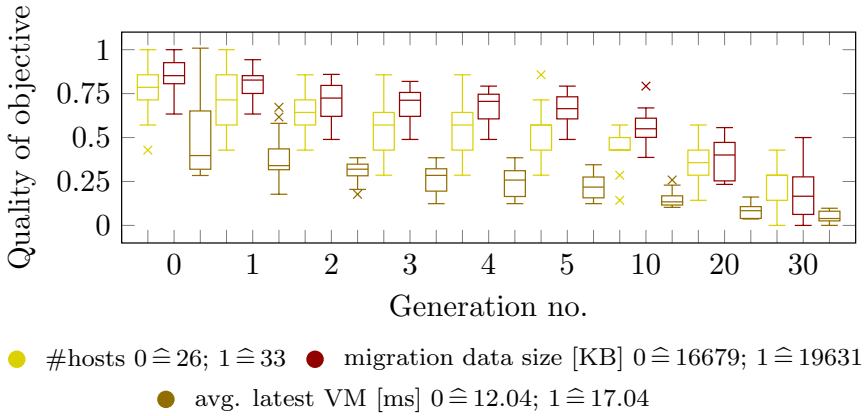


Figure 4.2: Quality of solutions for selected generations for a model of size 50. Each box plot represents 15 independent runs on the same data.

4.4.3 *Experiments*

This section presents the results of the experiments that allow to assess the quality and applicability of our approach.

4.4.3.1 *Generation Improvement Ratio*

The first experiment, depicted in Figure 4.2, shows the improvement of the placement quality across generations with respect to three criteria: the amount of needed hosts, migration data size and average response times of the applications. The underlying model size is 50, meaning, the algorithm is computing a placement for a system with 50 vCPSA. The x-axis denotes the generation number and the y-axis the results of the normalized quality for the objectives. For each of the criteria, the result range is being mapped to an interval between 0 and 1, where 0 represents the best solution and 1 the worst value. For example, the best achieved placement in respect of the number of needed hosts amounts to 26 and is mapped to 0 while the worst solution requires 33 and is mapped to 1. As can be seen, the improvement of the placement quality over the generations is significant. The initial solution requires 33 hosts with an average response times of 17.04 ms, while the best solution uses 26 hosts with an average response time of applications amounting to 12.04 milliseconds. Note that all of the solutions – regardless of their quality – are feasible placements, in other words, they are guaranteed to meet the timing requirements of the 50 virtualized CPS applications in the system.

4.4.3.2 *Comparing EA with ILP solutions*

This experiments compares the quality of the EA with the ILP results. In order to facilitate the computation, we evaluate a small system of

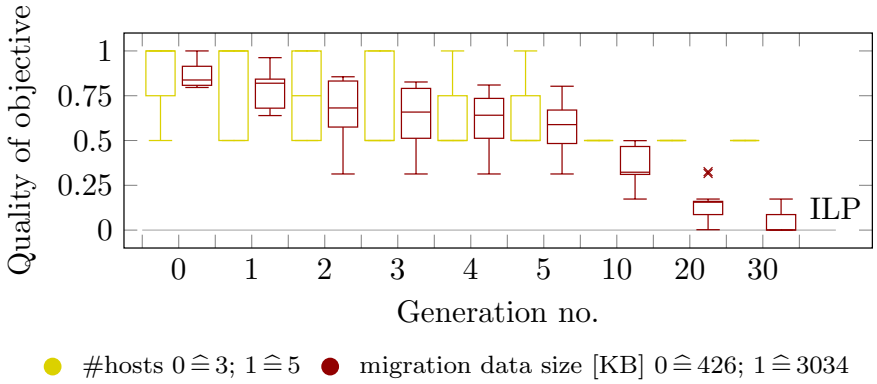


Figure 4.3: Quality of solutions for selected generations for a model of size 10 in comparison with ILP solution. Each box plot represents 15 independent runs on the same data.

size 10. The underlying ILP model abstracts from detailed network communication formulation, thus $B_i^{\text{in}} = B_i^{\text{out}} = H_i^{\text{regular}} = 0$. The two inspected optimization criteria are the number of used hosts and migration overhead. Again, each box plot depicts data from 15 independent runs on the same vCPS.

Figure 4.3 shows the quality of two objectives in comparison with the calculated ILP solutions for a model of size 10. As can be seen, the solutions computed by our algorithm in respect of the migration overhead can compete with the ILP results after 20 generations. However, none of the individuals encoding a placement reaches the ILP optimum in respect of the number of used hosts. The ILP solution requires 3 while the EA solutions need at least 4 hosts. We further investigated this aspect and rerun MAST models encoding the solution computed by the ILP. We found out that due to the more pessimistic nature of the algorithms employed by the holistic analysis, MAST assesses the optimal ILP solution as not schedulable and, as a consequence, the EA discards these individuals after their fitness has been evaluated. The overly pessimistic nature of assessment is, however, a known aspect of formal performance analysis and not inherent to MAST. Other techniques, e.g. RTC, exhibit similar behavior. This is the price that has to be paid for fast and exhaustive guarantees provided by these techniques. Nonetheless, note that Figure 4.3 only depicts selected individuals from different generations, which already encode optimized solutions. The first unoptimized yet feasible solutions computed by the algorithm in this scenario require 10 hosts, as the algorithm initially maps each of the vCPSA to a dedicated host. In other words, the EA optimizes the solutions from using 10 to using 4 hosts, while the ILP from using 10 to using 3 hosts.

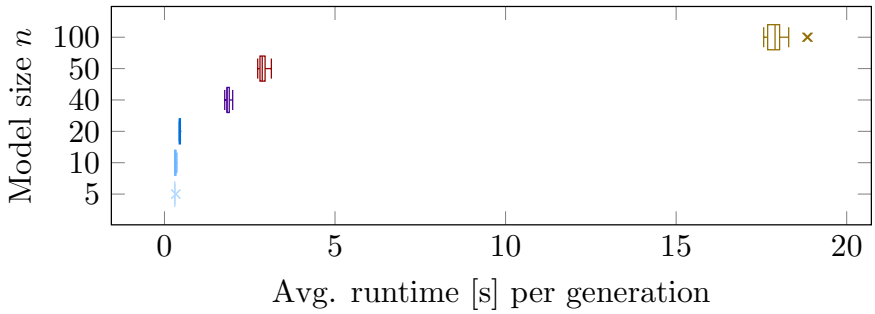


Figure 4.4: Runtime with different model sizes per generation.

4.4.4 *Time and Scalability*

In order to evaluate the runtime efficiency and scalability of our approach, we employed a machine with four Intel Xeon E5-4650 processors, each providing eight cores.

Figure 4.4 depicts the results of the runtime experiments. For smaller models (less than 20 vCPSA) the time to evaluate a candidate does not exceed one second. Realistic scenarios, consisting of ca. $50 - 60$ vCPSA, take about 5 seconds per generation to evaluate. Larger models sizes, exceeding 100 vCPSA, need considerably longer to compute. In our experiment, for a model of size 100, the computation time amounts to 18 seconds. These results indicate an exponential growth rate of the runtime function with increasing model size. Still, for many practical applications, the obtained runtime values are acceptable. Significantly larger models, counting hundreds of applications, would have to consider a relaxation of the approach, in order to remain computationally feasible. A possible solution is to partition the VMs beforehand and execute the algorithm separately on smaller clusters. Although, in this case the algorithm will never be able to reach an optimal solution, the quality loss of the solution should remain reasonably small. Finally, our inquiries have shown that the increase in evaluation time is mainly to be attributed to the MAST solver.

4.4.5 *Convergence*

The next experiment concerns with convergence times of our algorithm. The results are depicted in Figure 4.5. Please note that there is a semantic difference in the notion of convergence in respect of ILPs and EAs. In the former case, this notion denotes the approaching towards a single, n-dimensional point (optimum) in the solution space whereas the latter can be interpreted as a stabilization process of a population resulting – in our multiobjective case – in a multi-dimensional pareto front. In other words, in the case of the EA, the convergence translates

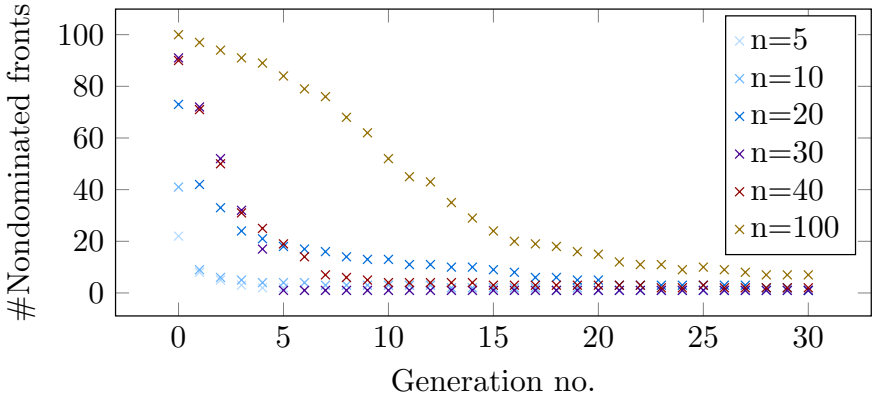


Figure 4.5: Number of nondominated fronts per generation for different model sizes (n). Each dot is representing the median of 15 different runs.

to the loss of genetic variety in the population and is therefore a good indicator for when to stop iterating.

As can be seen in Figure 4.5, for bigger models the convergence process takes longer, though in both cases, at generation 30, the number of nondominated fronts is 2, which means that there are only few individuals, which are not part of the pareto front, but form a second nondominated front.

4.4.6 *Ping-Pong Effect*

The "Ping-Pong" effect is a scheduling problem arising when newer scheduling decision revert previous ones and the schedule starts to oscillate between two or more similar states. There are scheduling algorithms, which counter this specific effect.

How vulnerable is our algorithm to this effect? As our algorithm computes a set of equal solutions under given constraints, the possibility can not be excluded that under an alternating set of similar constraints the proposed solutions could bounce between similar solutions. However, the criterion to reduce the number of migrations should result in at least a few solutions, which do not exhibit oscillation effects. Finally, it is up to the system designer or administrator to select or apply a proposed placement.

4.4.7 *Online or Offline Placement*

During discussions about scheduling algorithms often the question arises whether an approach can be considered as online or offline. In Section 4.4.4, our experiment shows that in the case of larger models (e.g. 100 VMs) the running time of the fitness function per generation

is about 18 seconds. If we additionally consider the convergence experiment (Section 4.4.5), which illustrates that for such sized models the stabilization process needs about 28 generation, then the total runtime of our algorithm takes about 8 minutes. Assuming we define online as "activated and deployed immediately after a triggering event occurred", then our approach should rather be considered as offline. However, if we define online as "at system runtime", then our placement algorithm can also be interpreted as online.

4.5 PRACTICAL ISSUES

The multiobjective approach of our algorithm allows for a variety of deductions from the computed solutions which can provide answers to numerous practical questions that arise when integrating or managing a virtualized CPS. Typical questions are:

- How to fulfill given requirements without an over-dimensioning of the necessary hardware?
- Can additional virtual machines be deployed in an already running system?
- How to efficiently plan maintenance?

All of these questions can be answered by examining the computed solutions for a specific model. In the following, we exemplarily examine a random model consisting of 50 VMs. Note that the necessary steps to interpret the results do not differ for other models.

In the following Figures 4.6, 4.7 and 4.9, every "x" mark represents a system configuration option in respect of two optimization criteria. For the purpose of visualization, we decided to use multiple 2-dimensional representations of the solution space, due to a rather unclear and chaotic graphical form the computed pareto front takes in a 3-dimensional representation. Now let's consider the following real world questions:

HOW TO FULFILL SYSTEM REQUIREMENTS WITHOUT OVER-DIMENSIONING THE NECESSARY HARDWARE? In order to reduce energy consumption, save procurement and maintenance costs or appropriately balance the quality-of-service against offered services prices, it would be desirable to have the possibility to chose from different system configurations representing trade-offs between the criteria of interest. Our approach provides the means to facilitate such an option. An adequate configuration can easily be chosen from the solution space computed by our algorithm.

Figure 4.6 shows 100 individuals encoding solutions which differ with respect to two optimization criteria, the amount of used hosts and the average response times of the VMs. The depicted individuals

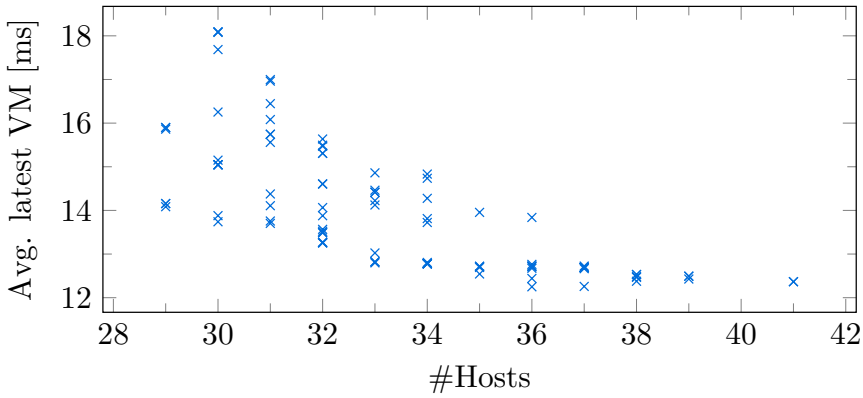


Figure 4.6: All 100 individuals of generation 30 for a model consisting of 50 VMs. The system designer can choose a placement balanced between number of used hosts and the average latest response time.

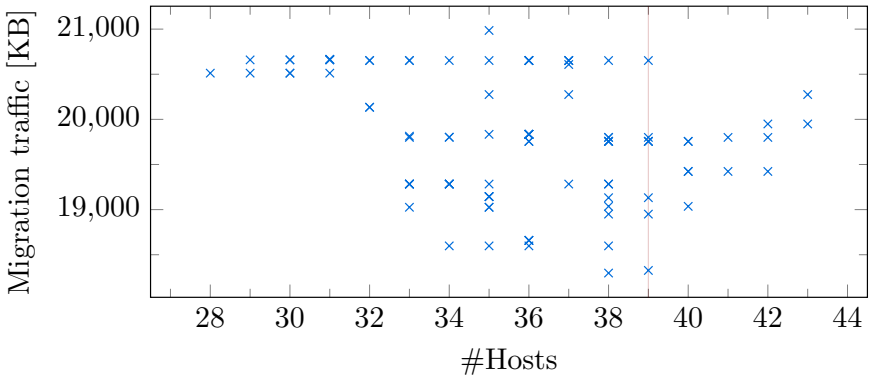


Figure 4.7: All 100 individuals of generation 30 for a model consisting of 50 VMs in which 2 VMs are new. The red line shows the number of hosts of the initial placement.

cover the entire solution space computed by the algorithm after 30 generations. All of the presented configuration are feasible solutions, meaning they do not exceed the computation load and bandwidth specifications as well as fulfill the given timing requirements. As can be seen, the system designer can choose from placements ranging from 27 to 39 hosts and exhibiting average latencies oscillating between 12 and 18 *ms*. Depending on the aim of the system, the designer can easily decide on the choice of the configuration. In practice, of course, only the solutions constituting the pareto front would be of interest.

CAN ADDITIONAL VIRTUAL MACHINES BE DEPLOYED IN AN ALREADY RUNNING SYSTEM? Our algorithm can cope with the issue of adding new workload to an already running system. Figure 4.7

shows 100 individuals after 30 iterations of our algorithm on a randomly generated model consisting of 50 VMs in which two new VMs have been added to the system. The red line marks the number of required hosts (39) for the original placement without the new VMs. The y-axis denotes the migration cost for state transitioning from the initial placement to a new system configuration. As before, all of the computed and depicted placements fulfill the given load constraints and timing requirements. Using these results, the system designer could choose an adequate placement considering the trade-off between the number of used hosts and the necessary migration traffic.

In the context of this question, another one of technical nature arises: In such cases, does it yield benefits to start the search for a new placement from the current setup? In other words, is it useful to search locally when only small changes have been made to the optimization problem? In our approach a local search translates to starting the EA with a population based on the previous solution and not from a randomly generated one. In order to investigate this issue, we conducted another experiment where for both initiation types we compare the number of nondominated fronts as well as the quality of the computed solutions.

Figure 4.8 depicts the number of nondominated fronts for two runs of the EA with different initialization. The blue solutions contain randomly initialized individuals and the red solutions consists to 50% of individuals that tightly reflect the previously deployed placement. As can be seen, the convergence process is very similar. This means that using individuals encoding precious placements, does not necessarily yields benefits in respect of the runtime of the algorithm.

We additionally compared the quality of the computed solutions. For this purpose, we calculated the width of the two pareto fronts with the standard euclidean distance metric $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$. The random initial population generated a pareto front with a width of 3616, whereas the biased initial population generated a pareto front with a width of 3035. This indicates that a randomly initialized population yields a more diverse front when compared to the biased one. We also compared the standard deviation values for each criteria separately and those are also slightly higher for runs with a randomly initialized population.

Since probably no runtime savings can be expected, but the width of the pareto front and also the standard deviation of the single criteria is slightly higher with a random initialization, and we can not put forward any empirical arguments for the biased approach, it seems that in the case when only small adjustments have to be made to the system the randomized approach is of advantage.

HOW TO EFFICIENTLY PLAN MAINTENANCE? Consider a scenario with a given placement and the necessity to power down multiple

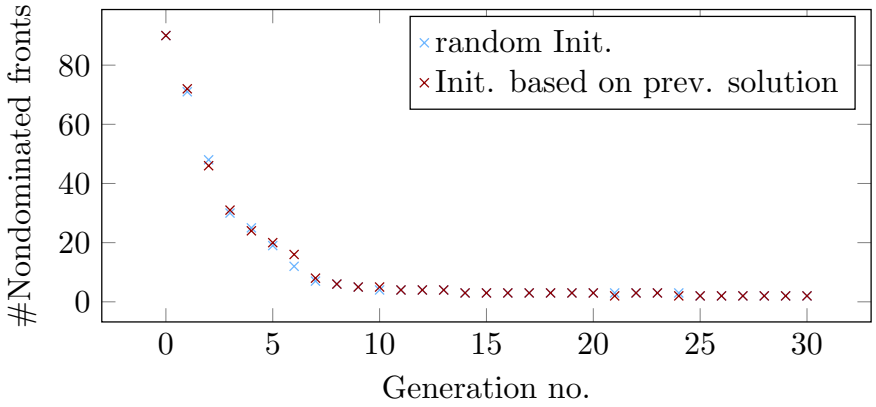


Figure 4.8: Number of nondominated fronts per generation for different initializations.

servers for the purpose of maintenance. If the already running VMs are constrained by certain quality-of-service conditions (e.g. minimal service downtime), then a reconfiguration of the system has to happen at runtime and therefore has to rely on migration. The alternative would be to shut down all of the affected VMs and restart them at their new locations. However, in this case, the service downtimes would be significant. Our approach can compute a wide spectrum of alternative placements taking into account the number of hosts that need to be powered down subject to the necessary migration network traffic. For a synthetic scenario, we compute alternative placements for a model consisting of 50 VMs. It uses 39 hosts in its initial form. The results after 30 iterations can be seen in Figure 4.9. The illustration also contains placements which use even more hosts than the initial solution and are thus marked red. From these results, a system designer could now choose an adequate placements and decide on how many hosts he wants to free with respect to quality-of-service constraints. Again, regardless of the choice, all real-time requirements for the services are guaranteed.

4.6 CHAPTER SUMMARY

In this chapter, we introduced a methodology for planning safe and efficient virtualized CPS compute and control clusters. We described how to enable an optimized integration of CPS by means of virtualization and discussed the used methods, the corresponding models and the assumed system architecture of the execution platform. We showed that the presented methodology allows not only to optimally dimension the virtualized CPS, but also provides strict guarantee regarding the timing predictability of the integrated CPS applications. The approach yields system configuration options representing trade-

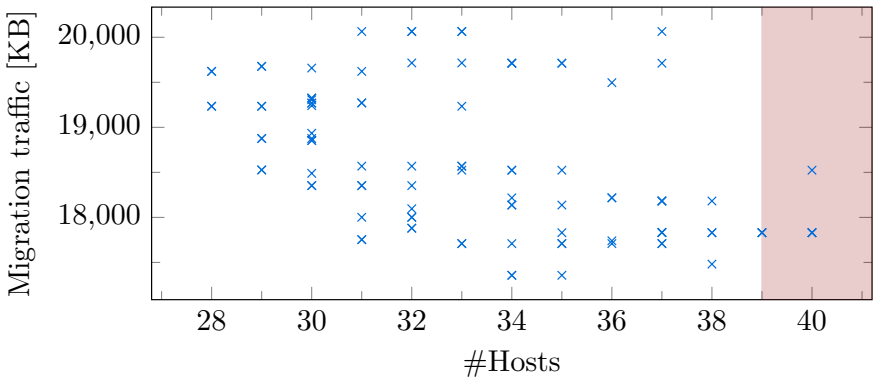


Figure 4.9: New placements for a model consisting of 50 VMs. The red area shows placements, which require more hosts than the initial placement.

offs between different non-functional requirements of which all are – in a mathematical sense – guaranteed to be fulfilled. From those, a system designer can safely choose the one that best meets his individual needs. Moreover, as our approach combines evolutionary algorithms with a mature formal performance analysis technique (MAST), the modeling process is – compared e.g. to ILP formulations – relatively easy and flexible and therefore accessible to a wide range of system designers. Finally, the presented methodology also provides answers to several practical questions that arise when integrating CPS by means of virtualization.

5 TESTING OF VIRTUALIZED CPS

The previous chapters discussed the technological aspects related to the development of virtual execution environments for CPS, in particular, the efficiency issues related to the system software of the underlying execution and communication infrastructure and the thereon based high availability solution, as well as a methodology for the planing of safe and efficient virtualized CPS. Those chapters exhibit a strong focus on the non-functional aspects related to the construction of virtualized CPS. The following chapter shifts the focus towards functional aspects, as it addresses the issue of testing virtualized CPS applications. As it was the case in the previous chapters, the background for our research is situated within the domain of cyber-physical energy systems. In particular, we describe two architectures for testing applications from the smart grid domain.

The first part describes a HiL co-simulation architecture, which aims at the validation and verification of virtualized CPS applications while taking into account the complex dependencies between the power system and the communicational and computational aspects of the infrastructure. The second part presents a testbed which aims at an empirical evaluation of the applications. In contrast to the first approach, it allows to induce realistic load levels on the infrastructure, and therefore provides the means for testing both the functional and non-functional characteristics of virtualized CPS applications as well as of the underlying ICT. It has, however, a smaller scope of application than the HiL co-simulation architecture. The findings presented in the following sections have been published in [64, 66].

5.1 HARDWARE-IN-THE-LOOP CO-SIMULATION ARCHITECTURE

The ongoing integration of renewable energy sources into the electrical transmission systems coupled with the cross-border liberalization of European electricity markets is causing a higher degree of volatility in power grids. To ensure a safe, secure and efficient grid operation, new concepts utilizing the advances in computing and communication technologies are being developed. Concepts of electricity transmission and distribution systems, in which digital information flow, use and management play a central role, are being referred to as *smart grids*.

In the field of modern power systems, the literature proposes a wide range of smart grid concepts, which utilize power system assets in combination with modern ICT infrastructure as well as different testbed architectures for the development and testing of suitable appli-

cations [22, 41, 71]. Consequently, besides power system aspects every process of development and testing of smart grid applications should take into account the link to the underlying ICT infrastructure. As an empirical evaluation is often not feasible in practice, most solutions rely on simulation tools. The main challenge in these approaches is the facilitation of a combined analysis of the power system and ICT simulation.

Fortunately, a solution to this problem is being provided by the *Hybrid Simulator Architecture* described in [44, 103]. It is a modular co-simulation environment for comprehensive analysis of mutual effects, taking into account communication networks, IT processing and power system response while investigating smart grid applications. However, the IT processing model it provides is very simplistic. The architecture abstracts from the characteristics of the execution platform and reduces the complexity of computation to run time estimates of the analyzed application. We argue that such an approach is not sufficient with respect of testing virtualized CPS applications, inter alia, due to the complexity of the virtualized execution platform. We aim at a more comprehensive approach that enables a thorough testing and analysis of virtualized smart grid applications as well as the virtualization hardware and software infrastructure. Therefore, in the following, we extend the *Hybrid Simulator Architecture* in order to facilitate the integration of our CPS-Xen execution platform. To the best of our knowledge, there exists no other work that describes how to develop and test virtualized power system applications while taking into account the mutual dependencies between the power system and the ICT. In detail, we make the following contribution to the research on virtualized cyber-physical energy systems [64]:

- We introduce a *HiL co-Simulation Architecture* for the development and testing of virtualized CPS applications by integrating the virtualized execution platform into the co-simulation of the power and ICT systems.
- We provide validation (simulation-based correctness analysis) for virtualized software functions in joint interaction with the power system, the computation and communication infrastructure.
- We show how to obtain detailed latencies of the virtualized system. These delays comprise of the measured computation times for the given algorithm running in the VM as well as of the delays caused by the packet processing on the virtualized platform.
- The obtained computation-related latencies allow for a precise parametrization of simulation models regarding information technology delays.

In the following, we briefly describe the IEEE Standard 1516 - *High Level Architecture* (HLA) [59] for distributed simulation as it forms the basis for the introduced HiL co-Simulation Architecture. For a comprehensive introduction that outlines the HLA-Standard in a practical way, please refer to [76].

5.1.1 *IEEE1516 HLA Standard*

The HLA IEEE 1516 Standard for distributed simulation defines an architecture comprising of the following three main components:

- A central administration instance, the *Runtime Infrastructure* (RTI) which provides a set of services for synchronization, information exchange and management
- HLA compliant simulation systems called *Federates*. Note that in our case, we also define the connected hardware-in-the-loop infrastructure as a federate.
- A model for describing shared objects, attributes and interactions called *Object Model Template* (OMT).

The overall co-simulation, composed of the three components, is called a *Federation*. In turn, a running co-simulation is referred to as a *Federation Execution*. The federates are connected to the RTI during federation execution. The RTI hosts and manages the federation execution and offers services to the federates, in order to synchronize their logical time and facilitate the information exchange. To use these services, every federate has to implement the corresponding interface. In addition, the HLA IEEE 1516 Standard defines features for the connection of federates to the RTI via Ethernet through the TCP/IP and UDP communication protocols. This enables HLA IEEE 1516 Standard based simulations to be distributed and run in parallel. The most important parts of the HLA IEEE 1516 Standard are:

- IEEE 1516: Framework and Rules - Fundamentals;
- IEEE 1516.1: Interface specification - Interface specification for federates and RTI services;
- IEEE 1516.2: Object Model Template specification - Specification of the structure for a valid *Federation Object Model* (FOM).

From the beginning of a federation execution till it's completion, the participating federates perform service calls on the RTI, which in turn uses the interfaces of the federates to perform callbacks. To exchange information every federate publishes and subscribes to the so called objects and interactions defined in the FOM. Non-persistent information - like events - is exchanged by updating and reflecting interactions, as well as their parameters. Persistent information - like

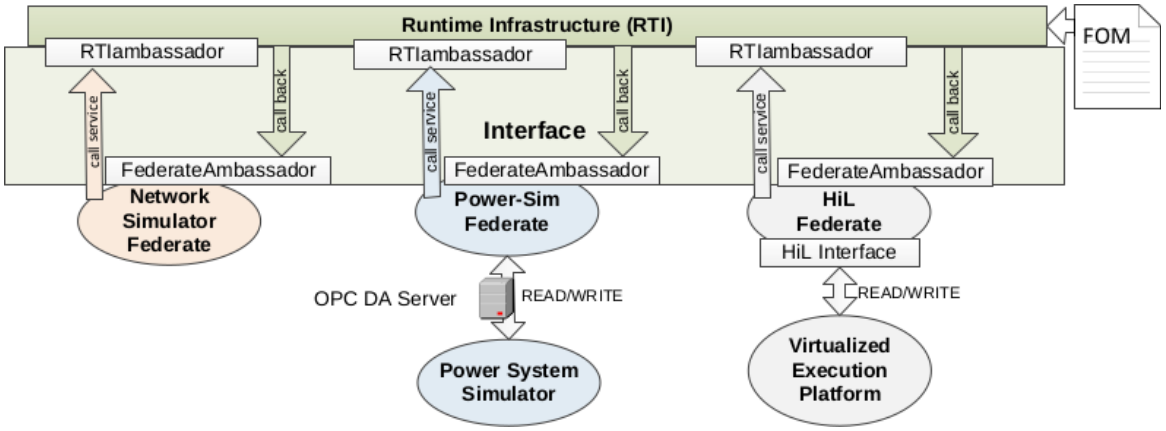


Figure 5.1: Hardware-in-the-loop co-Simulation Architecture comprising the three federates: the power system and the communication network simulators as well as the execution platform.

related data - is exchanged by updating and reflecting objects and their attributes.

5.1.1.1 *Types of Federates*

We divide federates in two classes. They differ in the connection mode to the RTI. The first class is called *common federate*. Common federates have integrated HLA components which allow for information exchange with other federates directly via the RTI. In most cases, simulators do not provide such components, therefore they have to publish and receive information through an additional instance that contains the appropriate HLA components. Federates utilizing such a component are referred to as *delegated federates* and constitute the second class.

5.1.2 *The HiL co-Simulation Architecture*

This section describes the *HiL co-Simulation Architecture*, which integrates our virtual execution platform into the HLA-based co-simulation of the power and ICT systems. It is being depicted in Figure 5.1.

The architecture (federation) comprises the following three federates:

- Network Simulator Federate - for the communication network simulation.
- Power-Sim Federate - for the simulation of the power system.
- HiL Federate - the interface to the execution platform for the virtualized CPS applications.

The power system simulator is connected to the *Power-Sim Federate* via a *OLE for Process Control Data Access* (OPC DA) server connection.

The OPC is a widely spread performant industrial communication standard and we use it to facilitate data exchange of measurement and control values between the power system simulator and the *Power-Sim Federate*. The synchronization of the logical time for the power system simulator, the *Power-Sim Federate* and the federation is realized in a conservative approach. This means that only one federate is advancing it's logical time at all times. The *Power-Sim Federate* translates the measurement values into the corresponding object attribute values. After the transformation, the updated object attribute values are published via RTI to the subscribed *HiL-Federate*. Only the object attributes and the interactions defined in the FOM are allowed to publish and subscribe. The FOM itself is being loaded by the RTI during the creation of a federation execution. All federates use services implemented in the *RTIambassador* whereas the interfaces for the service callbacks are located in the *FederateAmbassador*. Figure 5.1 also illustrates the structural difference between the delegated and the common federates. The network simulator includes the implementation of the HLA components in the *FederateAmbassador* interface which makes the *Network Simulator Federate* an example for a common federate. In contrast, the *Power-Sim Federate* and the *HiL-Federate* are delegated federates. Therefore, neither the power system simulator nor the virtualized execution platform includes HLA components. The updated object attributes are committed from the *HiL-Federate* to the virtualized execution platform through the *HiL-Interface*. The virtualized execution platform sends control commands back to the *HiL-Federate* which delivers these control commands via RTI, the *Power-Sim Federate* and the OPC DA server to the power system simulator.

5.1.3 *HiL-Interface*

The *HiL-Interface* creates the simulation entry point for the virtualized execution platform. Through this entity the virtualized CPS applications issue their control commands to the simulated primary power system equipment. Further, the *HiL-Interface* is responsible for the protocol translation. Depending on the communication direction, the interface transforms either the power simulation data into UDP protocol stream or the UDP stream into the format of the power system data. The *HiL-Interface* is also responsible for the synchronization of the data exchange between the co-simulation and the virtualized execution platform. This is guaranteed by forcing the *Hil-Federate's FederateAmbassador* to pause execution during the time when the control application is running on the virtualized execution platform. The *HiL-Federate* and the virtualized execution platform exchange data in a conservative approach.

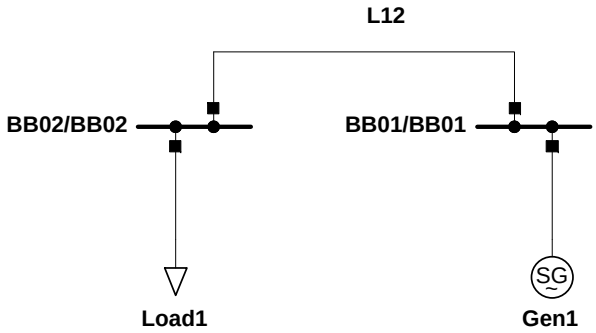


Figure 5.2: Power system test network

5.1.4 *Proof of Concept*

This sections provides a proof of concept for our approach by the means of a step-by-step analysis of a message flow of our *HiL-co-simulation Architecture*. The chart describes the testing process of a virtualized overcurrent protection application. For this purpose we investigate a simplistic scenario.

SCENARIO The scenario comprises of an elementary power and communication system. The test network for the power system is illustrated in Figure 5.2. It depicts the generator $Gen1$ with a constant feed-in, two substations $BB01$ and $BB02$, and the transmission line $L12$ to interconnect the load and the generator. The power line $L12$ is being monitored by a measurement device situated in the substation $BB01$. The state determining values - in form of a current phasor - are being continuously transmitted to the protection application over the substation LAN. The protection function, executed on the virtualized platform and located within the substation $BB01$, analyses these values. In case of an overcurrent or a short circuit, the protection algorithm dispatches a control command to the circuit breaker $CB4$ (situated in substation $BB01$), in order to interrupt the current flow. The test network is modeled and simulated in the power system simulator PowerFactory [33]. The underlying communication system assumes a simplistic topology and is modeled in the *Network Simulator Federate*. The overall *HiL co-Simulation Architecture* was implemented in JAVA and the protection algorithm, running within the VM, in the C programming language.

In the evaluated scenario the sequence of events is as follows: At the beginning of the scenario the current on the line $L12$ is stable and the switch is set to open. $100ms$ after the start of the simulation a short circuit occurs on the monitored transmission line $L12$ between the generator and the connected load, causing a dynamic increase in the current functions. The overcurrent is detected by the measurement device at $110ms$ and the corresponding current phasor values are being send to the VM executing the protection algorithm. This information

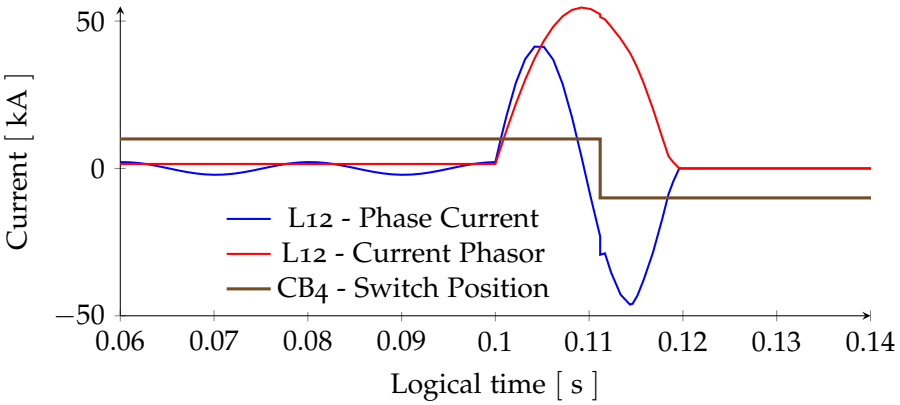


Figure 5.3: 3-phase short circuited line L12

arrives at $110,5ms$. The protection function analyses the corrupt state of the transmission line and sends a control command to the circuit breaker $CB4$ after $0,072ms$. The computation time of the algorithm is $0,001ms$, the remaining $0,071ms$ account for packet processing on the virtualized execution platform. The control command arrives at the switch at $111,072ms$ triggering the operation to interrupt the current flow. At the time-stamp $120ms$, after about another $9ms$ - the time elapsed due to the process level equipment - the current flow on the transmission line $L12$ amounts to 0.

The results for our scenario are depicted in Figure 5.3. The blue line represents the phase current, the red line shows the value of the current phasor and the brown line depicts the position of the circuit breaker $CB4$. All values are presented in relation to logical time.

5.1.5 *Simulation Process and Results*

This section details the simulation process by analyzing a message flow chart from our scenario. The chart is given in Figure 5.4 and represents a sequence of events that occur in our simulation between the time a state change of the power system is released and the execution of the related response control command. The message flow chart can be elucidated as follows: At time-stamp t_0, after simulating the recent processes in the test network, the power system simulator releases its new state values to the HiL-Federate. The HiL-Federate initiates the communication between the measurement device and the virtualized execution platform. This causes the RTI to invoke the network simulator in order to compute the corresponding communication delay t_1. Next, the RTI schedules the HiL-Federate, which translates the simulation protocol into the UDP protocol and forwards the data to the protection algorithm. The response time of the virtualized execution platform - given in t_2 - is composed of two values. The first represents the actual computation delay of the protection algorithm

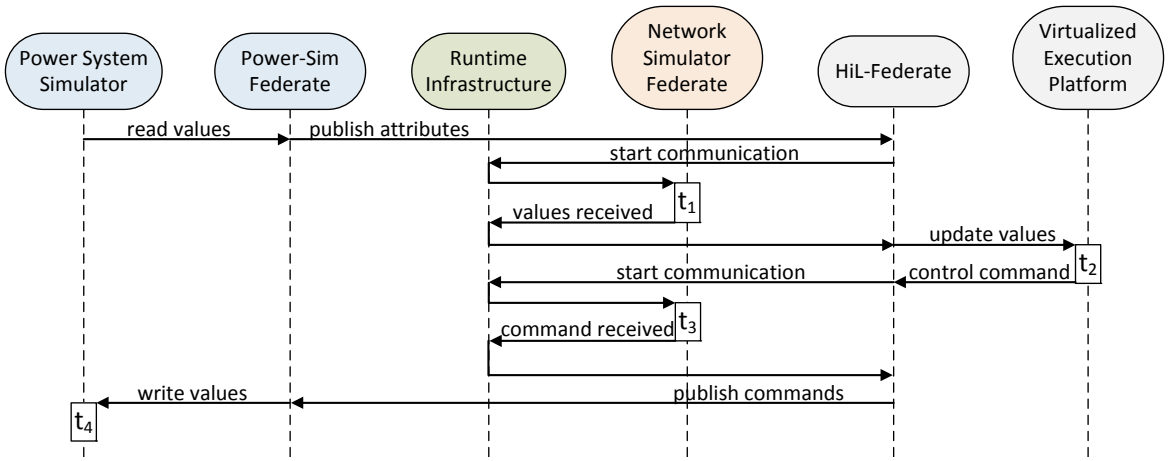


Figure 5.4: Message flow chart

and the second one quantifies the latencies occurred due to packet processing on the virtualization platform. In contrast to the other delays, the t_2 value is not the outcome of a simulation, instead it is the *actual time* that passed while computing the algorithm. In the case of t_2, the logical time and the wall-clock time are one. After releasing the control command by the execution platform, the HiL-Federate translates it from UDP back to the simulation protocol and again initiates a communication. However, this time from the host on which the protection VM is running to the desired equipment device. This transmission latency is given by t_3. Next, a new event for the power system simulator has to be scheduled to execute the corresponding operations. The time elapsed due to the operation of the process-level equipment is denoted by t_4. The sum of the delays t_1 to t_4 represents the logical time for the entire process, from the release of new power system state values to the execution of the control operation. It includes the latencies from the information and communication infrastructure and the delays induced by the operation of the primary equipment of the power system.

The obtained results show that our *HiL-co-Simulation Architecture* is working as intended. It correctly handles the different type of simulators and successfully interfaces the virtualized execution platform with its encapsulated CPES applications. To achieve this behavior, the architecture synchronizes all of its members and appropriately translates the corresponding data traffic. The obtained results confirm that our approach is able to analyze virtualized CPS applications with respect to their functional correctness as well as their interdependencies with the power system and the communication network. Further, the detailed response time analysis, including the execution time of the algorithms as well as the latencies induced by the packet processing of the virtualized execution platform, facilitates the preliminary

performance evaluation of both the virtualization hardware and the virtualized CPES software.

5.1.6 *Discussion*

In order to ensure optimal performance and a meaningful execution time analysis, our test CPS application was implemented in the C programming language. Therefore, to analyze the monitor, protection or control algorithm, which are in practice often implemented or analyzed in JAVA, GNU R or MATLAB, in order to be able to analyze them on the *HiL co-Simulation Architecture*, they would have to be rewritten in C or C++.

Another issue relates to the preliminary performance evaluation of the virtualization hardware. As shown in Section 5.1.4, our architecture is able to evaluate – to the extend provided by simulation – the functional correctness and the exact execution delays of the CPES application under study. However, due to the co-simulation timing constraints - given by the individual simulators - a realistic workload level can not be induced on the virtualization hardware. Therefore, the analysis of the packet processing times should be treated with caution, as they represent timing values for an architecture with sufficient free resources. A solution to this approach would be to interface the virtualized execution platform with real-time simulators. Then, a thorough analysis of the hardware related characteristics would be possible. A testbed architecture which enables such an analysis will be discussed in the next section.

Finally, although the presented scenario was concerned with substation level equipment, the *HiL-co-Simulation* approach can also analyze scenarios related to control centers and wide area applications.

5.2 VGRIDLAB – A TESTBED FOR VIRTUALIZED SMART GRIDS

One of the shortcomings of the HiL co-simulation architecture, introduced in the previous chapter, is its inability to induce realistic workloads on the tested hardware, in particular, the integrated execution platform. In this section, we discuss a testbed architecture for the development, deployment and validation of virtualized smart grid functions, which fixes this deficiency and allows for an evaluation of the underlying ICT infrastructure and devices used in power systems automation under realistic load situations. In detail, the novelty of this architecture is given by the following characteristics:

- It represents a consistently virtualized ICT infrastructure. The hardware resources of both the communication network and the execution platform are being virtualized.

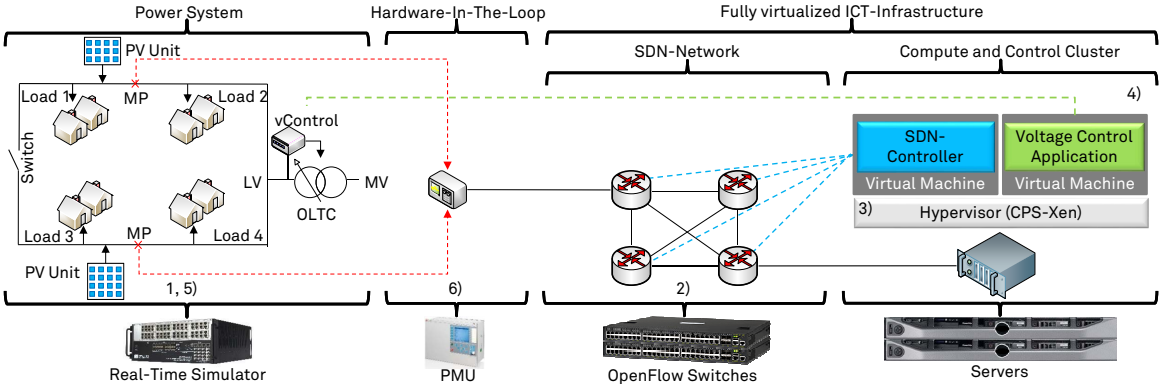


Figure 5.5: Overview of the vGridLab architecture, comprising power system simulation and hardware components for measurement, communication and computing as well as measuring points for deriving application and platform characteristics.

- The virtualized applications can be evaluated in combination with real devices for power system automation.
- Due to the interfaced real-time power system simulator, both the application as well as the infrastructure can be evaluated in real-time. The use of a real-time simulator also enables the inducement of realistic workloads on the infrastructure or even the possibility to conduct stress tests of the underlying hardware.

Figure 5.5 depicts the implementation of our virtualized smart grid testbed architecture that is adopted for the needs of a case study that involves a virtualized smart grid application for the controlling of an *On Load Tap Changer* (OLTC). The architecture and its description is based on [65].

The testbed architecture consists of three components: the energy system simulator, the HiL electrical devices and the virtualized ICT infrastructure. These components can be flexibly interconnected and adjusted in order to fit the requirements of the experiment under study.

The power system is being represented by the real-time simulator OP5600. In the given example, the RTS simulates a low-voltage distribution network connected via an OLTC to medium voltage network. The OLTC is operated by a virtualized control (vControl) being executed as a virtual machine in the compute and control cluster. The low-voltage network consists of two feeders including two loads and a photovoltaic unit operated as open loop. In the discussed experiment, the HiL component of the architecture comprises one *phasor measurement unit* (PMU) of type ABB RES 670. The PMU is used to distribute timestamped voltage measurements in compliance with IEEE C37.118.1 that are to be processed by the vControl. For this purpose, the analogue outputs of the OP5600 simulator are connected to the PMU.

The ICT system consists of a communication network and a compute and control cluster. *Software-defined networking* (SDN) is applied to enable dynamic and flexible configuration of the communication infrastructure. Therefore, the control logic is concentrated at the so-called SDN controller, which determines network behavior via the OpenFlow protocol based on its global view. In total, five Pica8 3290 bare metal switches and four software switches, running Open vSwitch on standard computing hardware, can be incorporated into the testbed. The compute and control cluster is responsible for hosting the virtualized smart grid application as well as the SDN controllers. It is running CPS-Xen [69] (for details on CPS-Xen see Section 3.3).

vGridLab provides multiple measuring points and interfaces for obtaining application and system characteristics. In the following, a brief overview of those is given. The numbers in brackets correspond to the numbers depicted in Figure 5.5.

The functional correctness of the virtualized algorithms can be verified by analyzing state-variables in the power system simulation (1). The OpenFlow switches (2) are adapted to allow for the measuring of communication and switching latencies as well as to quantify overheads generated by high-availability solutions (e.g. failover latencies in case of link failures). In a privileged domain of the virtualized execution platform (3) system response times (comprising the latencies constituted by the entire platform software stack, including TCP/IP-stacks latencies and the application execution times) can be measured. Further, by using clock cycle precise measurements techniques – embedded in guest VMs (4) – the testbed allows to obtain highly accurate execution delays for the analyzed smart grid algorithms. The real-time simulator (5) allows to quantify round-trips times, which are defined as the time interval between the sending of a changed variable state and the corresponding control decision of the algorithm. Finally, through the analysis of the logged system data, characteristics of the HiL-device (6) can be deduced.

Overall, the presented architecture allows for a detail analysis and evaluation of both the function and non-functional characteristics of the virtualized smart grid applications as well as the underlying ICT infrastructure and real electrical devices used in power system automation. It has standardized and well-defined interfaces enabling for loose coupling of different ICT and power system hardware. This facilitates the adaptation of our architecture to a variety of different scenarios. Finally, due to the real-time simulator, realistic workloads can be induced on the to be evaluated systems, allowing for the implementation of stress test scenarios, which are of high interest to the power system automation community.

5.3 CHAPTER SUMMARY

This chapter introduced two architectures for developing, verification and validation of virtualized CPS applications from the smart grid domain. Both architectures aim at taking into account the interwoven dependencies between the applications, the power system and the underlying ICT as well as the possibility to enable the testing of involved hardware components, including power system and ICT equipment. However, the two architectures differ in their scope of application. While the *HiL-co-Simulation Architecture* enables the analysis of large scale scenarios involving wide area applications, it exhibits shortcomings with respect to run times and the possibility to induce realistic workloads on the ICT hardware. In turn, the *vGridLab* testbed architecture allows for a real-time evaluation of the applications and the underlying hardware components – including stress test scenarios, yet it has only a limited scope that includes low-scale distribution networks or individual substations. However, the architectures are not mutually exclusive and could be combined in order to assess larger networks or wide area applications in all details.

6 DISCUSSION AND OUTLOOK

The previous chapters presented and discussed in detail technological and methodological approaches for the planing and construction of safe and efficient virtualized cyber-physical systems. This chapter extends and generalizes the discussion. Based upon the expertise from previous findings as well as some new considerations, it gives an outlook towards a generic execution platform architecture with virtualization support for emerging CPS applications. Before that however, it provides a discussion regarding the conformity of virtualized environments with standards of functional safety.

6.1 SAFETY CERTIFICATION

Although we did not conduct any research on the topic of safety certification in CPS, in the following, we want to briefly address this topic as it constitutes an important aspect of constructing safe and secure virtualized CPS. Achieving conformity with standards of functional safety has to be regarded as an additional requirement.

Most cyber-physical systems have strict functional safety certification requirements that have to be met before a system can be deployed. Depending on the deployment domain, different standards have to be considered. Examples are: the IEC 61850 – for power system automation (briefly discussed in Chapter 3, Section 3.1), IEC 61508 [57] – functional safety standard covering a broad spectrum of industry applications, ISO 26262 [61] – function safety for road vehicles, or DO-178C [118]– for software-based aerospace systems, as well as many others. Besides safety standards, embedded software products often also have to comply with software development guidelines. A prominent example is the MISRA C [54] set of rules for the C programming language aiming at facilitating safety, reliability and portability of the implemented code.

In general, the certification process of software is a time and money consuming process. As virtualization was originally neither designed with real-time safety requirements in mind nor for a specific embedded domain, most of the fully-fledged hypervisors – of which some are open source projects – do not provide any level of safety certification. There exist products that emerged from solutions for real-time operating systems or hardware partitioning approaches and possess various levels of certifications for different domains. Along the way, some added virtualization support to their portfolio. Prominent examples are SYSGO PikeOS [17, 73], Green Hills Integrity Multivisor or

Windriver. In respect of Xen, the community around the hypervisor is, as of 2019, making efforts to certify the hypervisor code for the ISO 26262 ASIL-B standard level.

The certification of hypervisors is an important and expensive aspect of adopting this technology to the domain of CPS. It should, however, not only be perceived as a challenge, as it can also be an asset. An already certified hypervisor can act as a safe and secure basis for strong isolation of mixed-criticality applications and thus helping manufacturers to certify specific system components separately in their own execution environments. Furthermore, due to the isolation property, the execution of environments with different certification levels is possible.

6.2 GENERIC ARCHITECTURE FOR EMERGING CPS

The last few years were characterized by a rapid development of the virtualization technology ecosystem. Novel hardware features facilitating virtualization technology emerged, COTS hardware products for embedded systems introduced or extended their support for virtualization, new hypervisors aiming at embedded systems were developed while prominent solutions added embedded and cyber-physical systems domains to their scope of application, and most important, also the expectations regarding the functionality of CPS applications evolved and with them their requirements.

Although we are not able to address all the new developments in this thesis, we wish to summarize the requirements of future CPS applications and propose extensions to our architecture that allow to meet those new expectations. Note that the extensions do not invalidate our results in any way. They just add new functionalities. All previously discussed findings hold.

In the presentation of the generic architecture, we abstract from concrete solutions or implementations and concentrate on discussing high level requirements that – in our opinion – should be fulfilled by platforms aiming for hosting future CPS applications.

However, before presenting the architecture, we first summarize the requirements of future CPS applications. Due to its generic character, the following enumeration lists requirements that correspond to the combined needs of different CPS domains. Solutions from the field of logistic will have a different subset of requirements with respect to the execution platform than automotive systems or systems aiming at controlling and managing electric power distribution grids. Obviously, the architecture should support the implementation of any desired subset.

6.2.1 *Requirements of Future CPS*

The generic architecture for virtualized CPS is designed to comply with the following requirements:

HARDWARE EXTENSIONS The architecture should support all available hardware-assisted virtualization capabilities, in order to facilitate virtualization as well as increase performance and security. Today, those include not only CPU and memory features, but cover the entire system architecture. Among them are interrupt, GPU, I/O and network virtualization. Note that x86 and ARM architectures support an equivalent spectrum of features.

COMPONENTIZATION At all levels the architecture should remain as modular as possible. This is important because componentization is a prerequisite for the construction of efficient and secure execution platforms. The possibility to tailor the infrastructure software translates to smaller footprints – and thus reduced attack surfaces, faster boot and execution times, higher certification-friendliness, eases the implementation and compliance with security policies, allows for the construction of driver domains and finally for an adequate isolation of workloads with different criticality levels. Therefore, the architecture should support static hardware partitioning for safety and security critical workloads as well as the possibility to concurrently host multiple and different execution environments to facilitate the development and deployment of mixed-criticality systems. The support for execution environments should include native execution, general-purpose OS's, unikernels as well as application and system containers.

RESOURCE MANAGEMENT The architecture should support options for flexible allocation, sharing and partitioning of resources. The implementation of these capabilities has to be based on available hardware extensions as well as concepts and technologies like paravirtualization, driver domains and pass-through.

Starting with the CPU, the execution platform should provide a set of best-effort, soft- and real-time schedulers as well as – for highly timing sensitive applications – the possibility to dedicate physical cores to a domain without the need to employ a scheduler. Also the grouping of CPUs into pools should be supported with the option for assigning schedulers on a per CPU pool basis.

When it comes to memory, besides the static on per VM basis allocation, the architecture should provide ways for dynamic memory reconfiguration as well as for the construction of shared memory regions. Ideally, also NUMA-awareness and cache allocation techniques should be available.

Regarding IO-processing and communication, a wide range of internal and external buses should be supported, including bandwidth allocation and balancing options. IO-processing is to be synergized with corresponding VMM-schedulers.

Due to the increase in artificial intelligence and pattern recognition use cases, the execution platform should provide support for GPU based machine and deep learning techniques. Another GPU related requirement is the possibility for sharing GPU capabilities and displays between different execution environments at run time.

In order to fulfill multi-media requirements, besides GPU virtualization, it should also be possible to share audio capabilities between different execution environments with routing support for various source and destination devices.

Finally, also workload, energy and software management as well as maintenance features should be available on the platform. For these, the architecture has to implement VM migration, support for advanced power management – including policies on per execution environment basis – as well as remote install, update and diagnostic capabilities for the entire infrastructure stack.

FAULT-TOLERANCE PROPERTIES The architecture should support a broad spectrum of configurable fault-tolerance methods. Beside different isolation levels for guaranteeing time and space fault containment, the execution platform should assume additional fault-tolerance techniques. A crucial aspect lies in the possibility to provide various forms of redundancy. This allows to significantly increasing system reliability.

In order to protect applications from hardware failures, redundancy can be delivered by implementing different standby modes utilizing rebooting and microrebooting, replication as well as parallel execution techniques. Our CPS-Remus solution for high availability with real-time failover and recovery is an example of such techniques. In order to protect against software faults, the technique of N-version programming can be adopted to provide build-in redundancy. Furthermore, system monitors for conducting sanity checks as well as watching over and verifying system functions should also be assumed.

TIMING PROPERTIES In order to facilitate planing and verification, the architecture should exhibit deterministic system latencies and the processing times have to correspond with the timing requirements encountered in the given application or domain of interest. In order to achieve this, the implementation of the architecture has to fulfill the requirements mentioned above in the resource management paragraph.

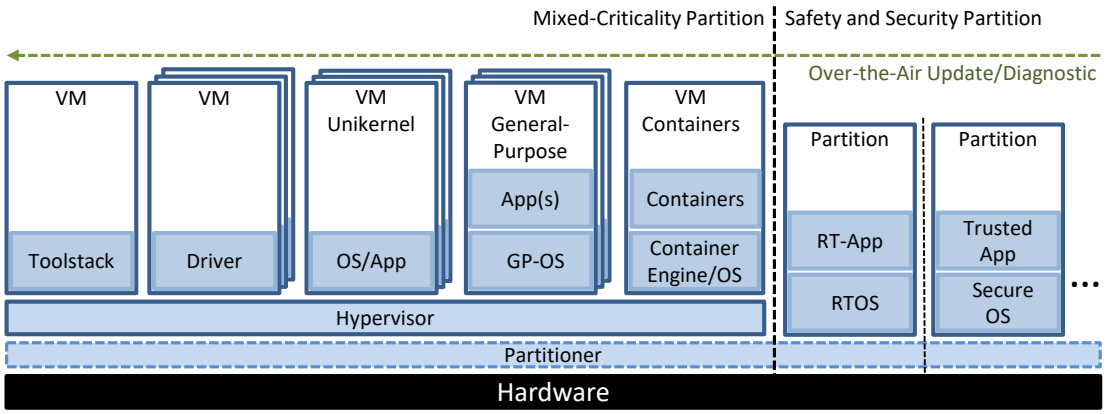


Figure 6.1: A generic execution platform architecture for emerging CPS applications.

CERTIFICATION AND SECURITY The implementation of the architecture should aim at minimizing the trusted computing base. A direct consequence of such an approach is the increase in system security. Moreover, it facilitates code management and certification. The latter is of special importance, as the execution platform has to comply with safety standards from the given domains of interest. A proven paradigm for implementing minimal trusted computing bases is the microkernel design. In fact, many of the available hypervisors are microkernels.

The platform should also provide a possibility to define fine-grained access policies for inter-domain communication, drivers and hypercalls. Finally, it has to support hardware security technologies like *Trusted Platform Module*, *Intel Trusted Execution Technology* or *ARM TrustZone*.

6.2.2 *Generic Architecture*

Most of the above discussed requirements are already being fulfilled by our CPS-Xen architecture. However, in order to cope with the additional ones, the generic architecture generalizes and extends our CPS-Xen architecture (compare Figure 3.2) by, inter alia, introducing and merging additional execution environments into its stack. The different types of execution environments have been discussed in Section 2.1.3. In the following, we will discuss selected extensions to the presented architecture, which is depicted in Figure 6.1.

Some workloads impose particularity strict timing or security requirements on systems. In order to facilitate a safe consolidation of such workloads with other applications on a common hardware platform, the generic architecture assumes hardware partitioning functionalities. Depending on the hardware capabilities, the partitioner can be implemented as part of the firmware software, as a separate system monitor or as a part of the hypervisor. Regardless of implementation

details, the partitioner should possess the functionality to divide the hardware in at least two partitions, e.g a *mixed-criticality partition* and a *safety and security partition*. Ideally, the safety and security partition should then be further dividable. Note that this can lead to compatibility and portability issues. Timing or security sensitive applications are often being developed for dedicated OS's. OEMs trying to ship a rich assortment of applications, which are bound to specific OS libraries, are forced to deploy a redundant set of functionalities and services encapsulated in different OS's. However, the new Armv8.4 architecture [83] provides means to mitigate this effect by introducing a new virtualization layer into its Secure world exception level stack and thus allowing for a secure isolation of trusted resources and generalization of trusted code.

Another important feature of the generic architecture is the enormous ecosystem of execution environments for encapsulating applications. This guarantees that regardless of how strict the requirements may be, there exist an adequate execution environment for the efficient development and deployment of a given applications. To this end, the generic architecture offers all forms of environments from bare-metal through virtualized general-purpose OS's, unikernels, containers or even complex nested virtualization solutions and this without having to compromise on isolation and security quality.

Last but not least, with the advent of highly softwarized automation solutions and increasing connectivity, the desire and interest in keeping products state of the art – by installing or updating software without the requirements of its physical presence – has grown significantly. Therefore, the architecture also assumes the realization of over-the-air software techniques. If extended with diagnostic functionalities and methods for live updating of the system software, the architecture comprises a comprehensive and efficient software management concept for the entire infrastructure stack. The new live patching features of the Xen hypervisor can act as an example of such techniques. It allows to substitute hypervisor's data and functions at runtime without the need to reboot the system or to migrate guest VMs.

7 SUMMARY AND CONCLUSION

The last decade revealed the vast economical and societal potential of cyber-physical systems. The unprecedented efficiency of today's hardware facilitates the exploitation of this potential and further fuels the adaptation of CPS to ever new areas of application. However, new opportunities brought to light new challenges. The main ones oscillate around the softwarization of CPS and the spreading adaptation of multi-core and manycore architectures. They have been described in more details in Chapter 1. These challenges, together with the various functional and often strict non-functional requirements of this domain, render CPS and their design highly complex.

Fortunately, these issues are not exclusive to the area of CPS. In other domains, comparable problems could have been successfully tackled by the technique of virtualization. The objective of this thesis is to investigate the applicability of this technique to the domain of CPS.

7.1 CONTRIBUTIONS

The idea of adopting virtualization technology to CPS yields several research questions of both technological as well as methodological nature. The contributions of this work are presented in accordance with these categories.

TECHNOLOGICAL CONTRIBUTIONS The first set of contributions focuses on the technological issues related to the infrastructure software and have been made subject of discussion in Chapter 3.

In Section 3.3, we analyze the shortcomings of the popular platform virtualization solution Xen with respect to hosting time sensitive workloads. We identify two major problems. First, the policies provided by Xen for scheduling VMs with soft- and firm real-time requirements (SEDF and RTDS) fail at fulfilling the timing constraints of the hosted CPS applications. Second, in the Xen architecture, we discover issues associated with I/O processing which introduce priority inversion into the system. Subsequently we propose solutions for the encountered problems – implemented in form of an extension called CPS-Xen – and quantify throughout an extensive set of experiments that our approach significantly improves the timing properties of the execution platform.

The contributions of the second part of Chapter 3 concern with the efficiency of high availability solutions that employ virtual machine replication. In Section 3.4, after analyzing the drawbacks of state of

the art techniques, we introduce our *self-determined* virtual machine replication model. Self-determined replication extends the high availability design space by a replication model that allows for an efficient protection of CPS applications. This claim is being scientifically substantiated by an exhaustive and in-depth evaluation of our approach, including a real-world applicability study which further demonstrates the feasibility of our approach in production environments.

METHODOLOGICAL CONTRIBUTIONS The second set comprises contributions of methodological nature.

Chapter 4 extends the state of the art by introducing a methodology for the planning and integration of safe and efficient virtualized CPS compute and control clusters. It shows how the presented methodology enables to optimally dimension a virtualized CPS, while at the same time allowing to provide strict guarantees regarding the timing predictability of the integrated CPS applications. Furthermore, by combining evolutionary algorithms with a mature performance analysis technique, our approach facilitates the modeling process – e.g. when compared with ILP optimization – and by this significantly improves its accessibility to designers and engineers. Finally, our contribution provides answers for several practical questions that could arise while integrating and managing a virtualized CPS cluster.

In Section 5.1 of Chapter 5, we present a HiL co-simulation architecture, which aims at the validation and verification of virtualized CPS applications while taking into account the complex dependencies between the power system and the communicational and computational aspects of the infrastructure. The proposed architecture enables the analysis of large scale scenarios involving wide area applications. It exhibits, however, shortcoming in respect of the possibility to induce high loads on the ICT hardware. Therefore, its mainly suited for the analysis of functional aspects of the application and the architecture.

The testbed vGridLab, presented in Section 5.2 of the same chapter, overcomes the shortcoming of the previous architecture and enables a real-time evaluation of the applications and the underlying hardware components – even including stress test scenarios. However, it has a reduced modeling scope when compared with the HiL co-simulation architecture. The testbed provides the means for testing the functional aspects of virtualized CPS applications as well as the non-functional characteristics of the underlying ICT in real-time.

Based upon the expertise from the findings presented in the previous chapters as well as some new considerations regarding the requirements of future CPS applications, Chapter 6 gives an outlook towards a generic execution platform architecture with virtualization support for emerging CPS applications.

SOFTWARE CONTRIBUTIONS Finally, in the course of our research work, we developed CPS-Xen with CPS-Remus. A virtual execution environment for a dependable and efficient hosting of virtualized CPS applications. It implements the above listed scientific contributions and serves as the evaluation platform for the results presented in this thesis. For reasons of openness and reproducibility of the presented results, the entire source code is available for download on our GitHub project page [1]. The software stack also includes all of the extensions made by us to the MiniOS-Unikernel.

7.2 FINAL REMARKS

This dissertation advances the state of the art in the domain of cyber-physical systems. It does so by describing several contributions which successfully address various technological as well as methodological issues and challenges related to the adoption of virtualization technology to this domain. This work proves the feasibility of the approach and demonstrates how to employ virtualization technology for the benefits of CPS. Regarding the future of virtualization and CPS, our conclusion is that it is not a question of if, but of to what extend virtualization will be employed for the construction of future CPS. We believe that its role in delivering safety, security and efficiency to CPS will be crucial.

1 https://github.com/cpsxen/cps-xen

A

APPENDIX: NUMERICAL EVALUATION RESULTS.

CPU Load	*Scheduler*	*VM*	*Maximum*	*σ*	*#Missed Deadlines*
55%	RM	VM_1	293 µs	13.4 µs	0
	RTDS	VM_1	1135 µs	59.6 µs	3 (0.15%)
65%	RM	VM_1	295 µs	13.6 µs	0
	RTDS	VM_1	2848 µs	126 µs	174 (0.87%)
80%	RM	VM_1	295 µs	13.4 µs	0
	RTDS	VM_1	4513 µs	290 µs	814 (4.07%)
55%	RM	VM_2	396 µs	17.5 µs	0
	RTDS	VM_2	1914 µs	173 µs	0
65%	RM	VM_2	396 µs	17.2 µs	0
	RTDS	VM_2	3805 µs	255 µs	2 (0.01%)
80%	RM	VM_2	397 µs	17.7 µs	0
	RTDS	VM_2	4589 µs	311 µs	6 (0.03%)
55%	RM	VM_3	490 µs	20.2 µs	0
	RTDS	VM_3	2433 µs	268 µs	0
65%	RM	VM_3	490 µs	20.2 µs	0
	RTDS	VM_3	4204 µs	330 µs	0
80%	RM	VM_3	485 µs	18.9 µs	0
	RTDS	VM_3	6069 µs	429 µs	3 (0.15%)
55%	RM	VM_4	584 µs	21.8 µs	0
	RTDS	VM_4	2159 µs	413 µs	0
65%	RM	VM_4	5078 µs	518 µs	0
	RTDS	VM_4	4256 µs	545 µs	0
80%	RM	VM_4	8446 µs	501 µs	0
	RTDS	VM_4	7679 µs	635 µs	0

Table a.1: Full numerical results of experiments described in Section 3.3.4, including the amount of missed deadlines, the maxima of measured latencies and standard deviation values.

LIST OF FIGURES

LIST OF TABLES

[1] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasey. "A VIRTUAL MACHINE SYSTEM FOR THE 360/40." In: (1966).

[2] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[3] Kyoungho An, Shashank Shekhar, Faruk Caglar, Aniruddha Gokhale, and Shivakumar Sastry. "A cloud middleware for assuring performance and high availability of soft real-time applications." In: *Journal of Systems Architecture* 60.9 (2014), pp. 757–769.

[4] Neil C Audsley, Alan Burns, MF Richardson, and AJ Wellings. *Deadline monotonic scheduling*. University of York, Department of Computer Science, 1990.

[5] Neil C Audsley, Alan Burns, Mike F Richardson, and Andy J Wellings. "Hard real-time scheduling: The deadline-monotonic approach." In: *IFAC Proceedings Volumes* 24.2 (1991), pp. 127–132.

[6] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. "Applying new scheduling theory to static priority pre-emptive scheduling." In: *Software Engineering Journal* 8.5 (1993), pp. 284–292.

[7] AUTOSAR. *AUTOSAR SOMEIP Protocol*. https://www.autosar.org/fileadmin/files/standards/foundation/1-1/protocols/specs/AUTOSAR_PRS_SOMEIPProtocol.pdf.

[8] AUTOSAR. *AUTOSAR SOMEIP-Service-Discovery Protocol*. https://www.autosar.org/fileadmin/files/standards/foundation/1-1/protocols/specs/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf.

[9] Algirdas Avizienis and Liming Chen. "On the implementation of N-version programming for software fault tolerance during execution." In: *Proc. IEEE COMPSAC*. Vol. 77. 1977, pp. 149–155.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." In: *ACM SIGOPS* 37.5 (Oct. 2003), pp. 164–177.

[11] Sanjoy Baruah and Enrico Bini. "Partitioned scheduling of sporadic task systems: an ILP-based approach." In: *Proceedings of the International Conference on Design and Architectures for Signal and Image Processing (DASIP 2008)*. Brussels, Belgium, Nov. 2008.

[12] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "Uppaal – a Tool Suite for Automatic Verification of Real–Time Systems." In: *Proc. of Workshop on Verification and Control of Hybrid Systems III*. Lecture Notes in Computer Science 1066. Springer–Verlag, Oct. 1995, pp. 232–243.

[13] Enrico Bini and Giorgio C. Buttazzo. "Schedulability Analysis of Periodic Fixed Priority Systems." In: *IEEE Trans. Computers* 53.11 (2004), pp. 1462–1473.

[14] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. "A Case for NUMA-aware Contention Management on Multicore Systems." In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010, pp. 557–558.

[15] Hendrik Borghorst and Olaf Spinczyk. "CyPhOS – A Component-based Cache-Aware Multi-Core Operating System." In: *Proceedings of the 32th International Conference on Architecture of Computing Systems (ARCS '19)*. 2019.

[16] T. C. Bressoud and F. B. Schneider. "Hypervisor-based Fault Tolerance." In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 1–11.

[17] Jacques Brygier, Rudolf Fuchsen, and Holger Blasum. *PikeOS: Safe and secure virtualization in a separation microkernel*. Tech. rep. Technical report, SYSGO, 2009.

[18] Liming Chen, Algirdas Avizienis, et al. "N-version programminc: A fault-tolerance approach to rellablllty of software operatlon." In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995*, IEEE. 1995, p. 113.

[19] M. Chereque, D. Powell, P. Reynier, J. L. Richier, and J. Voiron. "Active replication in Delta-4." In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. July 1992, pp. 28–37.

[20] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[21] J. Choi, H. Oh, S. Kim, and S. Ha. "Executing synchronous dataflow graphs on a SPM-based multicore architecture." In: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. June 2012, pp. 664–671.

[22] Mehmet Hazar Cintuglu, Osama A Mohammed, Kemal Akkaya, and A Selcuk Uluagac. "A Survey on Smart Grid Cyber-Physical System Testbeds." In: *IEEE Communications Surveys and Tutorials* 19.1 (2017), pp. 446–464.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines." In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[24] Jeffery K Cochran, Shwu-Min Horng, and John W Fowler. "A multi-population genetic algorithm to solve multi-objective scheduling problems for parallel machines." In: *Computers & Operations Research* 30.7 (2003), pp. 1087–1102.

[25] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*. Vol. 5. Springer, 2007.

[26] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. "An Experimental Time-sharing System." In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE '62 (Spring). San Francisco, California: ACM, 1962, pp. 335–344.

[27] *CPS-Xen: A Virtual Execution Environment for Cyber-Physical Applications*. `https://github.com/cpsxen/cps-xen`.

[28] R. J. Creasy. "The Origin of the VM/370 Time-sharing System." In: *IBM J. Res. Dev.* 25.5 (Sept. 1981), pp. 483–490.

[29] *Credit scheduler*. `https://wiki.xen.org/wiki/Credit_Scheduler`.

[30] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. "Remus: High availability via asynchronous virtual machine replication." In: *In Proc. NSDI*. 2008.

[31] R. I. Davis and A. Burns. "Hierarchical fixed priority preemptive scheduling." In: *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. Dec. 2005.

[32] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan. "A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II." In: Springer, 2000, pp. 849–858.

[33] DIgSILENT. *DIgSILENT PowerFactory*. `http://www.digsilent.de/`. [Online; 2018].

[34] *Docker: Enterprise Container Platform*. `https://www.docker.com/`.

[35] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service." In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 3:1–3:16.

[36] Yaozu Dong, Zhao Yu, and Greg Rose. "SR-IOV Networking in Xen: Architecture, Design and Implementation." In: *Workshop on I/O Virtualization*. Ed. by Muli Ben-Yehuda, Alan L. Cox, and Scott Rixner. USENIX Association, 2008.

[37] Nils Dorsch, Boguslaw Jablkowski, Hanno Georg, Olaf Spinczyk, and Christian Wietfeld. "Analysis of Communication Networks for Smart Substations Using a Virtualized Execution Platform." In: *Proceedings of the International Conference on Communications (ICC '14)*. IEEE Press, 2014.

[38] Arvind Easwaran, Insik Shin, and Insup Lee. "Optimal virtual cluster-based multiprocessor scheduling." In: *Real-Time Systems* 43.1 (Sept. 2009), pp. 25–59.

[39] Christof Ebert and Capers Jones. "Embedded Software: Facts, Figures, and Future." In: *Computer* 42.4 (Apr. 2009), pp. 42–52.

[40] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management." In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266.

[41] X. Fang, S. Misra, G. Xue, and D. Yang. "Smart Grid – The New and Improved Power Grid: A Survey." In: *IEEE Communications Surveys Tutorials* 14.4 (2012), pp. 944–980.

[42] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. "The Flux OSKit: A Substrate for Kernel and Language Research." In: *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*. 1997, pp. 38–51.

[43] Christian Gagné and Marc Parizeau. "Genericity in Evolutionary Computation Software Tools: Principles and Case Study." In: *International Journal on Artificial Intelligence Tools* 15.2 (2006), pp. 173–194.

[44] H. Georg, C. Wietfeld, S. C. Müller, and C. Rehtanz. "A HLA based simulator architecture for co-simulating ICT based power system control and protection systems." In: *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*. Nov. 2012, pp. 264–269.

[45] Frank Ghenassia et al. *Transaction-level modeling with SystemC*. Vol. 2. Springer, 2005.

[46] Robert P. Goldberg. "Architectural Principles for Virtual Computer Systems." PhD thesis. Cambridge, MA, 1972.

[47] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. "Xen and Co.: Communication aware CPU Scheduling for Consolidated Xen-based Hosting Platforms." In: *VEE '07*. New York, USA: ACM, 2007.

[48] José C. Palencia Gutiérrez, José Javier Gutiérrez García, Michael González Harbour, and Juan María Rivas Concepción. *Analysis Techniques used in MAST*.

[49] M. González Harbour, J. J. Gutiérrez Garcıa, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. "MAST: Modeling and Analysis Suite for Real Time Applications." In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. ECRTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 125–.

[50] Michael González Harbour, J. Javier Gutiérrez, José M. Drake, Patricia López Martínez, and J. Carlos Palencia. "Modeling distributed real-time systems with MAST 2." In: *Journal of Systems Architecture* 59.6 (2013), pp. 331–340.

[51] Gernot Heiser. "The role of virtualization in embedded systems." In: *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. ACM. 2008, pp. 11–16.

[52] Gernot Heiser and Kevin Elphinstone. "L4 microkernels: The lessons from 20 years of research and deployment." In: *ACM Transactions on Computer Systems (TOCS)* 34.1 (2016), p. 1.

[53] Martin Hilbert and Priscila López. "The World's Technological Capacity to Store, Communicate, and Compute Information." In: *Science* 332.6025 (2011), pp. 60–65. eprint: `http://science.sciencemag.org/content/332/6025/60.full.pdf`.

[54] HORIBA MIRA Ltd. *MISRA C and MISRA C++*. MISRA Consortium, 2018.

[55] E. S. H. Hou, N. Ansari, and Hong Ren. "A genetic algorithm for multiprocessor scheduling." In: *IEEE Transactions on Parallel and Distributed Systems* 5.2 (Feb. 1994), pp. 113–120.

[56] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. "NetVM : High Performance and Flexible Networking Using Virtualization on Commodity Platforms." In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 445–458.

[57] IEC. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. Geneva, Switzerland: International Electrotechnical Commission, 2018.

[58] IEC TC57. *IEC 61850: Communication networks and systems for power utility automation*. Geneva, Switzerland: International Electrotechnical Commission, 2018.

[59] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules." In: *IEEE Std. 1516-2000* (2000), pp. i–22.

[60] David M.E. Ingram, Pascal Schaub, Richard R. Taylor, and Duncan A. Campbell. "Network Interactions and Performance of a Multifunction IEC 61850 Process Bus." In: *IEEE Transactions on Industrial Electronics* (Sept. 2012).

[61] ISO. *ISO 26262-1: Road vehicles – Functional safety*. International Organization for Standardization, 2018.

[62] B. Jablkowski, M. Mueller, and O. Spinczyk. "High Availability in Cyber-physical Systems by Self-determined Virtual Machine Replication." In: *Proceedings of the 13th IEEE International Symposium on Industrial Embedded Systems (SIES 2018)*. June 2018, pp. 1–10.

[63] Boguslaw Jablkowski, Ulrich Thomas Gabor, and Olaf Spinczyk. "Evolutionary Planning of Virtualized Cyber-Physical Compute and Control Clusters." In: *Journal of Systems Architecture* 73 (Feb. 2017), pp. 17–27.

[64] Boguslaw Jablkowski, Markus Küch, Olaf Spinczyk, and Christian Rehtanz. "A Hardware-in-the-Loop Co-Simulation Architecture for Power System Applications in Virtual Execution Environments." In: *Proceedings of the Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES 2014)*. IEEE Press, 2014, pp. 1–6.

[65] Boguslaw Jablkowski, M. Kuech, N. Dorsch, A. Kubis, O. Spinczyk, C. Wietfeld, and C. Rehtanz. "vGridLab — a testbed for virtualized smart grids." In: *Journal of Computer Science - Research and Development* (Aug. 2017). Extended abstract.

[66] Boguslaw Jablkowski, Markus Kuech, Nils Dorsch, Andreas Kubis, Olaf Spinczyk, Christian Wietfeld, and Christian Rehtanz. "vGridLab: A Testbed for Virtualized Smart Grids." In: *Journal of Computer Science-Research and Development* (Aug. 2017). Extended abstract.

[67] Boguslaw Jablkowski and Olaf Spinczyk. "Continuous Performance Analysis of Fault-Tolerant Virtual Machines." In: *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*. (Braunschweig, Germany). Lecture Notes in Informatics. German Society of Informatics, Sept. 2012, pp. 494–505.

[68] Boguslaw Jablkowski and Olaf Spinczyk. "CPS-Remus: Eine Hochverfügbarkeitslösung für virtualisierte cyber-physische Anwendungen." In: *Tagungsband zum Thema Betriebssysteme und Echtzeit (Echtzeit 2015)*. Ed. by Wolfgang A. Halang and Olaf Spinczyk. Springer, Nov. 2015.

[69] Boguslaw Jablkowski and Olaf Spinczyk. "CPS-Xen: A Virtual Execution Environment for Cyber-Physical Applications." In: *28th International Conference on Architecture of Computing Systems (ARCS '15)*. Porto, Portugal: Springer, Mar. 2015, pp. 108–119.

[70] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whalley, and Ed C. Snible. "Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement." In: *IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, 4-9 July, 2011*. 2011, pp. 72–79.

[71] A. Johnson, J. Wen, J. Wang, E. Liu, and Y. Hu. "Integrated system architecture and technology roadmap toward WAMPAC." In: *ISGT 2011*. Jan. 2011, pp. 1–5.

[72] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System." In: *The Computer Journal* 29.5 (May 1986), pp. 390–395.

[73] Robert Kaiser and Stephan Wagner. "Evolution of the PikeOS microkernel." In: *First International Workshop on Microkernels for Embedded Systems*. Vol. 50. 2007.

[74] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "KVM: the Linux Virtual Machine Monitor." In: *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*. 2007.

[75] Andreas Kubis et al. "Validation of ICT-Based Protection and Control Applications in Electric Power Systems." In: *PowerTech Conference (POWERTECH)*. Eindhoven, Netherlands: IEEE Press, June 2015.

[76] Frederick Kuhl, Judith Dahmann, and Richard Weatherly. *Creating computer simulation systems: An introduction to the high level architecture*. Upper Saddle River and NJ: Prentice Hall PTR, 2000.

[77] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. MIT Press, 2017.

[78] J. Lehoczky, L. Sha, and Y. Ding. "The rate monotonic scheduling algorithm: exact characterization and average case behavior." In: *[1989] Proceedings. Real-Time Systems Symposium*. Dec. 1989, pp. 166–171.

[79] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. "The design and implementation of an operating system to support distributed multimedia applications." In: *IEEE journal on selected areas in communications* 14.7 (1996), pp. 1280–1297.

[80] C. Li, S. Xi, C. Lu, C. D. Gill, and R. Guerin. "Prioritizing soft real-time network traffic in virtualized hosts based on Xen." In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2015, pp. 145–156.

[81] Stan Liao, Steve Tjiang, and Rajesh Gupta. "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment." In: *In Proceedings of the 34th Design Automation Conference*. 1997, pp. 70–75.

[82] J. Liedtke, H. Härtig, and M. Hohmuth. "OS-controlled cache predictability for real-time systems." In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. June 1997, pp. 213–224.

[83] ARM Limited. *Isolation using virtualization in the Secure world. Secure world software architecture on Armv8.4*. ARM Limited, 2018.

[84] *Linux Containers*. `https://linuxcontainers.org/`.

[85] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61.

[86] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. "CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems." In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2009, pp. 215–228.

[87] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. "Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-tolerant Services." In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 575–588.

[88] M. Lu and T. c. Chiueh. "Fast memory state synchronization for virtualization-based fault tolerance." In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. June 2009, pp. 534–543.

[89] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary. "Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions." In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. May 2015, pp. 98–106.

[90] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, and Jo. "Jitsu: Just-In-Time Summoning of Unikernels." In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 559–573.

[91] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Crowcroft. "Unikernels: Library Operating Systems for the Cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 461–472.

[92] Anil Madhavapeddy et al. "Jitsu: Just-In-Time Summoning of Unikernels." In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 559–573.

[93] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. "My VM is Lighter (and Safer) Than Your Container." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 218–233.

[94] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the Art of Network Function Virtualization." In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473.

[95] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer, 2nd ed., 2011.

[96] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Soonhoi Ha, Chanhee Lee, Qiang Xu, and Lin Huang. "Mapping of applications to MPSoCs." In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2011, pp. 109–118.

[97] Alejandro Masrur, Thomas Pfeuffer, Martin Geier, Sebastian Drössler, and Samarjit Chakraborty. "Designing VM Schedulers for Embedded Real-time Applications." In: *CODES+ISSS*. New York, USA: ACM, 2011, pp. 29–38.

[98] C. May. "Mimic: A Fast System/370 Simulator." In: *Papers of the Symposium on Interpreters and Interpretive Techniques*. SIGPLAN '87. St. Paul, Minnesota, USA: ACM, 1987, pp. 1–13.

[99] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. "RemusDB: Transparent High Availability for Database Systems." In: *The VLDB Journal* 22.1 (Feb. 2013), pp. 29–45.

[100] *Mobile cellular subscriptions*. `http://data.worldbank.org/indicator/IT.CEL.SETS.P2?end=2015&start=1990`. Date accessed: 02.03.2017.

[101] Hendrik Moens and Filip De Turck. "VNF-P: A model for efficient placement of virtualized network functions." eng. In: *10e International Conference on Network and Service Management, Proceedings*. Rio de Janeiro, Brazil, 2014, pp. 418–423.

[102] Frank Mueller. "Compiler Support for Software-based Cache Partitioning." In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers & Tools for Real-time Systems*. LCTES '95. La Jolla, California, USA: ACM, 1995, pp. 125–133.

[103] S. C. Müller, H. Georg, C. Rehtanz, and C. Wietfeld. "Hybrid simulation of power systems and ICT for real-time applications." In: *2012 3rd IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe)*. Oct. 2012, pp. 1–7.

[104] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. "Speculative Execution in a Distributed File System." In: *SIGOPS Oper. Syst. Rev.* 39.5 (Oct. 2005), pp. 191–205.

[105] Heikki Orsila, Tero Kangas, Erno Salminen, Timo D Hämäläinen, and Marko Hännikäinen. "Automated memory-aware application distribution for multi-processor system-on-chips." In: *Journal of Systems Architecture* 53.11 (2007), pp. 795–815.

[106] Andrzej Osyczka. "7 - Multicriteria optimization for engineering design." In: *Design Optimization*. Ed. by John S. Gero. Academic Press, 1985, pp. 193–227.

[107] Gabriele Paoloni. "How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures." In: (2010).

[108] PCI-SIG. *PCI-SIG Single Root I/O Virtualization*. `http://pcisig.com/specifications/iov/single_root/`.

[109] Simon Perathoner. "Modular performance analysis of embedded real-time systems: improving modeling scope and accuracy." PhD thesis. ETH Zurich, 2011, pp. 1–230.

[110] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. "Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases." In: *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*. VDTS '09. Nuremberg, Germany: ACM, 2009, pp. 18–23.

[111] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Commun. ACM* 17.7 (July 1974), pp. 412–421.

[112] Donald E. Porter, Galen Hunt, Jon Howell, Reuben Olinsky, and Silas Boyd-Wickizer. "Rethinking the Library OS from the Top Down." In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, Inc., Mar. 2011.

[113] D. Powell, I. Bey, and J. Leuridan, eds. *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1991.

[114] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. "Locating system problems using dynamic instrumentation." In: *2005 Ottawa Linux Symposium*. 2005, pp. 49–64.

[115] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. "Look mum, no vm exits!(almost)." In: *arXiv preprint arXiv:1705.06932* (2017).

[116] Hans P. Reiser and Rudiger Kapitza. "Hypervisor-Based Efficient Proactive Recovery." In: *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*. SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 83–92.

[117] John Scott Robin and Cynthia E. Irvine. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor." In: *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*. SSYM'00. Denver, Colorado: USENIX Association, 2000, pp. 10–10.

[118] RTCA and EUROCAE. *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics and European Organisation for Civil Aviation Equipment, 2018.

[119] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. "Beyond Softnet." In: *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*. ALS '01. Oakland, California: USENIX Association, 2001.

[120] Fred B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial." In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319.

[121] L. H. Seawright and R. A. MacKinnon. "VM/370: A Study of Multiplicity and Usefulness." In: *IBM Syst. J.* 18.1 (Mar. 1979), pp. 4–17.

[122] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. "Mapping on multi/many-core systems: survey of current and emerging trends." In: *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 1.

[123] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. "Binary Translation." In: *Commun. ACM* 36.2 (Feb. 1993), pp. 69–81.

[124] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[125] N. A. Speirs and P. A. Barrett. "Using passive replicates in Delta-4 to provide dependable distributed computing." In: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. June 1989, pp. 184–190.

[126] Brinkley Sprunt, Lui Sha, and John Lehoczky. "Aperiodic task scheduling for Hard-Real-Time systems." In: *Real-Time Systems* 1.1 (1989), pp. 27–60.

[127] Yoshi Tamura. "Kemari: Virtual machine synchronization for fault tolerance using domt." In: *Xen Summit* 2008 (2008).

[128] L. Thiele, S. Chakraborty, and M. Naedele. "Real-time calculus for scheduling hard real-time systems." In: *Proceedings of the IEEE 2000 International Symposium on Circuits and Systems*. 2000, 101–104 vol.4.

[129] L. Thiele, S. Chakraborty, and M. Naedele. "Real-time calculus for scheduling hard real-time systems." In: *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*. Vol. 4. May 2000, 101–104 vol.4.

[130] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. "Mapping applications to tiled multiprocessor embedded systems." In: *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*. IEEE. 2007, pp. 29–40.

[131] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. "Making Shared Caches More Predictable on Multicore Platforms." In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 157–167.

[132] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. *Denali: Lightweight virtual machines for distributed and networked applications*. Tech. rep. Technical Report 02-02-01, University of Washington, 2002.

[133] S. Xi, J. Wilson, C. Lu, and C. Gill. "RT-Xen: Towards real-time hypervisor scheduling in Xen." In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. Oct. 2011, pp. 39–48.

[134] S. Xi, M. Xu, C. abd Phanm L.T.X Lu, C.D Gill, O. Sokolsky, and I. Lee. "Real-Time Multi-Core Virtual Machine Scheduling in Xen." In: *EMSOFT'14* (Oct. 2014).

[135] Sisu Xi, Chong Li, Chenyang Lu, and Christopher D. Gill. "Prioritizing local inter-domain communication in Xen." In: *IWQoS*. IEEE, 2013, pp. 73–82.

[136] Yufeng Xin, I. Baldine, J. Chase, T. Beyene, B. Parkhurst, and A. Chakrabortty. "Virtual smart grid architecture and control framework." In: *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. Oct. 2011, pp. 1–6.

[137] F. Z. Yousaf and T. Taleb. "Fine-grained resource-aware virtual network function management for 5G carrier cloud." In: *IEEE Network* 30.2 (Mar. 2016), pp. 110–115.

[138] Jun Zhu, Wei Dong, Zhefu Jiang, Xiaogang Shi, Zhen Xiao, and Xiaoming Li. "Improving the performance of hypervisor-based fault tolerance." In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–10.

[139] Jun Zhu, Zhefu Jiang, Zhen Xiao, and Xiaoming Li. "Optimizing the performance of virtual machine synchronization for fault tolerance." In: *IEEE Transactions on Computers* 60.12 (2011), pp. 1718–1729.

[140] Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. "Optimization of task allocation and priority assignment in hard real-time distributed systems." In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.4 (2012), p. 85.