Mateus Krepsky Ludwich

ON TIME-DETERMINISTIC MULTICORE VIRTUALIZATION TECHNIQUES

Tese submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Doutor em Ciência da Computação.
Orientador: Antônio Augusto Fröhlich, Prof. Dr.

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Ludwich, Mateus Krepsky
On Time-deterministic Multicore Virtualization Techniques / Mateus Krepsky Ludwich ; orientador, Antônio Augusto Fröhlich, 2018.
205 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2018.

Inclui referências.

1. Ciência da Computação. 2. Virtualização. 3. Modelagem de Interferência. 4. Tempo Real Multicore. 5. Escalonamento Composto. I. Fröhlich, Antônio Augusto. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Mateus Krepsky Ludwich

ON TIME-DETERMINISTIC MULTICORE VIRTUALIZATION TECHNIQUES

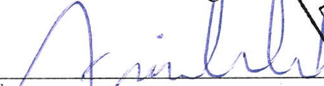
Esta Tese foi julgada aprovada para obtenção do
Título de Doutor em Ciência da Computação, área de concentração
Ciência da Computação e aprovada em sua forma final pelo
Programa de Pós-Graduação em Ciência da Computação
da Universidade Federal de Santa Catarina.

Florianópolis (SC), 19 de Julho de 2018.

Prof. José Luís Almada Güntzel, Dr.
Coordenador do Programa

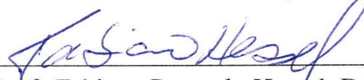
VANIA BOGORNY
Subcoordenadora do PPGCC/UFSC
Portaria nº 2393/2017/GR, de 27/10/20

Banca Examinadora:



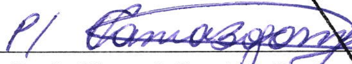
Prof. Antônio Augusto Fröhlich, Dr.
Universidade Federal de Santa Catarina
Orientador

Prof. Rivalino Matias Júnior, Dr.
Universidade Federal de Uberlândia (videoconferência)

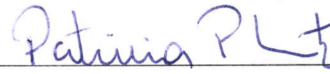


Prof. Fabiano Passuelo Hessel, Dr.
Pontifícia Universidade Católica do Rio Grande do Sul (videoconferência)

VANIA BOGORNY
Subcoordenadora do PPGCC/UFSC
Portaria nº 2393/2017/GR, de 27/10/2



Prof. Giovani Gracioli, Dr.
Universidade Federal de Santa Catarina (videoconferência)



Profa. Patricia Della Mea Plentz, Dra.
Universidade Federal de Santa Catarina

To my grandfather Egon Arno Krepsky,
in memoriam.

ACKNOWLEDGMENTS

First and foremost I would like to thank Jesus Christ, which is God, for his active presence in my live. I would like to thank my mother Eleonora K. Krepsky Ludwich, my father Adriano Brognoli Ludwich, and my brother Tiago Krepsky Ludwich for hearing my complaints and for their words of encouragement. I would like to thank my friends Jonas Godtsfriedt, Rodrigo Bruch, and Thiago Possenti for more than twenty years of friendship. I would like to thank my family, in especial my cousins Andrei Krepsky de Melo, and Daniel Ludwich for their long stand friendship.

I would like to thank my colleagues and friends at LISHA, especially João Gabriel Reis, and Davi Resner for the conversations about work, science, and how the machines are going to take over. :) I would like to thank also Rodrigo Meurer for accepting the challenge of integrating Linux with a hypervisor under construction.

I would like to thank Professor Sebastian Fischmeister for welcoming me during my stay at the University of Waterloo. I would also to thank Professor Borzoo Bonakdarpour for the conversations about formal methods, and for stressing the importance of defining concepts in order, before using them.

I would like to thank Professor Giovani Gracioli for the prolific discussions about multicore and real-time systems.

I would like to especially thank Guto (A.K.A. Professor Antônio Augusto Fröhlich), my advisor, for the innumerous conversations about my research, science, and other themes; and for giving me the opportunity of learning, teaching (as teaching assistant for the Advanced Operating System course), and researching for more than a decade at LISHA. May the hunger for knowledge of the dining philosophers never end!

This research was supported by the Brazilian Federal Agency for Coordination for the Improvement of Higher Education Personnel (CAPES), "CAPES DS" grant, and supported in part by "CAPES/D-FAIT" grant.

On ***Time-deterministic***
"A wizard is never late. Nor is he early; he arrives precisely when he means to." (Peter Jackson et al., The Fellowship of the Ring, 2001).
Multicore
"...single-chip systems with multiple cores..." (Hennessy and Patterson, Computer Architecture - A Quantitative Approach, 2012)
Virtualization *Techniques*
"All problems in computer science can be solved by another level of indirection." (David Wheeler apud Diomidis Spinellis, Beautiful Code: Leading Programmers Explain How They Think, 2007).

RESUMO

A complexidade crescente dos sistemas de tempo real aliada aos avanços nas tecnologias de processadores está proporcionando o uso de arquiteturas multiprocessadas também no domínio de tempo real embarcado. Como consequência, funcionalidades que eram implementadas utilizando-se *hardware* dedicado agora estão sendo implementadas por uma coleção de tarefas executando em um sistema operacional de tempo real. A mudança de *hardware* dedicado para multiprocessadores, os quais compartilham tanto memória como Entrada e Saída (do inglês, *Input/Output* - I/O), levanta questões complexas sobre o isolamento espacial e temporal de tarefas. Neste cenário, máquinas virtuais executando sobre um *hypervisor* representam uma solução bem estabelecida com respeito ao isolamento espacial, uma vez que cada máquina virtual utiliza o seu próprio espaço de endereçamento e é protegida das demais por meio de uma Unidade de Gerenciamento de Memória (do inglês, *Memory Management Unit* - MMU). Isolamento temporal, entretanto, tem sido alvo de intensa pesquisa ultimamente. Escalonamento tempo real, tratamento de interrupção multinível e troca de contexto com precisão de ciclo são exemplos de técnicas incorporadas neste cenário. Entretanto, isolamento temporal ainda não foi completamente explorado, assim como diversos aspectos de arquiteturas multiprocessadas tem sido insuficientemente investigados.

Este trabalho investiga o impacto de interferência temporal que um domínio pode causar em outro enquanto executando em um cenário onde memória Cache de Último Nível (do inglês, *Last-level Cache* - LLC) e interconexões (de memória e de I/O) são compartilhadas. Um sistema de criticalidade mista é levado em consideração o qual é composto por dois níveis de criticalidade: LO (baixo / não crítico) e HI (alto / temporalmente crítico). São considerados dois tipos de domínios, *domínios-HI* que executam tarefas temporalmente críticas (*tarefas-HI*) e *domínios-LO* que executam tarefas não críticas (*tarefas-LO*). Em um determinado instante do tempo, o sistema pode operar em modo LO ou modo HI, dependendo da interferência temporal que é percebida. Enquanto o sistema está operando em modo LO, ambas as *tarefas-LO* e *tarefas-HI* podem executar. Enquanto o sistema está operando em modo HI, todas as *tarefas-LO* são suspensas e somente *tarefas-HI* são permitidas a executar. Um mecanismo para ligar e desligar dispositivos de I/O é empregado para reduzir a interferência causada por contenção

nas interconexões. A semântica do desligamento de dispositivos depende da classe do dispositivo e se ele é físico ou virtual. Por exemplo, enquanto o desligamento de uma Unidade Central de Processamento Física (do inglês, *Physical Central Processing Unit* - PCPU) implica em alterar os níveis de tensão e corrente elétrica enviados ao dispositivo, o desligamento de uma Unidade Central de Processamento Virtual (do inglês, *Virtual Central Processing Unit* - VCPU) pode significar apenas suspender a *thread* do *hypervisor* utilizada para implementar a VCPU em questão.

Este trabalho introduz uma abordagem de modelagem que captura os requisitos de processamento e I/O de tarefas periódicas de um sistema operacional convidado (as quais compõem um domínio) e as abstrai em um Modelo de Recursos de Domínio (do inglês, *Domain Resource Model* - DRM). Um DRM define as reservas de recursos (I/O e processamento) disponíveis a um domínio como também o período de recarga destas reservas. *Escalonabilidade de domínio* é então definida comparando a demanda gerada pelas tarefas do sistema operacional convidado com a reserva que é provida pelo DRM. *Escalonabilidade de sistema* é definida compondo a reserva de todos os DRMs em conjunto e verificando se a plataforma de *hardware* subjacente é capaz de atendê-la. É assumido que os requisitos dos *domínios-HI* são sempre conhecidos, enquanto os requisitos dos *domínios-LO* podem ser conhecidos ou não. Também é assumido que a topologia física empregada (composta por processadores, dispositivos de I/O e suas interconexões) é conhecida. É conhecida também a alocação física, isto é, quais domínios utilizam quais recursos.

A abordagem de modelagem proposta possui duas fases principais, uma em tempo de projeto e outra em tempo de execução. Durante a fase de projeto, *domínios-HI* e *domínios-LO* conhecidos executam e os eventos relacionados ao *overhead* são medidos, como o número de faltas na LLC e o tempo de contenção nas interconexões. Tais informações são utilizadas para inflar os requisitos das tarefas de sistema operacional convidado e para computar DRMs cientes de *overhead*. Durante tempo de execução, novos domínios podem ser admitidos no sistema desde que eles não rompam com a escalonabilidade de *domínios-HI* em modo HI nem com a escalonabilidade de sistema. Durante tempo de execução, informação relacionada ao *overhead* é continuamente monitorada e usada para inflar a reserva dos domínios e na execução de testes de escalonabilidade destes. Sempre que é detectado que algum *domínio-HI* se tornará não escalonável em modo LO, o sistema muda de modo LO para modo HI evitando que *domínios-HI* percam *deadlines*.

O sistema muda de volta para modo *LO* sempre que a interferência decresce a ponto de *domínios-HI* serem escalonáveis novamente em modo *LO*.
A abordagem de modelagem proposta, baseada no DRM, foi avaliada em domínios com requisitos de processamento e I/O distintos e com níveis de criticalidade distintos. As avaliações abordam não apenas tempo de projeto, como também simulam mudanças em tempo de execução, mostrando como a admissão de novos domínios pode impactar na escalonabilidade. A abordagem proposta foi avaliada de acordo com decisões de escalonamento frente a mudanças na quantidade de interferência percebida como o número de faltas na LLC e mudanças na topologia física na medida em que dispositivos eram ligados ou desligados. A abordagem proposta foi também avaliada quanto ao uso de recursos físicos. Comparando a abordagem proposta com o modelo de Recursos Periódicos de Multiprocessados (do inglês, *Multiprocessor Periodic Resource* - MPR), a abordagem proposta apresenta uma maior demanda por PCPUs, pois ela utiliza PCPUs para implementar não somente VCPUs de execução como também VCPUs que tratam operações de I/O. No entanto, ao comparar decisões de escalonamento, a abordagem proposta apresenta melhores resultados que MPR uma vez que a abordagem proposta é ciente de *overhead* e pode escolher desligar *domínios-LO* evitando que *domínios-HI* percam *deadlines* em modo *HI*.
Palavras-chave: Virtualização, Modelagem de Interferência, I/O, LLC, Tempo Real, Multicore, Escalonamento Composto

RESUMO EXPANDIDO

Introdução

A complexidade crescente dos sistemas de tempo real aliada aos avanços nas tecnologias de processadores está proporcionando o uso de arquiteturas multiprocessadas também no domínio de tempo real embarcado. Como consequência, funcionalidades que eram implementadas utilizando-se *hardware* dedicado agora estão sendo implementadas por uma coleção de tarefas executando em um sistema operacional de tempo real. A mudança de *hardware* dedicado para multiprocessadores, os quais compartilham tanto memória como Entrada e Saída (do inglês, *Input/Output* - I/O), levanta questões complexas sobre o isolamento espacial e temporal de tarefas. Neste cenário, máquinas virtuais executando sobre um *hypervisor* representam uma solução bem estabelecida com respeito ao isolamento espacial, uma vez que cada máquina virtual utiliza o seu próprio espaço de endereçamento e é protegida das demais por meio de uma Unidade de Gerenciamento de Memória (do inglês, *Memory Management Unit* - MMU). Isolamento temporal, entretanto, tem sido alvo de intensa pesquisa ultimamente. Escalonamento tempo real, tratamento de interrupção multinível e troca de contexto com precisão de ciclo são exemplos de técnicas incorporadas neste cenário. Entretanto, isolamento temporal ainda não foi completamente explorado, assim como diversos aspectos de arquiteturas multiprocessadas tem sido insuficientemente investigados.

Objetivos

Este trabalho investiga o impacto da interferência temporal que um domínio pode causar em outro enquanto executando em um cenário onde memória Cache de Último Nível (do inglês, *Last-level Cache* - LLC) e interconexões (de memória e de I/O) são compartilhadas. Deste modo, este trabalho pretende responder as seguintes perguntas de pesquisa: (i) Como o compartilhamento de recursos interfere com comportamento temporal de tarefas em um *hypevisor multicore*? (ii) Como quantificar a extensão dessa interferência? (iii) Quais técnicas podem ser utilizadas para reduzir essa interferência ou mantê-la dentro de limites conhecidos? (iv) Como tal interferência pode ser utilizada para alimentar um modelo do sistema de forma que possa ser possível testar a escalonabilidade de tarefas e a viabilidade de domínios? (v) Como tais testes de escalonabilidade cientes de interferência podem ser utilizados em uma estratégia (em tempo de projeto e execução) a qual garanta que tarefas

temporalmente críticas nunca percam um *deadline*? Portanto, o principal objetivo deste trabalho é *demostrar que a abordagem de modelagem proposta a qual é ciente de interferência e os testes de escalonabilidade propostos combinados em uma estratégia de tempo de projeto e execução asseguram que tarefas de sistemas operacionais convidados temporalmente críticas nunca percam um* deadline *em um ambiente virtualizado* multicore *embarcado no qual recursos são compartilhados.*

Metodologia

Os seguintes passos foram seguidos para demonstrar que o objetivo deste trabalho foi alcançado. (i) Apresentar os tipos de interferência de LLC e interconexão. (ii) Discutir como técnicas existentes podem ser utilizadas no cenário de virtualização *multicore* embarcado. (iii) Desenvolver um modelo ciente de interferência juntamente com um teste de análise de escalonabilidade composto (níveis virtual e físico). (iv) Elaborar uma estratégia de tempo de projeto e execução que utiliza as técnicas apresentadas e o modelo proposto para assegurar a escalonabilidade de tarefas de sistema operacional convidado temporalmente críticas. (v) Avaliar a estratégia proposta utilizando domínios sintéticos com conjunto de tarefas distintos em uma topologia encontrada em uma arquitetura *multicore* usual.

Neste trabalho são considerados dois tipos de domínios, *domínios-HI* que executam tarefas temporalmente críticas (*tarefas-HI*) e *domínios-LO* que executam tarefas não críticas (*tarefas-LO*). Em um determinado instante do tempo, o sistema pode operar em modo LO ou modo HI, dependendo da interferência temporal que é percebida. Enquanto o sistema está operando em modo LO, ambas as *tarefas-LO* e *tarefas-HI* podem executar. Enquanto o sistema está operando em modo HI, todas as *tarefas-LO* são suspensas e somente *tarefas-HI* são permitidas a executar.

Resultados e Discussão

A abordagem de modelagem proposta, baseada no Modelo de Recursos de Domínio (do inglês, *Domain Resource Model* - DRM), foi avaliada em domínios com requisitos de processamento e I/O distintos e com níveis de criticalidade distintos. As avaliações abordam não apenas tempo de projeto, como também simulam mudanças em tempo de execução, mostrando como a admissão de novos domínios pode impactar na escalonabilidade. A abordagem proposta foi avaliada de acordo com decisões de escalonamento frente a mudanças na quantidade de interferência percebida como o número de faltas na LLC e mudanças na topologia física na medida em que dispositivos eram ligados ou desligados. A abordagem proposta foi também avaliada quanto ao uso de

recursos físicos. Comparando a abordagem proposta com o modelo de Recursos Periódicos de Multiprocessados (do inglês, *Multiprocessor Periodic Resource* - MPR), a abordagem proposta apresenta uma maior demanda por CPUs físicas, pois ela utiliza CPUs físicas para implementar não somente CPUs virtuais de execução como também CPUs virtuais que tratam operações de I/O. No entanto, ao comparar decisões de escalonamento, a abordagem proposta apresenta melhores resultados que MPR uma vez que a abordagem proposta é ciente de *overhead* e pode escolher desligar *domínios-LO* evitando que *domínios-HI* percam *deadlines* em modo HI.

Considerações Finais

Baseado nos resultados obtidos é possível concluir que o objetivo principal deste trabalho foi alcançado uma vez que, utilizando-se a abordagem proposta, é garantido que HI-tasks sempre cumpram seus *deadlines*, como é demonstrado pelos testes de escalonabilidade.

Até onde foi possível identificar, este trabalho é o primeiro que apresenta uma abordagem para avaliar a escalonabilidade de tarefas de sistema operacional convidado e para computar os requisitos de domínio considerando o compartilhamento de memória e Input/Output (I/O). Espera-se que a abordagem proposta possa ajudar projetistas de sistemas a avaliar se uma dada plataforma é capaz de suportar requisitos de tempo real e como tarefas de sistema operacional convidado podem ser distribuídas entre domínios distintos.

A abordagem proposta foi avaliada apenas analiticamente, utilizando cenários que simulam execuções práticas em uma plataforma real. Por um lado a eficácia e eficiência das técnicas apresentadas neste trabalho já foram demonstradas em cenários não virtualizados, como demostrado no capítulo de revisão do estado da arte, e espera-se que sejam mantidas em ambientes virtualizados. Por outro lado, até onde foi possível identificar, não existe atualmente um único *hypervisor* que implemente todas as técnicas necessárias para a implementação do proposto neste trabalho, como escalonamento composto de criticalidade mista, desligamento e religamento de dispositivos físicos e virtuais e tratamento de interrupção multinível temporalmente determinístico. O sistema operacional Embedded Parallel Operating System (EPOS) contempla a maioria das técnicas necessárias porém, em seu estado atual, não possui um suporte completo à virtualização. Uma extensão do EPOS, denominada de *HyperEPOS* foi planejada no contexto deste trabalho. Uma versão preliminar do projeto do *HyperEPOS* é apresentada no apêndice A, e uma adaptação do sistema operacional LINUX para rodar sobre o *HyperEPOS* é descrita em (MEURER; LUDWICH;

FRöHLICH, 2016). Os mecanismos para suportar tratamento de interrupções de forma temporalmente determinística em uma plataforma física são apresentados em (LUDWICH; FRöHLICH, 2015) e em (LUDWICH et al., 2014). Em estágios iniciais desta pesquisa avaliaram o escalonador do EPOS de acordo com correção funcional e são apresentados em (LUDWICH; FRöHLICH, 2013) e (LUDWICH; FRöHLICH, 2012). O mesmo escalonador que foi utilizado na ocasião para escalonar tarefas de sistema operacional, pode ser utilizado no escalonamento de CPUs virtuais no *HyperEPOS*.

Palavras-chave: Virtualização, Modelagem de Interferência, I/O, LLC, Tempo Real, Multicore, Escalonamento Composto

ABSTRACT

The growing complexity of real-time systems and the advances in processor technology are enabling the use of multicore architectures also in the real-time embedded system domain. As a consequence, features that used to be implemented using dedicated hardware are now being implemented by a collection of tasks running on a Real-time Operating System (RTOS). The shift from dedicated hardware to multicore processors, which share both I/O and memory, raises complex issues about the spatial and temporal isolation of tasks. In this scenario, Virtual Machines atop a hypervisor is a well-established solution with respect to spatial isolation, since each Virtual Machine (VM) uses its own address space and are protected from each other by ordinary MMUs. Temporal isolation, however, has been a subject of intense research lately. Real-time scheduling, multi-level interrupt handling, and cycle-accurate context switching are examples of techniques incorporated in this scenario. However, temporal isolation has not been fully explored yet, and several aspects of multicore architectures have been poorly investigated.
This work investigates the impact of temporal interference that a domain can cause on another while running in a scenario where Last-level Cache (LLC) and interconnects (memory and I/O) are shared. It is taken into account a mixed criticality system, composed of two levels of criticality LO (non-critical) and HI (time-critical). Two types of domains are considered, HI-domains that run time-critical tasks (HI-tasks), and LO-domains that run non-critical tasks (LO-tasks). In a given moment of time, the system can operate either in LO or HI mode depending on the perceived temporal interference. While the system is operating in LO mode, both LO-tasks and HI-tasks can execute. While the system is running in HI mode, all LO-tasks are suspended, and only HI-tasks are allowed to run. A mechanism of powering ON and OFF I/O devices is employed to reduce the interference caused by the interconnect contention. The semantics of powering OFF devices depends on the device class, and on either the device is physical or virtual. For instance, while the powering OFF of a Physical Central Processing Unit (PCPU) means to change the voltage and electric current levels that are sent to the device, the powering OFF of a Virtual Central Processing Unit (VCPU) means to suspend the hypervisor thread employed to implement such a VCPU.

This work introduces a modeling approach, which captures the processing and I/O requirements of periodic guest Operating System (OS) tasks that compose a domain and abstracts them into a Domain Resource Model (DRM). A DRM defines the resource (CPU and I/O) budgets available to a domain as well the replenishment periods of such budgets. *Domain schedulability* is then defined by comparing the demand generated by the guest OS tasks to the budget supplied by the DRM. *System schedulability* is defined by composing the budget of all DRMs together, and by checking whether the underlying physical platform is able to attend them. It is assumed that requirements of HI-domains are always known, while the requirements of LO-domains can be known or not. It is also assumed that the employed physical topology (composed by processors, I/O devices, and their interconnections) is known. The physical allocation it is also known (i.e. whiches domains use whiches resources).

The proposed modeling approach has two main phases, one at design time and other at runtime. During the design phase, HI-domains and known LO-domains execute and overhead related events are measured, such as the number of LLC misses, and the interconnect contention time. Such information is used to inflate the requirements of guest OS tasks and to compute overhead-aware DRMs. During runtime, new domains might be admitted in the system as long they do not disrupt the schedulability of HI-domains in HI mode neither the system schedulability. During runtime, overhead information is monitored continuously and used to inflate the budget of domains and to run scheduling test of domains. Whenever is detected that some HI-domain will become unschedulable in LO mode, there is a HI to LO mode switch, preventing HI-domains from missing deadlines. The system switches back to LO mode whenever the interference decreases and HI-domains are schedulable again in LO mode.

The proposed modeling approach, based on DRM, was evaluated for domains with distinct processing and I/O requirements, and distinct criticality levels. The evaluations comprise not only design time requirements but simulate changes in the runtime scenario, showing how the admission of new domains can impact with the schedulability. The proposed approach was evaluated according to scheduling decisions facing changes in the perceived interference such as the number of LLC misses, and changes in the physical topology as devices were powered ON and OFF. The proposed approach was evaluated also regarding the use of physical resources. Comparing the proposed approach with the Multiprocessor Periodic Resource (MPR) modeling, it presents a

higher PCPU demand since it uses PCPUs for implementing not only execution VCPUs but also for implementing VCPUs that handle I/O operations. However, while comparing scheduling decisions, the proposed approach presents better results than MPR since the proposed approach is overhead-aware and can choose to power OFF LO domains avoiding deadline misses of HI-domains in HI mode.
Keywords: Virtualization, Interference Modeling, I/O, LLC, Real-Time, Multicore, Compositional Scheduling

LIST OF FIGURES

Figure 1 Mapping physical pages to cache locations (GRACIOLI, 2014). 57
Figure 2 Interrupt handling by an ISR. 60
Figure 3 Interrupt handling by an IST. 60
Figure 4 Evaluated Queuing Petri Net (QPN) model. Adapted from (NOORSHAMS et al., 2014). 66
Figure 5 Domain Resources Mapping. 100
Figure 6 Interference due to contention caused by I/O device. . . . 103
Figure 7 Power off dynamics. 106
Figure 8 Interrupt Serviced by Hypervisor (ISbyH). 108
Figure 9 Interrupt Servicing VCPU (ISVCPU). 109
Figure 10 Semaphore Observer. 111
Figure 11 *Semaphore_Observed::notify*. 112
Figure 12 *Semaphore_Observer::updated*. 113
Figure 13 Interference due to cache line eviction caused by another PCPU. 115
Figure 14 Interference due to cache line eviction caused by I/O device. 116
Figure 15 Division of WTU_i^{HI} 153
Figure 16 Schedulability Analysis DRM vs. global EDF (gEDF). . . . 154
Figure 17 Base topology. Known-*LO*-domains case. 155
Figure 18 Runtime scenario, Known-*LO*-domains case. 165
Figure 19 AVG_{miss} over time (known *LO*-domains case). 166
Figure 20 PCPU demand. Known-*LO*-domains case. 168
Figure 21 Fraction of schedulable domains. (known *LO*-domains case). 169
Figure 22 VCPU class. 192
Figure 23 Domain class. 193
Figure 24 Hypercall invocation. 201
Figure 25 VCPU processing messages. 202
Figure 26 Concurrent Observer. 202
Figure 27 *Concurrent_Observed::notify*. 202
Figure 28 *Concurrent_Observer::update*. 203

LIST OF TABLES

Table 1 $\lim_{n\to\infty} SICPr_p(n)$ for different replacement policies (AXER et al., 2014). 55
Table 2 Comparison of approaches for modeling temporal interference. 74
Table 3 Comparison of approaches for detecting temporal interference. 79
Table 4 Comparison of approaches for dealing with I/O virtualization. 88
Table 5 Comparison of approaches for dealing with interrupts virtualization. 92
Table 6 Comparison of approaches for dealing with memory virtualization. 98
Table 7 Generated DRM. 153
Table 8 Base topology insertion and servicing rates. Known-*LO*-domains case. Units in bytes per seconds (B/s). 156
Table 9 Original design-time domain requirements. Known-*LO*-domains case. Units in seconds (s). 158
Table 10 Physical allocation. Domains known at design time. Known-*LO*-domains case. 159
Table 11 Interconnect contention overhead (in seconds). Known-*LO*-domains case. 159
Table 12 LLC miss values and overhead (in seconds). Known-*LO*-domains case. 159
Table 13 Inflated Dom_1 (sum of all tasks) budgets. Known-*LO*-domains case. 160
Table 14 DRMs. Known-*LO*-domains case. 161
Table 15 Non-inflated requirements of new domains. Known-*LO*-domains case. 162
Table 16 Updated physical allocation. All domains (known *LO*-domains case). 163
Table 17 Comparison of hypervisors, including the proposed one (HyperEPOS). 200

LIST OF ACRONYMS

AMC Adaptive Mixed-Criticality

API Application Programming Interface

APIC Advanced Peripheral Interrupt Controller

AVET Average Execution Time

BE Best-effort

BFD Best-fit Decreasing

CAP Cache Partitioning

CC Correlation Coefficient

CCP Complete Cache Partitioning

CCS Complete Cache Sharing

cEDF clustered EDF

COTS Commercial off-the-shelf

CPN Colored Petri Net

CPI Cycle per Instruction

CPS Cyber-Physical System

CPU Central Processing Unit

CRPMD Cache-Related Preemption or Migration Delay

DID Direct Interrupt Delivery

DMA Direct Memory Access

DMPR Deterministic Multiprocessor Periodic Resource

DRAM Dynamic Random-access Memory

DRM Domain Resource Model

DSM Distributed Shared Memory

DP Dynamic Priority

DVT Domain Virtual Time

EDF Earliest Deadline First

EDF-VD EDF-Virtual Deadline

EDF-VD-TM EDF-Virtual Deadline-Task Mode

EPT Extended Page Table

EPOS Embedded Parallel Operating System

FFD First-fit Decreasing

FFSB Flexible File System Benchmark

FIFO First-in First-out

FJP Fixed Job Priority

FPGA Field-programmable Gate Array

FSB Front Side Bus

FTP Fixed Task Priority

gDM global DM

gEDF global EDF

GPFN Guest Page Frame Number

GPOS General Purpose Operating System

GPP Guest Physical Page

GSPN Generalized Stochastic Petri Net

gRM global RM

HPC Hardware Performance Counters

HPP Host Physical Page

HRT Hard Real-time

HVM Hybrid Virtual Machine

IDC Inter Domain Communication

I/O Input/Output

IOMMU Input/Output Memory Management Unit

IOVCPU Input/Output Virtual Central Processing Unit

IPI Interprocessor Interrupt

ISR Interrupt Servicing Routine

IST Interrupt Servicing Thread

ISVCPU Interrupt Servicing VCPU

ISbyH Interrupt Serviced by Hypervisor

KVM Kernel-based Virtual Machine

LAPIC Local APIC

LLC Last-level Cache

LLF Least Laxity First

LP Linear Programming

LRU Least Recently Used

MAE Mean Absolute Error

MC Mixed-Criticality

MFN Machine Frame Number

ML Machine Learning

MMU Memory Management Unit

MPEG Moving Picture Experts Group

MPIOV Multi-Processor I/O Virtualization

MPR Multiprocessor Periodic Resource

NIC Network Interface Card

NTB Non Transparent Bridge

NPB NAS Parallel Benchmark

NPT Nested Page Table

NUMA Nonuniform Memory Access

OS Operating System

PC Personal Computer

PCH Platform Controller Hub

PCI Peripheral Component Interconnect

PCIe PCI Express

PCI-X PCI eXtended

PCPU Physical Central Processing Unit

PD Performance Degradation

pDM partitioned DM

pEDF partitioned EDF

PF Physical Function

p-gate Peripheral Gate

PIBS Priority Inheritance Bandwidth-preserving Server

PLRU Pseudo-LRU

PN Petri Net

PMU Performance Monitoring Unit

pRM partitioned RM

PTP Page Table Prefetching

PUART Physical UART

QoS Quality-of-Service

QPN Queuing Petri Net

RM Rate Monotonic

RMI Remote Method Invocation

RMSE Root Mean Squared Error

RRSE Root Relative Squared Error

RT Real-time

RTOS Real-time Operating System

RVI Rapid Virtualization Indexing

SBF Supply Bound Function

SLAT Second Level Address Translation

SMP Symmetric Multiprocessor

SPT Shadow Page Table

SR-IOV Single-Root Input/Output Virtualization

SRT Soft Real-time

SS Sporadic Server

ST System Time

SICPr State-Induced Cache Predictability

TCP Transmission Control Protocol

TDMA Time Division Multiplexed Access

TLB Translation Lookaside Buffer

TSC Time Stamp Counter

UART Universal Asynchronous Receiver Transmitter

UML Unified Modeling Language

UMA Uniform Memory Access

USB Universal Serial Bus

VCI VM Contention Intensity

vColoring Virtual Coloring

VCPU Virtual Central Processing Unit

VCS VM Contention Sensitivity

VDT Virtual Device Thread

VF Virtual Function

vLLC Virtual LLC

VM Virtual Machine

VMCS Virtual Machine Control Structure

VMM Virtual Machine Monitor

VNIC Virtual NIC

VUART Virtual UART

WCT Wall-clock Time

WCET Worst Case Execution Time

WCRD Worst Case Resource Demand

WCRT Worst Case Response Time

WFD Worst-fit Decreasing

WSS Work Set Size

LIST OF SYMBOLS

Symbol	Description	Page
Ω	Set of criticality levels.	122
$\omega_{\mathcal{S}}$	System criticality.	122
$\mathcal{R}$	Set of virtual I/O device classes.	122
Γ	Set of I/O operations.	122
τ_i	The ith guest OS task.	123
T_i	Period of τ_i.	123
D_i	Relative deadline of τ_i.	123
ω_i	Criticality level of τ_i.	123
$C_i^{\omega_{\mathcal{S}}}$	VCPU budget of τ_i.	123
$R_i^{v,\omega_{\mathcal{S}},\gamma}$	Virtual I/O device budget of τ_i.	123
$\mathcal{D}$	A domain.	123
f^v	Resource unit to time unit mapping function.	124
D_i^{LO}	Artificially tightened deadline for the LO mode.	124
$\delta_{LO_{to}HI}$	LO to HI mode switch overhead.	124
$\omega_{\mathcal{D}}$	Domain criticality.	125
τ	Domain task set.	125
$U_{\tau\Theta}^{\omega_{\mathcal{S}}}$	Total VCPU utilization of a Domain.	125
$U_{\tau\Theta}^{\omega_{\mathcal{S}}}$	Total virtual I/O device utilization of a Domain.	125
$WDU_{\tau}^{\omega_{\mathcal{S}}}$	Whole domain utilization.	125
$\mathcal{M}$	A Domain Resource Model.	126
$\omega_{\mathcal{M}}$	Criticality of $\mathcal{M}$.	126
Π	VCPU period of $\mathcal{M}$.	126
$\Lambda^{v,\gamma}$	Virtual I/O device period of $\mathcal{M}$.	126
$\Theta^{\omega_{\mathcal{S}}}$	VCPU budget of $\mathcal{M}$.	126
$\Xi^{v,\omega_{\mathcal{S}},\gamma}$	Virtual I/O device budget of $\mathcal{M}$.	126
$m^{\omega_{\mathcal{S}}}$	Number of VCPUs in $\mathcal{M}$.	126
$n^{v,\omega_{\mathcal{S}},\gamma}$	Number of virtual I/O devices in $\mathcal{M}$.	126
sbf_{Θ}	VCPU Supply Bound Function.	129
sbf_{Ξ}	Virtual I/O device Supply Bound Function.	130
LV	Set of bandwidth levels.	130
$cl2lv(\omega, lv)$	Criticality level to bandwidth level function.	130
pe	Processing element.	131

$\lambda_{pe}^{\gamma,lv}$ Processing element insertion rate. 131
$\boldsymbol{PE}$ Set of processing elements. 131
me Memory element. 131
$\mu_{me}^{\gamma,lv}$ Memory element servicing rate. 131
$\boldsymbol{ME}$ Set of memory elements. 131
$inter$ Interconnect element. 131
$\lambda_{inter}^{\gamma,lv}$ Interconnect element insertion rate. 131
$\mu_{inter}^{\gamma,lv}$ Interconnect element servicing rate. 131
$\boldsymbol{INTER}$ Set of interconnect elements. 131
$Top({}^{\omega}\mathcal{S})$ Physical topology. 132
$PhyAlloc(pe)$ Physical allocation relation. 133
$W_{\Theta i}^{\omega}(t)$ VCPU workload upper bound. 134
N_i Number of jobs that occur in the time interval a, b. . 134
$\epsilon_{\Theta i}^{\omega}(t)$ VCPU carry-in demand. 134
$W_{\Xi i}^{v,\omega,\gamma}(t)$ Virtual I/O device workload upper bound. 134
$\epsilon_{\Xi i}^{v,\omega,\gamma}(t)$ Virtual I/O device carry-in demand. 134
A_k Length of interval a, r. 135
D_k Length of interval r, b. 135
$I_{\Theta i}^{\omega}$ Total VCPU demand in the interval a, b. 135
$I_{\Xi i}^{v,\omega,\gamma}$ Total virtual I/O device demand in the interval a, b. 135
$I_{\Theta 1,i}^{\omega}$ Total type-1 VCPU demand in the interval a, b. 135
$I_{\Xi 1,i}^{v,\omega,\gamma}$ Total type-1 virtual I/O device demand in the interval a, b. 135
$I_{\Theta 2,i}^{\omega}$ Total type-2 VCPU demand in the interval a, b. 135
$I_{\Xi 2,i}^{v,\omega,\gamma}$ Total type-2 virtual I/O device demand in the interval a, b. 135
$\bar{I}_{\Theta i,1}^{\omega}$ Total type-1 VCPU demand upper bound. 135
$\bar{I}_{\Xi i,1}^{v,\omega,\gamma}$ Total type-1 virtual I/O device demand upper bound. 135
$\bar{I}_{\Theta i,2}^{\omega}$ Total type-2 VCPU demand upper bound. 135
$\bar{I}_{\Xi i,2}^{v,\omega,\gamma}$ Total type-2 virtual I/O device demand upper bound. 136
$\hat{I}_{\Theta i,2}^{\omega}$ Total type-2 VCPU demand upper bound discounting carry-in demand. 136
$\hat{I}_{\Xi i,2}^{v,\omega,\gamma}$ Total type-2 virtual I/O device demand upper bound discounting carry-in demand. 136
DEM_{Θ} VCPU Worst Case Resource Demand upper bound. . . 137

DEM_{Ξ}	Virtual I/O device Worst Case Resource Demand upper bound. 137
$A^{\omega\mathcal{S}}_{\Theta kmax}$	Maximum length of interval a, r for VCPU workloads. 137
$C^{\omega\mathcal{S}}{}_{\Sigma}$	Sum of the m - 1 largest VCPU budgets in a task set. 137
$A^{v,\omega\mathcal{S},\gamma}_{\Xi kmax}$	Maximum length of interval a, r for virtual I/O device workloads. 138
$R^{v,\omega\mathcal{S},\gamma}{}_{\Sigma}$	Sum of the n - 1 largest virtual I/O device budgets in a task set. 138
$\tau_{\mathcal{M}}$	DRM task set computed from $\mathcal{M}$. 139
τ_{Θ}	VCPU task set computed from $\mathcal{M}$. 139
$\tau^{v,\gamma}_{\Xi}$	IOVCPU task set computed from $\mathcal{M}$. 139
cm	Number of tasks in the VCPU task set computed from $\mathcal{M}$. 139
$cn^{v,\gamma}$	Number of tasks in the IOVCPU task set computed from $\mathcal{M}$. 139
ND	Number of domains in the system. 140
MS	Set of all DRMs in the system. 140
τ_{sys}	The task set of all VCPUs and IOVCPUs in the system. 140
$\tau^{v}_{\Xi sys}$	The task set of all IOVCPUs in the system. 140
g^{v}	Time unit to resource unit mapping function. 140
φR	Set of physical I/O device classes. 140
X^{v}	Physical I/O device budged in resource units provided by one element of the v class. 140
$\delta^{\omega\mathcal{S}}_{\Theta}$	LLC miss overhead. 145
Δ_{miss}	LLC miss delay. 145
WC_{miss}	Maximum number of LLC misses events during a VCPU period. 145
AVG_{miss}	Average number of LLC misses events during a VCPU period. 145
$C^{\omega\mathcal{S}''}_{i}$	VCPU inflated budget of τ_i. 145
$CT^{v,\omega\mathcal{S},\gamma}$	Interconnect contention time. 146
$\delta^{v,\omega\mathcal{S},\gamma}_{\Xi}$	Interconnect contention overhead. 146
$R^{v,\omega\mathcal{S},\gamma''}_{i}$	Virtual I/O device inflated budget of τ_i. 146
τ''	Domain inflated task set. 147
$f_{C\delta\Theta}$	VCPU to LLC miss overhead mapping function. 148
$C^{\omega}_{all\mathcal{S}}$	Sum of VCPU budgets of all domains. 148

CONTENTS

1 INTRODUCTION 39
1.1 RESEARCH QUESTIONS AND GOALS 40
1.2 ASSUMPTIONS AND SCOPE 41
1.3 METHODOLOGY 41
1.4 CONTRIBUTIONS 44
1.5 STRUCTURE OF THE DOCUMENT 44
2 BACKGROUND 47
2.1 REQUIREMENTS FOR VIRTUALIZATION 47
2.2 MULTIPROCESSOR REAL-TIME SCHEDULING 49
2.3 MEMORY 52
2.3.1 Cache Mapping 52
2.3.2 Cache Replacement Policies 53
2.3.3 Types of Interference 55
2.3.4 Page Coloring 57
2.3.5 Virtualization 58
2.4 I/O 59
2.4.1 Interrupt Handling 59
2.4.2 Virtualization 61
3 RELATED WORK 63
3.1 INTERFERENCE MODELING 63
3.1.1 Discussion 73
3.2 INTERFERENCE DETECTION AND PREDICTION 75
3.2.1 Discussion 78
3.3 I/O 79
3.3.1 Discussion 87
3.4 INTERRUPTS 89
3.4.1 Discussion 92
3.5 MEMORY 93
3.5.1 Discussion 97
4 PROPOSED TECHNIQUES USAGE 99
4.1 DETECTING INTERFERENCE 99
4.2 I/O 102
4.3 INTERRUPTS 108
4.4 MEMORY 114
4.4.1 Domain-oriented Page-coloring 117
4.4.2 VCPU-oriented Page-coloring 117
4.4.3 Guest OS task-oriented Page-coloring 118

4.5 DISCUSSION 119
5 PROPOSED MODEL 121
5.1 COMMON DEFINITIONS 121
5.2 GUEST OS TASK AND DOMAIN 122
5.3 DOMAIN RESOURCE MODEL 125
5.3.1 Supply Bound Functions of Domain Resource Model 127
5.4 INTERCONNECT TOPOLOGY 130
5.5 OVERHEAD-FREE COMPOSITIONAL SCHEDULABILITY ANALYSIS 133
5.6 DRM GENERATION 140
5.7 VIRTUALIZATION OVERHEAD 141
5.8 OVERHEAD-AWARE COMPOSITIONAL SCHEDULABILITY ANALYSIS 144
5.8.1 LLC miss overhead 145
5.8.2 Interconnect Contention overhead 145
5.8.3 Schedulability Analysis 146
5.9 OVERALL DYNAMICS 147
5.9.1 Variant 1: Knowing the requirements of *LO*-domains 148
5.9.2 Variant 2: Using the Slack Domain 149
5.10 DISCUSSION 150
6 EVALUATION OF THE PROPOSED MODEL 151
6.1 SCHEDULABILITY EVALUATION 151
6.2 ANALYSIS CASE 154
6.3 DISCUSSION 170
7 CONCLUSION 173
References 179
APPENDIX A – Embedded Multicore Hypervisor 191

1 INTRODUCTION

The growing complexity of real-time embedded systems is demanding more processing power due to the evolution and integration of features. In an automotive environment, for instance, new safety functionalities like “automatic emergency breaking” and “night view assist” must read and fuse data from sensors, process video streams, and rise warnings when an obstacle is detected on the road under real-time constraints (MOHAN et al., 2011). Allied to that the continuous evolution of processor technology, together with its decreasing cost, has enabled multicore architectures to be also used in the real-time embedded system domain (CHO; RAVINDRAN; JENSEN, 2006; DAVIS; BURNS, 2011). As consequence, features that used to be implemented using dedicated hardware are now being implemented in software. The usual approach in this case could be to implement such features as tasks to be managed by an RTOS. However, infotainment subsystems (e.g. video player, games and web browsing) usually are too complex to be modeled in the context of critical applications (as tasks of an RTOS) and are better served by conventional OSes, such as Linux, mainly because the level of human interaction they demand and the intensive use of I/O devices. At the same time, conventional OSes are not able to guarantee the stringent time-requirement of critical tasks. Besides, such critical tasks must not suffer interference from other tasks neither spatially (reading data that should not be accessed or corrupting data structures from others) nor temporally (changing the time behavior of a task and making it losing its deadlines).

In this scenario, VMs atop a hypervisor is a well established solution with respect to spatial isolation, since each VM uses its own address space and is protected from each other by ordinary Memory Management Unit (MMU)s. The use of hypervisors instead of multitask alone allows for code reuse, particularly on the infotainment side where non-real time operating system and applications can be used with none or small modifications.

While spatial isolation is a well solved problem, temporal isolation has not been fully explored in the context of virtualized systems, and several architectural aspects of it have been poorly investigated. One of the faced problems is how to ensure temporal isolation between distinct VMs in a hypervisor taking into account that all VMs run on the same physical hardware. Even in multicore architectures, where VMs can execute in distinct processors there is still interference due to shar-

ing of resources. I/O operations such as Direct Memory Access (DMA) transactions that copy data from the device to the main memory or vice-versa, which are triggered by a task running on a VM can cause interference on tasks running on another VM. The interference can be caused by *contention* in the case a task on a VM wishes to access the main memory which is done through the memory interconnect but the memory interconnect is already being used by an I/O device performing a DMA for a task on another VM. Additionally, in a multicore processor the LLC is shared between cores, so a task running on a VM can evict cache lines used by tasks running on another VM. That can happen either due to *intercore interference*: a task access memory and evict cache lines used by tasks running on another VM or due to *I/O interference*: an I/O device performing DMA which was issued by a task on a VM evicts cache lines used by tasks running on another VM. Works targeting I/O temporal interference usually don't target multicore neither virtualization (PELLIZZONI et al., 2008; MALKA et al., 2015). Other works focus on real-time virtualization (including multicore) but not on architectural aspects such memory or I/O (XI et al., 2014; NELSON et al., 2014; HEISER; LESLIE, 2010; BECKERT et al., 2014). There is still works that focus on real-time multicore but are not applied to virtualization (GRACIOLI; FRöHLICH, 2013).

1.1 RESEARCH QUESTIONS AND GOALS

This work investigates the impact that shared resources have on the temporal behavior of tasks running on guest OSes in the context of embedded multicore virtualized platforms. The shared resources addressed by this work are the Last-level Cache (LLC) (shared by all CPUs) and the memory and I/O buses of a system (shared by CPUs and I/O devices). As such, this work intends to answer the following research questions: (i) *How the sharing of resources interferes with the temporal behavior of tasks in a multicore hypervisor?* (ii) *How to quantify the extent of such interference?* (iii) *What are the techniques that can be used to reduce such interference and keep it bounded?* (iv) *How such interference can feed a model of the system in order to check tasks schedulability and domain feasibility?* (v) *How such interference-aware schedulability tests can be used in a (design and runtime) strategy that ensures that temporally critical guest OS tasks never miss a deadline?* Therefore, the main goal of this work is ***to demonstrate that the proposed interference-aware modeling approach and schedul-***

ing tests combined in a design time and runtime strategy can ensure that temporally critical guest OS tasks never miss a deadline in an embedded multicore virtualized platform with shared resources.

1.2 ASSUMPTIONS AND SCOPE

Regarding the physical platform where the hypervisor runs, this work focuses on Symmetric Multiprocessor (SMP) (homogeneous multicore) in which all PCPU have identical processing capacity. Regarding the interconnect, the proposed modeling approach assumes that the physical topology is known. It focuses on physical topologies that present a single main memory. Regarding the interference, this work takes into account the LLC but does not take into account potential interferences inside the main memory that come from mapping physical addresses to the same Dynamic Random-access Memory (DRAM) bank. Differently from the LLC properties such as associativity, used in this work, the mapping of physical addresses to DRAM banks is usually not publicly available (YUN et al., 2014). It assumes that the LLC miss rate is available at runtime. Additionally, the proposed approach assumes that the time critical guest OS tasks requirements are known.

1.3 METHODOLOGY

The following steps were devised to demonstrate that the goal of this work is reached.

1. To present the types of LLC and interconnect interferences.

2. To discuss how existent techniques can be employed in the embedded multicore virtualized scenario.

3. To develop an interference-aware model along with composite (physical and virtual) schedulability analysis tests.

4. To devise a design time and runtime strategy that uses the presented techniques and the proposed model to ensure the schedulability of time-critical guest OS tasks.

5. To evaluate the proposed strategy using synthetic domains and task sets in a topology that follows a usual multicore architecture.

Among the presented techniques are:

- Bus load monitoring, and selective powering-OFF of devices used by non-critical domains or the powering-OFF of non-critical domains themselves to deal with I/O sharing interference.
- Interrupt Servicing Thread (IST)-like mechanism: to bound the time of interrupt handling.
- Page-coloring: to deal with LLC interference.
- Real-time Mixed-Criticality (MC) scheduling algorithms: to schedule VCPUs among the system PCPUs.
- Composite scheduling, scheduling of guest OS tasks virtual processing elements, and virtual processing elements on physical processing elements.

The proposed model takes into account two levels of criticality, *LO* (non-critical), and *HI* (time-critical), two operations for I/O devices ("read" and "write"), and multiple I/O device classes.

The proposed design time and runtime strategy, which uses the proposed model and the selected techniques, was evaluated analytically by simulating practical executions. Such approach was selected instead of practical experiments since, as far was possible to identify, no single hypervisor implements all the required techniques discussed in this work, composite MC real-time scheduling, powering ON and OFF physical and virtual devices, page-coloring, and time-deterministic multilevel interrupt handling.

This work was developed in the Software/Hardware Integration Laboratory (LISHA) in *Universidade Federal de Santa Catarina* and is build upon research conducted in the lab over the last two decades and supervised by Professor Antônio Augusto Fröhlich. The works that had direct influence on this work and their respective authors, in chronological order were:

- First version of EPOS with support for kernel mode (FRöHLICH, 2001).
- Power management interface and mechanism for embedded systems components (HOELLER; WANNER; FRöHLICH, 2006), (HOELLER, 2007).
- Multicore support for EPOS (FRöHLICH; LISHA, 2010).

- An embedded operating system API for monitoring hardware events in multicore processors (GRACIOLI; FRöHLICH, 2011).
- Implementation and evaluation of page coloring cache partitioning on real-time multicore systems (GRACIOLI; FRöHLICH, 2013), (GRACIOLI, 2014).
- Real-time multicore support for EPOS (GRACIOLI, 2014).
- Evaluation of I/O interference on component reconfiguration (REIS; FRöHLICH; JR., 2015), (REIS, 2016).

Regarding the literature review, it started in a more ad-hoc way searching for seminal papers on virtualization, papers regarding interference, and papers published by the group (LISHA). For each paper read, it was identified not only the references but the papers that made citations to the paper read. Afterwards, a more systematic review was employed targeting more "recent" papers (from 2012 to 2016). Some of the employed search strings were

- "virtualization", or "hypervisor" and "real-time"
- "I/O", and "virtualization", and "real-time", and "mixed criticality", or "determinism", or "predictable"
- "performance counter", and "virtualization"

Some of the inclusion criteria were

- Papers focusing on multicore
- Papers focusing on embedded systems

Some of the exclusion criteria were

- Papers targeting performance instead of time-determinism
- Papers focusing mostly on storage, or cloud virtualization

The search was conducted using the following digital libraries and search engines

- IEEEXplore
- ACM DL
- Cite Seer X
- Science Direct

The final selection of papers was a merge between the ad-hoc selection and the papers identified by the search describe above.

1.4 CONTRIBUTIONS

Contributions of this work are

- Discussion of bus load monitoring and selective powering-off of devices and domains to cope with I/O interference.
- Introduction of a mechanism to bound the time of interrupt handling and its refactoring as a design pattern.
- Discussion of page-coloring to deal with LLC interference on virtualized platforms.
- Quantification of time overhead caused by private and shared LLC cache memories.
- Quantification of I/O overhead caused by contention on interconnects.
- Introduction of a model that captures such overheads and can be used for computing domain and whole-system processing and I/O requirements.
- Introduction of schedulability analysis for this model on the virtual and physical levels: guest OS tasks on domains, and domain virtual resources on the platform physical resources.
- Introduction of a strategy for using the proposed model to ensure the deadlines of HI-tasks are always met.

The ideas discussed on this work intend to serve as guidelines for the design and implementation of a time-deterministic multicore hypervisor. Furthermore, the model presented in this work is expected to help system designers checking the feasibility of their task sets given their requirements and target platform.

1.5 STRUCTURE OF THE DOCUMENT

The remaining of this document is organized as follows:

Chapter 2 reviews fundamental concepts concerning this work including the requirements for virtualization, multiprocessor real-time scheduling, and memory and I/O hierarchies. It also reviews the main virtualization approaches for I/O, and memory subsystems.

Chapter 3 reviews related work, which is divided into five groups. The first group presents approaches for interference modeling. The second group presents approaches for interference detection and prediction. The third to fifth groups discuss techniques for making I/O, interrupts, and memory time-deterministic, respectively. In that chapter are presented techniques already employed on hypervisors and also techniques usually employed on RTOSes but that can be potentially extended for virtualized platforms.

Chapter 4 presents the first part of this thesis proposal, a set of guidelines on how techniques for interference detection and handling can be used for ensuring time-deterministic virtualization. The chapter investigates aspects of I/O, interrupts, and memory by the physical and virtual point-of-view.

Chapter 5 presents the second part of thesis proposal, a modeling approach that captures the processing and I/O requirements of guest OS tasks and that computes domain requirements, and the whole system requirements. That chapter also presents scheduling tests for overhead-free and overhead-aware scenarios. Such tests take into account the virtual and physical levels, two levels of criticality, and distinct classes of I/O devices. After presenting how to build domain models and schedulability tests, the chapter presents how they can be used in a design and runtime strategy that ensures HI-tasks never miss their deadlines.

Chapter 6 presents the evaluation of the proposed modeling approach and strategy. First the proposed scheduling tests are evaluated for distinct domains and guest OS tasks. Then, it is presented an analysis case that mimics an real system execution and the proposed approach is compared with the MPR approach.

Chapter 7 presents the final conclusions of this work, points out this work limitations, and the envisioned future works.

2 BACKGROUND

This Chapter reviews some of the base concepts used in later chapters of this document. Section 2.1 reviews the formal requirements for system virtualization introduced by Popek and Goldberg in their seminal paper (POPEK; GOLDBERG, 1974). Section 2.2 reviews multiprocessor architecture and real-time scheduling concepts. Section 2.3 reviews concepts related to memory hierarchy including cache mapping, cache replacement policies, and which are the types of cache interference. Section 2.4 reviews concepts regarding I/O, and interrupt handling. Additionally, Sections 2.3 and 2.4 review virtualization approaches for memory and I/O. By no means is this a comprehensive review of these subjects. Instead, this chapter tries to give a general overview of the presented topics, zooming in on the specific concepts mentioned in further chapters of this document.

2.1 REQUIREMENTS FOR VIRTUALIZATION

Popek and Goldberg present formal requirements that can be used to verify whether a third generation computer architecture supports for virtualization (POPEK; GOLDBERG, 1974). By third generation computer architecture the authors mean architectures that support for memory addressing using relocation register (base address and size of a block of memory), supervisor and user modes, and trap mechanisms that change the control of the program to a specific point up to the execution of certain instructions. They define Virtual Machine Monitor (VMM) (A.K.A hypervisor) as a control software that manages VMs where user software runs.

According to them, instructions in a third generation computer can be classified in privileged, sensitive, and innocuous. A privileged instruction is an instruction that causes a trap and is executed by the control software (the VMM). A sensitive instruction is *control sensitive* if it changes the amount of memory available or affects the processor mode without going through the memory trap sequence. A *behavior sensitive* instruction is an instruction whose execution depends on the value of the relocation-bounds register (R) or on the processor mode (M). A function that is not sensitive is innocuous. Using such instruction classification, according to the Theorem 1 of the paper, a given architecture is virtualizable if the set of sensitive instructions for that

computer is a subset of the set of privileged instructions.

A VMM must satisfy tree properties, named: efficiency, resource control, and equivalence. The efficiency property states that all innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program. The resource control property states that it must be impossible for an arbitrary program to affect the system resources, i.e. memory, available to it. The allocator of the control program (a VMM module) is to be invoked upon any attempt. The equivalence property states that any program K executing with a control program resident performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.

The authors also present the concept of recursively virtualizable and the Theorem 2 of the paper states that a conventional third generation computer is recursively virtualizable if it is: (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it.

Finally the authors relax criteria for virtualization and the Theorem 3 of the paper defines that an Hybrid Virtual Machine (HVM) may be constructed for any conventional third generation machine in which the set of *user sensitive* instructions is a subset of the set of privileged instructions. Where user sensitive is either a user control sensitive (a control sensitive instruction executed at user mode) instruction or a user behavior sensitive (a behavior sensitive instruction executed at user mode) instruction. Additionally, the authors state that in the HVM monitor, all instructions in the virtual supervisor mode will be interpreted.

The paper defines in a formal manner the requirements for classifying a system as virtualizable and represents a seminal work for the area. However, the authors model take into account as resource only memory. Therefore, there are no resource-bounded nor time-critical programs in their model. Additionally, their formal proof assumes that I/O instructions and interrupts do not exist.

Despite more than forty years have passed since Popek and Goldberg work, as Chapter 3 aims to show, memory hierarchy and I/O are still the less explored aspects in current hypervisor solutions.

2.2 MULTIPROCESSOR REAL-TIME SCHEDULING

This section reviews concepts regarding real-time scheduling in multiprocessor architecture. First, it recalls the definitions of multiprocessor architecture and its sub-classifications according to memory organization. Then, it presents real-time computing concepts, defines schedulability, and presents the gEDF algorithm, which is used in this work, along with a scheduling test for gEDF.

A classic *multiprocessor* architecture is composed by two or more processors, and a *shared main memory*. When such processors are grouped together in the same encapsulation the architecture may be refereed as *multicore*. A multiprocessor architecture is typically used for *parallel* and *concurrent programming* (ANDREWS, 1999). The multiprocessor architecture, especially the multicore architecture is focus of this work. Another kind of architecture is the *multicomputer* architecture composed by two or more processors, each processor having its own memory, and communicating to each other by *exchanging messages* (ANDREWS, 1999). Such architecture is mainly used for *distributed programming* and does not belong to the scope of this work.

The multiprocessor architecture can be further classified, according to its memory organization, into SMP and Distributed Shared Memory (DSM) (HENNESSY; PATTERSON, 2012). In the SMP organization, each processor is relatively closed to each other and the time each one takes to access the shared memory is the same. Because of that, SMP is also refereed as Uniform Memory Access (UMA) (HENNESSY; PATTERSON, 2012). In despite of that, each SMP processor can have one of more level of private cache, and that is the case of several multicore processors such as the Intel Core i7 family. In the DSM organization, each processor can contain memory and I/O that is grouped together into a *node*. Processor nodes, communicate to each other by using an interconnect located in the same chip as the processor nodes. In such a case, the memory is still shared since the address space is shared, but is also distributed among the nodes (hence the DSM term). Since processors in a DSM organization are relatively far from each other, the time for accessing memory can vary depending on where is located the target processor node memory. Because of that, the DSM organization it is also referred as Nonuniform Memory Access (NUMA) (HENNESSY; PATTERSON, 2012). This work focuses on multiprocessors using the SMP organization and with identical processing capacity.

One of the main characteristics of real-time computing is that the correctness of a computation depends not only on the value of the

generated result but also on the time the generated result was produced. The result must be generated within a specific time known as *deadline*. Therefore, even if a result has a correct value, it might be incorrect from the real-time point of view if obtained after the specified deadline. In such situation, one says that a *deadline miss* has occurred. Real-time systems, are commonly composed of *tasks* that perform a specific real-time computation. Regarding the severity of a deadline miss, a real-time task can be classified into *hard*, *firm*, and *soft*. The deadline miss in a *hard* real-time task represents a *negative* quality-of-service after the deadline, meaning a catastrophic event such as the loss of human lives or an economical disaster. The deadline miss in a *firm* real-time task represents quality-of-service decreasing to *zero* after the deadline, the system might still operate but the value obtained from the computation will be useless. The deadline miss in a *soft* real-time task represents quality-of-service that *gradually decreases* after the deadline, and the value obtained from the computation, despite having a worse quality, might be still useful (SHIN; RAMANATHAN, 1994). A system that is composed by more than one type of task regarding deadline severity is classified as a multi or Mixed-Criticality (MC) system (VESTAL, 2007). This work focuses on MC systems executing in a virtualized platform. Two levels of criticality are envisioned, one HI and another LO. As such, HI-tasks referrer to *hard* real-time tasks, while LO-tasks referrer to *soft* real-time tasks or tasks that executed according to a best-effort policy.

One of the most employed models for describing a real-time task it the "tree-parameter" model, denoted as (T_i, E_i, D_i). In such a model a sporadic task τ_i has a lower bound value for period (T_i) that specifies the minimum time between two distinct occurrences of a task (known as *job*). Whenever T_i denotes not the minimum but the exact time between jobs, the task is refereed as *periodic*. Each job has a relative deadline (D_i) that specifies the time the job has to complete after it is released. Failing to do so incurs in a deadlines miss. The other parameter, E_i is the upper bound in which the job is expected to complete its computation once is released, known as Worst Case Execution Time (WCET) (BARUAH; BERTOGNA; BUTTAZZO, 2015).

A set of tasks that are expected to run in the same platform is known as *task set* or *task system*. A task set is *feasible* if exists a *correct* (all deadlines are met) *schedule* (sequence of jobs execution) for every collection of jobs in the task system. A task set is *schedulable* if all deadlines are met by the scheduling algorithm (which constructs schedules online during the system execution). Additionally, *schedul-*

ing tests inform at design time (offline) whether a given task set can be scheduled by a specific scheduling algorithm, based on the number of tasks in the set and their parameters (BARUAH; BERTOGNA; BUTTAZZO, 2015). This work focuses on scheduling virtual processing elements on physical resources, and guest OS tasks on such virtual processing elements in a multiprocessor (both physical and virtual) platform. Furthermore, it introduces scheduling tests for such systems in the presence of architectural-related overheads.

Tasks in a task set are scheduled according to their *priorities*, from the tasks of higher priority to the tasks of lower priority. Tasks sets and their respective scheduling algorithms can be classified according to how task priority is assigned. There are three main classes, Fixed Task Priority (FTP), Fixed Job Priority (FJP), and Dynamic Priority (DP). In FTP, the priority is assigned to the task and all jobs of that task will have the same priority. One example of FTP scheduling algorithm is the Rate Monotonic (RM), which defines the task of higher priority as the one with the smaller period. In the FJP, the priority is assigned to the job of a task and does not change within that job execution. One example of FJP scheduling algorithm is the Earliest Deadline First (EDF), which defines as the highest priority job the one that is closer to its deadline. In DP, the priority of a job can change during its execution. One example of DP scheduling algorithm is the Least Laxity First (LLF), in which the job that has the least *laxity* (has more computation to perform in relation to its deadline) as the one of highest priority. The laxity of a job remains the same while the job is executing and increases as the job is not executing (e.g. because it has being preempted) (BARUAH; BERTOGNA; BUTTAZZO, 2015). This work focuses on FJP scheduling algorithms for intra-domain scheduling. More precisely, on the gEDF scheduling algorithm, a EDF version for multiprocessor platforms in which every task can run on any processor, thus being "globally" scheduled. Such scheduling is suitable for this work proposal since, as explained in Chapter 5, all tasks inside the same domain share the same criticality.

Several scheduling tests have being proposed for gEDF. Mostly of them are refereed as *sufficient* meaning that they reject all unschedulable task sets but can also reject some task sets that are schedulable while executing the gEDF algorithm in a running system. Equation 2.1 presents the condition used in the scheduling test for gEDF proposed by (GOOSSENS; FUNK; BARUAH, 2003). However, instead of the original notation the notation used is the one presented in (BARUAH; BERTOGNA; BUTTAZZO, 2015).

$$U_{sum}(\tau) \leqslant m - (m-1)U_{max}(\tau) \tag{2.1}$$

where

- $U_{sum}(\tau)$ is the total task set utilization, computed as $\sum_{\tau_i \in \tau}(\frac{Ei}{Ti})$.
- m is the quantity of processors in the target platform.
- $U_{max}(\tau)$ is the maximum utilization among all task utilizations of the task set, computed as $max_{\tau_i \in \tau}(\frac{Ei}{Ti})$.

According to (GOOSSENS; FUNK; BARUAH, 2003) test, a task set τ is schedulable by gEDF if the condition described in Equation 2.1 evaluates to true.

2.3 MEMORY

Ideally, computer system memory would be unlimited and would have instantaneous access time. In practice, however, as close to the Central Processing Unit (CPU) faster (and more expensive) memories are. Cache memories, which are closer to the CPU are typically tens to hundred of times faster than main memory (PATTERSON; HENNESSY, 2005). Caches implement mechanisms for hiding access latency to the main memory, which assume that data that is being used now is more likely to be used again in a near future (principle of temporal locality) and that data close to the one that being used now is more likely to be used next (principle of spatial locality) (PATTERSON; HENNESSY, 2005). However, most of these mechanisms such as cache mapping and replacement policies were designed with efficiency in mind but not with temporal predictability which can difficult their use (without further adaptations) in real-time systems (AXER et al., 2014).

This section first reviews cache mapping types and replacement policies, then it presents the sources of unpredictability relate to the use of cache in multicore processors. Finally, it points out solutions, presenting page coloring, the cache-partitioning strategy that was chosen to be employed on this work.

2.3.1 Cache Mapping

The cache mapping strategy defines where in the cache a main memory address can be mapped. Consequently, it also defines how the

cache system will determine whether the data related to a given address is in the cache or not. There are three types of cache mapping: direct, fully-associative, and N-way associative (PATTERSON; HENNESSY, 2005).

In *direct mapping*, there is a single place on the cache (cache line) that a main memory block can be mapped to. The cache line is usually determined by the modulo congruent (%) operation as shown by Equation 2.2.

$$\begin{aligned} \text{Cache line} = (\text{Block address}) \\ \% \ (\text{Number of cache blocks in the cache}) \end{aligned} \tag{2.2}$$

In *fully-associative* mapping, a block in main memory can be mapped to anywhere in the cache. In this strategy, the main memory address is decomposed in a tag field and block offset where the *tag* field identifies the cache line and the *block offset* identifies the position within the cache line a byte is. Additionally, there is a *validity bit* that identifies whether the cache line contains valid data, a *dirty bit* that identifies whether the cache line has been modified, and, in the case of multicore processors, coherence protocol bits.

N-way set-associative mapping combines direct and fully associative mapping strategies. As such, a block in main memory can be mapped to a fixed number of locations in the cache, the *sets*. Similarly to Equation 2.2, Equation 2.3 defines how a cache set is determined.

$$\begin{aligned} \text{Cache set} = (\text{Block number}) \\ \% \ (\text{Number of sets in the cache}) \end{aligned} \tag{2.3}$$

In N-way set-associative mapping, in addition to the tag and block offset fields, there is also an *index* field that identifies the cache set that is being addressed.

2.3.2 Cache Replacement Policies

A cache line replacement policy defines which cache line is evicted on the occurrence of a cache miss. Some of the main cache line replacement policies are Least Recently Used (LRU), Random, First-in First-out (FIFO), and Pseudo-LRU (PLRU), which are briefly presented below.

- **Least Recently Used (LRU)**: The cache line that has been "least recently used" is the one replaced by the cache controller

(AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004).

- **Random**: Chooses the cache line to be replaced randomly from all cache lines in the set (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004).

- **FIFO**: Replaces the cache lines in a sequential order, which after the first round, will replace the oldest line in the set. Each cache set has a circular counter which points to the next cache line to be replaced and the counter is updated on every cache miss (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004).

- **PLRU**: Employs approximations of the LRU mechanism to speed up operations and reduce the complexity of implementation. Due to approximations, the least recently accessed cache line is not always the location to be replaced. There are several different methods to implement PLRU. Two examples are the Most Recently Used (MRU)-based Pseudo-LRU (PLRUm) and the Tree-based Pseudo-LRU (PLRUt). See (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004) for an explanation of these two methods.

Distinct cache line replacement policies have different predictability. Axer et al. have compared the four aforementioned cache policies, according to predictability (AXER et al., 2014). Before present their findings, a definition of cache predictability is presented. The source of unpredictability in caches is the state of the cache (formed by the values in the cache) at the moment it start to perform a sequence of memory accesses. On one hand, an unpredictable cache line replacement policy will present a completely distinct number of deadline misses for the same sequence of memory accesses, depending on the cache initial state. On the other hand, a complete predictable cache line replacement policy will present the same number of deadline misses every time it performs a given sequence of memory access regardless of the initial state of the cache. Equation 2.4 defines State-Induced Cache Predictability (SICPr) that is a measure of how predictable a cache replacement policy is.

$$SICPr_p(n) := \min_{q_1, q_2 \in Q_p} \min_{s \in B_n} \frac{M_p(q_1,\ s)}{M_p(q_2,\ s)} \tag{2.4}$$

Where:

- p is the cache replacement policy;

- n is the length of a sequence of memory accesses;

Table 1: $\lim_{n\to\infty} SICPr_p(n)$ for different replacement policies (AXER et al., 2014).

	2	3	4	5	6	7	8
LRU	1	1	1	1	1	1	1
FIFO	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$
PLRU	1	-	0	-	-	-	0
RANDOM	0	0	0	0	0	0	0

- Q_p is the set of cache states;
- B_n is the set of sequences of memory accesses of length n;
- $M_p(q, s)$ number of misses of policy p accessing sequence s starting in cache state q.

The quotient given by $\frac{M_p(q_1,\ s)}{M_p(q_2,\ s)}$ in the Equation 2.4 works as a quality measure for predictability. The minimal value of such a quotient will be obtained when the fraction's numerator and denominator represent the number of cache misses is the best case worst case scenarios, respectively. If the number of cache misses in the worst case is much bigger than the number of cache misses in the best case, such quotient will be near to zero. Therefore, as close such quotient is from 1 the better, in the case it is exactly 1 means that there is not difference in the number of misses in the worst and best case and the policy is totally predictable. As a consequence, $SICPr_p(n)$ is one value over the interval $[0, 1]$ and as close to 1, more predictable the cache replacement policy is.

Reineke and Grund have studied the $\lim_{n\to\infty} SICPr_p(n)$ that investigates the behavior of cache replacement policies when the sequence of memory access tends to infinite. Their tool named "Relacs" can compute $\lim_{n\to\infty} SICPr_p(n)$ for several cache replacement policies (REINEKE; GRUND, 2013). Table 1 present the results for LRU, FIFO, PLRU and random policies with associativities ranging from 2 to 8.

As the table indicates, LRU is the optimal policy and, therefore, the more robust to cache state. FIFO and PLRU are much more sensitive to their state than LRU. Depending on its state, FIFO(k) may have up to k times as many misses. PLRU is only defined for powers of two. At associativity 2, PLRU and LRU coincide. In PLRU, for greater associativities, the number of misses incurred by a sequence s starting in state

q_1 cannot be bounded by the number of misses incurred by the same sequence s starting in another state q_2 . For a truly random replacement, the state-induced cache predictability is 0 for all associativities (AXER et al., 2014).

2.3.3 Types of Interference

According to Gracioli, cache interference can be classified in three types regarding their cause: intra-task, intra-core, and inter-core (GRACIOLI, 2014).

In *intra-task* interference a task evicts its own cache lines. That can occur whenever two memory entries in the working set of a task are mapped in the same cache set. One reason for that to happen is having the working set of a task larger than a specific cache level. Such kind of interference also happens in single-core processors.

In *intra-core* interference, a task evicts cache lines from another (preempted) task. Intra-core interference, in this case, is also an inter-task interference caused whenever a running task evicts cached data from a preempted task. As a consequence, the preempted task will suffer cache misses and an increase in its memory access time as soon it is rescheduled.

In *inter-core interference*, which is also a kind of inter-task interference, tasks running in different cores access the same cache line on the shared level of cache. One task writes on the shared cache block and the invalidation-based coherence algorithm causes an invalidation to establish ownership of that block. Then, another task attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred.

A consequence of time unpredictability caused by cache interference on real-time systems is the increasing of the WCET that becomes more pessimistic which, in some cases, make the system implementation impracticable (GRACIOLI, 2014; WILHELM et al., 2008). One can suggest to disable the cache to eliminate the sources of unpredictability associated with them. However, direct access to main memory would also increase the WCET making it prohibitive in some cases. WCET are much better with caches than without, even if neither data nor instructions are accessed twice (LIEDTKE; HAERTIG; HOHMUTH, 1997).

Therefore, instead of eliminating caches, two kinds of solutions have been explored to make the use of cache more predictable for their use in real-time systems: *cache locking* and *cache partitioning*. Cache

locking is a mostly hardware-based solution in which a group of cache lines (or an individual cache line) is “locked” and cannot be evicted. Cache partitioning, which has hardware and software-based solutions, divides the cache in mutual-exclusive regions (partitions) and assigns specific partitions to tasks or cores. The next section details the cache-partition strategy named *page coloring*, one of the techniques that are used in this work. An extensive review of cache locking and partitioning methods can be found in (GRACIOLI, 2014).

2.3.4 Page Coloring

Page coloring is a cache partition technique that divides the cache in groups of indexes (groups of sets) where which one of those groups represents a color. Each color is then assigned to a group of pages and page frames that are managed by the OS.

Figure 1 shows the mapping of pages and page frames into cache lines. The maximum number of available colors in one platform, given by Equation 2.5, depends on the cache size, the size of each page, and on the cache associativity.

$$\begin{aligned}\text{number of colors} = \text{cache size} \;/\; \text{number of ways} \\ /\; \text{page size}\end{aligned} \tag{2.5}$$

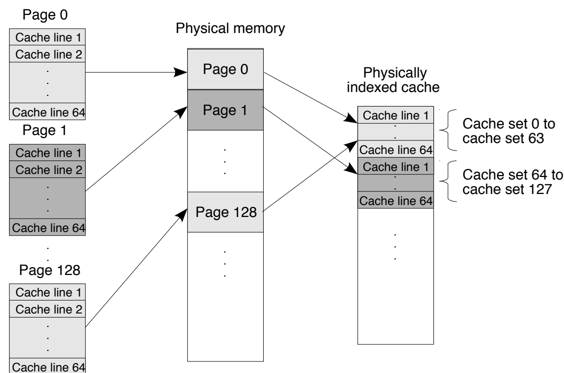


Figure 1: Mapping physical pages to cache locations (GRACIOLI, 2014).

For example, the i7 2600 processor has at its shared level cache

(L3) a 8MB 16 way associative cache. Taking into account a page size of 4KB, according to Equation 2.5 such processor will present 128 colors available for page coloring (8M / 16 / 4K).

As can be noticed in Figure 1, a page color is given to every page and, after the maximum number of available color is reach, the next page (e.g. page 128 in the figure) will get the color 0 again. Each line that composes a color belongs to a distinct cache set and distinct colors will use distinct groups of cache sets, therefore pages from different colors will not evict cache lines from one another.

2.3.5 Virtualization

Memory virtualization is usually performed by providing virtual MMU or virtualizing MMU's page tables. The employed strategy is usually architecture-dependent and depends on whether para or full-virtualization is being employed and whether architectural support for virtualization is available.

In the case of para-virtualization, where the guest OS kernel is modified, one strategy usually employed is to replace direct MMU interaction by hypercalls that perform the MMU update. For para-virtualization the Xen, hypervisor uses the concept of *pseudo-physical* memory (CHISNALL, 2007). In such approach, the application at the guest OS uses *virtual* memory, guest OS uses *pseudo-physical* memory, and hypervisor uses *machine* memory. The guest OS uses the pseudo-physical memory to perform memory operations. From the point of view of the guest OS, the "physical" memory is identified by Guest Page Frame Number (GPFN)s. GPFN have per-domain meaning and identify pseudo-physical memory frames. The actual physical memory is identified by Machine Frame Number (MFN)s that have the same meaning within the hypervisor or any domain. The use of pseudo-physical memory frames enables the guest OS have a contiguous vision of page frames, even if the underlying machine page frames are sparsely allocated and in any order. The hypervisor maintains a machine-to-pseudo-physical table which records the mapping from machine page frames to pseudo-physical ones. The domain maintains a pseudo-physical-to-machine table which performs the inverse mapping.

In the case of full-virtualization, the guest OS perform memory management as if it was not virtualized (i.e. it is not aware of the virtualization). To accomplish that, the hypervisor must somehow trap all pages updates and translate them in such a way the principle of re-

source control (POPEK; GOLDBERG, 1974) is not violated. One strategy for page table virtualization is known as Shadow Page Table (SPT). In SPT, the guest has a copy of the page table in a set of pages marked by the hypervisor as read-only. When the guest updates these, it causes a fault, and control is transferred to the hypervisor. The hypervisor then translates the updates into a real page table and continues. SPT can be aided by architectural support for virtualization as it is provided by Intel VT-x and AMD-V x86 extensions. While using SPT in the case of the x86 (IA32) architecture, whenever the guest attempts to update the CR3 register, or modify the page tables, the CPU traps into the hypervisor and allows it to emulate the update.

Another virtualization strategy provided by architectural extensions is the Second Level Address Translation (SLAT). The SLAT implementation of Intel is known as Extended Page Table (EPT) and AMD's is known as Rapid Virtualization Indexing (RVI) (or Nested Page Table (NPT)). SLAT adds a higher level to the page table. Each guest is allowed to manipulate CR3 directly; however, the semantics of this register are modified. The guest sees a completely virtualized address space, and only sets up mappings within the range allocated by a hypervisor. The hypervisor controls the MMU to manipulate the mappings but does not need to get involved while they are running as in software-managed SPT (CHISNALL, 2007).

2.4 I/O

This section reviews the usual approaches for interrupt handling in non virtualized systems and explains how usually I/O and interrupts are virtualized.

2.4.1 Interrupt Handling

Traditionally, interrupt handling is performed by having an Interrupt Servicing Routine (ISR) that is invoked every time an interrupt occurs to service. Servicing an interrupt usually combines a data transfer phase, followed by some processing and either actuation or event-signaling phases. As Figure 2 shows, for every interrupt occurrence, the interrupt reception, handling, and acknowledgment are executed in the context of the running thread. In this way, ordinary ISRs "invade" the running thread, causing jitter and consuming time that, in some

cases, even compromise the job's deadline.

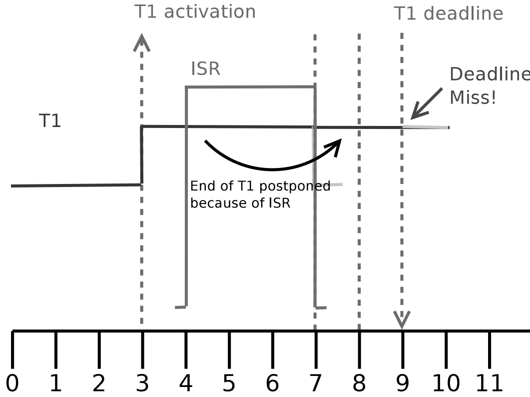


Figure 2: Interrupt handling by an ISR.

Nevertheless, ordinary ISRs have the advantage of presenting very low latency and a virtually constant servicing activation time. This characteristic is exploited by some RTOS, including Sloth (HOFER; LOHMANN; SCHRODER-PREIKSCHAT, 2011), to ensure a predictable execution environment for embedded software. In Sloth, threads and interrupts are handled by an interrupt request arbitration unit. A thread can be activated by a hardware interrupt or by another thread that sets its IRQ bit. The hardware interrupt dispatching mechanism takes over the role of the scheduler. If needed, a task prologue can be executed during thread dispatching to handle context saving and restoring. Schemes like this fit well in some classes of embedded systems for which events can be fully modeled at design-time (such as OSEK (PORTAL, 2008)). Each interrupt must be assigned a unique priority on the same scheme used by tasks and the latency of higher level interrupts or tasks must be accounted in the execution time of lower priority ones. If two interrupts happen at the same time, then the lowest priority one will experience a latency that is much higher than that of the hardware platform. The possible accumulated delays quickly build up to unpredictable execution time for some tasks that are forced from a real-time category into a best-effort one (FOYO; MEJIA-ALVAREZ; NIZ, 2006).

A long-standing alternative to deal with the interference caused by ISRs on more complex systems, for which it is usually impossible to specify a non-interfering model of interrupt triggering and task execution, are *Interrupt Service Threads* (IST), shown in Figure 3. RTOS

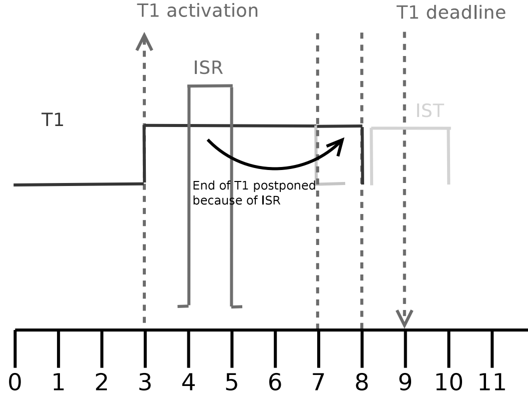


Figure 3: Interrupt handling by an IST.

supporting this mechanism, decouple interrupt reception from handling. Interrupt handling is performed by ordinary threads, scheduled following the same global policy applied to other threads (INTEL, 2014; KLEIMAN; EYKHOLT, 1995).

2.4.2 Virtualization

There are three basic approaches for I/O devices virtualization (HEISER; LESLIE, 2010):

1. To export a (usually simplified) device interface to the guest OS, with the actual driver residing inside the hypervisor.
2. To make the device directly accessible to the guest OS by mapping device registers into the guest's memory and virtualizing interrupts.
3. To run device drivers as separated user-level process, which communicate with the rest of the system using Inter Domain Communication (IDC).

The advantage of approach 1 is to allow for device multiplexing between distinct domains. However, sometimes the device interface is simplified too much and some additional features may not be available. The advantage of approach 2 is efficiency and its main disadvantage is being almost impossible to multiplex devices. The approach 3 allows

for device multiplexing with the cost of additional context switches to implement IDC.

In the case of para-virtualization Xen exports abstract device interfaces to the guest OS. For example, instead of an SCSI device and an IDE device, an abstract block device that supports only two operations (read and write) is exported. Xen employs a *split device driver model* where the *front end* of the device driver lies at a Domain U and the *back end* of the device driver lies at Domain 0. Events can be used for communicating between device driver front and back ends. The front end will send to the back end *requests* events and the back end will send to the front end *response* events. Additionally, I/O ring buffer using shared memory can used for front and back end communication in the case polling is employed. Alternatively, Xen can perform device emulation in the case of full-virtualization and unmodified guest OSses. For advertising device drivers to others domains, Domain 0 can use the XenStore data structure. XenStore is a tree-like data structure that the guest OS can transverse in order to discover device drivers they wish to use (CHISNALL, 2007).

In the case of full-virtualization hypervisors can use architectural extensions such as Input/Output Memory Management Unit (IOMMU). Similarly to MMUs, devices initiate a DMA operation providing a virtual address (even not being aware the address is virtual), rather than a physical one, and the IOMMU translates the address into a physical address. IOMMU can be used to prevent a device controlled by one domain from interfering with another. Both AMD and Intel have proposed interfaces for IOMMU for x86 systems. Additionally, Intel (VT-d) provides for interrupt remapping. While working in conjunction with VT-x, VT-d can be used to assign interrupts to a virtual, rather than physical, CPU. Each interrupt is uniquely identified by its interrupt number and originator ID (derived from the Peripheral Component Interconnect (PCI) device ID) used to generate a mapping to a virtual CPU number.

3 RELATED WORK

This chapter presents the related work regarding aspects of temporal interference. Section 3.1 reviews works that focus mainly on modeling temporal interference, including the computation of resources interfaces, allocation, and scheduling of guest OS task, VCPUs, PCPUs, and domains. Section 3.2 reviews techniques used for deciding when the temporal interference has reached a high level and some action needs to be taken. Section 3.3 presents related work regarding how to deal with temporal interference due to the interconnect contention. Section 3.4 reviews what are the approaches for implementing time-deterministic interrupt handling on virtualized platforms. Section 3.5 reviews the techniques used for dealing with interference caused on the memory hierarchy, mainly at the LLC. By the end of each section, it is presented this work's proposal to handle each one of these aspects.

3.1 INTERFERENCE MODELING

Gopalakrishnan et al. focus on Hard Real-time (HRT) communication in multiple-hop bus-based networks (GOPALAKRISHNAN et al., 2004). The core of their idea is based on perceiving messages as schedulable elements that are periodically transmitted and have an associated end-to-end deadline. Therefore, a message schedule is a route, and the authors propose an off-line technique for synthesizing such routes.

Each message has a source host, a destination host, a period, a transmission time, and an end-to-end deadline. The end-to-end deadline is the time the message has to reach its destination host. The transmission time divided by the message period defines the message resource utilization requirement. A route (or path) consists of all hosts and all buses a message needs to follow in order to reach its destination. For each message to be routed, is computed a set of *effective paths* (paths that contain neither cycles nor strict subset) that are capable of meeting the end-to-end deadline of the message.

The problem of routing generation is formulated as a Linear Programming (LP) problem, whose goal is to maximize the slack available among all buses while keeping some restrictions. The main restriction is that for every message, for every effective path of that message, and for every bus that belongs to that path, the message required utilization plus some slack must be smaller or equal to the bus utilization

bounds. That means each bus in the path must be able to fulfill the transmission rate required by that fraction of the message. The variables in the LP problem define a utilization factor (a value between 0 and 1) that defines the fraction of the message resource utilization requirement for every bus that composes a message path. The solution of the LP formulation assigns a value between 0 and 1 to every utilization factor showing how to route the fraction of the total traffic that a message generates along a given path. Therefore, the solution to the linear program is a solution to the multihop routing problem.

The off-line technique for synthesizing routes proposed by Gopalakrishnan et al. can deal with HRT requirements. It also is capable of dealing with distinct resources, in a sense that each a host can represent a distinct resource (e.g. memory controller, I/O device) that is connected to an interconnect since each message can have a distinct destination. However, knowing all messages and all message requirements (which are derived from parameters such as message size and end-to-end deadline) beforehand might be unfeasible to obtain in practice. Additionally, the model proposed by the authors assumes that all hosts have equal capacity, that all messages are integer multiples of packet lengths, and that transmission times are synchronized.

Kim et al. propose a static estimation method for computing the waiting time due to bus contention for every processing element. The method targets an iterative design space exploration approach where the developer can evaluate distinct buses topologies. The proposed method is based on queuing analysis, and it was developed for three bus arbitration schemes: (i) fixed priority, (ii) round-robin, and (iii) two-level Time Division Multiplexed Access (TDMA) (KIM; IM; HA, 2005).

The method was evaluated according to accuracy while comparing it against trace-based simulation. The static estimation proposed by the authors is several orders of magnitude faster than a trace-driven simulation while keeping the estimation error within 10% consistently in various communication architecture configurations.

While presenting an interesting method for static estimation, Kim et al. focus on design space exploration rather than using the model to predict states of a real running system. Except for the number of memory accesses (obtained from traces), everything is simulated (or statically-estimated). Furthermore, their method's accuracy is compared with trace-driven simulation instead of with data collected on a physical platform.

Noorshams et al. propose modeling I/O performance interference in virtualized environments by using QPNs (NOORSHAMS et al., 2014).

An ordinary Petri Net (PN) consists of tokens, places, transitions, and arcs. Tokens lie in places. A transition can be fire if it is enabled consuming the tokens from its input place and generating tokens to its output place. Arcs connect places to transitions. QPNs extend PNs by defining *queuing places*. A queuing place is a place that has a *queue* where the tokens wait to be serviced and a *depository* for tokens that have completed their service at the queue. The time interval between a token that has entered in the queue until the time the token goes to the depository is known as *servicing time*. Additionally, a QPN is a type of Generalized Stochastic Petri Net (GSPN) meaning that transitions can either be *immediate* (firing at time zero after being enabled) or *timed*, having a fire delay that follows an exponential probability distribution. Furthermore, whenever more than one transition is enabled at the same time, the next transition to fire is randomly chosen based on the transition firing weight (probability). Moreover, a QPN is also a Colored Petri Net (CPN), thus both tokens and transitions can have a color. Token colors enabled the tokens to be classified into groups, while transition colors can define different firing modes.

The approach has evaluated by using QPN to model a virtualized platform composed of the IBM System z mainframe and the IBM DS8700 storage system. Applications in IBM System z run inside Logical Partitions , which is the equivalent to a domain. While IBM System z focuses on providing processor and memory resources, IBM DS8700 focuses on providing storage. The evaluation was performed by running the Flexible File System Benchmark (FFSB) that performs "read" and "write" operations of files while using a given number of threads. The authors have collected the response time for "read" and "write" operations and have built a single VM model using such values. Then, the authors have evaluated the model while simulating a scenario composed of two VMs. After that, the values obtained in the simulation were compared to values obtained with measurements of practical experiments and the estimation error of the model was computed. The mean estimation error was of 8% for "read" operations and 11% for write operations.

The evaluated QPN model, shown in Figure 4 models "read" and "write" requests separately using token colors for distinguishing one type of request from another. "Read" requests wait on the "Reads Waiting" place, while write requests wait on the "Writes Waiting" palace. Such waiting places model requests that come from applications running on the IBM System z mainframe. The servicing time of "Reads Waiting" and "Writes Waiting" places is modeled using the

Gamma-distribution, whose parameters (shape and scale) are obtained from practical measurements. The "Servicing" place represents a request ("read" or "write") being serviced on the IBM DS8700 storage service. The "Scheduler" place is an immediate queuing place (tokens are serviced immediately) that contains one token and ensures that only one request is served at a time and schedules the next request according to the relative priorities. The transition in between the waiting and the servicing states is an immediate transition, firing at zero time after being enabled. Such a transition is ruled by a weighted incidence function that defines which request ("read" or "write") is fired based on their relative weights. Such "read" and "write" weights are chosen in a way (in an iterative process) that minimizes the difference between the measurements of the considered configurations and the simulation results. The authors also define a function that gives the relative the "read" and "write" weights based on the number of workload threads so the model can be used for unseen configurations.

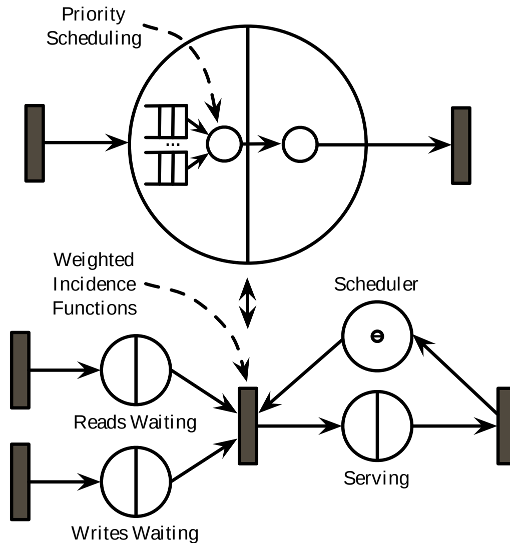


Figure 4: Evaluated QPN model. Adapted from (NOORSHAMS et al., 2014).

QPN seems an interesting approach for modeling interference for performance evaluation. However, the authors evaluate the approach in a coarse-grained model (whole computers instead of computers interconnect and devices) and only for a small number of components

(two computers only).

Easwaran et al. introduce the MPR model that abstracts the processing budget that a domain can collectively provide to a task set of a given domain. The models supported by their proposal are based on sporadic tasks $\tau_i = (T_i, C_i, D_i)$ where T_i denotes the minimum release separation between two jobs of the task, C_i denotes the WCET of a task job, and D_i its relative deadline.

A MPR model is defined as $\mu = \langle \Pi, \Theta, m' \rangle$ where Π denotes the replenishment period of the processing budget that the model collectively provides, which is denoted as Θ, given the model has m' VCPUs.

A domain composed of a task set is schedulable by a MPR model if, for all tasks of the domain, the model is capable of providing resources for the task worst case processing demand. The minimal resource budget a MPR model can provide is specified by its Supply Bound Function (SBF) that is defined according to Equation 3.1.

$$sbf_\mu(t) = \begin{cases} 1^{st}\ case : t' < 0 \\ 0 \\ 2^{nd}\ case : t' \geqslant 0\ \wedge x \in [1, y] \\ \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta + max(0, m'x - (m'\Pi - \Theta)) \\ 3^{rd}\ case : t' \geqslant 0\ \wedge x \notin [1, y] \\ \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta + max(0, m'x - (m'\Pi - \Theta)) - (m' - \beta) \end{cases} \tag{3.1}$$

where

- $k = \frac{\Theta}{m'}$
- $t' = t - (\Pi - \lceil k \rceil)$
- $\alpha = \lfloor k \rfloor$
- $\beta = \Theta - m'\alpha$
- $x = (t' - \Pi \lfloor \frac{t'}{\Pi} \rfloor)$
- $y = \Pi - \alpha$

The task worst case processing demand is defined by Equation 3.2.

$$DEM(A_k + D_k, m') = m'C_k + \sum_{i=1}^{n} \hat{I}_{i,2} + \sum_{i: i \in L(m'-1)} (\bar{I}_{i,2} - \hat{I}_{i,2}) \tag{3.2}$$

where $D_k - A_k$ is the time interval of duration t, and $\bar{I}_i$ and $\hat{I}_i$ are related to the total workload of the task during that interval.

Easwaran et al. describes an algorithm on how computing a MPR model given a domain task set. The algorithm generates one task for each VCPU in the model (m') with processing budgets dimensioned to attend what shall be provided by the model (Θ).

Since there is no known algorithm for scheduling MPR models, Easwaran et al. also introduce an algorithm for converting a MPR model to a task set. Finally, the whole system schedulability can be verified by using a scheduling algorithm such as gEDF for inter-domain scheduling.

Despite the prominent ideas on composite scheduling analysis on multi-processed virtualized systems, Easwaran et al. do not investigate cache memory nor interconnect interferences.

Xu et al. introduce a method for computing the resource bandwidth requirement for each domain of a virtualized platform and for the system as a whole (XU et al., 2013). The resource bandwidth is expressed as the number of resource units (usually time) that are required each given period. The authors extend the MPR model of Easwaran et al. (EASWARAN; SHIN; LEE, 2009) introducing the Deterministic Multiprocessor Periodic Resource (DMPR) model. The idea of DMPR is that every domain will be composed of m full VCPUs and by one partial VCPU. A full VCPU is pinned to an exclusive PCPU so it will be always available for executing guest OS tasks of the domain that it belongs to. A partial VCPU, on the other hand, shares one or more PCPUs with partial VCPUs of other domains. Therefore, a partial VCPU might be suspended and not be available at a given time interval. A partial VCPU is defined by a budget and period. The budget specifies how many resources units the VCPU can provide to guest OS tasks. If the VCPU exhausts its budget within its period, the VCPU is suspended until the next period when its budget is replenished. VCPUs are scheduled by the hypervisor using clustered EDF (cEDF), with a cluster for each full VCPU (composed of one PCPU each) and a cluster with the remaining PCPUs (that are shared among the partial VCPUs of all domains). Within the domains, tasks are scheduled using gEDF. Tasks are specified by their periods, WCET, and deadline. Combining period, WCET, and deadline of all tasks in the task set of the domain, a DMPR model can be computed for the domain. The model describes how many full VCPUs the domain task set needs and what is the required bandwidth of the domain partial VCPU. Then, combining the models of all domains a DMPR model can be derived for the whole system. With that

information is possible to determine the minimal number of required PCPUs for scheduling the whole system.

First the authors present methods for computing a DMPR model (for the domain and for the system) taking into account an overhead-free scenario. After that, the events that can generate a Cache-Related Preemption or Migration Delay (CRPMD) are identified and two methods are devised for computing DMPR models which are aware of the cache overhead: a task-centric approach and a hybrid approach. In the overhead-free scenario a DMPR model for domain is feasible if and only if the demand of its task set is smaller or equal to what is supplied by the domain. By summing the number of required full VCPUs of each domain and combining all partial VCPUs budgets of each domain a DMPR model for the whole system can be derived. Such model is feasible if the number of PCPUs provided by the system is equal to the number of required PCPUs plus one.

In the paper, the authors take into account only private caches (i.e. a shared LLC cache is not taken into account) that could correspond to the inner cache levels (e.g. L1 or L2) of processors commonly found nowadays. Three kinds of events are related to CRPMD (i.e. events that will cause the eviction and further replacement of cache lines): (i) preemption of guest OS tasks within a domain, (ii) preemption of VCPUs, and (iii) the completion of a VCPU (that occurs whenever the VCPU exhausts its budget). Based on these three events the task-centric approaches computes the maximum CRPMD each task of every domain can suffer. The feasibility of the overhead-aware DMPR model is computed by inflating the WCET of each task with the maximum CRPMD overhead and by using the same feasibility check used in the overhead-free of the DMPR model.

The hybrid approach combines the task-centric approach with the model-centric approach and it chooses the DMPR with the smaller minimal bandwidth between the two. In the model-centric approach, the WCET of each task is inflated with the overhead caused by tasks preemption only, as opposed to the task-centric approach that takes into account also the preemption and completion events of VCPUs (named VCPU stops events). In the model-centric approach, for each domain, the overhead caused by VCPU stops events is combined and subtracted from the resource supply of the domain resulting into what is called effective resource supply. The feasibility of the overhead-aware DMPR model of a domain is computed by inflating the WCET of the domain task set, by and using its effective resource supply, and by using the same feasibility check used in the overhead-free of the DMPR model.

The authors have evaluated their DMPR modeling approach in four experiments. The first experiment compares DMPR with the MPR and the authors show that for all employed task sets the DMPR models require less resources then MPR models. The second and the third experiments compare the hybrid and the task-centric DMPR computation approaches. The second experiment evaluates the two approaches while changing the total task set utilization. The third experiment compares how the two approaches perform for task sets with distinct sizes (number of tasks). In both experiments, the hybrid approach requires fewer resources than the task-centric approaches for most of the task sets. The fourth experiment compares the hybrid and task-centric approaches in theoretical and practical scenarios. For that the experiment chooses task sets that are unschedulable according to theory and in practice and measure the deadline miss ratio. The systems deployed following the hybrid approach present a smaller deadline miss ratio then the ones using the task-centric approach. The fourth experiment also compares the overhead-free versions of MPR and DMPR in theory and practice and show that task sets that are schedulable according to theory are not schedulable in practice scenarios, showing that both models underestimate the CRPMD overhead.

While the approach proposed by the authors advances the analysis of cache-related overhead in the scheduling of virtualized systems, it presents two main limitations. The first one is imposing that all domains will have only one partial VCPU, all others VCPUs of every domain need to have a dedicated PCPU each. The second limitation is that only private caches are investigated. The effects of cache-line eviction caused by false sharing of a shared LLC are not taken into account.

In the invited paper for ACM SIGBED Review of 2016, Phan et al., present the challenges for dealing with cache-related interference in the context of virtualized systems where the LLC is shared among the PCPUs (PHAN; XU; LEE, 2016). The authors refer to virtualized systems as “two-level compositional systems”. Previously, Easwaran et al. have proposed methods for schedulability analysis of CPUs in the context of two-level compositional system: at the first level tasks are scheduled among VCPU and at the second level VCPUs are scheduled among PCPUs (EASWARAN; SHIN; LEE, 2009). Easwaran et al. refer as *component* a task set and its scheduling policy, which in virtualization terminology can be denoted as a domain. Easwaran at al. have defined a method for computing how much resource a domain will need, which they refer to as “defining the interface of the component”. The interface

of all components can be further combined forming the interface of the whole system, which specifies how many physical resources (e.g. PCPUs) the system will be need for scheduling all task sets. In another work, Xu et al. have extended their compositional analysis to deal with private caches only, including how to compute cache-aware component interfaces (XU et al., 2013). In ACM SIGBED paper, Phan et al. take into account CPUs, private, and shared LLC (PHAN; XU; LEE, 2016). First, the paper identifies the challenges that shared LLC brings to the compositional schedulability analysis. Then, it delineates the idea of cache scheduling. Finally, the paper delineates how cache-aware component (and system) interfaces could be computed while using CPU and cache scheduling.

In the first part of the paper, the authors identify four main challenges that shared LLC brings to the compositional schedulability analysis: (i) concurrent cache accesses, (ii) cyclic dependency between components' interfaces and tasks' overheads, (iii) dynamic concurrent resource supply patterns, and (iv) interactions between the LLC and private caches. Regarding (i), since PCPUs access the LLC simultaneously is difficult to estimate with precision when a cache miss will occur. In the worst-case, one must assume that every access to the LLC is a cache miss, which is a very pessimistic assumption. Regarding (ii), one of the key ideas of the compositional analysis is having independence between components in the sense they can be computed in isolation. However, it is not possible to know how often each VCPU of a domain will be preempted if one does not have information about the VCPUs of other domains. Also it is not possible to know how often each VCPU will exhaust its budget without knowing the VCPU budget (which is a cyclic dependency). One solution is to define a VCPU period a priori, and make such information available to all other domains. The challenge (iii), pointed out by the authors is not specific of cache sharing but is related to PCPU sharing. A VCPU might not be available 100% of the time, therefore the resource supply changes dynamically according to how the VCPUs are scheduled. That must be taken into account while computing the demand of each task and task set. Regarding (iv), the overhead scenarios are dependent on cache policies (e.g. inclusive vs. exclusive) implemented by each cache level.

In the second part of the paper, the authors present the idea of treating cache as another schedulable dimension. To deal with that problem, the authors mention software-based approaches such as page coloring, and hardware-based approaches such as way-partitioning and Intel Cache Allocation Technology (CAS). The former approach (page

coloring), partitions the cache into disjoint groups of cache sets (the cache colors). The later approach (Intel CAS), provides the facility of assigning portions of the LLC to a given core (specified by its cpuid). The authors point out that software approaches such as two-level page-coloring can present the disadvantage of making cache (re)allocation more expensive since there is the management of colors at the host and guest levels. On the other hand, hardware based approaches despite being more efficient are limited to a smaller number of partitions while compared to software-based techniques. The authors point out that the definitive solution might be hybrid combining software and hardware approaches. The authors also mention static versus dynamic cache partitioning and also suggest a hybrid approach. They point out as a potential solution to perform static allocation at the domain level and dynamic allocation at task level.

In the third part of the paper, the authors discuss extensions towards cache-aware analysis for compositional systems with cache scheduling. Regarding the specification of components, the authors identify that might be necessary to specify the WCET of a task (or VCPU) as a function of the number of cache partition it has, instead of using a single WCET value. They also propose use memory access pattern and working set size, as a refinement to such a component specification as these two factors can greatly influence the overheads.

Being a position paper, that paper points out interesting challenges but does not give definitive answer for any of them. Kim et al. (detailed in Section 3.5) already exploits the idea of defining tasks and VCPUs WCET as function of the number of cache partition (page colors in the case) the task or VCPU has (KIM; RAJKUMAR, 2016). Additionally, Cheng et al. (detailed in Section 3.2) also points out memory access pattern and working set size as major factors regarding influence on overheads (CHENG et al., 2016).

In the invited paper to ACM SIGBED, Lee et al., propose a system model that combines MC and compositional scheduling targeting *open systems* (LEE et al., 2016). By *open systems*, the authors mean Cyber-Physical System (CPS)s composed of components developed by different vendors and later integrated to compose the whole system. The system model proposed by the authors encompasses two criticality levels: high (HI) and low (LO). Regarding the compositional aspect, a component can be composed of a set of components or a set of tasks, which suggests that the system model can be N-level. Regarding MC in open systems, the authors propose two types of component isolation and the concept of *component group*. Components are *weakly isolated*

if HI-tasks in one component may affect the correctness of LO-tasks in other components. Components are *strongly isolated* if no tasks in one component may affect the correctness of tasks in other components. The authors propose that components designed in the same standard (or vendors) to be grouped in the same group and that components designed by different standards to compose distinct groups. Additionally, the authors propose to use weak isolation within a group and strong isolation between groups. The authors suggest extending the EDF-Virtual Deadline (EDF-VD) adding criticality-level, referred as *task-mode* (HI or LO). The envisioned algorithm is named EDF-Virtual Deadline-Task Mode (EDF-VD-TM). Finally, the authors point out compositional methods for computing components' interfaces and schedulability analysis of the proposed algorithm and the whole system needs to be further developed.

The system model proposed by Lee et al. is applicable for two-level virtualized systems. The inner level defines components composed of tasks, composing domains. The outer level defines a component composed of domains, composing the whole system. Their work is still at initial state and does not aim on targeting overhead caused by memory nor I/O hierarchy.

3.1.1 Discussion

Table 2 summarizes the approaches presented by each related work for modeling temporal interference. Column **TD** (Time-deterministic) identifies whether the approach focus on time determinism rather than on performance. Column **Inter.** (Interconnect) identifies whether the approach focus on I/O devices or interconnect. Column **Mem.** (Memory) identifies whether the approach focus on memory hierarchy. Column **MC** (Mixed-Criticality) identifies whether the approach supports for mixed-criticality. Column **MP** (Multiprocessor) identifies whether the approach deals with multiprocessing. Column **Virt.** (Virtualization) identifies whether the approach focus on virtualized systems.

Gopalakrishnan et al. focus on time-determinism and interconnect, proposing scheduling periodic messages among a bus hierarchy until the message reaches its destination (GOPALAKRISHNAN et al., 2004). In despite of the work of Gopalakrishnan et al. did not exploit that, messages period could potentially be related to transaction rate of I/O devices and the presented interconnect network could be used to model the interconnect of physical machines. Their work, however, does not

Table 2: Comparison of approaches for modeling temporal interference.

Work	TD	Inter.	Mem.	MC	MP	Virt.
(GOPALAKRISHNAN et al., 2004)	X	X				
(KIM; IM; HA, 2005)		X				
(NOORSHAMS et al., 2014)		X				X
(EASWARAN; SHIN; LEE, 2009)	X				X	X
(XU et al., 2013)	X		X		X	X
(PHAN; XU; LEE, 2016)	X		X		X	X
(LEE et al., 2016)	X			X	X	X
Proposal	X	X	X	X	X	X

target memory hierarchy, nor mixed-criticality (all messages are HRT), nor multiprocessing, nor virtualization.

Kim et al. focus on interconnect of a physical machine and uses queuing theory to compute contention time of processing elements (KIM; IM; HA, 2005). Noorshams et al. goes further employing both queuing theory and PN to model the scheduling of requests (NOORSHAMS et al., 2014). However, Kim and Noorshams focus mainly on performance, instead of time-determinism. Noorshams et al. claims to focus on virtualization since they use a virtualization platform as case of study. However, they do not model two level of processing elements (physical and virtual) separately. They model the requests made by guest OS tasks on the physical level. Additionally, neither Kim nor Noorshams focus on memory, MC, nor multiprocessing.

Easwaran et al. (EASWARAN; SHIN; LEE, 2009) focus on computing interface (computational requirements) of domains in a multiprocessed virtualized system. Their work does not explore memory nor interconnect interference and targets a single criticality mode. However, their seminal ideas on composite scheduling for multi-processed system were of high importance for works that have extended it such as (XU et al., 2013) and the work of this thesis.

Xu et al., Phan et al., and Lee et al., focus on computing domain interfaces and the whole system in a time-deterministic virtualized platform (XU et al., 2013), (PHAN; XU; LEE, 2016), (LEE et al., 2016). Additionally, Xu et al. takes into account the overhead caused by cache misses on private caches. Phan et al. aims on extending Xu et al. in order to take into account a shared LLC as well. Lee et al. does not fo-

cus on memory, instead it focus on components developed by different vendors and that attend distinct mixed-criticality levels. Neither Xu et al., Phan et al., nor Lee et al focus on interconnect.

This work proposes employing queuing theory for computing the contention time on physical interconnects. Such contention time is used to compute domains and whole-system requirements in a similar way that is proposed by Easwaran et al., Xu et al., Phan et al., and Lee et al. This work, however, also takes into account the interconnect and I/O devices (both in the host and guest levels). Additionally, as in Lee et al., this work focus on two level of criticality (HI and LO), computing two versions of the domain and system interfaces, one for each criticality level.

3.2 INTERFERENCE DETECTION AND PREDICTION

Cheng et al. propose an approach for performance prediction on virtualized multicore systems (CHENG et al., 2016). The authors claim their approach is contention-aware. However, "contention", according to the authors, includes not only contention time waiting for access an interconnect but also the time for repopulating evicted cache lines, and for accessing the main memory. The authors assume that the contention a VM suffers will cause the performance of that VM to degrade. Equation 3.3 (CHENG et al., 2016) defines the Performance Degradation (PD) of a VM A (VM of interest) while executing simultaneously with another VM B.

$$PD^{A}_{co-run/B} = \frac{CPI^{A}_{co-run/B} - CPI^{A}_{alone}}{CPI^{A}_{alone}} \tag{3.3}$$

The metric used for performance measurement is Cycle per Instruction (CPI) of the VM, which is defined by the average CPI of all its VCPUs.

The proposed performance prediction approach uses two *features* for performance prediction, VM Contention Sensitivity (VCS) and VM Contention Intensity (VCI), which are defined in Equation 3.4 (CHENG et al., 2016) and Equation 3.5 (CHENG et al., 2016), respectively.

$$VCS^{A}_{bench_i} = \frac{CPI^{A}_{co-run/bench_i} - CPI^{A}_{alone}}{CPI^{A}_{alone}} \tag{3.4}$$

$$VCI_{bench_i}^{A} = \frac{CPI_{co-run/A}^{bench_i} - CPI_{alone}^{bench_i}}{CPI_{alone}^{bench_i}} \quad (3.5)$$

The VCS of a VM A gives how much A will *suffer* from contention, while VCI of A will give how much A will *cause* contention. VCS and VCI are defined using the CPI of the VM of interest running alone and the CPI of the same VM while running simultaneously with a given benchmark (that executes on another VM).

The performance prediction approaches is based on Machine Learning (ML) algorithms and it is composed of two phases, a training phase and an online monitoring phase. There is a performance predictor for every VM of interest. During the training phase, the ML algorithm is feed with the VCS and VCI features (taking into account all benchmarks of interest), and with the actual PD (taking into account the VM of interest and all other VMs). With that, the predictor learns how VCS and VCI change the actual PD. Afterwards, during the online monitoring phase, the predictor is feed with the monitored VCS and VCI, and gives as output the predicted PD. By comparing the actual PD with the predicted PD it is possible to determine the prediction accuracy.

The proposed approach was evaluated according to prediction accuracy while using the NAS Parallel Benchmark (NPB) suite (benchmarks NPB-OMP and NPB-MPI) and SPEC CPU 2006. NPB evaluates performance of parallel systems, while SPEC CPU 2006 evaluates performance of processor and memory resources. The evaluation uses five machine learning algorithms and four metrics for error computation (actual vs. predicted PD). The metrics used for error computation were Mean Absolute Error (MAE), Correlation Coefficient (CC), Root Mean Squared Error (RMSE), and Root Relative Squared Error (RRSE). The authors have used the Weka data mining software, which has a collection of machine learning algorithms (WAIKATO, 2017). The employed algorithms were (i) linear regression, (ii) REPTree, (iii) M5P, (iv) Bagging, and (v) the Weka default for neural network. REPTree and M5P are based on decision tree and are able to capture non-linear correlations between the features (VCS and VCI) and PD presenting better performance than linear regression for most the cases. The Bagging algorithm, which is a variation of REPTree that employs multiple trees and uses the mean value of all trees as output result, has presented the better performance of all five algorithm. The average MAE is 2.83%, which the authors claim to be a good value for prediction accuracy. Regarding the online monitoring, it is not specified in the paper the

interval used for sampling CPI but the authors say that the general overhead of the monitoring was kept under 0.5% of CPU utilization.

While presenting interesting results, the approach presented by Cheng et al. presents also some limitations. The paper does not focus on contention caused by I/O and network. It only focuses on contention related to memory access. Besides that, the only metric used is CPI, which is mainly CPU-related. Memory related metrics such as cache accesses, cache misses, time spent on memory interconnect are not used. Additionally, the paper does not mention how many PCPUs are used on each experiment, thus it is not clear whether the employed benchmarks run on a single or more than one PCPU. The authors mention the possibility of updating the predictor online, retraining it with new values of VCS and VCI, whenever the prediction error trespass a given threshold. However, that is not explored in the paper. Another application mentioned but not explored is the use of the proposed approach for designing a contention-aware VM schedule algorithm.

Scheffel et al. propose an approach for analyzing processes behavior on multicore systems (SCHEFFEL; FRöHLICH, 2016). Their approach uses a Bayesian network for building a specialist system that is capable of telling, with a given probability, the *type* of the interference and the *origin* of the interference. By *type* of the interference, the authors mean whether the temporal interference is *CPU bounded* (caused by CPU intensive processes) or *Cache-related* (caused by processes that cause LLC interference). By *origin* of interference, the authors mean whether the temporal interference is caused by a process running (i) on the same CPU, (ii) on another CPU but on the same core, or (iii) on another core. The distinction of CPU and core is made since a processor based on the Intel *hyperthreading* technology is used. In that case, "same CPU" means the same hyperthread in the same core, and "another CPU on the same core" means another hyperthread on the same core. The processor taken into account is of the Intel Core i7 family. Sharing the same core means sharing L1 and L2 caches. All cores share the L3 (the LLC).

The Bayesian network is built during a *training* phase where a *process of interest* that is both CPU and cache intensive runs in isolation during 10 seconds. After 10 seconds, an interference process (which causes either CPU or cache interference) is scheduled to run simultaneously with the process of interest for another 40 seconds. After those 40 seconds, the interference process is suspended and the process of interest runs until completion. During the training phase both the inputs and outputs of the specialist system are known. The inputs (or

features) are (i) the number of executed instructions, (ii) the number of cache references, (iii) the number of cache-misses, whether there are other processes running on (iv) another core, (v) another CPU (but on the same core), or (vi) within the same CPU. Such inputs where obtained using Hardware Performance Counters (HPC) implemented on the processor's Performance Monitoring Unit (PMU). The employed sampling rate was 200 ms. The outputs of the specialist system are (i) process(es) of interest running without interference, concurrent CPU-bounded process (causing CPU-related interference on the process of interest) (ii) running on the same CPU, (iii) running on another CPU within the same core, (iv) running on another core, concurrent cache-intensive process (causing cache-related interference) (v) running on the same CPU, (vi) running on another CPU within the same core, (vii) running on another core.

During the training phase the output is known since the concurrent processes are executed in a controlled scenario. In fact, for the paper experiments, all collected data is sampled in a controlled scenario (where the location and type of all process are known). The collected data is then divided (using the k-fold method) into *training* (1/6 of the samples) and *test* (1/5 of the samples) data. By using this method, is possible to compute the accuracy of the generated Bayesian network that, in the executed experiments, was of 98.56%.

The feature selection is manual and experimented-based. For reaching an accuracy of 98.56%, the authors have employed the history of the last two sampled number of instructions, and cache-references. Regarding instruction-related data, while comparing to Cheng et al., the history of last two samples seams reasonable since Cheng et al. uses CPI, which already represents an average value.

3.2.1 Discussion

Table 3 summarizes the approaches presented by each related work for detecting or predicting temporal interference. Column **Inst.** (Instructions) identifies whether the approach uses the number of executed instructions during a period of time (or number of cycles) as input for evaluation. Column **Cache** identifies whether the approach uses the number of cache references or the number of cache misses as input for evaluation. Column **Inter.** (Interconnect) identifies whether the approach uses interconnect-related events as input for evaluation. Column **MP** (Multiprocessing) identifies whether the approach deals

Table 3: Comparison of approaches for detecting temporal interference.

Work	Inst.	Cache	Inter.	MP	Virt.	TD	Sched.
(CHENG et al., 2016)	X			X	X		
(SCHEFFEL; FRÖHLICH, 2016)	X	X		X			
Proposal	X	X	X	X	X	X	X

with multiprocessors (VCPUs or PCPUs). Column **Virt.** (Virtualization) identifies whether the approach takes into account readings of virtual processing elements counters. Column **TD** (Time-deterministic) identifies whether the approach itself is aware of time in the sense of not disturbing the temporal behavior of tasks while performing the sampling of variables or deciding whether there is interference or not. Column **Sched.** (Scheduling) identifies whether the approach uses the performance counters information in order to take scheduling decision.

Cheng et al. focus on a virtualized platform and uses metrics obtained form VCPUs. The authors aim at analyzing contention at memory devices (CHENG et al., 2016). However, the only employed metric is CPI, which is instruction-related. Scheffel et al. do not target virtualized systems. However, their approach uses both instructions and cache-related events (number of references and number of misses) for predicting temporal interference. Neither Cheng et al. nor Scheffel et al. uses interconnected-related information for interference prediction. Additionally, neither of the two approaches investigates how to keep bounded the overhead of sampling, training, and taking decisions. Both Cheng et al. and Scheffel et al. envision that such interference-related data could be used for taking scheduling decisions, however, none of the works explores that point.

This work aims at detecting temporal interference during runtime. As in Cheng et al. and Scheffel et al., the relevant features are manually selected. Such features are instructions, cache, and interconnect related data. As in Cheng et al., this work proposes to use data from virtual processing elements, which are in turn updated by their physical counterparts. Furthermore, this work aims at using MC scheduling, turning OFF non-critical components whenever the current interference makes guest OS tasks exhausts their *LO* computation time within a job.

3.3 I/O

Missimer et al. present an approach for scheduling guest OS tasks and guest-level I/O ISRs in the context of a mixed-criticality and virtualized system (MISSIMER et al., 2016). The authors employ the LINUX terminology referring to guest-level ISR as “bottom half” interrupt handler. The “top half” interrupt handler, on the other hand, represents the host-level and is handled by the hypervisor. Their approach focuses on fixed-priority systems and uses Sporadic Server (SS)s for scheduling guest OS tasks and Priority Inheritance Bandwidth-preserving Server (PIBS) for scheduling guest-level ISRs. There is no scheduling of VCPUs among PCPUs or virtual I/O devices among physical I/O devices. Instead, all physical processing elements are partitioned among domains (referred as *sandbox*) statically.

A SS is defined by (C, T), where C denotes a budget capacity and T is the server period. A SS can be implemented as a periodic task that servers sporadic guest OS tasks as long the server capacity is not empty. A PIBS is defined by U, which denotes utilization. A PIBS runs on behalf of a SS and inherits both the priority and period of the SS. The system dynamics is the following. A guest OS task that wishes to perform an I/O operation makes such a request to the hypervisor and gets blocked waiting for the operation to complete. The hypervisor then programs the physical I/O device to perform the I/O operation. When such an I/O operation completes its physical I/O device generates an interrupt that is handled by the top half interrupt handler. The top half interrupt handler wakes the PIBS responsible for such interrupt. The PIBS handles the bottom half of the interrupt (e.g. by copying interrupt-related data from the host to the guest level) and wakes up the guest OS task that is blocked waiting the I/O operation.

The authors approach combines SS and PIBS with Adaptive Mixed-Criticality (AMC) scheduling. In AMC scheduling system, both tasks and the whole system can have several criticality levels. In AMC, a task is defined as $(T_i, D_i, \vec{C_i}, L_i)$, where T_i represents the task period, D_i represents the task deadline, L_i represents the task criticality level, and $\vec{C_i}$ is a vector that represents the task computation time (one position in the vector for each criticality level). The authors define two criticality levels HI, which represents hi-criticality, and LO, which represents low-criticality. Therefore, $L_i \in LO, HI$. While applied to the system the criticality levels (also HI and LO) are referred as “modes”. The authors extended SS to have a capacity vector $\vec{C}$. In such a context, $C(LO)$ denotes the server capacity while the system is in the LO

mode, and $C(HI)$ denotes the server capacity while the system is in the HI mode. Similarly, PIBS are extended to have a utilization vector $\vec{U}$ that specifies $U(LO)$ and $U(HI)$.

In the original AMC, when the system is in LO mode all tasks run and when the system is in HI mode all LO tasks are suspended. The authors use an extension of AMC that allows for LO tasks keep running when system mode is HI. A HI-task defines its $C(HI) \geqslant C(LO)$, while a LO-task defines the opposite $C(LO) \geqslant C(HI)$. By having a $C(HI)$ non null a LO-task can keep running while the system is in HI mode. The system starts in LO mode and it switches to HI mode whenever a HI-task exhausts its $C(LO)$ before finishing its current job.

The authors' approach is named IO-AMC-rtb, where AMC-rtb denotes AMC-Response Time Bound, which is the name of the schedulability test used in AMC. The authors present the Worst Case Response Time (WCRT) for a guest OS task and for a guest-level ISR using IO-AMC-rtb.

The approach was evaluated using the Quest-V, a mode of the Quest OS that supports virtualization. The practical experiments were conducted on an Intel Core i3-2100 processor running at 3.10 GHz.

One of the experiments evaluates how I/O throughput is affected by a system mode change form LO to HI mode. The system is composed of two guest OS tasks a HI and a LO criticality one by two PIBS a HI and a LO criticality one. The two PIBS represent two Universal Serial Bus (USB) cameras that generate stream to be processed and perform interrupts. The PIBS CAMERA1 represents the HI PIBS, while the PIBS CAMERA2 represents the LO PIBS. The two guest OS tasks, served by SSs, represent applications that will consume the camera stream. The guest OS task APP1 represents the HI task, while the guest OS task APP2 represents the LO task. CAMERA1 is used by APP1 and CAMERA2 is used by APP2. The utilization of CAMERA1 for LO mode is of 0.1% ($U(LO) = 0.1\%$) while the utilization of CAMERA1 for HI mode is of 1% ($U(HI) = 1\%$). The utilization of CAMERA2 is the opposite: $U(LO) = 1\%$ and $U(HI) = 0.1\%$. In, the experiment the system executes in LO mode on the first 30 seconds, then it switches to HI mode. The results show that while executing in the LO mode CAMERA2 presents a high throughput while CAMERA1 presents a small throughput. When the system mode changes to HI, it is observed the opposite, CAMERA1 throughput becomes high while CAMERA2 throughput becomes near zero. Additional experiments compare the approach proposed by the authors, which uses SS for guest OS tasks and PIBS for bottom half interrupt handling (SS+PIBS) to a pure SS approach (SS

used for both tasks and interrupt). Results show while using SS+PIBS, the application tasks complete their jobs in regular intervals. The pure SS approach, however, presented a higher jitter in the case of *LO* tasks.

While the approach innovates presenting how co-scheduling I/O operations and tasks in a mixed-criticality system, it contains some limitations. Because the approach performs a static partitioning of PCPUs and physical I/O devices among the domains it does not deal with the problem of shared physical processing elements. Additionally, the approach proposed by the authors does not intent to reduce or control the interference caused by physical I/O devices it aims only at being aware of it. The PIBS approach for handling I/O devices solves only part of the problem allowing for I/O devices to be "scheduled" among tasks. However, it does not prevent an *LO* virtual I/O device from receiving data (e.g. a virtual Network Interface Card (NIC)) that can cause contention on the interconnect even when the correspondent PIBS is not scheduled to run, ultimately affecting *HI* tasks.

Liu et al. presents a credit-based fairing scheduling algorithm for I/O virtualization named VCF. The idea of VCF is that every domain gets a number of I/O credits for executing I/O operations and that the hypervisor can dynamically redistribute such credits among the domains. A domain that has more credits than it needs, can lend credits to another domain that needs more credits (LIU; TONG; SHEN, 2013).

VCF is based on the Xen structure of split device drivers where each driver is composed of a front-end that lies on a Domain U and by a back-end that lies on Domain 0. The guest OS running on a Domain U uses the front-end part of the driver to invoke I/O operations that are delivered to the back-end part of the driver inside the Domain 0, which is the one that accesses the physical device and executes the I/O operation. According to VCF, all Domains U receive I/O credits on each time interval t_i. The initial distribution of credits if made according to the domain weight, higher the weight is, more credits it receives. After a t time still inside the time interval t_i, an *I/O monitor* inside the hypervisor updates the number of allocated, consumed, and remained credits for every Domain U. It is assumed that during the interval t_i, a Domain U will consume its credits uniformly. It is also assumed a domain will consume the number of credits it has consumed on the last time interval. Thus, is possible to estimate at time t how much credit a Domain U will still need during t_i. The number of remaining credits in Domain U at time t is also taken into account. If is estimated in t that Domain U will require more credits, the difference between the

credits the domain still needs and its remaining credits is borrowed from a bank of credit (named B_{bank}) inside the hypervisor. On the other hand, if the Domain U has at time t more credits it will need, the extra credits is deposited at B_{bank} to be lent to other domains. If there is no credit to be lent and a domain has consumed all its I/O credits the domain will need to wait until the next time interval when the credits of all domains are replenished.

To evaluate VCF, the authors have conducted experiments using the httperf (MOSBERGER; JIN, 1998; MOSBERGER et al., 2017) benchmark. On the experiments, Domains U act as clients making HTTP requests. All Domains U, Domain 0 and Xen (version 4.1.2) are placed in the same physical machine, named PM1. Therefore, all Domains U share the same physical NIC for making the HTTP requests. PM1 is an Intel quad-core Xeon running at 2.40 GHz, equipped with 32 GB RAM, and running the Ubuntu 12.04 64-bit LINUX distribution as host OS. The same OS is used on the guest side. Another physical machine, named PM2 has used as the HTTP server. PM2 is an Intel dual core running at 2.96 GHz, equipped with 2 GB RAM, and running Ubuntu 12.04. Three experiments were conducted, comparing the original Xen scheduler (ORIG_XEN) to VCF. In all experiments, the variable measured was the connection rate (number of connections per second made with the server) of each domain, which the authors refer to that as *throughput*. In all experiments, three Domains U were used. On the first experiment, all Domains U were set with the same weight and were evaluated to distinct HTTP request rates at the clients (Domains U). While using ORIG_XEN, throughput differs (up to 50% of difference) between Domains U. While using VCF Domains U present similar throughput because VCF limits the request rate performed by each Domain U. The second experiment, set distinct weight for all domains on the ratio 1:2:3, giving the highest weight to Domain 3 and the lowest weight to Domain 1. While using ORIG_XEN, domains present similar throughput, since Xen discards such weight information. While using VCF domains with a higher weight present a higher throughput. The third experiment, set the same weight for all domains but domains perform HTTP requests using distinct rates using a 1:2:3 ratio. While using ORIG_XEN, the throughput is related to the request rate. While using VCF, the throughput is also proportional to the request rate but following a limited variation imposed by the weight.

While a credit-based approach seems interesting for co-scheduling CPUs and I/O, the authors do not explain in details how I/O credits are spent. An example is given in the case of *sending* packets of known

size using the NIC. The size of the packet, which is known at the sending, is somehow co-related to the number of credits. However, it is not explained how credits are decremented on *receiving* packets by the NIC, for example. The authors also do not describe what would be a good time interval for replenish credits (t_i) neither when, during t_i the credits shall me monitored and redistributed (t). It is not described how the credit-based approach would work for multiple I/O devices, whether each domain would be a credit amount per device class of a single credit amount for all devices.

Pellizzoni et al. propose an algorithm for computing the overall delay suffered by a task due to interference caused by I/O peripherals and an algorithm and infrastructure for coscheduling I/O transactions and tasks in a Commercial off-the-shelf (COTS) platform (PELLIZZONI et al., 2008).

The authors propose an algorithm for computing an upper bound on the cumulative delay suffered by a task due to fetches of evicted cache lines. Such cache lines are evicted due to peripheral transactions over a shared Front Side Bus (FSB). In order to compute the cumulative delay suffered by a task, the task control flow is divided into *superblocks*. Superblocks can include branches and loops but must be executed atomically. The cumulative delay is computed given the WCET of the superblock (obtained in the case there is no peripheral interference) and given the worst number of cache misses that can occur in that superblock. The proposed algorithm can give overly pessimistic results, since it accounts for the worst case delay inflicted by all peripherals in the execution time budget of each task. Additionally, the authors propose a coscheduling algorithm for tasks and I/O and a supporting infrastructure for controlling I/O bus accesses. The goal of the coscheduling is to maximize the amount of allowed peripheral traffic while guaranteeing all real-time tasks constraints. The infrastructure is composed of a Peripheral Gate (p-gate) and a Reservation Controller. The p-gate is a PCI/PCI eXtended (PCI-X) device that allows for controlling peripheral access to the gate. The Reservation Controller controls all p-gates and implements the coscheduling algorithm for tasks and I/O in Field-programmable Gate Array (FPGA).

Their proposal takes into account a mixed-criticality system scenario, which is composed of *safety critical* tasks, *mission critical* tasks, and *best effort* or software real-time tasks. Safety critical tasks are HRT tasks with high criticality such as flying control in the avionic domain. Mission critical tasks are also HRT tasks but have a lower criticality (when compared to safety critical). The authors propose to schedule

safety critical tasks blocking all I/O traffic except the one from peripherals used by the task. For mission critical, the authors propose to use coscheduling of I/O and tasks. For best effort or software real-time tasks, the authors propose to let all p-gates open as long there is time for it. The guaranteed real-time tasks (safety critical and mission critical) are further scheduling according to a hard reservation mechanism, where the reservation for each task is equal to its time budget. That ensures that a critical task will not miss its deadline as long as each job executes for no longer than the assigned budget.

The experiments for evaluating the proposal where conducted on a Intel Core2 CPU, using the Intel 975X system controller (chipset) and the Reservation Controller implemented in the Xilinx ML505 board.

The first experiment, measured the increase in the execution time of a task caused by peripheral interference. The experiment was composed of a traffic generator that produces write transactions to the main memory (and can vary the period and length of and transactions) and a task that generates a cache miss for each memory read. The task allocates a memory buffer of double the size of the CPU level 2 cache (the LLC in the case), and then cyclically reads from the buffer, one word for each 128-byte cache line. First they ran the task without the traffic generator and measured execution time and number of cache misses. Then, the traffic generator was turn on with maximum load and they have measured an increase of 43.85% in execution time.

The second experiment evaluates an Moving Picture Experts Group (MPEG) decoder that used hard reservation of resources and could be seem as a mission critical HRT task. In this experiment was also used a higher priority task that preempts the MPEG decoder every 1ms and replaces its cache contents. The code that executes within a period of the MPEG decoder was divided into 20 superblocks. For that experiment, it was measured the percentage of time that the p-gate is open during the task reservation time. Regarding the experiment on the MPEG decoder, the greedy algorithm allows for opening the p-gates at 31.21% percent of the task reservation time and the predictive algorithm allows for opening at 36.65% of the time. The theoretical bound in this case is 40.85%.

The analysis of temporal interference caused by I/O peripherals in real-time systems is essential in the embedded system scenario where the interaction with the environment is performed using I/O devices and the paper proposes a method for computing it.

The goal of the coscheduling algorithm “maximize the amount of allowed peripheral traffic while guaranteeing all real-time tasks con-

straints" does not seem to fit the Real-time (RT) scenario. Regarding RT, the goal should not be "to maximize peripheral access". It should be "to guarantee that all tasks use the I/O when they need it and to keep a bounded interference". A way to accomplished that is incorporating the I/O time in the execution time of tasks and coscheduling I/O and CPU using an RT policy.

Despite being feasible, the p-gate and Resource Controller approach require for dedicated hardware including customized PCI devices for implementing the p-gate and FPGA for the Resource Controller, which might not be the reality of all COTS. Despite using an Intel Core 2 in for the experiments, which is a dual core processor, the authors' experiments use such processor as a single core processor not running tasks simultaneously. Additionally, the approach proposed by Pellizzoni et al. only targets physical I/O devices not focusing on virtualized platforms.

Reis et al. presents an approach for time-deterministic reconfiguration of components in an embedded system scenario (REIS; FRöHLICH; JR., 2015), (REIS, 2016). Their approach executes a the reconfiguration procedure only when the OS is in *idle* state and if the reconfiguration manager evaluates that there is enough slack time for the reconfiguration to occur. Additionally, their approach suggest to power OFF components including I/O devices used by non-critical tasks and the non-critical tasks themselves in order to "create" more slack time for the reconfiguration to occur. That takes into account that the reconfiguration is required by a task of higher priority then the non-critical tasks using the I/O devices. In order to power OFF components, their approach relies on the method proposed by Hoeller et al. that proposes keeping a tree of component dependencies, including execution threads and I/O devices (HOELLER; WANNER; FRöHLICH, 2006), (HOELLER, 2007). Similarly to Pellizzoni's approach, powering OFF components also restricts the access that I/O devices have on the interconnect, managing the contention time suffered by tasks. Furthermore, the powering OFF approach does not require a dedicated hardware as the p-gate proposed by Pellizzoni et at. It only requires components to implement distinct operation modes, including an OFF mode in which components cease their operations. Similarly to Pellizzoni's approach, Reis et al. do not target virtualized platforms.

Munch et al. present an approach for I/O virtualization, called Multi-Processor I/O Virtualization (MPIOV), which focuses on mixed-criticality multiprocessed embedded real-time systems such as avionics (MüNCH et al., 2015). The idea of MPIOV is to separate one *application*

interface from another by using Non Transparent Bridge (NTB)s. An *application interface* can be either a Physical Function (PF) or Virtual Function (VF) of a PCI Express (PCIe) device that supports Single-Root Input/Output Virtualization (SR-IOV). Each *application partition* (the equivalent to a domain) will be mapped to its own application interface. The processor core where the application partition is executing will interface with its own NTB. Domain separation is achieved by configuring its NTB to allow transactions (from and to) the application interface assigned to that domain and to block all others transactions. Since separation is NTB-based, MPIOV does not require the use of IOMMU.

The MPIOV approach was evaluated in a multiprocessor setup by executing DMA transactions (reads and writes) issued by two distinct domains. The experiments measured transference time and rate for distinct transaction sizes. The authors claim that is no virtual difference between the two application interfaces evaluated neither for read nor write operations. Then, the results of one application interface where compared while using MPIOV and while not using it (NTBs configured to not separate one application interface bus address space from another). The authors show that both for transference time and transference rate the performance degradation while using MPIOV is 0.01 % being negligible.

The MPIOV approach seems interesting since it allows domain separation without requiring IOMMU. However, MPIOV was not evaluated according to separation (e.g. reduction of the contention time). It was only evaluated according to performance (transference time and rate).

3.3.1 Discussion

Table 4 summarizes the approaches presented by each related work for dealing with I/O virtualization. Column **Inter.** (Interconnect) identifies whether the approach uses some technique to restrict access from the devices to the interconnect. Column **Sched.** (Scheduling) identifies whether the approach performs either CPU and I/O devices co-scheduling or I/O-aware scheduling. Column **QoS** (Quality-of-Service (QoS)) identifies whether distinct domains can access virtual devices with distinct QoS (e.g. distinct transmission rates in the case of a virtual NIC) or with distinct priorities (e.g. whenever two domains try to use a shared I/O device, the I/O device is hand over to the domain of higher priority). Column **SW** (Software-based) identifies whether

Table 4: Comparison of approaches for dealing with I/O virtualization.

Work	**Inter.**	**Sched.**	**QoS**	**SW**	**TD**	**Virt.**
(MISSIMER et al., 2016)		X	X	X	X	X
(LIU; TONG; SHEN, 2013)		X	X	X	X	X
(PELLIZZONI et al., 2008)	X	X			X	
(REIS, 2016)	X			X	X	
(MüNCH et al., 2015)	X*			X*	X	X
Proposal	X	X	X	X	X	X

the approaches is mainly software-based not requiring dedicated hardware (hardware not found in the original COTS platform). Column **TD** (Time-deterministic) identifies whether the approach focus on time determinism rather than on performance. Column **Virt.** (Virtualization) identifies whether the approach focus on virtualized platforms.

Missimer et al. does not restrict the access of physical I/O devices to the interconnect. Instead it is based on co-scheduling guest OS tasks and guest-level ISRs. QoS can be achieved through the HI and LO criticality-levels, which can address either tasks or guest-level ISRs. However, in this case, QoS is only intra-domain, since processing elements are statically assigned to domains in Quest-V hypervisor. The scheduling technique presented by Missimer et al. is software-based, time-deterministic (since it employs RT algorithms), and focus on virtualized platforms.

Liu et al. also does not restrict the access of physical I/O devices to the interconnect. Instead, it employs a credit-based scheduling approach, which is aware of I/O devices (domains have I/O credits to spent), time-deterministic, and focuses on virtualized platforms. QoS can be achieved through the weight each domain has, that defines how much I/O credits it is going to receive at each replenishment period.

Pellizzoni et al. restricts the access of physical I/O devices to the interconnect by using dedicated hardware, the p-gates that control the access to the PCI bus and the reservation controller that manages them. Using such a dedicated hardware it implements the co-scheduling of I/O devices and tasks by allowing DMA transactions only when the p-gates are open. Such an approach is time-deterministic, however it does not target virtualized platforms.

Reis proposes to restrict the access of physical I/O devices to

the interconnect by powering OFF such physical I/O devices, however it does not explores the co-scheduling of tasks and I/O devices. It focusing on time-deterministic operations in the context of components reconfiguration, and it does not focus on virtualized platforms.

Münch et al. does not restrict the access of physical I/O devices to the interconnect in the sense of eliminating DMA transactions as in (PELLIZZONI et al., 2008) or (REIS, 2016). However, by configuring interconnect bridges, it allows for a given domain (that is assigned to a bridge) to filter transactions that do not use the VF designated to that domain. NTBs are available in COTS, therefore the MPIOV approach is mainly static. On the other hand, it relies on SR-IOV PCIe extension, not available in all physical I/O devices. Such an approach is time-deterministic and focuses virtualized platforms. However, it does not provide QoS nor is integrated to a co-scheduling mechanism.

This work proposes powering OFF/ON physical I/O devices in order to restrict their access to the interconnect. It proposes to use dedicated VCPUs, named Input/Output Virtual Central Processing Unit (IOVCPU) for handling I/O access to the interconnect. An IOVCPU runs on behalf of a virtual I/O device of a domain and can either retrieve DMA transaction data in the case of a "read" operation or to be used for issuing a new DMA transaction in the case of a "write" operation. IOVCPUs are co-scheduled with execution VCPUs following MC RT policies. Following MC scheduling policies, IOVCPUs of non-critical domains are suspended whenever the system is in HI criticality level, realizing a QoS control. In the practical point-of-view, IOVCPUs can also be used for ensuring that a domain cannot perform I/O operations using parameters (e.g. bytes to be written) above what is agreed for that domain.

3.4 INTERRUPTS

Beckert et al. present an approach for reducing interrupt latency on hypervisors (BECKERT et al., 2014). Interrupt latency, in this case, is defined by the elapsed time between the arrival of an interrupt in the hardware and the time it is started to be serviced in a specific hypervisor domain, thus including the time to route the interrupt from the hypervisor to the guest OS in the domain. Usually in real-time hypervisors, interrupts regarding one domain are serviced within the time slot such domain has to execute. However, in the case an interrupt arrives at the end of the time slot of a domain, it will be serviced only

when the domain is scheduled to run again, which in turns increase interrupt latency. The idea behind the approach proposed by Beckert et al. is to allow interrupts to be serviced in time slots of others domains as long their interference is kept to a maximum value that does not compromise the temporal isolation among partitions (domains). Their work employs the idea of "sufficient temporal independence" as oppose to total time isolation among domains. The idea is that the system is allowed to miss some deadlines as long the number of misses is bounded. A monitoring function, implemented at the hypervisor's interrupt mechanism evaluates at runtime whether an interrupt can be handled in the slot of another domain.

The approach of Beckert et al. is evaluated at a modified version of the uC/OS-MMU (of the uC family of microkernels) in an ARM926ejs (ARMv5) processor running at 200MHz. The approach was evaluated in three scenarios while generating 40% of interrupts at the time of the appropriate domain and 60% outside the domain slot. There were generated a total of 15000 interrupts. In the first scenario, in which the monitoring is disabled the average latency was around 2500us. In the second scenario, in which the monitoring is enabled, but some interrupts are delayed (happened outside the slot of the domain and are handled only when the domain is scheduled again) the average latency was around 1200us (around 2 $\times$ of improvement compared to the unmonitored case). In the third scenario, in which the monitoring is enabled and all interrupts can be handled either in the domain slot or using the slot of others, the average domain was around 150us (around 16 $\times$ of improvement compared to the unmonitored case). While the approach has obtained promising results, it demands for an (off-line) analysis to ensure not only schedulability of tasks and interrupts within a partition but also to ensure that the interference caused by handling interrupts outside the partition does not break temporal independence of such a partition. That, in turn, demands arrival functions for every event in the system.

Tu et al. propose an approach for delivering hardware interrupts to guest OSes named Direct Interrupt Delivery (DID). DID targets full-virtualization, not requiring modifications on the guest OS. The idea of DID is to reduce the number of VM exits (avoiding involving the hypervisor) while delivering interrupts. DID deliveries interrupts from SR-IOV devices, virtual devices, and timers, directly to the target VM/domain. The DID approach relies on the Intel VT-x extensions, using a Virtual Machine Control Structure (VMCS) for each VM. Additionally, DID is implemented on LINUX Kernel-based Virtual Machine (KVM).

In the case of PCIe devices that support the SR-IOV extension, a VF of the device is assigned to a VM. In that way, that VM can issue operations to the device directly (through the VF). Also via the VF, the device can interrupt the VM directly. In this case, the strategy employed by DID is following. If the system is running on guest mode (according to the VT-x extensions) and an interrupt is generated by the device, the interrupt of the VF goes through the PCIe hierarchy, reaches the IOMMU, and it is delivered to the Local APIC (LAPIC) of the target VM. However, if the system is running on host mode, and an interrupt is generated by the device, the hypervisor performs a *virtual interrupt injection* and then resumes the VM target of the interrupt. Such a virtual interrupt injection is performed by using a self Interprocessor Interrupt (IPI) where the PCPU sends an IPI to itself. By performing virtual interrupt injection, which converts a virtual interrupt into a physical one, all interrupts are forced to go through the Advanced Peripheral Interrupt Controller (APIC). That avoids the priority inversion problem that can happen when a lower-priority virtual interrupt may override higher-priority directly delivered interrupt.

KVM abstracts each virtual device with a Virtual Device Thread (VDT). A VDT runs on host mode, checks for physical interrupts, transforms them into virtual device interrupts, and delivers them to the target VM. In the case of virtual devices, the strategy employed by DID is following. For all virtual devices, physical interrupts are converted by the VDT into *IPI-based virtual devices interrupts*. Usually, VDTs run on a dedicate PCPU, separated from the VMs. Therefore, if the system is running on guest mode, the VDT sends the virtual device interrupt to the target VM by using an IPI. However, if the system is running on host mode, the hypervisor accepts the IPI sent by the VDT on the behalf of the VM, converts it into a *virtual interrupt*, and performs a virtual interrupt injection using a self IPI, as described previously.

In the case of timers interrupts, the strategy employed by DID is following. A specific PCPU D is designated to store the timers set up by the hypervisor. When a VM M is scheduled to run on a PCPU C, the hypervisor install the timer(s) related to M on C LAPIC timer. When M is descheduled, its timer(s) are removed from C and installed on D. If the system is running on guest mode, a timer interrupt interrupts C directly. If the system is running on host mode, a timer interrupt occur on D, then the hypervisor creates a virtual interrupt and performs virtual interrupt injection on M when M resumes its execution.

The DID approach was evaluated for several benchmarks while running with DID and running without it (interrupts delivered by the

hypervisor). The *Cyclictest* benchmark, developed by the authors, evaluates interrupt invocation latency, which is the time difference between the interrupt arrival on the hypervisor and the time it start to being serviced on the guest OS. The experiments show that, while using DID, the interrupt invocation latency decreases by 80% (from 14 us to 2.9 us). Another benchmark used in the experiments, was the *Iperf* that evaluates Transmission Control Protocol (TCP) throughput between two machines. The results show that DID improves intra-machine TCP throughput by up to 21%. Furthermore, the authors have evaluated DID while using the *Memcached*, which is a key-value store server. As evaluation benchmark, the authors have emulated a "twitter-like" workload (140 characters). The results show that, while using DID, the throughput of a VM running Memcached was improved by a factor of 3 as a result of reducing the VM exit rate from 97 K/s to less than 1 K/s.

While the experiments conducted using DID show promising results, the approach targets performance instead of time-determinism. The authors claim that the approach does not need the Intel APICv (APIC virtualization). However, their approach relies on the Intel VT-x extensions, which are vendor-specific.

3.4.1 Discussion

Table 5 summarizes the approaches presented by each related work for dealing with interrupt virtualization. Column **TD** (Time-deterministic) identifies whether the approach focus on time determinism (bound the interrupt handling time) rather than on performance. Column **MC** (Mixed-Criticality) identifies whether the approach supports for mixed-criticality. Column **SW** (Software-based) identifies whether the approaches is mainly software-based not requiring dedicated hardware (hardware not found in the original COTS platform). Column **Virt.** (Virtualization) identifies whether the approach focus on virtualized platforms.

Beckert et al. supports for time-determinism in the sense all interrupt patterns are analyzed off-line and their impact on domains is computed (BECKERT et al., 2014). However, their work tolerates bounded deadline losses, which is not suitable for HRT systems. Their approach uses a software-based implementation, changing the hypervisor to allow or deny interrupt interposing (the handling of the interrupt in another domain slot). Their work only supports one criticality level.

Tu et al. focus on performance instead of time-determinism and

Table 5: Comparison of approaches for dealing with interrupts virtualization.

Work	TD	MC	SW	Virt.
(BECKERT et al., 2014)	X*		X	X
(TU et al., 2015)			X	X
(MISSIMER et al., 2016)	X	X	X	X
Proposal	X	X	X	X

also supports only one criticality level (TU et al., 2015). Their technique of virtual interrupt injection is implemented in software, as long the underlining hardware support for IPI, which is the reality of mostly multiprocessors.

Missimer et al. (presented on Section 3.3) employs PIBS for interrupt handling at guest-level (MISSIMER et al., 2016). Their approach supports for two criticality level both for tasks and interrupts handling.

This work proposes to use dedicated VCPUs, named IOVCPU for handling interrupts. Such IOVCPUs will handle guest-level interrupts signalized by the host-level in a time-deterministic way. As in Missimer et al., this work proposes using two criticality levels for such IOVCPUs, HI and LO.

3.5 MEMORY

Kim et al. present two approaches for dealing with cache management for virtualized platforms, Virtual LLC (vLLC), and Virtual Coloring (vColoring) (KIM; RAJKUMAR, 2016). The vLLC approach targets guest OSes that support page-coloring, while vColoring targets guest OSes that do not support page-coloring. The idea of vLLC is to provide each VM with a portion of the physical LLC. A vLLC maps Guest Physical Page (GPP)s to Host Physical Page (HPP)s such that *guest colors* are mapped to their corresponding *host colors*. The idea of vLLC assumes that the guest OS is able to query (by using architecture specific instructions such as `CPUID` in the x86 architecture) cache parameters (such as size, number of sets, and associativity) at boot time. In the idea of vColoring, a guest OS task can be aware of coloring without the guest OS being color-aware. That is achieved by mapping the guest OS task GPPs to HPPs that have the same color. Both approaches assume that tasks do not migrate from one VCPU to another, and that every

VCPU is pinned to a specific PCPU. Still, there can be more than one VCPU competing for the same PCPU but only if these VCPUs belong to distinct VMs.

vLLC and vColoring were implemented in the LINUX KVM hypervisor. In the first set of experiments, the authors have evaluated vLLC and vColoring in x86 and ARM platforms using a single guest OS task and a single VM. The employed guest OSes were the LINUX/RK (that supports page-coloring), a "vanilla" LINUX (no page-coloring support), and MS Windows Embedded (also no page-coloring support). A guest OS task, named *latency*, traverses a randomly-ordered linked list (the task's working set), which has the size of half the LLC. The results show that assigning more colors to the guest OS task reduces the task execution time. Such reduction on execution time reaches a plateau when the number of colors assigned to the task is enough to make the task's working set fit in the LLC entirely.

In the second set of experiments, only the vLLC approach was evaluated and only in the x86 platform. The second set of experiments evaluated PARSEC benchmarks, except the *dedup* and *facesim* that focus mainly on disk operations. In this set of experiments, each PARSEC benchmark ran with three instances of the *latency* guest OS task simultaneously. The idea was to use the *latency* guest OS task to create interference on the benchmark and measure the benchmark execution time variation. Firstly, the benchmark and the instances of *latency* ran in distinct VCPUs, then (on another execution) on the same VCPU. When vLLC is enabled, all colors but one are assigned to the benchmark, while the instances of *latency* share the remaining one color. When vLLC is disabled, the benchmark and the instances of *latency* share all colors. In the multiple VCPUs experiment, while vLLC is disabled there is up to 30% of execution time increasing, versus up to 2% of execution increase time increasing while vLLC is enabled. In the single VCPU experiment, while vLLC is disabled the execution time increases up to 15%, versus unperceptive execution time increasing (for mostly benchmarks) while vLLC is enabled.

In addition to the practical experiments, the authors have evaluated their cache management scheme that defines how tasks are allocated to VCPUs, how colors are allocated to VCPUs, and how the replenishment budget of VCPUs is defined. Their approach is compared to six other approaches that combine task-to-VCPU allocation and cache-to-task allocation heuristics. The employed task-to-VCPU allocation heuristics are Best-fit Decreasing (BFD), Worst-fit Decreasing (WFD), and First-fit Decreasing (FFD). The employed cache-to-task allocation

heuristics are Complete Cache Partitioning (CCP) (where each task has a distinct set of colors), and Complete Cache Sharing (CCS) (where all tasks in the same VCPU share all colors). Therefore, the six approaches are, BFD+CCP, WFD+CCP, FFD+CCP, BFD+CCS, WFD+CCS, and FFD+CCS. Their approach outperforms six other approaches, resulting in a smaller VM total utilization.

While the approaches proposed by Kim et al. advance the state-of-the-art for cache management in virtualized systems, their approaches present some limitations. The assumption of statically allocating each guest OS task to a VCPU suggests that all tasks are single-threaded. By doing that, the authors can assume that a set of colors is assigned to a VCPU instead to the domain. Their approach also assumes that every VCPU is pinned to a specific PCPU. That, on the other hand, might be interesting since avoids eviction of cache lines due to VCPU migration. Additionally, their experiments suggest that their approach was not evaluated while executing two VMs (which distinct guest OSes) on the same host platform. With more then one VM, VCPUs would compete for the same PCPUs potentially causing a higher number of cache misses, especially while page-coloring was not employed.

Ma et al. present two approaches for dealing with cache management for virtualized platforms, Cache Partitioning (CAP), and Page Table Prefetching (PTP) (MA et al., 2013). Both approaches were implemented in LINUX KVM and evaluated while using an x86 host platform. In all considerations and for all evaluations, two VM instances are taken into approach, one executing an RTOS and another executing a General Purpose Operating System (GPOS). The employed RTOS is a LINUX patched with the Preempt-RT extension and focus on running Soft Real-time (SRT) tasks. Each VM is pinned to a specific PCPU in a dual-core processor.

CAP is based on page-coloring and performed statically. As KVM is a hosted hypervisor, each VM runs as a process in the LINUX host. Therefore, page-coloring has implemented by modifying the host operating system's memory page allocator.

PTP is a dynamical approach. It implements a periodic task that runs on a dedicated core and its purpose is to make available on the LLC the contents that are going to be needed by the RTOS. The PTP module runs on the core dedicated to the GPOS. Each time PTP module runs on the host LINUX, it gets the process id (PID) of the RTOS, locates the page table of the RTOS and reads it. Reading the page table of the RTOS will bring such pages to the LLC (and inner cache levels) of the core pinned to the GPOS. There is no mechanism to prevent the

guest OS tasks of the GPOS from evicting cache lines updated by PTP while performing memory access. Still, the authors claim that PTP is effective since it runs periodically.

The Cyclitest program, a *Cache Load* program, the CPU-SPEC-2006, and the Sysbench benchmarks were used for evaluating CAP and PTP approaches. The Cyclictest sets up a timer to run periodically and, each time it runs it reads a time stamp and computes the time difference between the current and last sampled time stamp. Such computed time interval (real interval) is compared to the time interval programmed on the timer (ideal interval). The difference between the real internal and the ideal interval will give the local latency. A *Cache Load* program, designed by the authors, runs on the GPOS and moves data from one place to another in such way that it cases cache interference. The CPU-SPEC-2006 benchmark is composed of a set of programs that evaluate mostly CPU performance. The Sysbench is a memory intensive benchmark.

The approaches evaluation was conducted on an Intel i5 processor (two cores, four hyper threads) running at 2.67GHz and on a computer equipped with 2 GB of main memory. The Intel i5 has 128 KB of L1 cache, 1 MB of L2, and 8 MB of L3 (LLC) cache. The host LINUX kernel version was 2.6.33.4-rt 20. Each VM was pinned to run in one core and configured with 512 MB of main memory. Additionally, the authors have disabled USB support of the host machine to reduce the number of generated interrupts.

The CAP approach was evaluated on three experiments. The first experiment executed Cyclictest on the RTOS and the *Cache Load* program on the GPOS. CAP was set to give half of the LLC to each VM. The experiment ran with and without (*base* case) CAP and the latency of Cyclictest was measured. While using CAP, the maximum latency is reduced by around 21% (222 us vs. 182 us). Additionally, CAP reduces the most frequent latency from a range from 66 to 85 us to a range from 31 to 70 us. The second experiment executed Cyclictest on the RTOS and the CPU-SPEC-2006 benchmark on the GPOS. It was measured latency of Cyclictest and execution time of the benchmark while changing the LLC ratio distribution. The latency measured by Cyclictest decreases with the increasing of LLC allocated to the RTOS. It decreases about 40% when the ratio goes from 1/8 to 4/8 (half of LLC allocated to the RTOS). Beyond that, the reduction on the latency is not expressive. From 4/8 to 7/8 the latency decreases about 3% only. CPU-SPEC-2006 execution time increases when more LLC is allocated to the RTOS but not much since the benchmark is mostly

CPU intensive. The third experiment executed Cyclictest on the RTOS and the Sysbench benchmark on the GPOS. It was the execution time of the benchmark (but not the latency of Cyclictest) while changing the LLC ratio distribution. From 4/8 to 6/8 the execution time of the benchmark increased by 3 times.

The PTP approach was evaluated on two experiments. The first experiment executed Cyclictest on the RTOS and the *Cache Load* program on the GPOS. The experiment ran with and without (*base* case) PTP and the latency of Cyclictest was measured. While using PTP, the maximum latency was reduced by around 56% (222 us vs. 142 us). The second experiment ran Cyclictest on the RTOS and CPU-SPEC-2006 benchmark on the GPOS while changing the period that PTP module executes. Increasing the period of PTP will cause the latency experienced by the RTOS to increase. For a PTP period of 200 us, the maximum latency is 142 us. For a PTP period of 10 ms, the maximum latency is 209 us, which is near to the *base* case where no PTP is used (222 us). The performance of GPOS, the other hand, is inversely proportional to PTP period since PTP "steals" CPU cycles of GPOS tasks.

In the presented experiments, while comparing PTP with CAP using 4/8, PTP can reduce the maximum latency experienced by the RTOS about 35% more than CAP. However, PTP caused higher performance degradation on GPOS tasks.

While the CAP approach based on page-coloring is well accepted and implemented by other works, the efficacy of the PTP approach seems questionable. There are no mechanisms to prevent the GPOS or the RTOS from evicting cache lines updated by PTP while performing memory accesses. Additionally, the choice of reading the page table of the RTOS instead of other data of the RTOS is not justified in the paper. Fetching page tables in any order is not the same as fetching the most recently used page tables or the most recently accessed addresses (that is the Translation Lookaside Buffer (TLB) function). Furthermore, the core that runs the RTOS has its own private TLB. The PTP approach was not evaluated for more than one RTOS running simultaneously so it is not clear how it would scale. In fact, the approach was not evaluated for more then one RT guest OS task on that RTOS. Moreover, measuring time-determinism only by using Cyclictest seems to be inadequate while evaluating how much an RTOS is affected by cache interference since Cyclictest does not perform memory operations. A better application to run on the RTOS would be running a program that reads and writes the memory so it would be affected by cache interference directly.

Table 6: Comparison of approaches for dealing with memory virtualization.

Work	**Apr.**	**TD**	**Sched.**	**Part.**	**SW**	**Virt.**
(KIM; RAJKUMAR, 2016)	vLLC	X		S	X	X
(KIM; RAJKUMAR, 2016)	vColoring	X		S	X	X
(MA et al., 2013)	CAP	X		S	X	X
(MA et al., 2013)	PTP	X*			X	X
Proposal		X	X	S	X	X

3.5.1 Discussion

Table 6 summarizes the approaches presented by each related work for dealing with memory virtualization. Column **Apr.** (Approach) identifies the approach name in the case of works that present more than one approach. Column **TD** (Time-deterministic) identifies whether the approach focus on time determinism rather than on performance. Column **Sched.** (Scheduling) identifies whether the approach uses memory hierarchy information (e.g. number of cache misses) into account for taking scheduling decisions. Column **Part.** (Partitioning) identifies the employed cache partitioning approach, which is either static (S), dynamic (D), or none (left in blank). A dynamic partitioning approach in this case, means that a cache partition can be reassigned at runtime. Column **SW** (Software-based) identifies whether the approaches is mainly software-based not requiring dedicated hardware (hardware not found in the original COTS platform). Column **Virt.** (Virtualization) identifies whether the approach focus on virtualized platforms.

Both vLLC and vColoring approaches proposed by Kim et al. are time-deterministic, software-based and focus on virtualized systems. vLLC virtualizes the LLC statically, and vColoring performs page-coloring cache partitioning also statically. The CAP approach, proposed by Ma et al. also uses page-coloring and performs the partitioning statically, and is time-deterministic. The PTP approach, on the other hand, does not perform partitioning, instead it employs a dedicated task on a dedicated core to fetch page tables of the SRT domain periodically. Since there is no isolation between the LLC areas each domain is allowed to use, PTP is not time deterministic.

This work proposes on using a cache partitioning also based on page-coloring, and also performed statically. It targets guests that are no color-aware thus the hypervisor handles the allocated colors. This work envisions the color distribution by domain, VCPU, and guest OS task as detailed in Chapter 4. During the modeling phase, HPC measured events will be used for computing the required time budget of guest OS tasks, and the whole system. Average values will be used for computing the requirements for the LO criticality level and worst-case values for computing the requirements for the HI criticality level.

4 PROPOSED TECHNIQUES USAGE

This chapter presents the first part of this work proposal. It presents a set of techniques and explains how they can be used for supporting time-deterministic virtualization. It starts by focusing on interference detection, then it investigates I/O aspects, interrupts, and memory. For each one of these aspects, it presents the sources of temporal non-determinism that is usually modeled as some sort of interference. Also for each aspect, after showing the causes of the interference, it is explained a technique to mitigate the interference itself.

The main contribution of this chapter does not consist in the techniques themselves, as many of them were introduced by related work, but on their application in the time-deterministic virtualization scenario.

4.1 DETECTING INTERFERENCE

Detecting interference can be divided in perceiving the existence of the interference and deciding whether it is tolerable or "too high" to the point of disrupting the time requirements of critical tasks. The infrastructure for perceiving the interference depends on hardware and software support.

Regarding hardware support, informational data can be provided per I/O device, per PCPU, per interconnect data, or no at all. An example of per I/O device data are the ones provided by statistic registers of some NICs that show the number of transmitted and received bytes. An example of per PCPU data are the *core* features provided by the processor PMU such as the number of accesses and misses to the fist level (L1) of cache, or the number of accesses and misses to the LLC caused by that processor. An example of per interconnect data are the *uncore* features such as the number of uncore cycles that a given DRAM channel is occupied with read requests (INTEL, 2016b).

Regarding software support, interference is perceived per domain, usually. In that case, it is desired to know which physical I/O devices and PCPUs are assigned to which domains. The Unified Modeling Language (UML) object diagram of Figure 5 illustrates the I/O devices and CPUs a domain owns. As one can notice, the domain don't own the physical processing elements (physical I/O devices and PCPUs) directly. Instead, it owns the virtual counterpart of them: virtual processing

elements (virtual I/O devices and VCPUs). The virtual processing elements, in turn, can be *in use* by the domain (*active* field set as *true*) or *not in use* by the domain (*active* field set as *false*).

Potentially, the assignment of virtual processing elements to a domain can occur before guest OS booting (e.g. at domain creation) or while the guest OS is running (which represents the *hot plugging* of processing elements). This work takes into account the former case only since the dynamic creation of virtual processing elements would change the load of their physical counterparts potentially disrupting the time-requirements of critical domains. Similarly, the mapping between virtual processing elements and physical processing elements can be performed before guest OS booting or while the guest OS is running. A static (pre guest OS boot) mapping can be performed for physical processing elements that are exclusive to a certain domain, which is the case, for example, of PCPUs dedicated to a critical domain. A dynamic (while guest OS is running) mapping is performed for shared physical processing elements, for example, the assignment of a PCPU to a VCPU that occurs whenever the hypervisor schedules VCPUs. Another example is a shared physical NIC: each domain will have its virtual NIC (the virtual counterpart of the physical NIC) and the hypervisor will handle which domain is using the physical NIC at a given time.

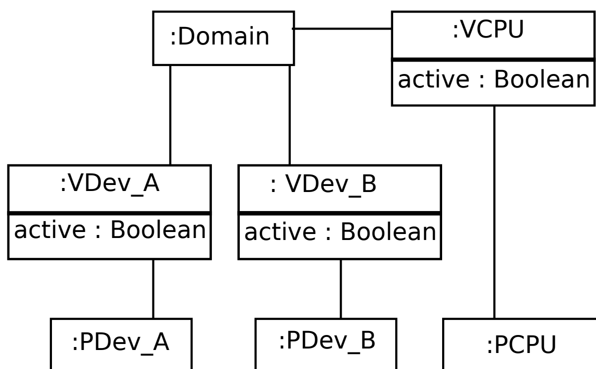


Figure 5: Domain Resources Mapping.

In cases where there is no dedicated hardware support such as PMU events or per I/O device counters, alternative approaches can be

employed for interference detection. For example, having a profile of the execution time of virtual processing elements methods (previously obtained) and at guest OS runtime monitoring such execution times. The increasing of execution time could be perceived as interference. While perception of interference on the guest OS task granularity (instead of domains) is potentially interesting, it is usually unfeasible since it would require the hypervisor to be aware of such tasks complicating its design and implementation.

Still regarding software support, the knowledge regarding interference can be either *from a global point-of-view* or *from a specific point-of-view*. In the former case, one knows how much each domain contributes to the general system load. In the latter case, one knows how much some domains (usually the critical ones) are suffering from interference caused by others. Regarding global knowledge, one could, for example, update for each virtual processing element how much it contributes the general system load (e.g. how many bytes each virtual NIC has sent in the last minute) by using the resources mapping of Figure 5. Regarding local knowledge, one could, for example, measure the LLC misses suffered by the PCPU that is in use by a critical domain.

Regardless of the technique used to perceive interference, one needs to judge whether the interference is acceptable or too high (risking disrupting the time constraints of critical tasks). In general, two approaches can be employed: static thresholds or training techniques. Are examples of static thresholds:

- physical I/O device in use by a non-critical task is presenting a high throughput (N% of its maximum transmission rate)
- the cache miss ratio (misses/access) from the point-of-view of one PCPU used by a critical domain has increased more than N%
- the interconnect is operating at more than N% of its maximum transmission rate
- a consecutive number of jobs of a critical task are presenting a jitter increasing of N% of the job execution time

Alternatively (or in conjunction with) using static thresholds, one can also employ training techniques. Such techniques aim at learning what a normal behavior looks like and, afterwards, judging non-normal behaviors as excessive interference. In the *training* phase critical domains are executed alone and the hypervisor collects data to define a *normal behavior*. Then, in a second moment, non-critical domains are

introduced and the hypervisor *monitors* how much the interference *diverges* from the normal behavior. The thresholds used to decide whether there is too much interference can be either computed from a reservation/QoS policy of resources (e.g. a certain critical domain requires a specific transmission rate of a NIC) or based on past experiences (e.g.: "every time the interconnected reached N% of its maximum transmission rate there were M% of deadline losses").

4.2 I/O

Regarding I/O, to enable time-deterministic critical tasks, a hypervisor must deal with the temporal interference caused by I/O operations that occurs mainly because contention time suffered by PCPUs or physical I/O devices in use by critical tasks whenever they need to wait for other devices to complete their operations. In the case of asynchronous I/O operations, the temporal determinism of I/O operations will also depend on the interrupt handling mechanism. This section focuses on DMA transactions issued by I/O devices. Section 4.3 explains how to handle I/O interrupts deterministically at the host-level.

Figure 6 shows the case in which the critical task T_2 has its access to memory delayed because it needs to wait for the DMA transaction issued by T_1, another guest OS task from another domain (possible a non-critical one), to finish before it can have access to the memory interconnect. In this case, and all others presented in this work, a multiprocessed platform is taken into account. Therefore, T_2 runs in parallel to T_1 since it is assumed those tasks are allocated to distinct PCPUs: PCPU_2 and PCPU_1 respectively. In the example, PCPU_2 is the element that suffers from interconnect contention.

In Figure 6, T_1 has issued a DMA transaction on time instant A that finishes on instant C. T_2 wishes to access the main memory on time instant B but first it needs to wait for the DMA transaction issued by T_1 to finish. Therefore, the contention time that T_2 suffers is the time interval between the instants B and C. The extent of the contention time depends on the access policy implemented by the arbiter of the interconnect as well from the number of processing elements (I/O devices or processors) sharing the interconnect. The behavior shown in Figure 6 is typical of many interconnects such as PCI buses that wait for a DMA transaction to complete before grating bus access to another processing element.

The general approach for handling I/O interference is to con-

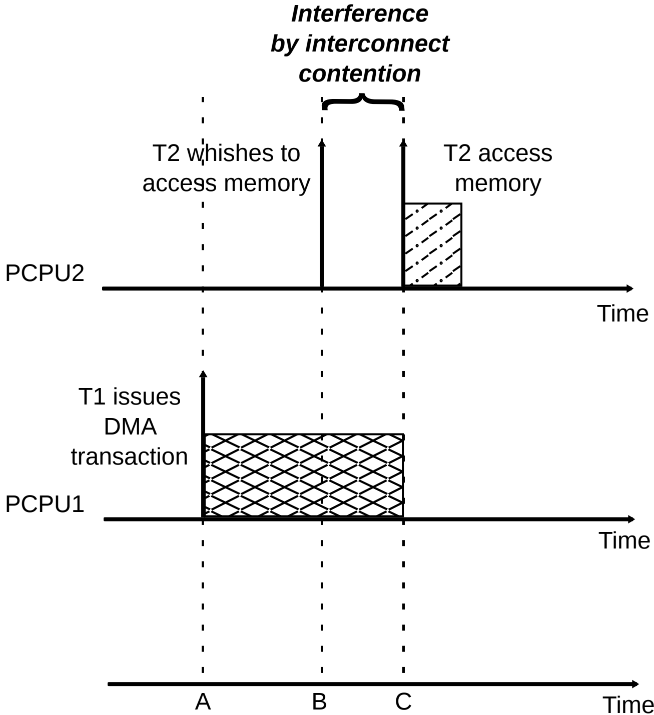


Figure 6: Interference due to contention caused by I/O device.

tinuously measure interference, to power OFF processing elements of non-critical domains whenever the interference is high, and to power such processing elements back ON whenever the interference becomes low again. The specific semantics of powering OFF/ON processing elements depends of the processing element itself and whether one is dealing with the physical processing element or its virtual counterpart. This work uses the following semantics convention:

- **Physical I/O devices**
  Physical I/O devices can implement several modes for energy management ranging from a *FULL* mode where all resources are ON to a complete *OFF* or *SLEEP* mode where all resources are powered OFF. Some intermediate states (e.g. *STANDBY*) can exist

as well. This work takes into account the two extremes (FULL and OFF). Therefore, powering a physical I/O device ON means to bring it to FULL mode, while powering a physical I/O device OFF means to bring it to OFF mode.

- **Virtual I/O devices**
  The OFF/ON semantics for virtual I/O devices depends on the underlining physical I/O device energy support as well on the operation that one is using. For example, to power a virtual NIC OFF could mean to suspend the guest OS tasks that are using the virtual NIC for sending frames. For the same virtual NIC, a power OFF could mean, for example, using some *filtering* on the physical level discarding frames that are addressed to the domain that had powered the virtual device OFF. Conversely, to power the same virtual NIC ON would mean to resume the guest OS tasks that are waiting for sending frames and to stop filtering frames that target the domain that owns the virtual NIC.

- **PCPUs**
  The OFF/ON semantics for PCPUs is the same than for physical I/O devices. However, powering ON and OFF PCPUs in the virtualization context is less common and is not taken into account in this work. Instead, the usual approach is to schedule another VCPU to a given PCPU instead of powering it OFF.

- **VCPUs**
  In the context of this work, to power a VCPU OFF means to *suspend* its execution, allowing the underling PCPU to be reassigned to another VCPU. Conversely, to power a VCPU ON means to *resume* its execution so it will be able to be assigned to some PCPU. Such assignment will depend on the VCPU scheduling policy implemented by the hypervisor and usually will not occur immediately after the resume of the VCPU.

Additionally, the following rules are used in the case of shared I/O devices (non-critical domains):

1. Powering OFF a virtual I/O device, will cause the aforementioned blocking and filtering policies (if any) to be enabled for the domain that owns the virtual I/O device. In the case the physical I/O device assigned to the virtual I/O device being powered OFF is shared with others virtual I/O devices (from other domains), the physical I/O device is kept ON. In the case the virtual I/O device

being powered OFF is the only user of its physical counterpart, the physical I/O device is also powered OFF.

2. Powering OFF a physical I/O device, will cause all its client virtual I/O devices to be powered OFF followed by the powering OFF of the physical I/O device itself.

3. Powering ON a virtual I/O device, will disable the blocking and filtering policies. Additionally, if the virtual I/O device being powered ON is the first client of the given physical I/O device and the physical I/O device is OFF, it will power ON the physical I/O device as well.

4. Powering ON a physical I/O device, will power the device ON *without* powering ON its client virtual I/O devices.

The UML communication diagram of Figure 7 shows the power OFF/ON dynamics that is considered by this work. In the figure, the hypervisor has taken the decision of powering OFF three I/O devices simultaneously: $VUART_{00}$ (a Virtual UART (VUART)), $VNIC_{10}$ (a Virtual NIC (VNIC)), and $PUART_1$ (a Physical UART (PUART)). Accordingly to Rule 1, the invocation of $power(OFF)$ on $VNIC_{10}$ will power OFF the virtual I/O device but will not power OFF $PNIC_0$ since it is shared with $VNIC_{20}$. Following the Rule 2, the invocation of $power(OFF)$ on $PUART_1$ will power OFF $VUART_{10}$ and $VUART_{20}$ then, it will power OFF of $PUART_1$ itself. The invocation of $power(OFF)$ on $VUART_{00}$ will cause $PUART_0$ to power OFF, accordingly to Rule 1. The worst case of interference caused by I/O devices assigned to critical domains must be taken into account at design time, therefore powering OFF an I/O device assigned to a critical domain at runtime aiming to reduce the interference is a design error. Anyway, the powering OFF of an I/O device assigned to a critical domain can occur when the whole domains is being shut down.

As mentioned in the beginning of this section, the general approach for handling I/O interference is to power OFF processing elements of non-critical domains whenever the interference is high and, as interference reduces, to power processing elements back ON. The specific powering OFF/ON order can be obtained from a design phase and coded statically or can be learned and changed as new domains are introduced (on distinct executions of the whole system).

Following are examples of powering OFF/ON order. **P** stands for physical I/O device, **V** for virtual I/O device, and **U** for VCPU. **PDev-OFF/PDev-ON** means to power physical I/O devices OFF/ON, **VDev-**

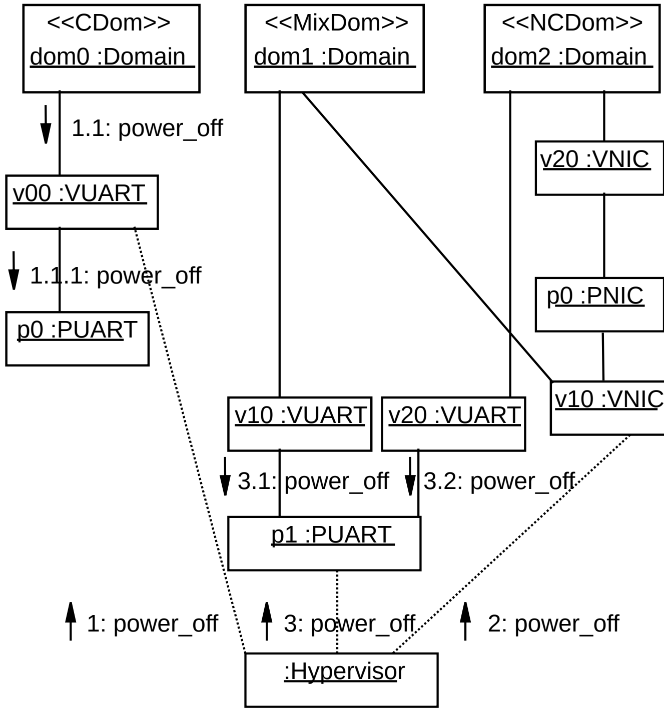


Figure 7: Power off dynamics.

OFF/VDev-ON means to power virtual I/O devices OFF/ON, and **VCPU-OFF/VCPU-ON** means to power VCPUs OFF/ON. “On high” means the actions performed whenever the interference is judged too high , while “On low” means the actions performed whenever the interference becomes acceptable.

1. P-V-U order
   - On high: PDev-OFF (implicit VDev-OFF), VCPU-OFF
   - On low: VCPU-ON, PDev-ON, VDev-ON
2. U-V-P order
   - On high: VCPU-OFF, VDev-OFF, PDev-OFF
   - On low: PDev-ON, VDev-ON, VCPU-ON

3. V-P-U order
   - On high: VDev-OFF, PDev-OFF, VCPU-OFF
   - On low: VCPU-ON, PDev-ON, VDev-ON
4. V-U-P order
   - On high: VDev-OFF, VCPU-OFF, PDev-OFF
   - On low: PDev-ON, VCPU-ON, VDev-ON
5. U-P-V order
   - On high: VCPU-OFF, PDev-OFF (implicit VDev-OFF)
   - On low: VDev-ON (implicit PDev-ON), VCPU-ON

As defined by Rule 2, powering OFF physical I/O devices causes virtual I/O devices to be powered OFF. Similarly, by Rule 3, powering ON virtual I/O devices will cause the correspondent physical I/O device to be powered ON. Each order can be used in a specific situation. For example, the *P-V-U* order can be used in the case there is a high I/O interference increase rate (I/O interference grows fast) since it powers OFF physical I/O devices first. The *U-V-P* order is more "optimistic" since it assumes powering OFF VCPUs first can reduce the interference. The *V-P-U* order assumes that some domains use a physical I/O device intensively while other domains use the same devices mildly. In some cases, powering off VCPUs can have similar effect than powering off virtual I/O devices since it will cause the guest OS tasks that run on the powered-OFF VCPU to stop (assuming such guest OS tasks will not migrate to another VCPU).

Additionally to the following orders, the hypervisor can power-OFF first the processing elements that are causing more interference (in the case there is such global knowledge as mentioned in Section 4.1) or can power-OFF first processing elements that are exclusive to non-critical domains, followed by processing elements that are shared between non-critical domains.

This work proposes the used of dedicated VCPUs, named IOVCPUs to handle operations of virtual I/O devices. IOVCPUs will execute periodically and, right before an IOVCPU is scheduled to run, the hypervisor will execute a Monitoring-decide-act procedure that will, based on the gathered data, decide whether processing elements must be powered OFF/ON, and issue of the respective operations to do so. Section 5.3 formalizes the virtual I/O devices parameters, and Section 5.5 (Definition 27) explains how IOVCPUs are computed from such parameters.

4.3 INTERRUPTS

Interrupts in virtualized platforms can be potentially handled in two ways, by servicing the interrupt synchronously immediately after it occurs or by servicing the interrupt latter (asynchronously) in the time allocated for the interrupt target domain. The former approach is similar to what ISR is for non-virtualized OSes and is referred in this work as ISbyH. The latter approach is similar to what IST is for non-virtualized OSes and is referred in this work as ISVCPU (KLEIMAN; EYKHOLT, 1995; FOYO; MEJIA-ALVAREZ; NIZ, 2006; BECKERT et al., 2014; TU et al., 2015).

The UML communication diagram of Figure 8 shows the dynamics of ISbyH. Once the interrupt generated by a physical I/O device is intercepted by the hypervisor, the virtualized version of the interrupt immediately occurs on the guest OS of the target domain regardless of which VCPU is currently running (the interrupt will occur on the context of the hypervisor). The interrupt is then handled and acknowledged by the guest OS by given an acknowledgment to the correspondent virtual I/O device. The reception of the acknowledgment by the virtual I/O device will generate an acknowledgment to the physical I/O device concluding the interrupt handling.

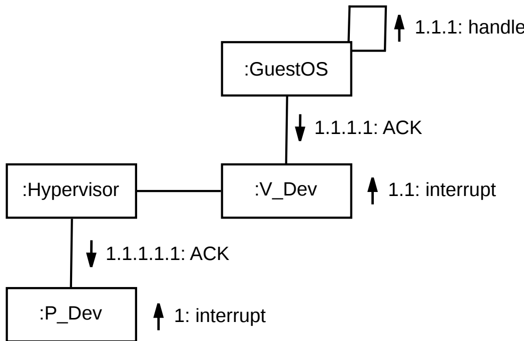


Figure 8: ISbyH.

The UML communication diagram of Figure 9 shows the dynamics of ISVCPU. Once the interrupt generated by a physical I/O device is intercepted by the hypervisor, the hypervisor signalizes the virtual I/O device of the target domain of the occurrence of the interrupt (generating a virtual interrupt) and, if the case is, pushing interrupt-related data into the virtual I/O device queue. After this step, the hypervi-

sor acknowledges the interrupt on the physical I/O device concluding the interrupt handling from the physical point-of-view (host/hypervisor level). Later, when the VCPU responsible for handling interrupts on the target domain (could be any VCPU or a specific one) is scheduled to run, the guest OS interrupt mechanism is invoked and the guest OS handles and acknowledges the interrupt to the virtual I/O device. Such an acknowledgment will then conclude the interrupt handling on the virtual point-of-view (guest OS level) and, if is the case, will free buffers on the virtual I/O device queue.

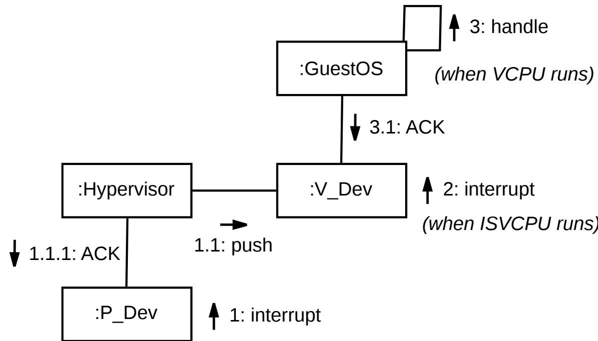


Figure 9: ISVCPU.

It is worthily mentioning that in both cases (ISbyH and ISVCPU) the guest OS can, in turn, handle its interrupts either by using ISR or IST.

The main potential advantage of using ISbyH is having zero *interrupt latency* which is defined as the elapsed time between (i) the moment the interrupt is intercepted by the hypervisor (step 1 of Figure 8) and (ii) the moment the guest OS perceives the interrupt (step 1.1 of Figure 8). The interrupt latency is zero because the hypervisor will immediately invoke the guest OS interrupt handling mechanism. In ISbyH, the hypervisor "invades" the context of the running VCPU which can be either a VCPU of the target domain or a VCPU of other domain. That will cause a time displacement (jitter) on the execution of VCPUs and, by consequence, on the guest OS tasks that run on it. Ultimately, such jitter can cause critical tasks to lose their deadlines. It is important to mention that the time displacement on the execution of a VCPU does not change the Domain Virtual Time (DVT), which is incremented only when the domain has a running VCPU. Non-critical tasks will usually depend only on DVT and will not perceive the VCPU

jitter. Critical tasks on the other hand, will usually use special timers provided by the hypervisor that operate using System Time (ST) for counting elapsed RT, therefore, suffering from VCPU jitter.

The main advantage of ISVCPU is to decouple in time the interrupt handling at host and guest levels. The role of the hypervisor is to intercept the interrupt, virtualized it, and acknowledge it to the physical I/O device (steps 1, 1.1, and 1.1.1 of Figure 9) while the interrupt servicing is performed by the guest OS during its VCPU time. As consequence, interrupt handling at host-level (steps 1, 1.1, and 1.1.1 of Figure 9) has constant time and there is no VCPU jitter. For that reason, ISVCPU is the strategy adopted by many hypervisors and is the strategy taken into account in this work.

Now it is presented the idea envisioned in this work to design and implement ISVCPU, decoupling interrupt reception and acknowledgment at the host-level from interrupt servicing at the guest-level, thus achieving time-bounded interrupt handling at host-level. The idea is a design pattern named *Semaphore Observer*. The Semaphore Observer is a concurrent variant of the *Publisher/Subscriber* design pattern (a.k.a. *Observed/Observer*) (GAMMA et al., 1995). It is depicted in Figure 10. The main difference between Semaphore Observer and the original Publisher/Subscriber is that a publisher notification does not cause its subscribers to be updated. Instead, the publisher only notifies subscribers that a state change has occurred through a synchronization mechanism. Subscribers are modeled as VCPUs that wait for such state change notification on the shared synchronization mechanism and then perform the updates. Such VCPUs are dedicated for handling interrupts and there can be one for each virtual I/O device on each domain. Also such VCPUs can be marked with a higher priority than the usual VCPUs in a domain so interrupts are updated on the guest-level before resuming guest OS tasks that might be waiting for them.

The synchronization mechanism in the pattern is abstracted as a *Semaphore* shared between the VCPUs assigned to the virtual I/O device (the `Semaphore_Observer`) and the physical I/O device hardware mediator (which implements the ISR) in the hypervisor (the `Semaphore_Observed`).

The `notify()` method of `Semaphore_Observed`, described in the sequence diagram of Figure 11, invokes an `update` method for each observer. Such `update` method, in turn, notifies the observer by invoking the `v()` operator of the semaphore. As depicted in Figure 11, the `update()` method can also push interrupt-related data into the virtual I/O device queue.

The `updated()` method of `Semaphore_Observer` depicted in Fig-

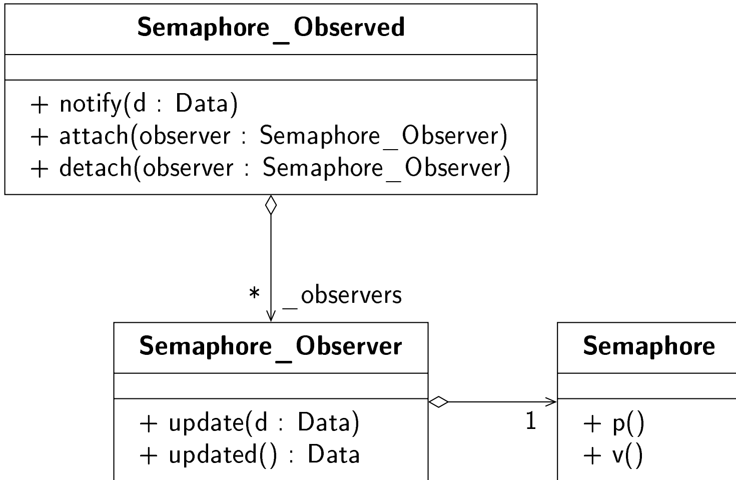


Figure 10: Semaphore Observer.

ure 12, will invoke the `p()` operator of the semaphore and, if needed, will wait for the physical I/O device interrupt notification. As depicted in Figure 12, the `updated()` method can also retrieve interrupt-related data from the virtual I/O device queue.

A semaphore is essential for interrupt handling. Since a semaphore has memory, interrupts notifications are never lost even if the frequency with which interrupts are generated is almost the same than the frequency with which interrupts are serviced. In such a case, however, in order to prevent data loss (in the case there is data generation associated with the interrupt occurrence) a larger buffer or a ring buffer strategy should be employed.

It should be noted that the use of semaphores in Semaphore Observer does not introduce priority inversion problems since they are not being used to guard a critical section. Instead, they are used only to synchronize the physical I/O device ISR with the corresponding ISVCPU, much in a Producer/Consumer way.

Regarding implementation, instead of using two kinds of VCPUs (an ordinary VCPU for executing guest OS tasks and an ISVCPU for updating I/O events on the guest OS) one might wish to use a single VCPU kind. However, in that case every VCPU in a domain would need to update I/O events as soon the VCPU is scheduled to run, before running the guest OS tasks.

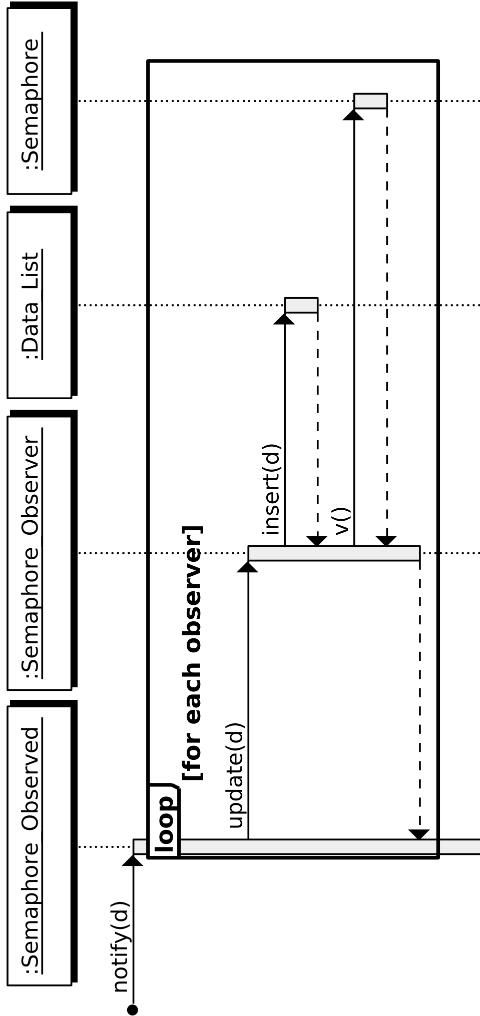


Figure 11: *Semaphore_Observed::notify.*

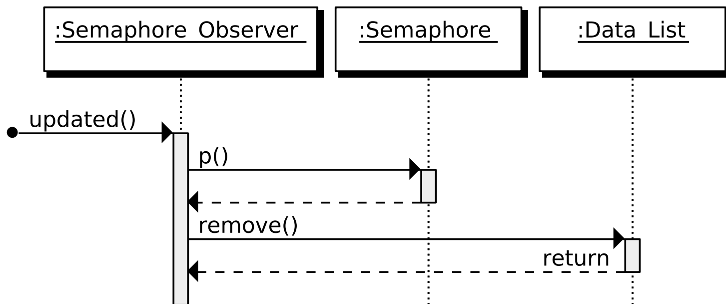


Figure 12: *Semaphore_Observer::updated*.

4.4 MEMORY

Regarding the memory hierarchy, temporal interference will occur mainly due to eviction of cache lines at the LLC that is shared between all PCPUs. Figure 13 shows the case where the critical task T_2 suffers temporal interference caused by the execution of T_1 a non-critical task that runs on another PCPU and has part of this page frames (physical memory) mapped to the same cache lines of T_2. Figure 14 shows a similar case in which the difference is that the non-critical task T_1 has issued a DMA transaction before accessing memory (which will then cause the eviction of lines that are used by T_2). The extent of the interference will depend on the time for accessing the main memory and replacing the evicted LLC lines.

In Figure 13 the LLC eviction interference is represented by the time interval between the time instants C and D. In Figure 14, the same kind of interference is observed between the time instants D and E.

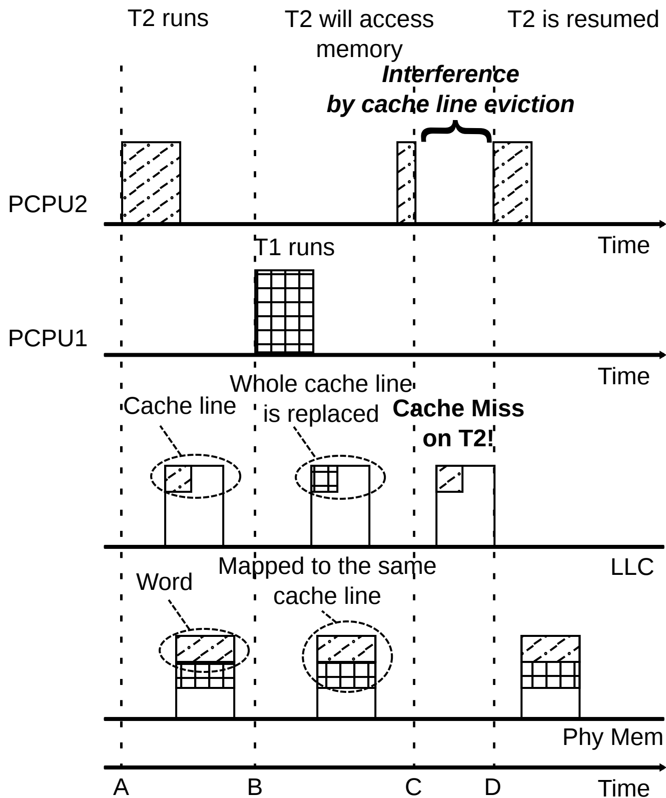


Figure 13: Interference due to cache line eviction caused by another PCPU.

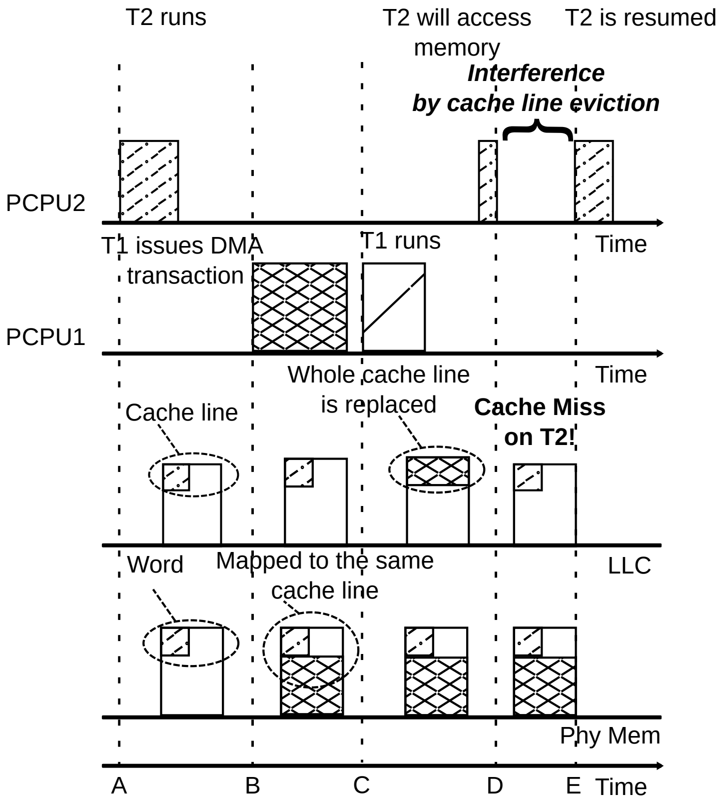


Figure 14: Interference due to cache line eviction caused by I/O device.

The general idea for dealing with memory interference at LLC is by partitioning the cache using page-coloring. The reasoning behind this idea is that partitions with distinct colors will not evict cache lines from one another, preventing the interference from happening. This work identifies three approaches for applying page-coloring on virtualized platforms: oriented to the domain, VCPUs, and guest OS tasks.

4.4.1 Domain-oriented Page-coloring

In the domain-oriented approach, the hypervisor will provide each critical domain with an exclusive color; non-critical domains will all share the same color. The main advantage of this approach is that it does not require any change on the guest OSes nor on the guest OS tasks. However, having one single color per domain is a too coarse granularity and will only be effective if all critical domains run a single task each. If a critical domain runs two or more critical tasks, one task will potentially evict cache lines from the other, disrupting the isolation and causing temporal interference on one another.

4.4.2 VCPU-oriented Page-coloring

In VCPU-oriented page-coloring, a color is given to a VCPU or group of VCPUs. This approach combines together (i) page-coloring, (ii) VCPU masking or affinity (that defines on which PCPUs a VCPU is allowed to run), and (iii) clustered scheduling (clusters of PCPUs). Five types of clusters are defined:

1. C-cluster is a cluster of size one (composed of one single PCPU) where each VCPU of the cluster has an exclusive color. As consequence of having size one, VCPUs in a C-cluster does not migrate, which helps preventing cache line eviction on inner-level caches (e.g. L1).

2. F-cluster is similar to a C-cluster regarding page-coloring, each VCPU has an exclusive color. However, can have size N (where $N \geqslant 1$).

3. D-cluster is an N-sized cluster with a cluster-exclusive color, i.e.: all VCPUs within the cluster share the same color but each D-cluster has its own exclusive color.

4. B-cluster is an N-sized cluster with a cluster-type color, i.e.: all VCPUs in all B-clusters share the same color.

5. N-cluster is an N-sized non-colored cluster. Having no color a guest OS task in an N-cluster can cause interference on any other guest OS task since it does not respect the cache partitioning. That is the default of systems that do not implement page-coloring.

The main advantage of this approach is that it does not require changes on guest OS tasks code. It might require the guest OS task to be pinned to specific VCPUs so it can use that VCPU color. However, this approach requires the guest OS to be aware of colors.

A possible mechanism for implementing color-aware on the guest OS is the following. On the creation of VCPUs, before loading any domain, PCPUs and VCPUs are organized into clusters according to the five types described above. At boot time, the guest OS asks the hypervisor for a list of colors it is allowed to have. Such list will depend on how the VCPUs that belong to the guest OS domain are organized (how many clusters and their types). For example, if the guest OS belongs to a domain composed of one C-cluster that has two VCPUs and by one D-cluster that has also two VCPUs, the hypervisor will grant the guest OS three colors (one for each VCPU of the C-cluster and one for the two VCPUs of the D-cluster) plus a fourth color (a system color) for the guest OS kernel. The guest OS must then have multiple page frame lists, one for each, color. At creation time, critical tasks must be pinned to VCPUs of C-clusters or F-clusters while non-critical tasks must be pinned to VCPUs of other clusters types. The heap of the guest OS process will be created with the color correspondent to the color of the VCPU it is pinned to.

4.4.3 Guest OS task-oriented Page-coloring

In the guest OS task-oriented page-coloring, each task is created informing the color it wishes to use. The guest OS will then create a dedicated heap for the task, according the color availability in the guest OS. The main advantage of this approach is that it gives more control (guest OS task-granularity) to system designers. However, it requires changes on both guest OS task and guest OS code. The mechanism is similar to the one used on VCPU-oriented page coloring. At boot time, the guest OS will ask the hypervisor for a set of colors which are going to

be used for creating dedicated list of free page frames. At task creation time, the guest OS task code needs to be modified in the sense the task itself must inform the color it wishes to use. In all approaches: domain-oriented, VCPU-oriented, and guest OS task-oriented it is important to notice that guest OS tasks will operate with guest-logical addresses while the colors are described in terms of host-physical addresses. The mapping of guest-logical addresses onto host-physical addresses must be handled by the hypervisor using mechanisms like SPTs.

4.5 DISCUSSION

The power OFF/ON techniques discussed in Section 4.2 are based on the ones evaluated by (HOELLER, 2007) and (REIS, 2016), extending them in order to take into account also VCPUs and virtual I/O devices. While (HOELLER, 2007) and (REIS, 2016) have thread as the *top-level* unit of dependencies, in this work the domain is the one to play such a role. Furthermore, the domain interacts with the virtual counterpart of a processing element and not with the physical processing element directly. In other works "a domain is client of a physical processing element that is represented by a virtual processing element proxy." instead of "a thread is a client of a physical processing element".

Mostly hypervisors adopt strategies similar to ISVCPU for handling interrupts. Xen uses the *split device driver* model for I/O devices and handles physical interrupts by mapping then onto events that, when happen, invoke the interrupt handling mechanism at guest OS level by using callback functions previously registered by the guest OS (CHISNALL, 2007). Some approaches such as the one presented by Beckert et al. explore something in between the ISVCPU and ISbyH approaches letting a interrupt be handled in the time slot of a domain distinct from its target domain if there is time for that (BECKERT et al., 2014). Their approach, however, demands for an off-line analysis to ensure that the interference caused by handling interrupts outside the domain does not break temporal independence of others. Such an analysis, in turn, demands for arrival functions for every event in the system. The approach presented in this section still requires a limited arrival of interrupts in order to keep the interference caused by interrupts bounded. However, rather than focusing on static analysis of interrupts interactions it focuses on keeping the interrupt handling time constant at the host level. The idea of Semaphore Observer pre-

sented in this Section was first presented in the published paper: *Proper Handling of Interrupts in Cyber-Physical Systems*, In Proceedings of the 26th IEEE International Symposium on Rapid System Prototyping (RSP 2015), 2015, pages 83-89, (LUDWICH; FRöHLICH, 2015).

Gracioli at al. have evaluated page-coloring while applied to multi task systems (GRACIOLI; FRöHLICH, 2013), (GRACIOLI, 2014). In their approach each critical task can have a unique color while non-critical tasks can share the same color. This work extends their approach, discussing the use of one or more exclusive colors to domains in a virtualized platform.

5 PROPOSED MODEL

This chapter presents the second part of this work proposal. It introduces the proposed model that captures guest OS tasks requirements regarding VCPUs and virtual I/O devices requirements. The guest OS tasks requirements will determine the computation of the domain requirements, which is abstracted in the form of a DRM. Finally, the domain requirements will compose the system requirements. By comparing the system requirements with the resources provided by the physical platform is possible to determine the feasibility of guest OS tasks and domains. Each interference source informally presented in Chapter 4 is formalized and feed into the model. Finally, interference-aware schedulability tests are devised, taking into account cache and I/O overheads.

Sections 5.1 to 5.3 introduce the proposed model that consists of guest OS tasks and the concept of DRM. The formal concept of guest OS task captures the processing and I/O requirements of a guest OS task. A DRM captures the VCPUs and virtual I/O devices budgets that can be provided to a domain as a whole. Section 5.4 introduces the concept of interconnect topology that specifies how PCPUs, and physical I/O devices are connected in a physical platform. Section 5.5 uses the concepts presented in previous sections to discuss schedulability of guest OS tasks by a DRM, how DRMs are combined to represent the requirements of the whole system, and whether the physical platform has resources for scheduling them. Section 5.6 presents the proposed algorithm for DRM generation. Section 5.7 formalizes the overhead caused by the virtualization itself. Section 5.8 formalizes memory and I/O overheads presented on previous sections and discuss the scheduling condition taking those overheads into consideration. Section 5.9 explains how the overhead-aware analysis can be used during runtime, in design and production phases, to take scheduling decisions encompassing virtual and physical processing elements. Section 5.10 closes the chapter discussing the proposed modeling approach.

5.1 COMMON DEFINITIONS

The following definitions are common to guest OS tasks, DRM, and the whole system.

Definition 1 *(Criticality Level)*

$\Omega = \{LO, HI\}$

The concept of *criticality level* is applied for Guest OS tasks, DRM, and the system. Two levels of criticality are specified, LO denotes low criticality and HI denotes high criticality. Regarding guest OS tasks (criticality level denoted by ω_i), there are HI-tasks and LO-tasks. The former are used for representing critical tasks, while the latter are used for representing non-critical tasks. A task set of HI-tasks forms a HI-domain, which those requirements is modeled by a HI-criticality DRM. Conversely, a task set of LO-tasks forms a LO-domain, which those requirements is represented by a LO-criticality DRM. The criticality level of a DRM is denoted by $\omega_{\mathcal{M}}$. Regarding criticality level of the system (also known as *system mode* and denoted by $\omega_{\mathcal{S}}$), when the system is in LO mode, both HI-tasks and LO-tasks execute. On the other hand, when the system is in HI mode, all LO-tasks are suspended and only HI-tasks execute.

Definition 2 *(Virtual I/O Device Class)*
$\mathcal{R} = \{x | x \textit{ is a Virtual I/O class.}\}$

A virtual I/O device class represents the virtual counterpart of a class of a physical I/O device such as NIC, Universal Asynchronous Receiver Transmitter (UART), and others. Instances of such classes are used by the domains to perform I/O operations.

Definition 3 *(I/O Operation)*
$\Gamma = \{rd, wr\}$

Two I/O operations are modeled for virtual I/O devices, “read” (denoted by rd), and “write” (denoted by wr). The precise meaning of the operations depends on the I/O device class. For example, to a video device “write” means writing data into the video device frame buffer, while for a NIC it means sending data through the link layer of a network.

5.2 GUEST OS TASK AND DOMAIN

A guest OS task τ_i is modeled as an MC periodic task, according to Definition 4.

Definition 4 *(Guest OS task)*
$\tau_i = (T_i, D_i, \omega_i, C_i^{\omega}, R_i^{v,\omega,\gamma})$ *where*

- *T_i is the task period.*
- *D_i is the task relative deadline.*
- *$\omega_i \in \Omega$ is the task criticality level.*
- *C_i^{ω}, where $\omega \in \Omega$, is the required budget of VCPU resource units for the task to perform its computation within one period while the system is running in LO ($\omega_{\mathcal{S}} = LO$) or HI ($\omega_{\mathcal{S}} = HI$) criticality level.*
- *$R_i^{v,\omega,\gamma}$, where $v \in \mathcal{R} \wedge \omega \in \Omega \wedge \gamma \in \Gamma$, denotes the required budget of resource units of a virtual I/O device class v for the task to perform its computation within one period while the system is running in LO ($\omega_{\mathcal{S}} = LO$) or HI ($\omega_{\mathcal{S}} = HI$) criticality level, and while performing a "read" ($\gamma = rd$) or "write" ($\gamma = wr$) operation on the virtual I/O device.*

Additionally,

1. *if $\omega_i = HI$*
   $C_i^{HI} > C_i^{LO} \wedge R_i^{v,HI,\gamma} > R_i^{v,LO,\gamma}$
2. *if $\omega_i = LO$*
   $C_i^{HI} = 0 \wedge R_i^{v,HI,\gamma} = 0$

Some examples, given that $v = 0$ represents the class of virtual NIC, and $v = 1$ represents the class of virtual UART.:

- $R_0^{1,HI,wr}$ means the budget per period required by task τ_0 to perform "write" operations on a virtual UART while the system is executing in HI mode.
- $R_1^{0,LO,rd}$ means the budget per period required by task τ_1 to perform "read" operations (i.e. receive packets) on a virtual NIC while the system is executing in LO mode.

In the case of VCPUs, the employed resource unit is usually a time unit. In the case of virtual I/O devices the employed resource unit can change from one class to another. However, to simplify the model presentation is assumed $R_i^{v,\omega,\gamma}$ is also specified in the same time unit as VCPUs. Additionally, is also assumed the existence of a function $f^v : R' \rightarrow R$ that maps R' specified in the usual resource units of the I/O device class (e.g. bytes per second) onto R specified in a time unit. Therefore, $R_i^{v,\omega,\gamma}$ denotes the time spend by task τ_i while using

the resource of class v within a period. In the case of a "write/send" operation, $R_i^{v,\omega,wr}$ usually denotes the time the task needs to program the device. In the case of a "read/receive" operation, $R_i^{v,\omega,rd}$ usually denotes the time the task needs to retrieve some data computed by the virtual I/O device. In the "read/receive" case, it is assumed a physical interrupt on the physical I/O device has already occurred and the hypervisor has signalized its virtual counterpart.

For HI-tasks, C_i^{LO} and $R_i^{v,LO,\gamma}$ are obtained from the task's Average Execution Time (AVET), while C_i^{HI} and $R_i^{v,HI,\gamma}$ are obtained from the task's WCET. For LO-tasks, C_i^{LO} and $R_i^{v,LO,\gamma}$ are obtained from the task's AVET, while C_i^{HI} and $R_i^{v,HI,\gamma}$ are zero by definition (since a LO-task is suspended whenever the system is in HI mode).

Regarding schedulability is assumed that

1. $D_i \leqslant T_i$
2. $C_i^{\omega} + \sum_{v \in \mathcal{R}, \gamma \in \Gamma} R_i^{v,\omega_{\mathcal{S}},\gamma} \leqslant D_i$ $(\forall \omega \in \Omega)$

The criterion 1 is usual in real-time systems. The criterion 2 states that all time, during the task's period, the task spends either processing or performing I/O operations must feet inside its deadline. In the case of HI-tasks, regarding system transition from LO to HI mode, it is usually assumed an artificially tightened deadline for the LO mode, according do Definition 5, such that $D_i^{LO} < D_i$.

Definition 5 *(Artificially tightened deadline for the LO mode)*

$D_i^{LO} = D_i - (C_i^{HI} + \sum_{v \in \mathcal{R}, \gamma \in \Gamma} R_i^{v,HI,\gamma} + \delta_{LO_{to}HI})$

where

- $\delta_{LO_{to}HI}$*, presented later in Definition 28, is the hypervisor overhead for switching from the LO to* HI *mode.*

D_i^{LO} assumes that the transition from LO to HI mode takes time. Therefore, the idea is to create a slack that allows the task to finish its current job before its actual deadline D_i. For LO-tasks $D_i^{LO} = D_i$ always, since they do not run in HI mode. The strategies used for deciding when criticality mode switches occur in the system are described in Section 5.9.

The *whole-task utilization* takes into account the VCPU and virtual I/O device budgets and is given by Definition 6.

Definition 6 *(Whole guest OS task utilization)*

$WTU_i^{\omega_S} = \frac{C_i^{HI} + \sum_{v \in \mathcal{R}, \gamma \in \Gamma} R_i^{v,\omega_{\mathcal{S}},\gamma}}{T_i}$

Tasks of the same criticality are grouped into a domain, which is defined according to Definition 7.

Definition 7 *(Domain)*
Dom $= (\omega_{\mathcal{D}}, \tau)$ *where*

- $\omega_{\mathcal{D}} \in \Omega$ *is the domain criticality.*
- τ *is the domain task set composed of guest* OS *tasks such that* $\forall \tau_i \in \tau : \ \omega_i = \omega_{\mathcal{D}}$.

The total utilization of a domain is defined for VCPUs and virtual I/O devices in Definition 8 and 9, respectively.

Definition 8 *(Total* VCPU *Utilization of a Domain)*
$U_{\tau\Theta}^{\omega_{\mathcal{S}}} = \sum_{\tau_i \in \tau} \frac{C_i^{\omega}}{T_i}$

Definition 9 *(Total virtual* I/O *device Utilization of a Domain)*
$U_{\tau\Xi}^{v,\omega_{\mathcal{S}},\gamma} = \sum_{\tau_i \in \tau} \frac{R_i^{v,\omega,\gamma}}{T_i}$

Additionally, the whole-domain utilization can be defined as shown in Definition 10.

Definition 10 *(Whole Domain Utilization)*
$WDU_{\tau}^{\omega_{\mathcal{S}}} = \sum_{\tau_i \in \tau} \frac{C_i^{\omega_{\mathcal{S}}} + \sum_{v \in \mathcal{R}, \gamma \in \Gamma} R_i^{v,\omega_{\mathcal{S}},\gamma}}{T_i}$

5.3 DOMAIN RESOURCE MODEL

A DRM models the processing and I/O budgets collectively provided by all VCPUs and virtual I/O devices of a domain. A DRM $\mathcal{M}$ is modeled according to Definition 11.

Definition 11 *(*DRM*)*
$\mathcal{M} = \langle \omega_{\mathcal{M}}, \Pi, \Theta^{\omega_{\mathcal{S}}}, m^{\omega_{\mathcal{S}}}, \Lambda^{v,\gamma}, \Xi^{v,\omega_{\mathcal{S}},\gamma}, n^{v,\omega_{\mathcal{S}},\gamma} \rangle$ *where*

- $\omega_{\mathcal{M}} \in \Omega$ *is the* DRM *criticality level. A* HI*-domain (*HI*-criticality domain) will contain only* HI*-tasks and its correspondent* DRM *will have* $\omega_{\mathcal{M}} = HI$*. A* LO*-domain (*LO*-criticality domain) will contain only* LO*-tasks and its correspondent* DRM *will have* $\omega_{\mathcal{M}} = LO$*.*

- Π *is the budget replenishment period for all VCPUs in the DRM.*

- Θ^{ω_S} *is the budget of resource units collectively provided by all VCPUs of that DRM in every* Π *time units while the system is running in* LO ($\omega_{\mathcal{S}} = LO$) *or* HI ($\omega_{\mathcal{S}} = HI$) *criticality level.*

- m^{ω_S} *is the number of VCPUs that the DRM can provide for each criticality level. In practice, however, since the system is at a given moment of time on either* HI *or* LO *mode, the actual number of VCPUs allocated to a domain is determined as* $max(m^{HI}, m^{LO})$.

- $\Lambda^{v,\gamma}$ *is a two-dimensional vector of budget replenishment period for each class (indexed by* v*) and each operation (indexed by* γ*) of virtual I/O device in the DRM.*

- $\Xi^{v,\omega_S,\gamma}$ *represents the resource units collectively provided by all virtual I/O devices of class* v *in that DRM every* $\Lambda^{v,\gamma}$ *time units while the system is running in* LO ($\omega = LO$) *or* HI ($\omega = HI$) *criticality level, and while performing a "read"* ($\gamma = rd$) *or "write"* ($\gamma = wr$) *operation.*

- $n^{v,\omega_S,\gamma}$ *is vector (indexed by* v, $\omega_{\mathcal{S}}$, *and* γ*) of the number of virtual I/O devices of each class, criticality level, and operation that the DRM can provide. Similarly to the number of VCPUs, the number of virtual I/O devices allocated to a domain is chosen as* $max(n^{v,HI,\gamma}, n^{v,LO,\gamma})$.

$\Lambda^{v,\gamma}$ indicates that each virtual I/O device will provide its resources periodically. This work proposes the concept of IOVCPU to make this possible. An IOVCPU is a dedicated VCPU to handle "read" and "write" operations of a virtual I/O device. An IOVCPU responsible for "read" can be realized by acting as a "Semaphore_Observer" as described by Section 4.3. A "read"-IOVCPU would run periodically, handling interrupts generated by the physical I/O device, virtualizing such interrupts and passing them to the guest OS along with data related to the interrupt when applicable. A "write"-IOVCPU will also run periodically, getting a "write" request made by the virtual I/O device and programming the physical I/O device to perform the "write" operation (e.g. a DMA transaction). Also in the case of "write" operations the IOVCPU can check whether the parameters (e.g. bytes to be written) passed to the virtual I/O device are in accordance to what is provided to that domain. Ideally the I/O requirements of a guest OS task are known at design time for both HI and LO domains. In

practice, such requirements might be unknown for some LO-domains. In such cases, if a guest OS task wishes to write more bytes than what is allowed by the virtual I/O device in that period, either the operation can be denied or the hypervisor is responsible for breaking the operation down on several periods. That will not occur with HI-domains, since the requirements of their guest OS tasks are assumed to be known at design time, always.

A feasible model should respect the condition

- $\Theta^{\omega_{\mathcal{S}}} \leqslant m^{\omega_{\mathcal{S}}}\Pi$
  $\wedge\ \Xi^{v,\omega_{\mathcal{S}},\gamma} \leqslant (n^{v,\omega_{\mathcal{S}},\gamma}\Lambda^{v,\gamma})$
  $\forall v \in \boldsymbol{R}, \forall \gamma \in \boldsymbol{\Gamma}$

This means that the provided VCPU and virtual I/O device budget cannot be grater then their respective replenishment periods.

The relation between the provided budget and its replenishment period is known as *resource bandwidth* of a DRM. Definitions 12 and 13 present how resource bandwidth for VCPUs and virtual I/O devices are computed, respectively.

Definition 12 *(VCPU bandwidth provided by a DRM)*
VCPU bandwidth $= \frac{\Theta^{\omega_{\mathcal{S}}}}{\Pi}$

Definition 13 *(virtual I/O device bandwidth provided by a DRM)*
Virtual I/O device bandwidth $= \frac{\Xi^{v,\omega_{\mathcal{S}},\gamma}}{\Lambda^{v,\gamma}}$

5.3.1 Supply Bound Functions of Domain Resource Model

A SBF specifies the quantity of resource provided by a DRM in a time interval of duration t. There two kinds of SBF associated with a DRM $\mathcal{M}$. The first one is $sbf_{\Theta}(\omega_{\mathcal{M}}, \omega_{\mathcal{S}}, t, m^{\omega_{\mathcal{S}}})$ that specify the VCPU resource units provided all VCPUs of the domain of criticality $\omega_{\mathcal{D}} = \omega_{\mathcal{M}}$ while the system is running on $\omega_{\mathcal{S}}$ mode in the time interval of duration t. The second kind if $sbf_{\Xi}(v, \omega_{\mathcal{M}}, \omega_{\mathcal{S}}, \gamma, t, n^{v,\omega_{\mathcal{S}},\gamma})$ that specify the resource units provided all virtual I/O devices of class v in a domain of criticality $\omega_{\mathcal{D}} = \omega_{\mathcal{M}}$ while executing the operation γ, while the system is running on $\omega_{\mathcal{S}}$ mode in the time interval of duration t.

$sbf_{\Theta}(\omega_{\mathcal{M}}, \omega_{\mathcal{S}}, t, m^{\omega_{\mathcal{S}}})$, is presented in Equation 5.1.

Definition 14 *(SBF for VCPUs)*

$$sbf_{\Theta}(\omega_{\mathcal{M}}, \omega_{\mathcal{S}}, t, m^{\omega_{\mathcal{S}}}) = \begin{cases} 1^{st}\ case: t' < 0 \vee \omega_{\mathcal{M}} = LO \wedge \omega_{\mathcal{S}} = HI \\ \qquad 0 \\ 2^{nd}\ case: t' \geqslant 0\ \ \wedge x \in [1, y] \\ \qquad\qquad \wedge \omega_{\mathcal{M}} = HI \wedge \omega_{\mathcal{S}} = HI \\ \qquad \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta^{HI} \\ \qquad + max(0, m^{\omega_{\mathcal{S}}} x - (m^{\omega_{\mathcal{S}}} \Pi - \Theta^{HI})) \\ 3^{rd}\ case: t' \geqslant 0\ \ \wedge x \notin [1, y] \\ \qquad\qquad \wedge \omega_{\mathcal{M}} = HI \wedge \omega_{\mathcal{S}} = HI \\ \qquad \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta^{HI} \\ \qquad + max(0, m^{\omega_{\mathcal{S}}} x - (m^{\omega_{\mathcal{S}}} \Pi - \Theta^{HI})) \\ \qquad -(m^{\omega_{\mathcal{S}}} - \beta) \\ 4^{th} case: t' \geqslant 0\ \ \wedge x \in [1, y] \\ \qquad\qquad \wedge (\omega_{\mathcal{M}} = HI \vee \omega_{\mathcal{M}} = LO) \\ \qquad\qquad \wedge \omega_{\mathcal{S}} = LO \\ \qquad \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta^{LO} \\ \qquad + max(0, m^{\omega_{\mathcal{S}}} x - (m^{\omega_{\mathcal{S}}} \Pi - \Theta^{LO})) \\ 5^{th} case: t' \geqslant 0\ \ \wedge x \notin [1, y] \\ \qquad\qquad \wedge (\omega_{\mathcal{M}} = HI \vee \omega_{\mathcal{M}} = LO) \\ \qquad\qquad \wedge \omega_{\mathcal{S}} = LO \\ \qquad \left\lfloor \frac{t'}{\Pi} \right\rfloor \Theta^{LO} \\ \qquad + max(0, m^{\omega_{\mathcal{S}}} x - (m^{\omega_{\mathcal{S}}} \Pi - \Theta^{LO})) \\ \qquad -(m^{\omega_{\mathcal{S}}} - \beta) \end{cases} \tag{5.1}$$

where

- $k = \frac{\Theta^{\omega_{\mathcal{S}}}}{m^{\omega_{\mathcal{S}}}}$
- $t' = t - (\Pi - \lceil k \rceil)$
- $\alpha = \lfloor k \rfloor$
- $\beta = \Theta^{\omega_{\mathcal{S}}} - m^{\omega_{\mathcal{S}}} \alpha$
- $x = (t' - \Pi \lfloor \frac{t'}{\Pi} \rfloor)$
- $y = \Pi - \alpha$

$sbf_{\Xi}(v, \omega_{\mathcal{M}}, \omega_{\mathcal{S}}, \gamma, t, n^{v,\omega_{\mathcal{S}},\gamma})$ is presented in Equation 5.2

Definition 15 *(SBF for virtual I/O devices)*

$$sbf_{\Xi}(v, \omega_{\mathcal{M}}, \omega_{\mathcal{S}}, \gamma, t, n^{v,\omega_{\mathcal{S}},\gamma}) = \begin{cases} 1^{st} \ case: t' < 0 \vee \omega_{\mathcal{M}} = LO \wedge \omega_{\mathcal{S}} = HI \\ \quad 0 \\ 2^{nd} \ case: t' \geqslant 0 \ \wedge x \in [1, y] \\ \qquad\qquad \wedge \omega_{\mathcal{M}} = HI \wedge \omega_{\mathcal{S}} = HI \\ \quad \left\lfloor \frac{t'}{\Lambda^{v,\gamma}} \right\rfloor \Xi^{v,HI,\gamma} \\ \quad + max(0, n^{v,\omega_{\mathcal{S}},\gamma} x - (n^{v,\omega_{\mathcal{S}},\gamma} \Lambda^{v,\gamma} \\ \quad - \Xi^{v,HI,\gamma})) \\ 3^{rd} \ case: t' \geqslant 0 \ \wedge x \notin [1, y] \\ \qquad\qquad \wedge \omega_{\mathcal{M}} = HI \wedge \omega_{\mathcal{S}} = HI \\ \quad \left\lfloor \frac{t'}{\Lambda^{v,\gamma}} \right\rfloor \Xi^{v,HI,\gamma} \\ \quad + max(0, n^{v,\omega_{\mathcal{S}},\gamma} x - (n^{v,\omega_{\mathcal{S}},\gamma} \Lambda^{v,\gamma} \\ \quad - \Xi^{v,HI,\gamma})) \\ \quad - (n^{v,\omega_{\mathcal{S}},\gamma} - \beta) \\ 4^{th} case: t' \geqslant 0 \ \wedge x \in [1, y] \\ \qquad\qquad \wedge (\omega_{\mathcal{M}} = HI \vee \omega_{\mathcal{M}} = LO) \\ \qquad\qquad \wedge \omega_{\mathcal{S}} = LO \\ \quad \left\lfloor \frac{t'}{\Lambda^{v,\gamma}} \right\rfloor \Xi^{v,LO,\gamma} \\ \quad + max(0, n^{v,\omega_{\mathcal{S}},\gamma} x - (n^{v,\omega_{\mathcal{S}},\gamma} \Lambda^{v,\gamma} \\ \quad - \Xi^{v,LO,\gamma})) \\ 5^{th} case: t' \geqslant 0 \ \wedge x \notin [1, y] \\ \qquad\qquad \wedge (\omega_{\mathcal{M}} = HI \vee \omega_{\mathcal{M}} = LO) \\ \qquad\qquad \wedge \omega_{\mathcal{S}} = LO \\ \quad \left\lfloor \frac{t'}{\Lambda^{v,\gamma}} \right\rfloor \Xi^{v,LO,\gamma} \\ \quad + max(0, n^{v,\omega_{\mathcal{S}},\gamma} x - (n^{v,\omega_{\mathcal{S}},\gamma} \Lambda^{v,\gamma} \\ \quad - \Xi^{v,LO,\gamma})) \\ \quad - (n^{v,\omega_{\mathcal{S}},\gamma} - \beta) \end{cases} \tag{5.2}$$

where

- $k = \frac{\Xi^{v,\omega_{\mathcal{S}},\gamma}}{n^{v,\omega_{\mathcal{S}},\gamma}}$
- $t' = t - (\Lambda^{v,\gamma} - \lceil k \rceil)$

- $\alpha = \lfloor k \rfloor$
- $\beta = \Xi^{v,\omega s,\gamma} - n^{v,\omega s,\gamma}\alpha$
- $x = (t' - \Lambda^{v,\gamma}\lfloor \frac{t'}{\Lambda^{v,\gamma}} \rfloor)$
- $y = \Lambda^{v,\gamma} - \alpha$

5.4 INTERCONNECT TOPOLOGY

A physical platform is composed of processing elements, memory elements, and interconnect elements. The model presented in this section uses queuing theory for representing such elements specifying them by their insertion and servicing rates. Such elements are connected among themselves composing the *physical interconnect topology.* Before introducing the definition of the physical interconnect topology, Definitions 16 to 20 formalize the elements that compose the topology, which is then presented at Definition 21.

Definition 16 *(Bandwidth level)*
$LV = \{AVG, MAX\}$

Insertion and servicing rates are defined in units of packets per time unit (usually Byte per second in this model), defining what is named *bandwidth* of the element.

Each topology element has two levels of bandwidth, its maximum level (denoted as MAX), which in this model is the nominal value provided by the device vendor and its AVG level that is the average used bandwidth taking into account a specific execution scenario.

A criticality-level (as defined by Definition 1) is mapped onto a bandwidth level according to Definition 17.

Definition 17 *(Criticality level to bandwidth level mapping)*

$$cl2lv(\omega, lv) = \begin{cases} 1^{st} case : \omega = HI \\ MAX \\ 2^{nd} case : \omega = LO \\ AVG \end{cases} \tag{5.3}$$

Definition 18 *(Processing Element and Processing Element Set)*
A Processing Element pe is defined as

pe = $(\lambda_{pe}^{\gamma,lv})$
where

- $\lambda_{pe}^{\gamma,lv}$ *denotes the pe insertion rate while executing the I/O operation* γ *(see Definition 3) at the lv (see Definition 16) bandwidth level.*

Additionally,
PE *is the set of all PCPUs and physical I/O devices in the system.*

Definition 19 *(Memory Element and Memory Element Set)*
A Memory Element me is defined as
$me = (\mu_{me}^{\gamma,lv})$
where

- $\mu_{me}^{\gamma,lv}$ *denotes the servicing rate of me while servicing the* γ *operation at the lv bandwidth level.*

Additionally,
ME *is the set of all memory elements in the system. Usually it represents a single main memory. Therefore,* $|\boldsymbol{ME}| = 1$.

Definition 20 *(Interconnect Element and Interconnect Element Set)*
A Interconnect Element inter is defined as
$inter = (\lambda_{inter}^{\gamma,lv}, \mu_{inter}^{\gamma,lv})$ *denoting insertion rate and servicing rate of inter for* γ *operation and lv level, respectively.*
Additionally,
INTER *is the set of all interconnect elements in the system.*

Given those definitions, a physical interconnect topology is presented by Definition 21.

Definition 21 *(Physical Interconnect Topology)*
A physical interconnect topology is defined as

$$Top({}^{\omega}\mathcal{S}) = \begin{cases} 1^{st} case : \omega_{\mathcal{S}} = HI \\ G^{HI} \\ 2^{nd} case : \omega_{\mathcal{S}} = LO \\ G^{LO} \end{cases} \tag{5.4}$$

where

- *G^{HI} denotes the topology configuration when the system is executing in HI-mode and G^{LO} denotes the topology configuration when the system is executing in LO-mode.*

  G^{HI} and G^{LO} are graphs $(\boldsymbol{V}, \boldsymbol{E})$ such that

- $\boldsymbol{V} = \boldsymbol{PE} \cup \boldsymbol{ME} \cup \boldsymbol{INTER}$

- $$\begin{aligned}\boldsymbol{E} = \{&(a,b) \mid a \in \boldsymbol{PE} \wedge b \in \boldsymbol{INTER} \\ &\vee a \in \boldsymbol{INTER} \wedge b \in \boldsymbol{INTER} \\ &\vee a \in \boldsymbol{INTER} \wedge b \in \boldsymbol{ME} \\ &\}\end{aligned}$$

The topology configurations are used to show which processing elements are powered ON (exist in the configuration) and which are powered OFF (do not exist in the configuration). Usually, G^{LO} will contain more processing elements than G^{HI} since the later will contain only processing elements used by HI-domains.

Additionally,

- *For any inter $\in \boldsymbol{INTER}$, $\lambda^{\gamma,lv}_{inter} = \sum_{x \,\in\, \boldsymbol{income}(inter)} \lambda^{\gamma,lv}_{x}$, where $\boldsymbol{income}(v)$ gives all incoming edges of vertex v.*

- *For any inter $\in \boldsymbol{INTER}$, it is assumed that $\mu^{\gamma,lv}_{inter} > \lambda^{\gamma,lv}_{inter}$. Such assumption is a queuing theory assumption and it ensures the time a device will wait to get access to the interconnect will not be infinite.*

It is assumed there is no virtual I/O device migration, and that the insertion and servicing rates as well the topology of virtual I/O devices is the same of their physical counterparts.

Given a physical topology, one can specify which processing elements are used by which domains. Such specification is named physical allocation, presented by Definition 22.

Definition 22 *(Physical Allocation)*

$$PhyAlloc(pe) = Dom^{*} \tag{5.1}$$

where

- *$pe \in PE$ (see Definition 18)*

- *Dom^* is a list of domains (see Definition 7) that use pe.*

HI-domains have exclusive access to their processing elements ($|PhyAlloc(pe)| = 1$), while LO-domains share processing elements. Memory and interconnect elements are assumed to be shared by all domains.

5.5 OVERHEAD-FREE COMPOSITIONAL SCHEDULABILITY ANALYSIS

This section presents the schedulability conditions for the DRM model. As will be explained, a task set of a domain can be scheduled by a DRM model if the worst case demand of its tasks is always inferior to what is provided by the DRM, which is given by its SBFs. Before introducing the schedulability condition, this section first defines the worst case demand of tasks. The resources (processing and I/O) demand of a task is defined by its own demand plus the interference caused by other tasks. It is important to mention that in this section, the term *interference* is merely the time a task does not run because other tasks of higher priority are using the resources. Interference due to cache and I/O sharing is introduced on Section 5.8.

With the aforementioned notion of interference in mind, the first step is to find an upper bound for it. This work extends the methodology used by (BERTOGNA; CIRINEI; LIPARI, 2005) and (EASWARAN; SHIN; LEE, 2009), which states that the interference a task τ_i can cause in a task τ_k cannot be higher than the workload upper bound of τ_i in a time interval of length t, where $t = b - a$. Such methodology assumes that there is a job of τ_k that has b as deadline, denoted as τ_k^b. In addition, is assumed that τ_k^b loses its deadline. Thus, the whole idea is that if the interference is always lower than this bound, then t_k is schedulable. The methodology also assumes an execution pattern in which τ_k^b and jobs of τ_i compete with each other, while one is running the other is not. Therefore, the workload upper bound assumes times in which τ_k is running and times in which τ_i is running. The methodology proposed by (BERTOGNA; CIRINEI; LIPARI, 2005) targets the gEDF scheduling algorithm for multiprocessor platforms. The methodology proposed by (EASWARAN; SHIN; LEE, 2009) extends the methodology proposed by (BERTOGNA; CIRINEI; LIPARI, 2005) for a virtualized scenario and it is also based on gEDF. This work extends the methodology proposed by (EASWARAN; SHIN; LEE, 2009) taking into account not only processing but I/O, and taking into account a MC scenario. Being an ex-

tension of (BERTOGNA; CIRINEI; LIPARI, 2005) and (EASWARAN; SHIN; LEE, 2009), this work also uses gEDF as the intra-domain scheduling algorithm, which reflects on the proposed scheduling tests presented this section and Section 5.8. gEDF is suitable for this work proposal since, inside the same domain, all tasks will share the same physical resources and will have the same criticality.

This work proposes two notions of workload bound of task τ_i in a time interval of length t, where $t = b - a$. The VCPU workload bound ($W_{\Theta i}$) and the virtual I/O device workload bound ($W_{\Xi i}$), which are defined in Equation 5.2 and Equation 5.5, respectively.

$$W^{\omega}_{\Theta i}(t) = N_i C^{\omega}_i + \epsilon^{\omega}_{\Theta i}(t) \tag{5.2}$$

where

- N_i is defined in Equation 5.3 and denotes the number of jobs of τ_i that start after (or at) a and finish before (or at) b.
- $\epsilon^{\omega}_{\Theta i}(t)$ is the carry-in demand generated by a job that has started before a and finishes during the interval $b - a$. $\epsilon^{\omega}_{\Theta i}(t)$ is defined in Equation 5.4.

$$N_i = \lfloor \frac{t - D_i}{T_i} \rfloor + 1 \tag{5.3}$$

$$\epsilon^{\omega}_{\Theta i}(t) = min(C^{\omega}_i, max(0, N_i T_i)) \tag{5.4}$$

$$W^{v,\omega,\gamma}_{\Xi i}(t) = N_i R^{v,\omega,\gamma}_i + \epsilon^{v,\omega,\gamma}_{\Xi i}(t) \tag{5.5}$$

where

- $\epsilon^{v,\omega,\gamma}_{\Xi i}(t)$ is the carry-in demand generated by a job that has started before a and finishes during the interval $b - a$. $\epsilon^{v,\omega,\gamma}_{\Xi i}(t)$ is defined in Equation 5.6.

$$\epsilon^{v,\omega,\gamma}_{\Xi i}(t) = min(R^{v,\omega,\gamma}_i, max(0, N_i T_i)) \tag{5.6}$$

The interval $[a, b]$ in which the workload bound is computed follows the Definition 23.

Definition 23 *(Workload Execution Pattern)*

Given the execution Interval $= [a, b]$ *where* a *and* b *are two points in time, expressed in some time unit (usually a sub-multiple of second).*

Such interval is divided into two parts, $(a, r]$ and $(r, b])$, where r represents the release time of the job τ_k^b.

It is assumed that in instant "a" there is at least one idle processing element and that in the interval $(a, b]$ there are no idle processing elements.

Additionally,

- *A_k (given by $r - a$) is the length of the interval $(a, r]$,*
- *D_k (given by $b - r$) is the length of the interval $(r, b])$.*

For VCPU workloads A_k can be expressed as $A^{\omega}_{\Theta k}$ to match with $W^{\omega}_{\Theta i}$. Conversely, for virtual I/O device workloads A_k can be expressed as $A^{v,\omega,\gamma}_{\Xi k}$ to match with $W^{v,\omega,\gamma}_{\Xi i}$. However, whenever is possible to infer from the context, just A_k is used to simplify the notation.

The total demand (or total workload) in this $[a, b]$ interval is denoted by $I^{\omega}_{\Theta i}$ (VCPU demand) or by $I^{v,\omega,\gamma}_{\Xi i}$ (I/O device demand). The total demand can be divided in times where τ_k is running that are defined as type-1 intervals, represented by $I^{\omega}_{\Theta 1,i}$ and $I^{v,\omega,\gamma}_{\Xi 1,i}$, and times where τ_k is not running that are defined as type-2 intervals, represented by $I^{\omega}_{\Theta 2,i}$ and $I^{v,\omega,\gamma}_{\Xi 2,i}$.

The total type-1 demand $I^{\omega}_{\Theta i,1}$ and $I^{v,\omega,\gamma}_{\Xi i,1}$ are respectively bounded by $\bar{I}^{\omega}_{\Theta i,1}$ and $\bar{I}^{v,\omega,\gamma}_{\Xi i,1}$. Equation 5.7 and Equation 5.8 define those bounds.

$$\bar{I}^{\omega}_{\Theta i,1} = C_k \tag{5.7}$$

$$\bar{I}^{v,\omega,\gamma}_{\Xi i,1} = R^{v,\omega,\gamma}_k \tag{5.8}$$

The upper bounds for type-1 demand is straightforward and states that τ_k^b cannot execute more than its required CPU and I/O device budgets.

The total type-2 demand $I^{\omega}_{\Theta i,2}$ and $I^{v,\omega,\gamma}_{\Xi i,2}$ are respectively bounded by $\bar{I}^{\omega}_{\Theta i,2}$ and $\bar{I}^{v,\omega,\gamma}_{\Xi i}$. Equation 5.9 and Equation 5.10 define those bounds.

$$\bar{I}^{\omega}_{\Theta i,2} = \begin{cases} 1^{st} case : \forall i \neq k \\ min(W^{\omega}_{\Theta i}(A_k + D_k), A_k + D_k - C^{\omega}_k) \\ 2^{nd} case : i = k \\ min(W^{\omega}_{\Theta k}(A_k + D_k) - C^{\omega}_k, A_k) \end{cases} \tag{5.9}$$

$$\bar{I}^{v,\omega,\gamma}_{\Xi i,2} = \begin{cases} 1^{st} case : \forall i \neq k \\ min(W^{v,\omega,\gamma}_{\Xi i}(A_k + D_k), A_k + D_k - R^{v,\omega,\gamma}_k) \\ 2^{nd} case : i = k \\ min(W^{v,\omega,\gamma}_{\Xi k}(A_k + D_k) - R^{v,\omega,\gamma}_k, A_k) \end{cases} \tag{5.10}$$

In the first case ($i \neq k$) of Equations 5.9 and 5.10, the first argument of the minimal function states that the interference cannot be bigger than the workload upper bound. In addition, the second argument of min states that interference does not need to be bigger than total interval ($A_k + D_k$) minus the execution time of τ_k itself since it is already taken into account on type-1 intervals. In the second case ($i = k$) of Equations 5.9 and 5.10, the execution time of τ_k is subtracted of the workload upper bound (first argument of min). In the second argument of min, the maximum time τ_k will not execute its the time interval previous of τ^b_k release (i.e. $(a, r]$ that has A_k length).

Terms $\hat{I}^{\omega}_{\Theta i,2}$ and $\hat{I}^{v,\omega,\gamma}_{\Xi i}$ are defined by Equation 5.11 and Equation 5.12. Their definition is similar definition to $\bar{I}^{\omega}_{\Theta i,2}$ and $\bar{I}^{v,\omega,\gamma}_{\Xi i}$, however, they discount the carry-in demand.

$$\hat{I}^{\omega}_{\Theta i,2} = \begin{cases} 1^{st} case : \forall i \neq k \\ min(W^{\omega}_{\Theta i}(A_k + D_k) - \epsilon^{\omega}_{\Theta i}(A_k + D_k), A_k + D_k - C^{\omega}_k) \\ 2^{nd} case : i = k \\ min(W^{\omega}_{\Theta i}(A_k + D_k) - C^{\omega}_k - \epsilon^{\omega}_{\Theta k}(A_k + D_k), A_k) \end{cases} \tag{5.11}$$

$$\hat{I}^{v,\omega,\gamma}_{\Xi i} = \begin{cases} 1^{st} case : \forall i \neq k \\ min(W^{v,\omega,\gamma}_{\Xi i}(A_k + D_k) - \epsilon^{v,\omega,\gamma}_{\Xi i}(A_k + D_k), A_k + D_k - R^{v,\omega,\gamma}_k) \\ 2^{nd} case : i = k \\ min(W^{v,\omega,\gamma}_{\Xi i}(A_k + D_k) - R^{v,\omega,\gamma}_k - \epsilon^{v,\omega,\gamma}_{\Xi i}(A_k + D_k), A_k) \end{cases} \tag{5.12}$$

Upper bound on the Worst Case Resource Demand (WCRD) of a task in the interval $(a, b]$

$$DEM_{\Theta}(\omega_k, \omega_{\mathcal{S}}, A_k + D_k, m^{\omega_{\mathcal{S}}}) = m^{\omega_{\mathcal{S}}} C_k^{\omega} + \sum_{i=1}^{n} \hat{I}_{\Theta i,2}^{\omega} + \sum_{i:i \in L(m^{\omega_{\mathcal{S}}}-1)} (\bar{I}_{\Theta i,2}^{\omega} - \hat{I}_{\Theta i,2}^{\omega}) \tag{5.13}$$

$$DEM_{\Xi}(v, \omega_k, \omega_{\mathcal{S}}, \gamma, A_k + D_k, n^{v,\omega_{\mathcal{S}},\gamma}) = n^{v,\omega_{\mathcal{S}},\gamma} R_k^{v,\omega,\gamma} + \sum_{i=1}^{n} \hat{I}_{\Xi i,2}^{v,\omega,\gamma} + \sum_{i:i \in L(n^{v,\omega_{\mathcal{S}},\gamma}-1)} (\bar{I}_{\Xi i,2}^{v,\omega,\gamma} - \hat{I}_{\Xi i,2}^{v,\omega,\gamma}) \tag{5.14}$$

A precondition for a domain to be schedulable by a given DRM is that the total domain utilization is smaller than the resource bandwidth provided by the DRM, as presented by Definition 24.

Definition 24 *(Domain Schedulability Precondition)*
$U_{\tau\Theta}^{\omega_{\mathcal{S}}} < \frac{\Theta^{\omega_{\mathcal{S}}}}{\Pi} \ \wedge U_{\tau\Xi}^{v,\omega_{\mathcal{S}},\gamma} < \frac{\Xi^{v,\omega_{\mathcal{S}},\gamma}}{\Lambda^{v,\gamma}}$

For scheduling analysis purposes, the time interval of length A_k can be bound to a maximum value. Definition 25 presents how to compute this bound for VCPUs and Definition 26 presents how to compute this bound to virtual I/O devices.

Definition 25 *(Maximum $A_{\Theta k}^{\omega_{\mathcal{S}}}$)*
$A_{\Theta kmax}^{\omega_{\mathcal{S}}} = \frac{C^{\omega_{\mathcal{S}}}{}_{\Sigma} + m^{\omega_{\mathcal{S}}} C_k^{\omega_{\mathcal{S}}} + B - D_k(\frac{\Theta^{\omega_{\mathcal{S}}}}{\Pi} - U_{\tau\Theta}^{\omega_{\mathcal{S}}})}{\frac{\Theta^{\omega_{\mathcal{S}}}}{\Pi} - U_{\tau\Theta}^{\omega_{\mathcal{S}}}}$
where

- $C^{\omega_{\mathcal{S}}}{}_{\Sigma} = \sum_{i:i \in L(m^{\omega_{\mathcal{S}}}-1)} C_i^{\omega_{\mathcal{S}}}$

  is the sum of the $(m^{\omega_{\mathcal{S}}} - 1)$ *largest VCPU budgets in the task set.*

- $B = \frac{\Theta^{\omega_{\mathcal{S}}}}{\Pi}[2 + 2(\Pi - \frac{\Theta^{\omega_{\mathcal{S}}}}{m^{\omega_{\mathcal{S}}}})]$

Definition 26 *(Maximum $A_{\Xi k}^{v,\omega_{\mathcal{S}},\gamma}$)*
$A_{\Xi kmax}^{v,\omega_{\mathcal{S}},\gamma} = \frac{R_{\Sigma}^{v,\omega,\gamma} + n'^{v,\gamma} R_k^{v,\omega,\gamma} + B - D_k(\frac{\Xi^{v,\omega_{\mathcal{S}},\gamma}}{\Lambda^{v,\gamma}} - U_{\tau\Xi}^{v,\omega_{\mathcal{S}},\gamma})}{\frac{\Xi^{v,\omega_{\mathcal{S}},\gamma}}{\Pi} - U_{\tau\Xi}^{v,\omega_{\mathcal{S}},\gamma}}$

A_{kmax} comes from a more pessimistic version for WCRD than the one defined by Equation 5.13 and Equation 5.14.

Finally, domain schedulability can be verified using Theorem 1.

Theorem 1 *(Domain Schedulability)*

A domain $\mathcal{D}$ composed of m^{ω_S} VCPUs, $n^{v,\omega_S,\gamma}$ virtual I/O devices, and a periodic task set τ is schedulable under gEDF using the DRM $\mathcal{M}$ if the precondition described by Definition 24 is satisfied and:

- $\forall \tau_k \in \tau$,
- $\forall A_k (A^{\omega}_{\Theta k}$ *or* $A^{v,\omega,\gamma}_{\Xi k}) \in [0, A_{kmax})$,
- $\forall \omega_{\mathcal{S}} \in \Omega$,
- $\forall v \in R, \forall \gamma \in \Gamma$:

$$DEM_{\Theta}(\omega_k, \omega_{\mathcal{S}}, A^{\omega}_{\Theta k} + D_k, m^{\omega_S}) \leqslant sbf_{\Theta}(\omega_{\mathcal{M}}, \omega_{\mathcal{S}}, t, m^{\omega_S})$$

$$\wedge\ DEM_{\Xi}(v, \omega_k, \omega_{\mathcal{S}}, \gamma, A^{v,\omega,\gamma}_{\Xi k} + D_k, n^{v,\omega_S,\gamma}) \leqslant sbf_{\Xi}(v, \omega_{\mathcal{M}}, \omega_{\mathcal{S}}, \gamma, t, n^{v,\omega_S,\gamma}) \tag{T1}$$

The reasoning behind A_{kmax} is that if there is some A_k for which the verification condition of Theorem 1 is violated, then such an A_k is smaller than A_{kmax}. Therefore, one needs to check for schedulability until A_{kmax}.

After defining all DRMs the system is expected to have (one for each domain), one needs to check whether the underlining physical platform is capable of providing the required resources. The idea is to transform each DRM $\mathcal{M}$ into a task set $\tau_{\mathcal{M}}$ as detailed in Definition 27 and then check whether that task set is schedulable.

Definition 27 *(Transformation of a DRM into a task set)*

- *Given* $\mathcal{M} = \langle \omega_{\mathcal{M}}, \Pi, \Theta^{\omega_S}, m^{\omega_S}, \Lambda^{v,\gamma}, \Xi^{v,\omega_S,\gamma}, n^{v,\omega_S,\gamma} \rangle$
- *Given the usual period task definition* $\tau_i = (T_i, E_i, D_i)$ *(period, execution time, deadline).*
- $\tau_{\mathcal{M}} = \tau_{\Theta} \cup \tau^{v,\gamma}_{\Xi}$ $(\forall v \in R, \forall \gamma \in \Gamma)$

- *Compute* τ_Θ *by*

$$\tau_1 = ... = \tau_k = (\Pi, \lfloor \frac{c\Theta}{cm} \rfloor + 1, \Pi)$$
$$\tau_{k+1} = (\Pi, \lfloor \frac{c\Theta}{cm} \rfloor + Z - k\lfloor \frac{Z}{k} \rfloor, \Pi)$$
$$\tau_{k+2} = ... = \tau_{cm} = (\Pi, \lfloor \frac{c\Theta}{cm} \rfloor, \Pi)$$

where:

  - $c\Theta = max(\Theta^{LO}, \Theta^{HI})$
  - $cm = max(m^{LO}, m^{HI})$
  - $Z = c\Theta - cm\lfloor \frac{c\Theta}{cm} \rfloor$
  - $k = \lfloor Z \rfloor$

- *Compute* $\tau_\Xi^{v,\gamma}$ *(*$\forall v \in R, \forall \gamma \in \Gamma$*) by*

$$\tau_1 = ... = \tau_k = (\Lambda^{v,\gamma}, \lfloor \frac{c\Xi^{v,\gamma}}{cn^{v,\gamma}} \rfloor + 1, \Lambda^{v,\gamma})$$
$$\tau_{k+1} = (\Lambda^{v,\gamma}, \lfloor \frac{c\Xi^{v,\gamma}}{n^{v,\omega_S,\gamma}} \rfloor + Z - k\lfloor \frac{Z}{k} \rfloor, \Lambda^{v,\gamma})$$
$$\tau_{k+2} = ... = \tau_{cn^{v,\gamma}} = (\Lambda^{v,\gamma}, \lfloor \frac{c\Xi^{v,\gamma}}{cn^{v,\gamma}} \rfloor, \Lambda^{v,\gamma})$$

where:

  - $c\Xi^{v,\gamma} = max(\Xi^{v,LO,\gamma}, \Xi^{v,HI,\gamma})$
  - $cn^{v,\gamma} = max(n^{v,LO,\gamma}, n^{v,HI,\gamma})$
  - $Z = c\Xi^{v,\gamma} - cn^{v,\gamma}\lfloor \frac{c\Xi^{v,\gamma}}{cn^{v,\gamma}} \rfloor$
  - $k = \lfloor Z \rfloor$

The task set τ_Θ defines all VCPUs of a domain. The τ_Θ task set will be composed of $cm = max(m^{LO}, m^{HI})$ tasks. Since the system is running in either LO or HI mode, there are no distinct VCPUs for each criticality level, instead the level that presents more units in the DRM is chosen. Similarly, task sets of type $\tau_\Xi^{v,\gamma}$ will define the IOVCPUs for each v class and γ operation. There will be $v \cdot \gamma$ task sets of the type$\tau_\Xi^{v,\gamma}$, each one with size $cn^{v,\gamma} = max(n^{v,LO,\gamma}, n^{v,HI,\gamma})$.

Theorem 2 *(System Schedulability)*

- *Given that the system is composed of ND domains, and that MS is the set of all of their respective* DRM*s.*
- $\tau_{sys} = \cup(\tau_{\mathcal{M}})_{\forall \mathcal{M} \in MS}$ *represents the task set of the whole system.*
- *Given* $\tau^{v}_{\Xi sys}$ *as the set of all* IOVCPU *task sets (indexed by* I/O *device class v).*
- *Given a function $g^{v} : R \to R'$ that maps time units onto the usual resource units of the* I/O *device class (e.g. bytes per second).*
- *Given the set of all physical* I/O *device φR classes that has a 1 to 1 correspondence with R.*
- *Given that the budget provided by a physical* I/O *device class v (in the usual resource units of the* I/O *device class) is $n^{v,\omega_S,\gamma} X^{v}$.*

The system is schedulable by the underling physical platform if

1. τ_{sys} *is schedulable by the platform* PCPU*s*
2. *and* $\forall v \in \varphi R :$
   $\sum_{\tau_i \in \tau^{v}_{\Xi sys}} g(E_i) \leqslant n^{v,\omega_S,\gamma} X^{v}$

Condition 1 of Theorem 2 ensures the schedulability of all VCPUs and IOVCPUs in the system. Condition 2 of Theorem 2 ensures that the required budget of all virtual I/O devices will not be grater then the provided by their physical counterparts.

5.6 DRM GENERATION

This section introduces the proposed algorithms for generating a DRM that is able to schedule a given domain. The proposed algorithm for DRM generation is Algorithm 1, which uses the Algorithm 2 as its core.

The inputs for Algorithm 1 are the domain task set (τ), the domain criticality ($\omega_{\mathcal{D}}$), the DRM VCPU (Π) and virtual I/O device ($\Lambda^{v,\gamma}$) periods, the set of system criticality-levels (Ω), the set of virtual I/O devices classes ($\mathcal{R}$), and the set of virtual I/O devices operations (Γ). The output of the algorithm it a DRM ($\mathcal{M}$) that has the minimum required resources to schedule the given domain. In this version of the algorithm the domain always have some VCPU load but the virtual I/O device load can be zero. In that case, the original virtual I/O device

ALGORITHM 1 DRM generation.

Input: τ, $\omega_{\mathcal{D}}$, Π, $\Lambda^{v,\gamma}$, Ω, $\mathcal{R}$, Γ
Output: $\mathcal{M} = \langle \omega_{\mathcal{M}}, \Pi, \Theta^{\omega_{\mathcal{S}}}, m^{\omega_{\mathcal{S}}}, \Lambda^{v,\gamma}, \Xi^{v,\omega_{\mathcal{S}},\gamma}, n^{v,\omega_{\mathcal{S}},\gamma} \rangle$
  $\omega_{\mathcal{M}} \leftarrow \omega_{\mathcal{D}}$
  $\Theta^{\omega_{\mathcal{S}}}, m^{\omega_{\mathcal{S}}} \leftarrow$ Budget and quantity for specifics $\forall\ \omega_{\mathcal{S}} \in \Omega$
  $\Xi^{v,\omega_{\mathcal{S}},\gamma}, n^{v,\omega_{\mathcal{S}},\gamma} \leftarrow$ Budget and quantity for specifics $\forall\ v \in \mathcal{R}, \omega_{\mathcal{S}} \in \Omega, \gamma \in \Gamma$
  if all $n^{v,\omega_{\mathcal{S}},\gamma}$ are zero **then**
    $\Lambda^{v,\gamma} \leftarrow 0$
  end if
  create $\mathcal{M} = \langle \omega_{\mathcal{M}}, \Pi, \Theta^{\omega_{\mathcal{S}}}, m^{\omega_{\mathcal{S}}}, \Lambda^{v,\gamma}, \Xi^{v,\omega_{\mathcal{S}},\gamma}, n^{v,\omega_{\mathcal{S}},\gamma} \rangle$
  ensure $\mathcal{M}$ schedules τ (using Theorem 1)

period is adjusted to zero. A similar adjustment can be performed for VCPU period if the VCPU load is allowed to be zero in the domain. However, a domain that has both VCPU and virtual I/O device loads as zero makes no practical sense. Algorithm 1 tries to generate a feasible quantity and budget for all criticality levels, and devices classes and operations. For such, it uses the Algorithm2.

The Algorithm 2 is based on the descriptions of how to generate a MPR from the (EASWARAN; SHIN; LEE, 2009) paper. In the (EASWARAN; SHIN; LEE, 2009) paper, however, the actual algorithm is not shown. Besides, Algorithm 2 is designed to support MC, and distinct virtual I/O device classes and operations. The $|\tau|$ of Algorithm 2 denotes the number of guest OS tasks in the domain. The Algorithm 2 performs a linear search on all possible quantities and budgets until it founds the minimal ones that are able to schedule the given domain while using the given parameters (criticality level and, for virtual I/O devices, the device class and operation).

5.7 VIRTUALIZATION OVERHEAD

Before introducing the formalization of sources of memory and I/O overhead, which is done in Section 5.8, it is important to take into account the overhead introduced by the hypervisor itself. This section formalizes hypervisor overheads, which are later used for inflating the required budget of guest OS tasks.

Two types of hypervisor overhead are associated with the MC nature of guest OS tasks scheduling employed in this proposal. The first

ALGORITHM 2 Budget and quantity for specifics.

Input: τ, isvcpu, $\omega_{\mathcal{S}}$, $\omega_{\mathcal{M}}$, (Π or $\Lambda^{v,\gamma}$), Ω, ($v, \gamma, \mathcal{R}$, Γ only if isvcpu = False)

Output: ($\Theta^{\omega_{\mathcal{S}}}, m^{\omega_{\mathcal{S}}}$ if isvcpu = True) or ($\Xi^{v,\omega_{\mathcal{S}},\gamma}, n^{v,\omega_{\mathcal{S}},\gamma}$ if isvcpu = False)

- **if** isvcpu = True **then**
  - $\Upsilon \leftarrow \Pi$
  - $minq \leftarrow U^{\omega_{\mathcal{S}}}_{\tau\Theta}$
  - $maxq \leftarrow \frac{\sum_{\tau_i \in \tau} C_i^{\omega_{\mathcal{S}}}}{min_{\tau_i \in \tau}(D_i - C_i^{\omega_{\mathcal{S}}})} + |\tau|$
- **else**
  - $\Upsilon \leftarrow \Lambda^{v,\gamma}$
  - $minq \leftarrow U^{v,\omega_{\mathcal{S}},\gamma}_{\tau\Xi}$
  - $maxq \leftarrow \frac{\sum_{\tau_i \in \tau}(\sum_{v \in \mathcal{R}, \gamma \in \Gamma} R_i^{v,\omega_{\mathcal{S}},\gamma})}{min_{\tau_i \in \tau}(D_i - R_i^{v,\omega_{\mathcal{S}},\gamma})} + |\tau|$
- **end if**
- **for all** $q \in [minq, maxq]$ **do**
  - $\omega_{\mathcal{M}} \leftarrow \omega_{\mathcal{D}}$
  - $minbud \leftarrow budstep$
  - $maxbud \leftarrow \Upsilon \cdot q$
  - $bud \leftarrow minbud$
  - **while** $bud < maxbud$ **do**
    - **if** Theorem 1 condition = True for the given parameters **then**
      - **returns** (bud, q)
    - **end if**
    - $bud \leftarrow bud + budstep$
  - **end while**
- **end for**
- **raise** Impossible to generate the DRM for the given parameters

one, given by Definition 28, is the overhead for switching the system mode from *LO* to *HI* mode. The second one, given by Definition 29, is the overhead for switching the system mode from *HI* to *LO* mode.

Definition 28 *(LO to HI mode switch overhead)*
$\delta_{LO_{to}HI} = \Delta_{P_{OFF}}$
where

- $\Delta_{P_{OFF}}$ *is the time needed to perform the power OFF of LO-domains or LO-domains processing elements.*

Definition 29 *(HI to LO mode switch overhead)*

$\delta_{HI_{to}LO} = \Delta_{P_{ON}}$
where

- $\Delta_{P_{ON}}$ *is the time needed to perform the power ON of LO-domains or LO-domains processing elements.*

The LO to HI mode switch occurs every time the hypervisor detects that HI-tasks will become unschedulable at LO mode (the scheduling test is presented in Section 5.8) taking into account LLC miss and interconnect contention overhead. The LO to HI mode switch overhead quantifies the cost of such mode switch. Conversely, the HI to LO mode switch overhead occurs whenever the system switches back from HI to LO mode.

Definition 30 summarizes the mode switch overhead depending on the current system mode and whether the decision of performing the mode switch has been taken.

Definition 30 *(Mode Switch Overhead)*

$$\delta_{ms} = \begin{cases} 1^{st}\ case : \omega_{\mathcal{S}} = LO \wedge\ testfails \\ \quad \delta_{LO_{to}HI} \\ 2^{nd}\ case : \omega_{\mathcal{S}} = HI \wedge noHIjob \\ \qquad \wedge\ ELAPSED \wedge\ testpass \\ \quad \delta_{HI_{to}LO} \\ Otherwise : \\ \quad 0 \end{cases} \tag{5.15}$$

The first case of Definition 30 indicates the HI to LO mode switch case, the system is in LO mode, and the overhead-aware scheduling test has failed. The second case indicates the LO to HI mode switch case, the system is in HI mode for a given $ELAPSED$ time, there is no job of HI-task pending ($noHIjob$), and the overhead-aware scheduling test has passed. The idea of $ELAPSED$ is to spend some time in HI mode before switching back to LO mode even if the scheduling test pass. That can be used to give the system some “inertia”, preventing needless back-and-forth HI to LO mode switching. The other cases represent the decision of not performing the mode switch (either because the system is in LO mode and the scheduling test pass or because the system is in HI mode and the conditions for switching back to LO mode have not being fulfilled yet).

The Mode Switch Overhead only includes the time for powering OFF or ON processing elements, but not the time taken to decide

whether to do so. That is part of the "Host Scheduling Overhead", introduced by Definition 31.

Definition 31 *(Host Scheduling Overhead)*
$\delta_{hyper} = \Delta_{test} + \delta_{ms} + \Delta_{rel} + \Delta_{sch} + \Delta_{cxtst} + \Delta_{cxtld}$
where

- Δ_{test} *is the scheduling test delay, defined as the time taken to decide whether a system mode switch shall happen.*
- δ_{ms} *is the mode switch delay, as given by Definition 30.*
- Δ_{rel} *is the release delay, defined as the time between the scheduling interrupt in the hypervisor (i.e. host-level), which indicates that a virtual processing element is ready for executing and the time it is selected to be scheduled.*
- Δ_{sch} *is the scheduling delay, defined as the time for choosing a virtual processing element to execute.*
- Δ_{cxtst} *is the context store delay, defined as the time taken to store the context of the virtual processing element that will leave the physical processing element.*
- Δ_{cxtld} *is the context load delay, defined as the time taken to load the context of the virtual processing element that was chosen to execute into it correspondent physical processing element.*

The host scheduling overhead includes three main components. The first one is the time the hypervisor takes to decide whether a mode switch shall me performed (Δ_{test}) based on the sampled interference values and on the overhead-aware scheduling test introduced in Section 5.8. The second one is the time to perform the mode switch itself (δ_{ms}). The third one is the time associated with the scheduling of virtual processing elements (VCPUs and virtual I/O devices on the physical processing elements). This overhead occurs on each Π (VCPU period) or $\Lambda^{v,\gamma}$ (virtual I/O device period).

5.8 OVERHEAD-AWARE COMPOSITIONAL SCHEDULABILITY ANALYSIS

This section formalizes the sources of memory and I/O overhead presented in Sections 4.4 and 4.2, respectively, and it presents the schedulability conditions taking into account such overheads.

5.8.1 LLC miss overhead

Definition 32 *(LLC miss overhead)*

$$\delta_{\Theta}^{\omega_{\mathcal{S}}} = \begin{cases} 1^{st}\ case : \omega_{\mathcal{S}} = HI \\ WC_{miss}\Delta_{miss} \\ 2^{nd}\ case : \omega_{\mathcal{S}} = LO \\ AVG_{miss}\Delta_{miss} \end{cases} \tag{5.16}$$

where

- Δ_{miss} *is the delay caused by one cache miss event (choice of which cache line will be evicted and its replacement by the contents retrieved from the main memory).*
- WC_{miss} *is the maximum number of cache misses events during a period.*
- AVG_{miss} *is the average number of cache misses events during a period.*

How tasks are affected, inflation of $C_i^{\omega_{\mathcal{S}}}$

Definition 33 *(Inflated required VCPU budget)*
$C_i^{\omega_{\mathcal{S}}''} = C_i^{\omega_{\mathcal{S}}} + \lceil \frac{T_i - (\sum_{x\in\mathcal{R},\gamma\in\Gamma} R_i^{x,\omega_{\mathcal{S}},\gamma})}{\Pi} \rceil \delta_{\Theta}^{\omega_{\mathcal{S}}}$

5.8.2 Interconnect Contention overhead

The interconnect overhead that each virtual I/O device will suffer comes from the interconnect time that its physical counterpart will experience on the interconnect element it is directly connected to. The interconnect element of the physical I/O device in question is obtained from the interconnect topology that follows the structure described in Section 5.4. Definition 34 shows how to compute the contention time $CT^{v,\omega,\gamma}$ on the interconnect used by the I/O device of class v while $\omega_s = \omega$ and I/O device is executing γ operation. The equation assumes the interconnect element (queue) with deterministic arrival and servicing times that follow a Poisson distribution (M/D/1 queue). M/D/1 queues assume one server and that is always valid in the considered topologies since each I/O device is connected to a single interconnect element.

Definition 34 *(Interconnect contention time)*

$$CT^{v,\omega_S,\gamma} = \frac{\lambda_{inter}^{\gamma,cl2lv(\omega_S)}}{2\mu_{inter}^{\gamma,cl2lv(\omega_S)}(\mu_{inter}^{\gamma,cl2lv(\omega_S)} - \lambda_{inter}^{\gamma,cl2lv(\omega_S)})} \tag{5.17}$$

where

1. $\lambda_{inter}^{\gamma,cl2lv(LO)}$ *and* $\mu_{inter}^{\gamma,cl2lv(LO)} \in V$ *of* $Top(LO)$
2. $\lambda_{inter}^{\gamma,cl2lv(HI)}$ *and* $\mu_{inter}^{\gamma,cl2lv(HI)} \in V$ *of* $Top(HI)$

Observations 1 and 2 remember that not only the bandwidth level (see Definition 16) changes but also the topology is different when changing ω_S. That is because in $Top(HI)$ the processing elements associated with LO-domains are powered OFF.

The interconnect contention overhead, as show by Definition 35 is exactly the interconnect contention time.

Definition 35 *(Interconnect contention overhead)*
$\delta_{\Xi}^{v,\omega_S,\gamma} = CT^{v,\omega,\gamma}$

Queuing theory usually works with distributions that are represented by their average time, thus becoming unsuitable for modeling the worst-case required while modeling HRT systems. This work proposes to overcome such limitation by using the worst-case time as the average time for computing the system requirements in the HI criticality level. Therefore, values above the average will not occur in practice. For the LO criticality level, this approach will use the average time values for computing system requirements. Therefore, this approach is pessimistic in the HI criticality level and takes into account the average case in the LO criticality level.

How tasks are affected, inflation of $R_i^{v,\omega,\gamma}$

Definition 36 *(Inflated required virtual I/O device budget)*
$R_i^{v,\omega_S,\gamma''} = R_i^{v,\omega_S,\gamma} + \lceil \frac{T_i - (C_i^{\omega_S} + \sum_{x \in \mathcal{R} \setminus v, \gamma \in \Gamma} R_i^{x,\omega_S,\gamma})}{\Lambda^{v,\gamma}} \rceil \delta_{\Xi}^{v,\omega_S,\gamma}$

5.8.3 Schedulability Analysis

A domain $\mathcal{D}$ composed of a task set τ is scheduled under gEDF by a DRM in the presence of memory and interconnect overhead if the

inflated task set τ'' is scheduled under gEDF by the same DRM while using the SBF in the absence of overhead.

The same goes for the whole system schedulability after transforming all DRM into tasks as explained by Definition 27.

5.9 OVERALL DYNAMICS

The scheduling strategy can be divided in two parts, one during *design time* and the other during *runtime*. The goal of the design time part is to compute DRMs for HI-domains in a way to ensure their schedulability in HI and LO modes while taking into account LLC miss and I/O interconnect contention overheads.

Two variants of the design time part are envisioned. The first one (summarized in Section 5.9.1) handles the cases where both HI and LO domains requirement are known beforehand. The second one (summarized in Section 5.9.2) handles the cases where only HI requirements are known beforehand.

The runtime part of both variants is similar, and the LO to HI mode switch can be triggered either reactively or predictively. In the reactive mode switch, the system switches from LO to HI mode whenever a HI-task job is not able to complete using its LO-budget (either C^{LO} or $R^{v,LO,\gamma}$). In the predictive mode the system will be operating on a *Monitoring-decide-act* cycle where decisions are taken. In such *Monitoring-decide-act* cycle, overhead-related information is gathered and used to predict the inflation of C^{LO} and $R^{v,LO,\gamma}$ of HI-tasks. Such inflated budgets are compared with the DRM of the domains obtained from the design phase and an overhead-aware schedulability test is performed. If the overhead-aware schedulability test fails the system switches to HI mode.

In the event of a LO to HI mode switch, processing elements assigned to LO-domains and the LO-domains themselves are powered OFF as described in Section 4.2. As the computed overhead starts to decrease, the inflation on the execution budgets will decrease and the overhead-aware schedulability will pass. Then, processing elements assigned to non-critical domains can be powered ON again, gradually. Additionally, a HI to LO mode switch will only occur when there are no pending HI-jobs (i.e. all HI-tasks have concluding their jobs in HI mode for that period).

Regarding the overhead-related information, Δ_{miss}, WC_{miss}, and AVG_{miss} can be obtained by consulting HPC registers. All vari-

ables are measured in the design time while executing HI-domains in conjunction with known LO-domains (or with the Slack Domain). Those LLC-related variables are sampled on each VCPU period. During runtime, all variables are continued to being measured in the *Monitoring-decide-act* cycle while executing all loaded domains.

5.9.1 Variant 1: Knowing the requirements of LO-domains

Design Time

1. Requirements of HI-domains and LO-domains are given
2. Physical Topology and physical allocation are given
   (a) $\delta_{\Xi}^{v,\omega_{\mathcal{S}},\gamma}$ is computed (using $Top(FULL)$ and the known $Top(LO)$)
3. HI-domains and known LO-domains run, one by one, cumulatively
   (a) WC_{miss} and AVG_{miss} are measured
   (b) δ_{Θ}^{HI} and δ_{Θ}^{LO} are computed
   (c) The mappings C^{HI} to δ_{Θ}^{HI} and C^{HI} to δ_{Θ}^{LO} are recorded
4. A function $f_{C\delta\Theta}$ that approximates the mapping $C^{\omega_{\mathcal{S}}}$ to $\delta_{\Theta}^{\omega_{\mathcal{S}}}$ is computed. Are also computed the total VCPU requirement (sum of all domains), denoted as $C_{all}^{\omega}{}_{\mathcal{S}}$.
5. Task's C^{HI} and $R^{v,HI,\gamma}$ are inflated using $\delta_{\Theta}^{HI} = f_{C\delta\Theta}(KC_{all}^{HI})$ and $\delta_{\Xi}^{v,FULL,\gamma}$, while task's C^{LO} and $R^{v,LO,\gamma}$ are inflated using $\delta_{\Theta}^{LO} = f_{C\delta\Theta}(KC_{all}^{LO})$ and $\delta_{\Xi}^{v,LO,\gamma}$. K is a constant.
6. DRMs are computed using the inflated tasks. The constant K opens the possibility (if $K > 1$) of accepting new domains that arrive at runtime.
7. Domains schedulability and system schedulability is checked/ensured (for all $\omega_{\mathcal{S}}$ modes).

Runtime

1. HI-domains and known LO-domains start to run

2. Eventually, new LO-domains can be added if their requirement is known and there is enough budget for them
3. *Monitoring-decide-act cycle*

In Step 2 of runtime, the hypervisor uses the new LO-domain requirements as inputs to check HI-domains (in HI mode) and system (all modes) schedulability. If it detects that one of the HI-domains (in HI mode) or the system will become unschedulable (overhead-aware test) the new LO-domain is not admitted in the system. Regarding LLC miss overhead, the overhead the new LO-domain can cause in the system is estimated using $f_{C\delta\Theta}$. Regarding interconnect contention overhead, the overhead is computed using an updated (the new domains can use devices that the domains in the design phase did not use) version of $Top(LO)$.

5.9.2 Variant 2: Using the Slack Domain

Design Time

1. Requirements of HI-domains are given
   (a) DRMs are computed *(optional intermediate step)*
2. Physical Topology is given
   (a) $\delta_{\Xi}^{v,\omega_S,\gamma}$ is computed
3. HI-domains run alone
   (a) WC_{miss} is measured
   (b) Task's C^{HI} and $R^{v,HI,\gamma}$ are inflated using WC_{miss} and $\delta_{\Xi}^{v,HI,\gamma}$
   (c) 3b $\mapsto$ Slack Domain is computed
   (d) (2a, 3a) $\mapsto$ DRMs are recomputed to attend the inflated tasks *(optional intermediate step)*
4. HI-domains and Slack Domain run
   (a) AVG_{miss} is measured
   (b) Task's C^{LO} and $R^{v,LO,\gamma}$ are inflated using AVG_{miss} and $\delta_{\Xi}^{v,LO,\gamma}$ (HI-domains only)

(c) 4b ↦ Slack Domain is recomputed

(d) (2a, 4a) ↦ DRMs are recomputed to attend the inflated tasks *(optional intermediate step)*

5. (2a, 3a, 4a) ↦ HI-domain DRMs are computed

6. (4c) ↦ DRM for Slack Domain is computed

7. HI-domains schedulability and system schedulability is checked/ensured.

Runtime Time

1. HI-domains run alone

2. LO-domains start to run (one by one or simultaneously), *each using a fraction of Slack Domain budgets*

3. *Monitoring-decide-act cycle*

The AVG_{miss} measured when LO-domains start to run (in conjunction with HI-domains) should have a similar value to the AVG_{miss} measured while running HI-domains and the Slack Domain together (in the design phase). In practice, due to hypervisor overhead (e.g. VCPU scheduling), it is possible that the former is higher than the latter since LO-domains will be composed of several tasks while Slack Domain has a single task.

5.10 DISCUSSION

The proposed model extends the VCPU-related definitions found in (EASWARAN; SHIN; LEE, 2009) to a MC scenario and presents virtual I/O device definitions that encompass MC, two operations ("read" and "write"), and multiple virtual I/O device classes.

Additionally, the proposed model extends the cache-related overhead definitions found in (XU et al., 2013) to a MC scenario and presents new virtual I/O device-related overhead definitions based on queuing theory.

The proposed hypervisor overhead definition, presented in Section 5.7 is based on the one presented by (NELSON et al., 2014), extending it for virtual I/O devices. The goals, however, are different. While (NELSON et al., 2014) uses similar definition for defining cycle-accurate virtual processor context switch, this works focuses on defining a scheduling overhead to be used into account on scheduling tests.

6 EVALUATION OF THE PROPOSED MODEL

This chapter evaluates the proposed design time and runtime strategy introduced in Section 5.9. Such strategy, in turn, employs the modeling approach proposed in Chapter 5, and the techniques described in Chapter 4.

All evaluations were performed analytically by simulating practical executions, instead of performing experiments in a real platform. The main reason for employing an analytical approach is that, as far was possible to identify, no single hypervisor implements all the techniques required by this work proposal such as, composite MC real-time scheduling, powering ON and OFF physical and virtual devices, page-coloring, and time-deterministic multilevel interrupt handling.

Section 6.1 evaluates the proposed scheduling test for distinct domains and guest OS tasks, while comparing it to a gEDF scheduling test.

Section 6.2 evaluates the impact that guest OS tasks of LO-domains have on guest OS tasks of a HI-domain. First, it evaluates the LLC miss overhead over execution time while the HI-domain executes together with a LO-domain expected at design time and with two additional LO-domains that come at runtime. Then, it compares the proposed approach with the MPR approach introduced by (EASWARAN; SHIN; LEE, 2009) regarding PCPU requirement, and schedulability decisions over time (DRM vs MPR).

Section 6.3 closes the chapter presenting a discussion of the obtained results and their implication while compared to related works.

6.1 SCHEDULABILITY EVALUATION

This section evaluates the overhead-free domain schedulability test introduced by Definition 1 in Section 5.5 for distinct task sets and distinct whole-domain utilizations. From the point-of-view of the DRM-based approach, each task set is a domain, composed of guest OS tasks. To give an idea of how the proposed domain schedulability test performs, it is compared to the gEDF schedulability test proposed by (GOOSSENS; FUNK; BARUAH, 2003) that is shown is Section 2.2. It should be noted, however, that the two tests are not direct comparable since the domain schedulability test takes into account a virtualized platform where the VCPUs are not available to the domain one hundred

percent of the time, which is the case of gEDF that is agnostic from whether the underling platform is virtual or physical. The evaluation takes into account a single HI-domain and, for comparing it to gEDF, it takes into account only the HI budget of tasks (the LO budget of every task is zero). The number of processors, used as input for the gEDF test (m) is taken from the DRM used for the domain schedulability test (i.e. $m = m^{HI}$).

The method for generating the task sets is based on the method presented by (BASTONI; BRANDENBURG; ANDERSON, 2010) for multiprocessor platforms. The proposed method extends (BASTONI; BRANDENBURG; ANDERSON, 2010) to take into account not only CPU but also I/O device budgets. The proposed method generates synthetic task sets, each task having distinct VCPU and virtual I/O device budgets. First, the proposed method generates task sets while ranging the whole-domain utilization from 2 to 8 using a step of 1, thus generating 6 whole-domain utilization "steps". The proposed method generates a DRM capable of scheduling all domains (all task sets) generated from the [2, 8] whole-domain utilization range. A whole-domain utilization is computed not only from VCPU budget but also from virtual I/O device budget. Therefore, the proposed method keeps generating task sets until the VCPU utilization of a task set ($U^{\omega_S}_{\tau\Theta}$) reaches the maximum established whole-domain utilization (i.e.: until $U^{\omega_S}_{\tau\Theta} = 8$). Each task is generated by using random values of a given distribution. The period of each guest OS task is randomly obtained from an uniform distribution in the interval [25 ms, 200 ms]. The whole-task utilization of each guest OS task is randomly obtained from an uniform distribution in the interval [0.1, 0.9]. The whole-task utilization if further divided in order to compute the VCPU and virtual I/O device budgets. The gEDF test only takes into account the VCPU budget, ignoring the virtual I/O device ones. The domain schedulability test takes into account all those budgets.

The Listing shown in Figure 15 illustrates how the whole-task utilization is divided in order to compute the VCPU and virtual I/O device budgets of each task. As show in the Listing, in the evaluation of this section two classes of virtual I/O device are taken into account, named CM and FS. The multiplication of the α_i values of Listing 15 with the whole-task utilization gives the utilization for each type of budget. The subsequent multiplication by the task period gives the budget. The Dirichlet distribution (BALAKRISHNAN; NEVZOROV, 2003) is employed in such a way the sum of all randomly generated α_is is 1 (one). Additionally, each α_i individually fits in the interval [0.01, 1].

- $C_i^{HI} = \alpha_0 \cdot WTU_i^{HI} \cdot T_i$
- $R_i^{CM,HI,rd} = \alpha_1 \cdot WTU_i^{HI} \cdot T_i$
- $R_i^{CM,HI,wr} = \alpha_2 \cdot WTU_i^{HI} \cdot T_i$
- $R_i^{FS,HI,rd} = \alpha_3 \cdot WTU_i^{HI} \cdot T_i$
- $R_i^{FS,HI,wr} = \alpha_4 \cdot WTU_i^{HI} \cdot T_i$

Figure 15: Division of WTU_i^{HI}.

Table 7: Generated DRM.

	DRM
$\omega_{\mathcal{M}}$	HI
Π	1.0E-03
Θ^{HI}	3.5E-03
m^{HI}	4
$\Lambda^{CM,rd=wr}$	1.0E-03
$\Xi^{CM,HI,rd=wr}$	3.5E-03
$n^{CM,HI,rd=wr}$	4
$\Lambda^{FS,rd=wr}$	1.0E-03
$\Xi^{FS,HI,rd=wr}$	3.5E-03
$n^{FS,HI,rd=wr}$	4

The first obtained α_i is the one that will define the VCPU budget, while the others will define the virtual I/O device budgets. The order of each virtual I/O device class and operation is also randomly obtained. The Listing 15 shows just one example of order in which the CM, rd appear first but it could be for example, first FS, wr, followed by CM, wr or any other possible combination.

Table 7 shows the generated DRM employed in the domain schedulability test analysis. For the gEDF test it was used $m = 4$.

Figure 16 shows the obtained results of the DRM-based and (GOOSSENS; FUNK; BARUAH, 2003) gEDF schedulability tests, while using the proposed method for task set generation. DRM can schedule all task sets up to whole-domain utilization equals 8. gEDF also goes up to 8 (in which the total VCPU utilization $U_{\tau\Theta}^{\omega_S} = 1.9$). DRM and gEDF fail to schedule with whole-domain utilization equals to 9 (in which

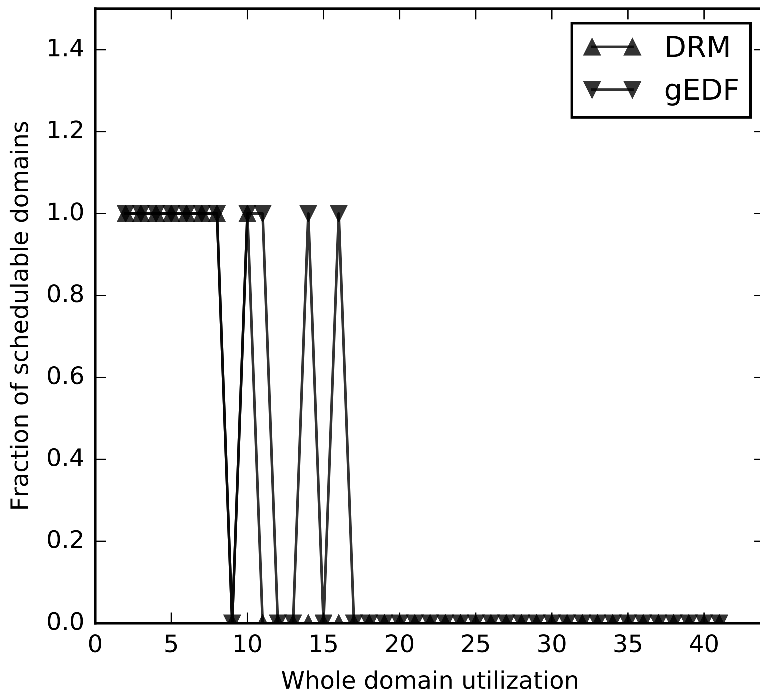


Figure 16: Schedulability Analysis DRM vs. gEDF.

$U^{\omega_S}_{\tau\Theta} = 2.4$). In that case, the gEDF schedulability test fails because the maximum task utilization is equal or higher than 0.6. gEDF can schedule some values of whole-domain utilization higher than 9 (such as 10) because in that case $U^{\omega_S}_{\tau\Theta}$ decreased to 1.8. Since the task set is randomly generated, it is ensured that whole-domain utilization increases monotonically, however, the $U^{\omega_S}_{\tau\Theta}$ can decrease or increase, thus a task set with higher whole-domain utilization can be scheduled again because it has a lower $U^{\omega_S}_{\tau\Theta}$. As show in the figure, gEDF can schedule all cases that DRM can. That is expected since the DRM scheduling test assumes gEDF will be used for intra-domain scheduling.

6.2 ANALYSIS CASE

The analysis case presented in this section demonstrates how guest OS tasks running on HI-domains can suffer interference caused by operations issued by guest OS tasks running on LO-domains. In this case, the interference is perceived as an increase in the LLC miss rate as described in Section 4.4 and depicted in Figure 14.

Figure 17 shows the considered physical topology. The presented topology models a usual Commercial off-the-shelf (COTS) topology, such as the one used by the Intel Sandy Bridge microarchitecture (INTEL, 2016a). While the processing elements P_0 through P_7, and the FS_1 device, together with the memory interconnect can be seen as the "processor" region of the topology (with the fastest interconnect); the CM_1 and CM_2 devices together with the I/O interconnect can be seen as the Cougar Point Platform Controller Hub (PCH) implemented by the H67 chipset (with the slowest interconnect) (INTEL, 2011).

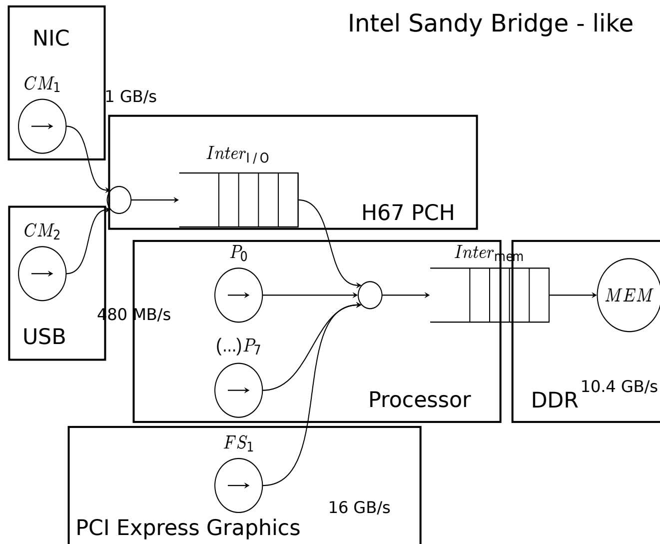


Figure 17: Base topology. Known-LO-domains case.

Table 8 shows the insertion and servicing rates of each element of the base topology. The symbols $\lambda_x^{\gamma,lv}$ and $\mu_x^{\gamma,lv}$ with a subscribed "x"

Table 8: Base topology insertion and servicing rates. Known-LO-domains case. Units in bytes per seconds (B/s).

PE	$\boldsymbol{\lambda_x^{rd=wr,MAX}}$	$\boldsymbol{\lambda_x^{rd=wr,AVG}}$	$\boldsymbol{\mu_x^{rd=wr,MAX}}$	$\boldsymbol{\mu_x^{rd=wr,AVG}}$
CM	2 x 1.98E-08	2 x 6.6E-07		
FS	1 x 7.26E-08	1 x 3.3E-08		
P	8 x 6.6E-08	8 x 3.3E-08		
$Inter_{I/O}$	3.96E-08	13.2E-07	6.6E-09	5.94E-09
$Inter_{mem}$	6.402E-09	3.102E-09	6.6E-09	5.94E-09
MEM			6.6E-09	5.94E-09

are used instead of the usual $\lambda_{pe}^{\gamma,lv}$, $\lambda_{inter}^{\gamma,lv}$, $\mu_{inter}^{\gamma,lv}$, and $\mu_{me}^{\gamma,lv}$ for generality. Cells are left in blank in the non-applicable cases (e.g. servicing rate for processing elements). While Figure 17 presents the nominal values of insertion and servicing rates of the elements, Table 8 presents the maximum values (MAX) and the average (AVG) used in this analysis case. The insertion and servicing rates of Table 8 are arbitrary, being bellow the nominal values of the Intel Sand Bridge microarchitecture, and respecting the assumptions of Definition 21 that defines the physical topology: (i) $\mu_{inter}^{\gamma,lv} > \lambda_{inter}^{\gamma,lv}$, (ii) $\lambda_x^{\gamma,MAX} > \lambda_x^{\gamma,AVG}$, (iii) $\mu_x^{\gamma,MAX} > \mu_x^{\gamma,AVG}$. Assumption (i) comes from queuing theory and prevents the contention time to be infinite. Assumptions (ii) and (iii) respect the MC definitions of the model (where the budgets in HI mode shall be greater than the budgets in LO mode). Respecting the guest OS task and DRM definitions, processing elements of the same class (e.g. CM_1 and CM_2 of CM class) have the same insertion and servicing rates. Additionally, it is assumed in this topology that the insertion and servicing rates of read and write operations are the same.

The interconnect servicing and insertion rates follows $\lambda_{inter}^{\gamma,lv} = \sum_{x \in \mathbf{income}(inter)} \lambda_x^{\gamma,lv}$ from Definition 21. For instance, $\lambda_{Inter_{mem}}^{rd=wr,MAX} = \lambda_P^{rd=wr,MAX} + \lambda_{FS}^{rd=wr,MAX} + \lambda_{Inter_{I/O}}^{rd=wr,MAX}$ (6.402E-09 = 8 x 6.6E-08 + 1 x 7.26E-08 + 3.96E-08). Table 8 present insertion and servicing rates for all processing elements ON (i.e. "full topology"). It is important to notice that as processing elements are powered OFF, the insertion rate of their classes' decreases.

In this analysis case, one HI-domain (Dom_1) and one LO-domain (Dom_2) are given at design time. Table 9 summarizes their original requirements taking into account an overhead-free scenario, while Table

10 shows the employed physical allocation. One HI-domain and one LO-domain is the minimal domain quantity needed to exercise the design part of the strategy proposed in Section 5.9.1 in which the requirements of LO-domains are known. The task set within each domain can be seen as *one instance* of the task sets generated using synthetic task set generation methods such as the ones employed in (BASTONI; BRANDENBURG; ANDERSON, 2010), and used again by (XI et al., 2014) in a virtualized platform. The physical allocation is arbitrary, respecting the Definition 22, which states that HI-domains have exclusive access to their processing elements.

In Table 9, values equal to zero are left as blank space. In the case of virtual I/O devices, Table 9 shows the sum of "read" and "write" budgets. In all computations, such budget is divided in such a way that 70% of the operations are "read" and 30% are "write". Such 70%/30% division is arbitrary serving to exercise the model but does not change the final results since, as Table 8 shows, the insertion rates for "read" and "write" are the same.

Table 11 shows the variations of the physical topology used in this case, along with the interconnect contention overhead experienced by each I/O device. $Full_{MAX}$ is the topology where all processing elements are ON using their maximum insertion rate (MAX). Conversely, $Full_{AVG}$ is the topology where all processing elements are ON but using their average insertion rate (AVG) instead. The third topology in the table, $Dom_{1.2}$, is the topology in place while running Dom_1 and Dom_2 in LO mode (using the average insertion rates of processing elements). In that case, the device FS_1 is OFF as it is not used. In this analysis case, it is considered that all PCPUs are ON in all topologies (only virtual power OFF of CPUs is employed). The interconnect contention overhead is computed using Definition 35, presented in Section 5.8.2, applied on the physical topology.

During the design-phase, as described in Section 5.9.1, domains start to run one by one, first Dom_1 and then $Dom_1$and Dom_2. On each execution scenario (Dom_1 alone and Dom_1+Dom_2) $C^{\omega}_{all\mathcal{S}}$ is computed, WC_{miss} and AVG_{miss} are measured, $\delta^{\omega s}_{\Theta}$ is computed, and the mapping $C^{\omega}_{all\mathcal{S}}$ to $\delta^{\omega s}_{\Theta}$ is recorded. Table 12 shows the values for WC_{miss}, AVG_{miss}, and $\delta^{\omega s}_{\Theta}$ for each execution scenario. In a practical experiment scenario, the values for WC_{miss}, AVG_{miss} are measured using HPCs and the techniques described in Section 4.1. In this analytical analysis case, the values for values for WC_{miss}, AVG_{miss} are arbitrary inputs, simulating values measured in practice. The WC_{miss}, AVG_{miss} values increase as more domains execute, which is expected

Table 9: Original design-time domain requirements. Known-LO-domains case. Units in seconds (s).

	$\boldsymbol{Dom_1(\omega_{\mathcal{D}}}$ =HI)			$\boldsymbol{Dom_2(\omega_{\mathcal{D}}}$ =LO)		
	$\boldsymbol{T_1}$	$\boldsymbol{T_2}$	$\boldsymbol{T_3}$	$\boldsymbol{T_1}$	$\boldsymbol{T_2}$	$\boldsymbol{T_3}$
T_i	25.0E-03	50.0E-03	50.0E-03	50.0E-03	100.0E-03	100.0E-03
D_i	25.0E-03	50.0E-03	50.0E-03	50.0E-03	100.0E-03	100.0E-03
C_i^{LO}	18.87E-03	18.87E-03	9.43E-03	33.02E-03	18.87E-03	28.3E-03
$R_i^{CM,LO,rd+wr}$		14.99E-03	12.49E-03		24.99E-03	14.99E-03
$R_i^{FS,LO,rd+wr}$						
C_i^{HI}	19.23E-03	21.37E-03	12.82E-03			
$R_i^{CM,HI,rd+wr}$		19.99E-03	14.99E-03			
$R_i^{FS,HI,rd+wr}$						

Table 10: Physical allocation. Domains known at design time. Known-LO-domains case.

PE	Domains
P_0, P_1	Dom_1
$P_2, ..., P_7$	Dom_2 (can be shared with others LO-domains)
CM_1	Dom_1
CM_2	Dom_2 (can be shared with others LO-domains)
FS_1	"free" (can be shared with others LO-domains)

Table 11: Interconnect contention overhead (in seconds). Known-LO-domains case.

Top	ω_S	m	n^{FS}	n^{CM}	$\delta_{\Xi}^{FS,\omega_S,\gamma}$	$\delta_{\Xi}^{CM,\omega_S,\gamma}$
$Full_{MAX}$	HI	8	1	2	2.45E-09	4.84E-12
$Full_{AVG}$	LO	8	1	2	9.20E-11	1.91E-12
$Dom_{1.2}$	LO	8	0	2	7.37E-11	1.91E-12

in a practical scenario as well. The $\delta_{\Theta}^{\omega_S}$ is computed using the Definition 32, presented in Section 5.8.1. The execution scenario $Dom_{1.2}$ is a design-phase scenario and indicates that Dom_1 and Dom_2 are running. The other scenarios occur at runtime phase and are explained later.

C^{ω_S} of Dom_1 tasks are inflated using $\delta_{\Theta}^{\omega_S} = f_{C\delta\Theta}(KC_{all}^{\omega_S})$, according to Definition 33 presented in Section 5.8.1, and the strategy presented in Section 5.9.1. $C_{all}^{\omega_S}$ takes into account the computation time of all domains that run in the design phase (Dom_1 and Dom_2).

Table 12: LLC miss values and overhead (in seconds). Known-LO-domains case.

Δ_{miss} = 4.00E-08				
Scenario	WC_{miss}	δ_{Θ}^{HI}	AVG_{miss}	δ_{Θ}^{LO}
Dom_1	2.38E+03	9.50E-05	8.75E+02	3.50E-05
$Dom_{1.2}$	3.13E+03	1.25E-04	1.13E+03	4.50E-05
$Dom_{1.2.3}$	4.25E+03	1.70E-04	1.50E+03	6.00E-05
$Dom_{1.2.3.4}$	5.38E+03	2.15E-04	1.88E+03	7.50E-05

Table 13: Inflated Dom_1 (sum of all tasks) budgets. Known-*LO*-domains case.

Scenario	ω_S	*Top*	$C^{\omega_S''}_{Dom_1}$	$R^{CM,\omega_S,rd+wr''}_{Dom_1}$
Free	*LO*	*N/A*	47.17E-03	27.48E-03
$Dom_{1.2}$	*LO*	$Dom_{1.2}$	49.29E-03	27.48E-03
$Dom_{1.2.3}$	*LO*	$Dom_{1.2}$	50.00E-03	27.48E-03
$Dom_{1.2.3.4}$	*LO*	$Full_{AVG}$	50.71E-03	27.48E-03

$R^{v,\omega_S,\gamma}$ of Dom_1 tasks are inflated using $\delta_{\Xi}^{v,\omega_S,\gamma}$. Since, in this analysis case, $C^{\omega_S}_{all}$ is not measured but $\delta^{\omega_S}_{\Theta}$ comes directly from given WC_{miss} and AVG_{miss} instead, neither $f_{C\delta\Theta}$ nor K are presented here.

Table 13 shows the inflated budgets of Dom_1 for distinct execution scenarios. In this case, the inflation does not use $\delta^{\omega_S}_{\Theta} = f_{C\delta\Theta}(KC^{\omega_S}_{all})$ but the $\delta^{\omega_S}_{\Theta}$ values from Table 12. Therefore, the slack needed for generating Dom_1's DRM is, in this analysis case, obtained by assuming there is another domain (named *synthetic domain*) running together with Dom_1 and Dom_2 in the design phase. As, in this case all domains (including the ones that come at runtime) are actually known beforehand, the requirements of the *synthetic domain* is the maximum of each runtime domain (Dom_3 and Dom_4 presented later). In that way and, as it is shown later in this section, the system will admit one extra domain at runtime but not two.

Table 13 also shows the physical topology used to compute $\delta_{\Xi}^{v,\omega_S,\gamma}$. $C^{\omega_S''}_{Dom_1}$ represents the sum of the inflated VCPU budgets of Dom_1. $R^{CM,\omega_S,rd+wr''}_{Dom_1}$ represents the sum of the inflated virtual I/O device budgets of the same domain.

The execution scenario $Dom_{1.2}$ is a design-time scenario and indicates that Dom_1 and Dom_2 are running. The execution scenarios $Dom_{1.2.3}$ and $Dom_{1.2.3.4}$ are runtime scenarios that happen when Dom_3 and Dom_4 (specifications given later), two *LO*-domains that arrive at runtime join Dom_1 and Dom_2. The *Free* execution scenario is the design-time scenario taking into account the original requirements of Dom_1, without taking into account overheads. Although not shown in the table, the budgets of Dom_2 are also inflated for checking system schedulability (overhead-aware test).

After that, DRMs are computed from the known domains. Their interfaces is shown in Table 14.

The design phase is concluded checking domains and system

Table 14: DRMs. Known-LO-domains case.

	$\boldsymbol{DRM_1}$	$\boldsymbol{DRM_2}$
$\omega_{\mathcal{M}}$	HI	LO
Π	1.0E-03	1.0E-03
Θ^{HI}	3.7E-03	0.0E-03
m^{HI}	4	0
Θ^{LO}	2.7E-03	1.9E-03
m^{LO}	3	2
$\Lambda^{CM,rd=wr}$	1.0E-03	1.0E-03
$\Xi^{CM,HI,rd+wr}$	8.0E-04	0.0E-03
$n^{CM,HI,rd+wr}$	2	0
$\Xi^{CM,LO,rd+wr}$	6.0E-04	5.0E-04
$n^{CM,LO,rd+wr}$	2	2
$\Lambda^{FS,rd=wr}$	0.0E-03	0.0E-03
$\Xi^{FS,HI,rd+wr}$	0.0E-03	0.0E-03
$n^{FS,HI,rd+wr}$	0	0
$\Xi^{FS,LO,rd+wr}$	0.0E-03	0.0E-03
$n^{FS,LO,rd+wr}$	0	0

schedulability.

In this case, two new LO-domains are loaded at runtime, Dom_3 and Dom_4. Table 15 shows their non-inflated requirements. Requirements for HI mode are all zero (and omitted) in the table since both Dom_3 and Dom_4 are LO-domains.

Table 16 shows the updated physical allocation taking into account the resources used by the new domains. Both Dom_3 and Dom_4 share PCPUs with Dom_2. Dom_3 also uses CM_2, while Dom_4 uses FS_1.

Table 15: Non-inflated requirements of new domains. Known-LO-domains case.

	$\boldsymbol{Dom_3}(\boldsymbol{\omega_{\mathcal{D}}}$ =LO)			$\boldsymbol{Dom_4}(\boldsymbol{\omega_{\mathcal{D}}}$ =LO)		
	$\boldsymbol{T_1}$	$\boldsymbol{T_2}$	$\boldsymbol{T_3}$	$\boldsymbol{T_1}$	$\boldsymbol{T_2}$	$\boldsymbol{T_3}$
T_i	25.0E-03	50.0E-03	50.0E-03	100.0E-03	150.0E-03	200.0E-03
D_i	25.0E-03	50.0E-03	50.0E-03	100.0E-03	150.0E-03	200.0E-03
C_i^{LO}	17.5E-03	35.0E-03	20.0E-03	90.0E-03	120.0E-03	50.0E-03
$R_i^{CM,LO,rd+wr}$			15.0E-03			
$R_i^{FS,LO,rd+wr}$						20.0E-03

Table 16: Updated physical allocation. All domains (known *LO*-domains case).

PE	**Domains**
P_0, P_1	Dom_1
$P_2, ..., P_7$	Dom_2, $\boldsymbol{Dom_3}$, $\boldsymbol{Dom_4}$
CM_1	Dom_1
CM_2	Dom_2, $\boldsymbol{Dom_3}$
FS_1	$\boldsymbol{Dom_4}$

The runtime time scenario considered for this analysis case is shown in Figure 18.

The first evaluation of this case was according to LLC miss overhead over execution time. Figure 19 shows the variation of AVG_{miss} along time, showing also scheduling decisions on each state of the runtime scenario. Marks P_{OFF} indicate that the hypervisor has decided to perform a LO to HI mode switch and to power OFF physical or virtual processing elements. Conversely, P_{ON} marks indicate that the hypervisor has decided to perform a HI to LO mode switch and to power ON physical or virtual processing elements. On State 1 AVG_{miss} has the same value as the one it was measured in the design phase. On State 2 LO-domains Dom_3 and Dom_4 start to execute. The AVG_{miss} is sampled and used for inflating the VCPU LO budget of Dom_1. The line $Dom_{1.2.3.4}$ of Table 13 show the inflated budget of Dom_1. Testing domain schedulability using the inflated tasks and the respective domains' DRMs, the hypervisor verifies on State 3 that HI-domains (one or more) will become unschedulable. Then, the hypervisor performs a LO to HI mode switch, choosing to virtual power OFF (VCPUs and virtual I/O devices) Dom_3 and Dom_4 (State 5). The hypervisor detects that HI-domains will become schedulable again in LO mode and decides to perform a HI to LO mode switch (State 6 and State 7). The hypervisor, in this case, decides to turn one the domains one by one. Powering ON Dom_3 the hypervisor detects that HI-domains are still schedulable in LO mode (State 8 and State 9). After that, the hypervisor powers ON Dom_4 but it notices that HI-domains will become unschedulable (State 10 and State 11). Then, the hypervisor decides to power OFF Dom_4 again (State 12 and State 13). The monitoring cycle goes on and, depending on the strategy employed, the hypervisor can learn that Dom_3 and Dom_4 can be used in isolation but not at the same time. Therefore, the hypervisor can choose to switch from one domain to another from time to time so neither becomes unserved. As shown in Table 11, the maximum contention time experienced by CM is at the order of pico seconds (E-12), reason why in Table 13, the task budget of CM shows no observable change. In the case of the maximum contention time observed, which is at the memory interconnect while using $Full_{max}$ topology, the overhead is still in the order of nanoseconds ($\delta_{\Xi}^{v,\omega_{\mathcal{S}},\gamma} = 2.45E-9$). In that case, the overhead from the virtual I/O device point-of-view ($\frac{\delta_{\Xi}^{v,\omega_{\mathcal{S}},\gamma}}{\Lambda^{v,\gamma}}$) will be in the order of micro seconds ($2.45E-6$), and The overhead experienced by the task ($\frac{R_i^{v,\omega_{\mathcal{S}},\gamma}}{\Lambda^{v,\gamma}}\delta_{\Xi}^{v,\omega_{\mathcal{S}},\gamma}$) will be in the order of nano-seconds ($E-3 * E-6$). In despite of the contention time experienced on each period of a task

1. Dom_1 and Dom_2 start to run
2. New (LO) domains: Dom_3 and Dom_4 start to run
   Monitoring-decide-act cycle:
3. Hypervisor detects Dom_1 will become unschedulable at $\omega_{\mathcal{S}} = LO$
4. Mode switch from $\omega_{\mathcal{S}} = LO$ to $\omega_{\mathcal{S}} = HI$
5. New LO-domains (Dom_3 and Dom_4) virtual power OFF
6. Hypervisor detects Dom_1 will be schedulable at $\omega_{\mathcal{S}} = LO$ again
7. Mode switch from $\omega_{\mathcal{S}} = HI$ to $\omega_{\mathcal{S}} = LO$
8. Dom3 virtual power ON
9. Hypervisor detects HI-domains are still schedulable at $\omega_{\mathcal{S}} = LO$
10. Dom4 virtual power ON
11. Hypervisor detects HI-domains will become unschedulable at $\omega_{\mathcal{S}} = LO$
12. Mode switch from $\omega_{\mathcal{S}} = LO$ to $\omega_{\mathcal{S}} = HI$
13. Dom4 virtual power OFF
14. (...)

Figure 18: Runtime scenario, Known-LO-domains case.

being small compared to the task period, tasks will still be affected especially in the case an operation needs to be break down on several periods.

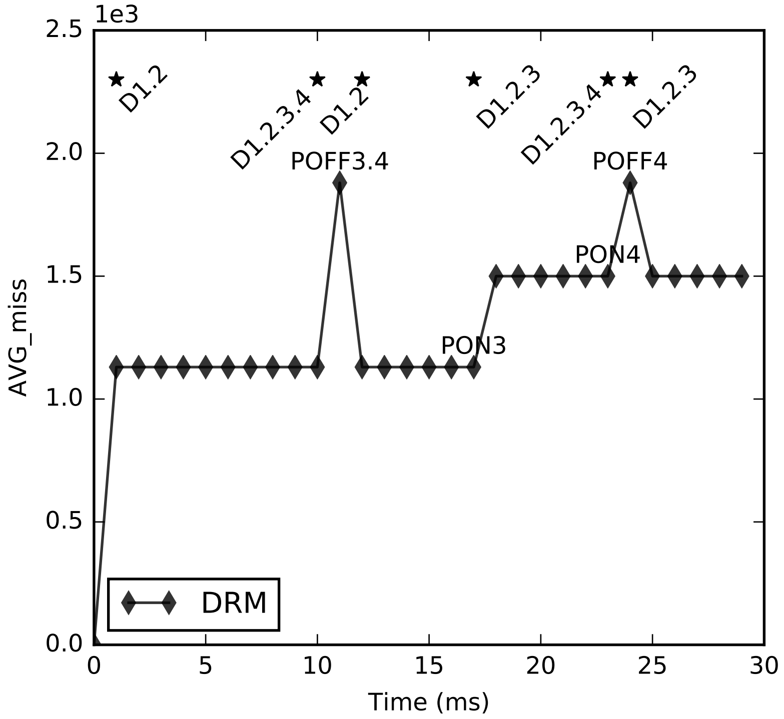


Figure 19: AVG_{miss} over time (known LO-domains case).

The second evaluation aims to compare the PCPU resources needed by DRM approach proposed in this thesis with the MPR approach proposed by (EASWARAN; SHIN; LEE, 2009). The first step is to execute the transformation of DRMs (see Definition 27) / MPRs (see (EASWARAN; SHIN; LEE, 2009)) into task sets. After transformation show in Definition 27, there are four tasks representing VCPUs, all of them have period of 1ms; the first three have execution time also of 1ms; the fourth one has execution time of 0.7ms. Such tasks are common between the DRM and MPR approaches. The DRM approach, in addition has two more additional tasks, both for representing the IOVCPU of the CM device, one for reading and another for writing.

Figure 20 shows the PCPU demand required by DRM and MPR approaches. The PCPU demand for MPR is 3.7, while for DRM is 5.7. The MPR approach will only require PCPU resources for implementing execution VCPUs, while the DRM approach will require PCPUs also for implementing IOVCPUs. In the presented configuration it was assumed that the domain would expend on the IOVCPU all the time it would spend on the virtual I/O device. However, in practice the time spent on IOVCPUs is only the time needed to program (or retrieve data from) the physical I/O device correspondent. After that, the PCPU allocated for the IOVCPU could be used by others VCPUs. The only thing that the proposed approach wishes to ensure is that the required physical I/O device budget is reserved for the correspondent virtual I/O device during all the device operation. I.e. if another domain tries to use the same physical I/O device it will needed to wait for the operation to finish and that is controlled at the IOVCPU level.

The third evaluation regards schedulability decisions over time taking into account the runtime scenario described on Listing 18. The graph shown in Figure 21 is similar to Figure 19 (first evaluation). However, while Figure 19 focus on LLC miss overhead, Figure 21 focus on the predicted fraction of schedulable domains on each step of the runtime scenario. The fraction of schedulable domains is the number of schedulable domains over the total number of domains and a domain is schedulable if all its guest OS tasks are schedulable. As in Figure 19 power OFF and power ON indicate schedulability decisions (criticality mode switch and power ON/OFF of processing elements). In the DRM approach, whenever the hypervisor detects that HI-domains schedulability fraction will become bellow 1 (one), there is a power OFF event. The figure compares the performance of the MPR approach. As the MPR approach if unaware of LLC miss overhead and the interconnect contention overhead it will never issue power OFF events what causes

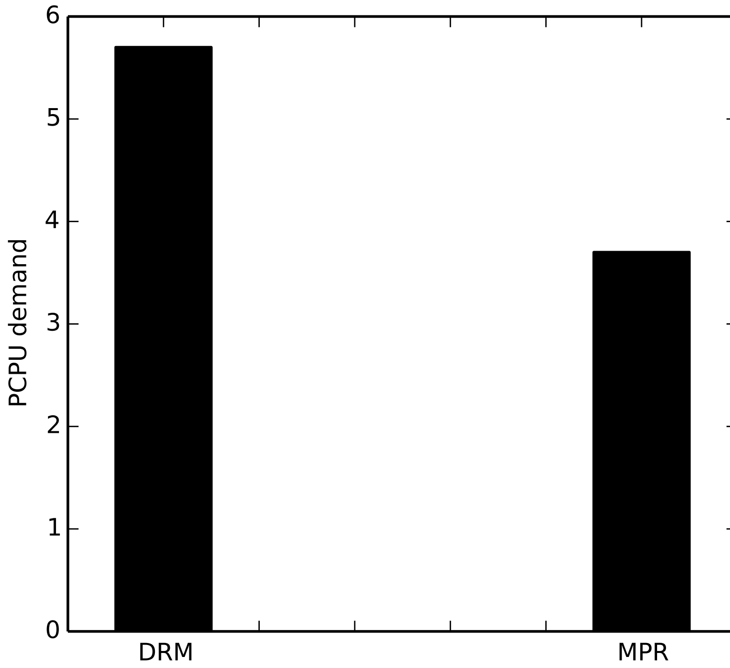


Figure 20: PCPU demand. Known-LO-domains case.

HI-domains to lose deadlines.

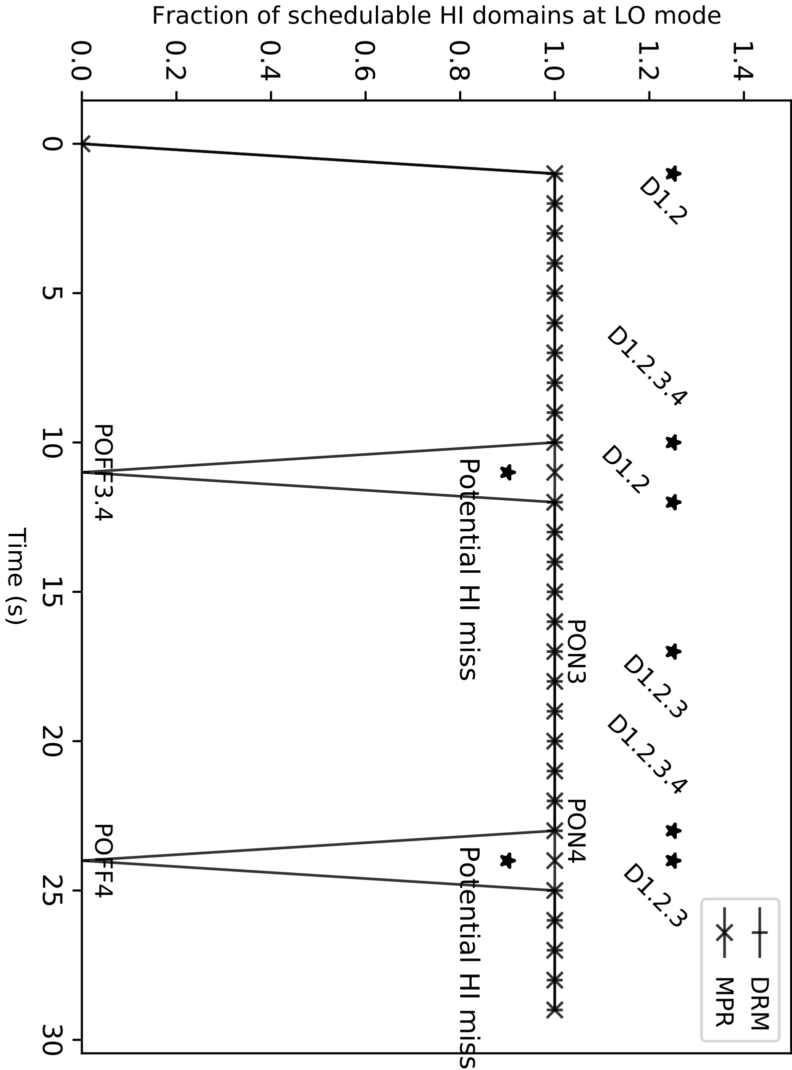


Figure 21: Fraction of schedulable domains. (known *LO*-domains case).

6.3 DISCUSSION

While comparing to MPR proposed by (EASWARAN; SHIN; LEE, 2009), the proposed DRM approach requires more PCPU resources since it employs PCPUs for implementing not only VCPUs but also for implementing IOVCPUs. On the other hand, the DRM approach presents a better schedulability decision while comparing to MPR since DRM takes into account LLC and interconnect overheads. As consequence, the proposed DRM approach will prevent LO-tasks from executing when the perceived interference is high. The MPR approach, on the other hand, can be over optimistic admitting the LO-tasks jobs that will later, due to the unperceived high interference, cause HI-tasks to miss their deadlines.

Similar conclusions can be inferred if comparing the proposed DRM approach to the DMPR approach proposed by (XU et al., 2013). Again, DRM requires more PCPU resources than DMPR since the former uses PCPUs for implementing IOVCPUs, while the latter does not handle interconnect overhead and does not require IOVCPUs for handling virtual I/O devices. While the proposed approach takes into account the LLC, (XU et al., 2013)'s approach only take into account private caches. In cases where the Work Set Size (WSS) of every task fits inside the private caches of PCPUs and the interconnect contention overhead is negligible, the schedulability decisions obtained with DRM and the DMPR approaches can reach equivalent results. However, in scenarios where the WSS of tasks are bigger than the private caches of PCPUs and thus it is required access to the LLC and thus can be LLC misses, or in scenarios where the interconnect contention overhead is significant, the proposed approach will present better schedulability decisions than (XU et al., 2013)'s approach since the former takes into account such overheads.

In addition, the proposed approach is distinct from (EASWARAN; SHIN; LEE, 2009) and from (XU et al., 2013) in its goal. The main focus of (EASWARAN; SHIN; LEE, 2009) and (XU et al., 2013) is at design time, and their goal is to decide whether domains can be scheduled by their models and whether the underlining physical platform is able to attend the models requirements. The guest OS tasks requirement are fixed and, in the case of (XU et al., 2013), the upper bound limits for private cache overheads are also fixed. The proposed approach, on the other hand, presents a more dynamic nature, presenting a design time and runtime phases as introduced in Section 5.9. While the guest OS tasks base requirements are also fixed, the overhead values are dynamically

sampled during the monitoring-decide-act at runtime, differently from (EASWARAN; SHIN; LEE, 2009) and from (XU et al., 2013). Such sampled values, feed the model back and are used for overhead-aware scheduling decisions as presented in Chapter 5.

7 CONCLUSION

The shift from dedicated hardware to software running on multiprocessor systems for implementing real-time embedded system features has been motivated by the growing in complexity of the features and the advances of processor technology. Such shift brings new challenges, mainly the separation between system components, once performed in hardware, which now needs to be ensured by software. In such scenario, the use of hypervisors has presented itself as a suitable solution especially regarding spatial isolation. Temporal, isolation, however, still lefts room for improvements. The main challenges are to quantify the interference caused on real-time tasks in a physical platform in which physical resources are shared among domains.

This work targets this problem by presenting a modeling approach which takes into account shared resources (CPUs, I/O devices, and interconnects) and that quantifies the impact of temporal interference on the guest OS tasks. The approach takes into account a compositing scenario, where guest OS tasks use VCPUs, and virtual I/O devices, while their domains are mapped on PCPUs and physical I/O devices. Furthermore, the modeling approach takes into account two distinct criticality levels, LO and HI. When the system is operating in LO mode, both HI-tasks and LO-tasks execute. However, when the system is operating in HI mode, all LO-tasks are suspended, and only HI-tasks can run. Moreover, this work proposes to power OFF the processing elements used by LO-tasks, preventing them from keeping interfering on HI-tasks.

In the proposed modeling approach task sets of the same criticality define a domain. Each task is defined by its period, deadline, computation (time on VCPUs), and I/O (time or rate of virtual I/O devices) requirements. The interconnect topology is expressed as a graph, and describes the connections among processing elements, interconnect, and memory elements. Additionally, it describes the insertion rate of each processing element, the servicing rate of memory elements, and the insertion and servicing rate of interconnect elements. The overhead associated with replacement of evicted cache lines, and the contention time on each interconnect elements is also an input for the model. The cache-related overhead can be obtained by measurements performed on the physical platform, while the contention time on interconnect elements is computed based on queuing theory using their insertion rates and servicing rates. Such rates, in turn, are obtained form nominal

values of the physical platform components or by measurement.

For each domain, is computed a DRM that abstracts the processing and I/O requirements of the domain tasks. A DRM is defined by a criticality, which is the same of the domain it is designed for, VCPU period, VCPU budgets (a budget for criticality level), VCPU quantities (for each criticality level), virtual I/O device periods (for each device class and operation), virtual I/O device budgets (for each device class, criticality level, and operation), and virtual I/O device quantities (for the same parameters as virtual I/O device budgets). The budgets of a DRM represent the capacity that all its virtual processing elements will collectively provide, its periods represent the replenishment interval of those budgets. Domain schedulability is then defined by whether the supply a DRM provides is able to fulfill the domain task set demand. To perform overhead-aware schedulability, memory and I/O overheads are used to inflate the budget of the domain tasks and it is checked whether the original DRM can still schedule them.

The overall approach is composed of a design and a runtime phase. In the design phase the known domains (HI and LO) run, overhead is measured, and the budgets of the tasks are inflated. During runtime, a LO to HI mode switch will occur whenever (i) a HI-task is unable to complete its job while using its LO budget or (ii) whenever the hypervisor predicts that the measured overhead will cause the LO budgets of HI tasks to inflate in such a way its DRM will not be able to scheduled it. As the algorithm that computes the DRMs interfaces is designed to compute the minimal required budgets that will attend the domain, in the design phase of the system development the task set of domains is inflated with the overhead measured on that scenario plus some slack. Otherwise during runtime, the system would not tolerate a minimal interference caused new domains. A HI to LO mode switch will occur whenever there are no pendent jobs of HI-tasks in HI mode and the hypervisor predicts the DRMs of all HI-domains are able to schedule their tasks taking into account the current interference.

The proposed modeling approach was evaluated while using synthetic domain requirements and while using a physical topology common in several platforms. For evaluation, it was taking into account a specific runtime scenario, so the approach could be evaluated while performing its design and runtime phases. The approach, which is based on DRM was compared to the MPR model introduced by (EASWARAN; SHIN; LEE, 2009). The DRM-based approach presents a higher PCPU demand than MPR since it uses PCPUs for implementing not only execution VCPUs but also for implementing IOVCPUs that are used by virtual

I/O devices. On the other hand, the proposed approach presents better domain schedulability than MPR since it takes into account LLC miss and interconnect contention overhead. As consequence, the proposed approach chooses to power OFF LO-domains avoiding HI-domains of losing their deadlines. Additionally, the proposed domain schedulability test was evaluated for distinct domains and whole-domain utilization values, and it achieves similar performance to the (GOOSSENS; FUNK; BARUAH, 2003) gEDF schedulability test.

From the obtained results, it is possible to conclude that the main goal of this work is reached, since while using the proposed approach, HI-tasks are ensured to always meet their deadlines as the scheduling tests demonstrate. It is also possible to affirm that the research questions brought up in the Introduction chapter are answered. Revisiting those questions and pointing out where and how they are answered:

- (i) *How the sharing of resources interferes with the temporal behavior of tasks in a multicore hypervisor?*.

  This question is answered in Chapter 4, Sections 4.2, and 4.4, regarding interconnect and LLC respectively. The types of interference are illustrated by Figure 6 (interference due to interconnect contention), Figure 13 (interference due to cache line eviction caused by another PCPU.), and Figure 14 (Interference due to cache line eviction caused by an I/O device).

- (ii) *How to quantify the extent of such interference?*

  Definition 32, which defines LLC miss overhead, and Definition 33, which defines how the VCPU budget of guest OS tasks are affected by such overhead answer this question regarding the LLC. Similarly, Definition 35, and Definition 36 present the equivalent for interconnect contention overhead. All these definitions are presented in Chapter 5, Section 5.8, which presents the overhead-aware compositional schedulability analysis.

- (iii) *What are the techniques that can be used to reduce such interference and keep it bounded?*

  Regarding LLC miss overhead, Section 4.4 answers this question presenting the domain-oriented, VCPU-oriented, and guest OS task-oriented approaches for implementing page-coloring in a virtualized platform. Regarding interconnect contention overhead, Section 4.2 answer this question discussing power OFF techniques, while Section 4.3, which presents the *Semaphore Observer*

design pattern, discuss how to keep the time for interrupt handling bounded.

- (iv) *How such interference can feed a model of the system in order to check tasks schedulability and domain feasibility?*

  The schedulability tests presented in Section 5.5 (Theorem 1 regarding domain schedulability and Theorem 2 regarding system schedulability) and the discussion of how they are used for overhead-aware tests in Section 5.8.3 answers this question.

- (v) *How such interference-aware schedulability tests can be used in a (design and runtime) strategy that ensures that temporally critical guest OS tasks never miss a deadline?*

  The strategies presented in Section 5.9 answer this question for the case when *LO*-domains requirements are known (Section 5.9.1) and for the case when they are not known (Section 5.9.2).

As far was possible to identify, this work is the first work that proposes an approach for guest OS task schedulability evaluation and computation of domain requirements that takes into account memory and I/O resource sharing. It is expected that the proposed approach can help system designers to evaluate whether a given platform is able to support real-time requirements and how guest OS tasks can be distributed among distinct domains.

The proposed modeling approach assumes cache-related overhead and contention time will affect all tasks in a domain equally, thus it can be overly pessimistic. Such pessimism is mitigated by using average cache-related overhead values and contention time while computing the *LO* criticality mode budget of each task but does not eliminate the fact that every task is affected equally. To overcome this limitation and regarding memory-related overhead specifically, (XU et al., 2013) proposes ways estimate an upper bound for VCPUs context switches and other events that would cause cache misses. Their approach, however, besides adding complexity to the model assumes only private caches and not a shared LLC. Regarding interconnect contention overhead the model presented in this work assumes a single memory element, and thus employs a single server queue. While that fits most real physical platforms, which have a single main memory that could be a limitation while employing other topologies. In this work processing elements are seem as source elements that make request to interconnect elements. An alternative approach would see each processing element as a complete queue system, composed of source, interconnect, and server. In

that way would be possible to explore contention time inside each processing element, which in turn could be used to investigate contention time focused on distinct servers and not only centered in the main memory. Such investigations and models extensions are left as potential future works.

The proposed approach was evaluated only analytically, using scenarios that mimic practical executions. On one hand the efficacy and efficiency of the proposed techniques was already demonstrated in non virtualized scenarios, e.g. page-coloring (GRACIOLI; FRöHLICH, 2013), (GRACIOLI, 2014), and powering ON and OFF devices (HOELLER, 2007) and (REIS, 2016), and is expected to be preserved on virtualized scenarios. On the other hand, as far was possible to identify, no single hypervisor implements all the required techniques proposed by this thesis, composite MC real-time scheduling, powering ON and OFF physical and virtual devices, time-deterministic multilevel interrupt handling. The EPOS OS is the one that has most of these required features, however, in its current state does not support virtualization fully. An extension of EPOS, named *HyperEPOS* has envisioned in the context of this work. A preliminary version of its design is presented in Appendix A, and an adaptation of LINUX to run atop *HyperEPOS* is described by (MEURER; LUDWICH; FRöHLICH, 2016). The mechanisms for supporting time-deterministic interrupt handling are presented and evaluated in a single-level interrupt handling by (LUDWICH; FRöHLICH, 2015) and by (LUDWICH et al., 2014). Early stages of this research and still in the context of this PhD, have evaluated the scheduler of EPOS according functional correctness (LUDWICH; FRöHLICH, 2013) and (LUDWICH; FRöHLICH, 2012). The same scheduler, then used for scheduling OS threads would be employed for scheduling VCPUs in *HyperEPOS*.

REFERENCES

AL-ZOUBI, H.; MILENKOVIC, A.; MILENKOVIC, M. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: *Proceedings of the 42Nd Annual Southeast Regional Conference*. New York, NY, USA: ACM, 2004. (ACM-SE 42), p. 267–272. ISBN 1-58113-870-9. Available from Internet: <http://doi.acm.org/10.1145/986537.986601>.

ANDREWS, G. R. *Foundations of Parallel and Distributed Programming*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201357526.

AXER, P. et al. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 13, n. 4, p. 82:1–82:37, mar. 2014. ISSN 1539-9087. Available from Internet: <http://doi.acm.org/10.1145/2560033>.

BALAKRISHNAN, N.; NEVZOROV, V. B. *A Primer on Statistical Distributions*. [S.l.]: John Wiley & Sons, 2003. ISBN 978-0-471-42798-8.

BARUAH, S.; BERTOGNA, M.; BUTTAZZO, G. *Multiprocessor Scheduling for Real-Time Systems*. 1st. ed. [S.l.]: Springer International Publishing, 2015. 228 p. (Embedded Systems). ISBN 978-3-319-08696-5.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In: *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2010. (RTSS '10), p. 14–24. ISBN 978-0-7695-4298-0. Available from Internet: <http://dx.doi.org/10.1109/RTSS.2010.23>.

BECKERT, M. et al. Sufficient temporal independence and improved interrupt latencies in a real-time hypervisor. In: *Proceedings of the 51st Annual Design Automation Conference*. New York, NY, USA: ACM, 2014. (DAC '14), p. 86:1–86:6. ISBN 978-1-4503-2730-5. Available from Internet: <http://doi.acm.org/10.1145/2593069.2593222>.

BERTOGNA, M.; CIRINEI, M.; LIPARI, G. Improved schedulability analysis of edf on multiprocessor platforms. In: *17th Euromicro*

Conference on Real-Time Systems (ECRTS'05). [S.l.: s.n.], 2005. p. 209–218. ISSN 1068-3070.

CHENG, Y. et al. Precise contention-aware performance prediction on virtualized multicore system. *Journal of Systems Architecture - Embedded Systems Design*, v. 72, p. 42–50, 2016. Available from Internet: <http://dx.doi.org/10.1016/j.sysarc.2016.06.006>.

CHISNALL, D. *The Definitive Guide to the Xen Hypervisor*. First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780132349710.

CHO, H.; RAVINDRAN, B.; JENSEN, E. An optimal real-time scheduling algorithm for multiprocessors. In: *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*. [S.l.: s.n.], 2006. p. 101–110. ISSN 1052-8725.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, n. 4, p. 35:1–35:44, out. 2011. ISSN 0360-0300. Available from Internet: <http://doi.acm.org/10.1145/1978802.1978814>.

EASWARAN, A.; SHIN, I.; LEE, I. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, v. 43, n. 1, p. 25–59, 2009. ISSN 1573-1383. Available from Internet: <http://dx.doi.org/10.1007/s11241-009-9073-x>.

FOYO, L. Leyva-del; MEJIA-ALVAREZ, P.; NIZ, D. de. Predictable interrupt scheduling with low overhead for real-time kernels. In: *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*. [S.l.: s.n.], 2006. p. 385–394. ISSN 1533-2306.

FRöHLICH; LISHA. *Open EPOS 1.0*. 2010. Available from Internet: <http://epos.lisha.ufsc.br/EPOS+Software>.

FRöHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. ISBN 3-88457-400-0.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GOOSSENS, J.; FUNK, S.; BARUAH, S. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 25, n. 2-3, p. 187–205, set. 2003. ISSN 0922-6443. Available from Internet: <https://doi.org/10.1023/A:1025120124771>.

GOPALAKRISHNAN, S. et al. Hard real-time communication in bus-based networks. In: *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*. [S.l.: s.n.], 2004. p. 405–414. ISSN 1052-8725.

GRACIOLI, G. *Real-Time Operating System Support for Multicore Applications*. 359 p. Tese (Doutorado) — Federal University of Santa Catarina, Florianópolis, 2014. Ph.D. Thesis. Available from Internet: <http://www.lisha.ufsc.br/pub/Gracioli_PHD_2014.pdf>.

GRACIOLI, G.; FRöHLICH, A. A. An embedded operating system api for monitoring hardware events in multicore processors. In: *Workshop on Hardware-support for parallel program correctness - IEEE Micro 2011*. Porto Alegre, Brazil: [s.n.], 2011. ISBN 978-1-4503-1053-6. Available from Internet: <http://www.lisha.ufsc.br/pub/Gracioli_HPPC_2011.pdf>.

GRACIOLI, G.; FRöHLICH, A. A. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In: *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Taipei, Taiwan: [s.n.], 2013. Available from Internet: <http://www.lisha.ufsc.br/pub/Gracioli_RTCSA_2013.pdf>.

HEISER, G.; LESLIE, B. The okl4 microvisor: Convergence point of microkernels and hypervisors. In: *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*. New York, NY, USA: ACM, 2010. (APSys '10), p. 19–24. ISBN 978-1-4503-0195-4. Available from Internet: <http://doi.acm.org/10.1145/1851276.1851282>.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture*: A quantitative approach. 5th. ed. 225 Wyman Street, Waltham, MA 02451, USA: Elsevier / Morgan Kaufmann, 2012. ISBN 978-0-12-383872-8.

HOELLER, A. S. *Gerenciamento do Consumo de Energia Dirigido pela Aplicação em Sistemas Embarcados*. Dissertação (Mestrado)

— Federal University of Santa Catarina, Florianópolis, 2007. M.Sc. Thesis.

HOELLER, A. S. J.; WANNER, L. F.; FRöHLICH, A. A. A hierarchical approach for power management on mobile embedded systems. In: *5th IFIP Working Conference on Distributed and Parallel Embedded Systems*. Braga, Portugal: [s.n.], 2006. p. 265–274. ISBN 978-0-387-39361-7.

HOFER, W.; LOHMANN, D.; SCHRODER-PREIKSCHAT, W. Sleepy sloth: Threads as interrupts as threads. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. [S.l.: s.n.], 2011. p. 67–77. ISSN 1052-8725.

INTEL. *Intel 6 Series Chipset and Intel C200 Series Chipset*: Datasheet. [S.l.]: Intel Corporation, 2011. 934 p. Document Number: 324645-006.

INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*: System programming guide, part 1. [S.l.]: Intel Corporation, 2014. 454 p. ISBN 3-88457-400-0.

INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. [S.l.]: Intel Corporation, 2016. 644 p. Order Number: 248966-032.

INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*: System programming guide, part 2. [S.l.]: Intel Corporation, 2016. 576 p. Order Number 253669-058US.

KIM, H.; RAJKUMAR, R. R. Real-time cache management for multi-core virtualization. In: *Proceedings of the 13th International Conference on Embedded Software*. New York, NY, USA: ACM, 2016. (EMSOFT '16), p. 15:1–15:10. ISBN 978-1-4503-4485-2. Available from Internet: <http://doi.acm.org/10.1145/2968478.2968480>.

KIM, S.; IM, C.; HA, S. Schedule-aware performance estimation of communication architecture for efficient design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 13, n. 5, p. 539–552, May 2005. ISSN 1063-8210.

KLEIMAN, S.; EYKHOLT, J. Interrupts as threads. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 29, n. 2, p. 21–26, abr. 1995. ISSN 0163-5980. Available from Internet: <http://doi.acm.org/10.1145/202213.202217>.

LEE, J. et al. Towards compositional mixed-criticality real-time scheduling in open systems: Invited paper. *SIGBED Rev.*, ACM, New York, NY, USA, v. 13, n. 3, p. 49–51, ago. 2016. ISSN 1551-3688. Available from Internet: <http://doi.acm.org/10.1145/2983185.2983193>.

LIEDTKE, J.; HAERTIG, H.; HOHMUTH, M. Os-controlled cache predictability for real-time systems. In: *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*. Washington, DC, USA: IEEE Computer Society, 1997. (RTAS '97), p. 213–. ISBN 0-8186-8016-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=523983.828369>.

LIU, X.; TONG, W.; SHEN, C. Qos-aware i/o schedule for virtual machines in cloud computing environment. In: *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. [S.l.: s.n.], 2013. p. 276–282.

LUDWICH, M. K.; FRöHLICH, A. A. System-level verification of embedded operating systems components. In: *Brazilian Symposium on Computing System Engineering*. Natal, Brazil: IEEE, 2012. ISBN 978-1-4673-5747-0.

LUDWICH, M. K.; FRöHLICH, A. A. On the formal verification of component-based embedded operating systems. *Operating Systems Review*, v. 47, n. 1, p. 28–34, 2013. ISSN 0163-5980.

LUDWICH, M. K.; FRöHLICH, A. A. Proper handling of interrupts in cyber-physical systems. In: *26th IEEE International Symposium on Rapid System Prototyping*. Amsterdam, The Netherlands: [s.n.], 2015. p. 83–89. Available from Internet: <http://www.lisha.ufsc.br/pub/Ludwich_RSP_2015.pdf>.

LUDWICH, M. K. et al. Run-time support system for models of computation in cyber-physical systems. In: *First Workshop on Cyber-Physical System Architectures and Design Methodologies*. New Delhi, India: [s.n.], 2014. p. 6. Available from Internet: <http://www.lisha.ufsc.br/pub/Ludwich_CPSARCH_2014.pdf>.

MA, R. et al. Cache isolation for virtualization of mixed general-purpose and real-time systems. *Journal of Systems Architecture*, v. 59, n. 10, Part D, p. 1405 – 1413, 2013. ISSN 1383-7621. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1383762113001288>.

MALKA, M. et al. riommu: Efficient iommu for i/o devices that employ ring buffers. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2015. (ASPLOS '15), p. 355–368. ISBN 978-1-4503-2835-7. Available from Internet: <http://doi.acm.org/10.1145/2694344.2694355>.

MEURER, R.; LUDWICH, M. K.; FRöHLICH, A. A. Virtualizing mixed-criticallity operating systems. In: *Brazilian Symposium on Computing Systems Engineering*. João Pessoa, Brazil: [s.n.], 2016. Available from Internet: <http://www.lisha.ufsc.br/pub/Meurer_SBESC_2016.pdf>.

MISSIMER, E. et al. Mixed-criticality scheduling with i/o. In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2016. p. 120–130.

MOHAN, S. et al. Using multicore architectures in cyber-physical systems. In: *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*. Michigan, USA: [s.n.], 2011.

MOSBERGER, D.; JIN, T. Httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 26, n. 3, p. 31–37, dez. 1998. ISSN 0163-5999. Available from Internet: <http://doi.acm.org/10.1145/306225.306235>.

MOSBERGER, D. et al. *The httperf HTTP load generator*. 2017. Available from Internet: <https://github.com/httperf/httperf>. Cited 31 Mar. 2017.

MüNCH, D. et al. Mpiov: Scaling hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. San Jose, CA, USA: EDA Consortium, 2015. (DATE '15), p. 579–584. ISBN 978-3-9815370-4-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=2755753.2755883>.

NELSON, A. et al. Comik: A predictable and cycle-accurately composable real-time microkernel. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014.

(DATE ’14), p. 222:1–222:4. ISBN 978-3-9815370-2-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=2616606.2616878>.

NOORSHAMS, Q. et al. Modeling of i/o performance interference in virtualized environments with queueing petri nets. In: *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 2014. (MASCOTS ’14), p. 331–336. ISBN 978-1-4799-5610-4. Available from Internet: <http://dx.doi.org/10.1109/MASCOTS.2014.48>.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1-55860-604-1.

PELLIZZONI, R. et al. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In: *Real-Time Systems Symposium, 2008*. [S.l.: s.n.], 2008. p. 221–231. ISSN 1052-8725.

PHAN, L. T. X.; XU, M.; LEE, I. Cache-aware interfaces for compositional real-time systems: Invited paper. *SIGBED Rev.*, ACM, New York, NY, USA, v. 13, n. 3, p. 52–55, ago. 2016. ISSN 1551-3688. Available from Internet: <http://doi.acm.org/10.1145/2983185.2983194>.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, ACM, New York, NY, USA, v. 17, n. 7, p. 412–421, jul. 1974. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/361011.361073>.

PORTAL, O. V. *OSEK VDX Portal*. 2008. Available from Internet: <http://www.osek-vdx.org/>. Cited 20 jun. 2015.

REINEKE, J.; GRUND, D. Sensitivity of cache replacement policies. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 12, n. 1s, p. 42:1–42:18, mar. 2013. ISSN 1539-9087. Available from Internet: <http://doi.acm.org/10.1145/2435227.2435238>.

REIS, J. G. *A Framework For Predictable Hardware/Software Component Reconfiguration*. 119 p. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, 2016. M.Sc. Thesis.

REIS, J. G.; FRöHLICH, A. A.; JR., A. H. On the fpga dynamic partial reconfiguration interference on real-time systems. In: *Brazilian Symposium on Computing Systems Engineering*. Foz do

Iguaçu, Brazil: [s.n.], 2015. p. 110–115. Available from Internet: <http://www.lisha.ufsc.br/pub/JGReis_SBESC_2015.pdf>.

SCHEFFEL, R. M.; FRöHLICH, A. A. Análise do comportamento de processos através de um sistema especialista probabilístico. In: *Brazilian Symposium on Computing Systems Engineering.* João Pessoa, Brazil: [s.n.], 2016. Available from Internet: <http://www.lisha.ufsc.br/pub/Scheffel_SBESC_2016.pdf>.

SHIN, K. G.; RAMANATHAN, P. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, v. 82, n. 1, p. 6–24, Jan 1994. ISSN 0018-9219.

TU, C.-C. et al. A comprehensive implementation and evaluation of direct interrupt delivery. In: *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2015. (VEE '15), p. 1–15. ISBN 978-1-4503-3450-1. Available from Internet: <http://doi.acm.org/10.1145/2731186.2731189>.

VESTAL, S. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007).* [S.l.: s.n.], 2007. p. 239–243. ISSN 1052-8725.

WAIKATO, M. L. G. at the University of. *Weka - Data Mining with Open Source Machine Learning Software in Java.* 2017. Available from Internet: <http://www.cs.waikato.ac.nz/ml/weka/>. Cited 31 Mar. 2017.

WILHELM, R. et al. The worst-case execution-time problem;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, n. 3, p. 36:1–36:53, maio 2008. ISSN 1539-9087. Available from Internet: <http://doi.acm.org/10.1145/1347375.1347389>.

XI, S. et al. Real-time multi-core virtual machine scheduling in xen. In: *Proceedings of the 14th International Conference on Embedded Software*. New York, NY, USA: ACM, 2014. (EMSOFT '14), p. 27:1–27:10. ISBN 978-1-4503-3052-7. Available from Internet: <http://doi.acm.org/10.1145/2656045.2656066>.

XU, M. et al. Cache-aware compositional analysis of real-time multicore virtualization platforms. In: *2013 IEEE 34th Real-Time Systems Symposium.* [S.l.: s.n.], 2013. p. 1–10. ISSN 1052-8725.

YUN, H. et al. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. [S.l.: s.n.], 2014. p. 155–166. ISSN 1545-3421.

APPENDIX A – Embedded Multicore Hypervisor

Most of hypervisors do not deal with the challenges imposed by modern multicore processors (such as cache hierarchy and interrupt handling) that need to be faced in order to provide for temporal isolation in such architectures. Instead, they either employ a dedicated hardware approach (e.g. CoMik) (NELSON et al., 2014) or do not address hard real-time (e.g. OKL4 and uC/OS-MMU) (HEISER; LESLIE, 2010; BECKERT et al., 2014). RT Xen (XI et al., 2014) is the most similar work related to this as it aims the same hardware architecture. However, it focuses mostly on multicore scheduling issues not performing a further investigation on memory neither I/O.

This Section describes how this work intends on dealing with process and memory management, time virtualization, access control, interrupt handling, and I/O devices in multicore platforms. For such its presents the design and current implementation of the HyperEPOS real-time multicore hypervisor, which was named after the EPOS RTOS.

The current (ongoing) HyperEPOS implementation runs on the IA32 architecture. Thus, this Section uses IA32 as an example. However, as EPOS confine architecture-dependent part in hardware mediators HyperEPOS shall be easily ported to others architectures. It is assumed that an architecture able to support for HyperEPOS will have at least two modes of processor execution (user and supervisor).

A.1 PROCESS MANAGEMENT

This section describes this thesis proposal for virtualized versions of CPUs, named VCPUs, as well how such CPUs are group together in *domains*, how such VCPUs are scheduled by the hypervisor and how they are assigned to PCPU (e.g. the cores in a multicore processor). This section also describes the proposed strategy for IDC and explains how new VMs start running.

A.1.1 Virtual CPU

According to the virtualization requirements of Popek and Goldberg (POPEK; GOLDBERG, 1974), all non-privileged instructions shall not suffer interference by the VMM, executing on the hardware directly. In the proposed approach, a VCPU is implemented as a periodic task that serves the guest OS tasks. A VCPU is assigned to a specific PCPU (e.g. CPU core). All non-privileged instructions in a VM task will exe-

cute directly on the PCPU that the VCPU is currently assigned. In order to perform operations that involve the execution of privileged instructions (e.g. disable interrupts) a task in a VM must use a IDC mechanism to communicate with a privileged domain that is responsible for handling I/O operations. Any attempt to execute a privileged instruction, will generate a fault in the PCPU taking into account that the guest OS is running on user mode and the current VCPU can propagate such fault as a virtual interrupt so the guest OS can terminate its current task.

EPOS Threads and Tasks compose a process. While EPOS Threads are the units of execution, EPOS Tasks represent the placeholder of threads, thus being the passive part of a process. A VCPU is implemented in HyperEPOS by extending the EPOS *Periodic_Thread* class as shown in Figure 22. Therefore, a VCPU can be scheduled using an unmodified version of the EPOS scheduler. The scheduling algorithm that chooses the VCPU that is going to run is configurable and both fixed priority (e.g. RM) and dynamic priority (e.g. EDF) in partitioned and global versions are supported.

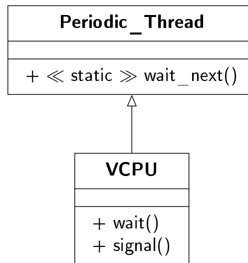


Figure 22: VCPU class.

The VCPU class contains the methods `wait` used for a VCPU that has sent a message and waits for its processing by another VCPU, and `signal` used to indicate that a message has been processed and the waiting VCPU can continue its execution.

A.1.2 Domain

A domain is defined as a VM that can contain one or more VCPUs. As depicted in Figure 23, a domain has a mailbox that stores messages addressed to it and any VCPU that belongs to the domain, whenever it

is scheduled to run, can get one of these messages and process it.

A domain is implemented as EPOS task and has a collection of VCPUs. Therefore, all VCPU in the same domain share the same address space. The guest OS sees a domain as a single machine that can have more than one CPUs.

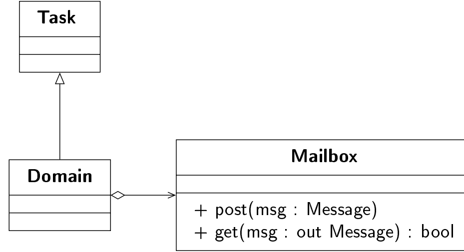


Figure 23: Domain class.

A.1.3 VCPU Scheduling and Migration

In the proposed approach, the VMM can schedule VCPU either by using global or partitioned real-time multicore algorithms such as partitioned EDF (pEDF), gEDF, partitioned RM (pRM), global RM (gRM), partitioned DM (pDM), and global DM (gDM). Besides that, a VCPU can migrate from one PCPU to another depending on the scheduling policy. The VMM scheduler is agnostic of the tasks running at the guest OS level. Therefore, the VMM will preserve the schedulability of guest OS tasks as long their schedule was feasible before the use of the VMM.

Given any particular PCPU at a specific time where the number of VCPUs that can run on it is denoted by $\boldsymbol{N_{vcpu}}$, given the period of such PCPU, denoted by $\boldsymbol{P_{pcpu}}$ and determined by the PCPU frequency of operation, the period each VCPU will provide for the guest OS tasks to run (denoted by $\boldsymbol{P_{vcpu}}$) is given by Equation A.1.

$$\boldsymbol{P_{vcpu}} = \frac{\boldsymbol{P_{pcpu}}}{\boldsymbol{N_{vcpu}}} - \boldsymbol{Overhead_h} \qquad \text{(A.1)}$$

where $\boldsymbol{Overhead_h}$ denotes the hypervisor overhead to schedule the VCPUs without taking into account VCPUs migration.

In the case global real-time multicore algorithms (e.g. gEDF) are

used, there is only one scheduling queue in the system and $\boldsymbol{N_{pcpu}}$ queue heads, where $\boldsymbol{N_{pcpu}}$ denotes the number of PCPUs the system has. In the case of partitioned algorithms, the system has $\boldsymbol{N_{pcpu}}$ scheduling queues and $\boldsymbol{N_{pcpu}}$ queue heads.

A.1.4 Inter Domain Communication

Domains communicate to each other by sending messages using an Remote Method Invocation (RMI)-like mechanism as detailed in Figure 24. Therefore, up to the invocation of a method a message is generated and sent to the hypervisor. If the message is to the kernel itself, it executes the method related to the message and returns the result if any. If the message is to a domain, the kernel posts the message in the mailbox of the target domain and puts the sender VCPU (source VCPU) to wait for the message handling.

As shown in Figure 25, every time a VCPU is scheduled to run, it checks for messages in its domain's mailbox. After getting a message from the domain's mailbox, the VCPU executes the method related to that message, writes the message result if any, and executes the *signal* hypercall so the source VCPU can continue its execution.

Messages have a fixed maximum size that determines the WCRT of a message passing.

A.1.5 Start of Day

The virtualization strategy, para-virtualization or full-virtualization, will define how a guest OS will start in a given VM. In the case of para-virtualization, the guest OS will start to run on a full operational VCPU (e.g. that is operating in protected mode in the case of the IA32 architecture). However, if full-virtualization is employed, the guest OS will execute a boot phase (e.g. in the case of IA32 architecture that means that guest OS will start to run at real-mode and will need to configure the protected mode, etc.).

In the current implementation of HyperEPOS, tasks running in a VCPU start at protected mode.

A.2 MEMORY MANAGEMENT

As for processors, where the guest OS must not execute privileged PCPU instructions, the guest OS must not have direct access to the registers and data structures related to page table management. In this section, we describe how such page table-related structures can be virtualized. In order to achieve real-time memory accesses must have a bounded maximum latency and should be preferably deterministic. This section also describes the adopted approach for reducing inter-domain last-level cache temporal interference, thus making memory access more deterministic and reducing access latencies.

Memory management in HyperEPOS is built using the MMU hardware mediator family of EPOS, responsible for allocating physical memory to the system and by *heap* abstractions, responsible for allocating logical memory to the domain tasks.

A.2.1 Page Table Management

The guest OS running on a VCPU muss not have direct access to data structures and registers related to page table management of the PCPU they are running on. In the case of para-virtualization the guest OS kernel is adapted to use virtual structures provided by the hypervisor. In the case of full-virtualization, some hardware virtualization support is usually employed, implementing techniques such as SPT: whenever the guest attempts to update a data structure or register related to page table management (e.g. CR3 register at IA32), or modify the page tables, the PCPU traps into the hypervisor and allows it to emulate the update.

The current implementation of HyperEPOS still does not virtualize data structures related to page table management. Therefore, HyperEPOS only admits at the VM level tasks that do not access such structures.

A.2.2 Page Coloring

Page coloring was chosen to be used in HyperEPOS since the cache partitioning it provides enables time predictability (LIEDTKE; HAERTIG; HOHMUTH, 1997; AXER et al., 2014). Partitioning suits well with domains since domains usually do not share memory and com-

municate through IDC. Additionally, page coloring does not depend on dedicated hardware and can be implemented in current multicore processors (AXER et al., 2014; GRACIOLI, 2014).

The strategy planned to be used in Hyper EPOS is to assign the same color to every VCPU running in the same domain. As consequence, tasks executing at the guest OS level will not cause temporal interference on tasks that belong to another domain (another color) and the domain will be able to keep the real-time requirements of its tasks.

The implementation of page coloring used in HyperEPOS is described in (GRACIOLI; FRöHLICH, 2013) and relies on the EPOS MMU hardware mediator family. While employing colors, the MMU provides for multiple lists for free-frames management, one list for each color. Similarly, there are multiple heaps, one for each color, to provide dynamic memory allocation for the domain tasks.

A.3 TIME

The hypervisor Application Programming Interface (API) shall provide to the guest OS tasks a way of dealing with real-time. Two key concepts here are Wall-clock Time (WCT) and DVT.

- Wall-clock Time, represents the real-time in the Unix time format: seconds and microseconds since $\mathbf{1^{st}}$ January 1970.
- Domain Virtual Time, represents the time as seen by the guest OS running in a domain. It is only incremented when a VCPU of such domain is scheduled to run.

Taking into account that a VCPU has only one part of the PCPU execution period, by following Equation A.1, a real-time periodic task running on the guest OS that has a period of $\boldsymbol{M}$ times units shall run in terms of VCPU execution with a period of $\boldsymbol{M} \times \frac{\mathbf{1}}{\boldsymbol{N_{cpu}}} - \boldsymbol{Overhead_h}$. In other words, for a DVT of $\boldsymbol{M}$ time units there will be $\boldsymbol{M} \times \frac{\mathbf{1}}{\boldsymbol{N_{cpu}}} - \boldsymbol{Overhead_h}$ time units of WCT.

In the API of HyperEPOS, a task running at the guest OS will be specified in terms of DVT and the hypervisor itself will handle the conversion to WCT, been transparent for whom has specified the task period of the guest OS.

In a Personal Computer (PC) platform, Wall-clock Time is measured by using a local real-time clock. For measuring DVT, a chronometer abstraction is used on each VCPU. Such chronometers are feed by

the Time Stamp Counter (TSC) of the PCPU where the VCPU is assigned to. If migration of VCPUs across distinct PCPUs is employed, the underlining micro architecture must keep TSCs of all PCPUs synchronized or else the time measured before the migration will not be valid after the migration.

A.4 ACCESS CONTROL

Every component in HyperEPOS is associated with a capability, and each domain manages the capabilities of the components it handles. Whenever a task running on a VCPU (source VCPU) wishes to invoke a method that belongs to a component that relies in a domain (target domain) it does so using the RMI mechanism. Inside the message used in the RMI, the source VCPU adds its capability and the access rights to the component it wishes to use. The VCPU that process such a message in the target domain then checks whether the capability provided by the source VCPU is valid and, only then, executes the method corresponding to the component in question.

By having fixed size messages and fixed size representations for capabilities values and access rights, the verification of the validity of a capability can be performed deterministically and with a bounded worst-case time.

A.5 INTERRUPTS AND EVENTS

One of the key ideas to achieve bounded interrupt handling, needed to fulfill real-time requirements, is to decoupling interrupt reception and acknowledgment from interrupt servicing. HyperEPOS achieves that by employing ISTs that are modeled and implemented using a design pattern, developed in the context of this work, named *Concurrent_Observer*. The *Concurrent Observer* is a concurrent variant of the *Publisher/Subscriber* design pattern (a.k.a. *Observed/Observer*) (GAMMA et al., 1995). It is depicted in Figure 26. The main difference between Concurrent Observer and the original Publisher/-Subscriber is that the `notify()` method of the Publisher does not invoke the `update()` method of its subscribers. Instead, it only notifies subscribers that a state change has occurred through a synchronization mechanism. Subscribers are modeled as threads (or VCPUs in the context of HyperEPOS) that wait for such state change notification on the

shared synchronization mechanism and then perform the updates.

The synchronization mechanism in the pattern is abstracted as a *Semaphore Handler*. The Handler is shared between the Concurrent Observer and the publisher (i.e. the Concurrent Observed), which holds a list of handlers as its subscribers. The `notify()` method of `Concurrent_Observed`, described in the sequence diagram of Figure 27, invokes the `call operator` for each handler. A *Semaphore Handler* is essential for interrupt handling. Since a semaphore has memory, interrupts notifications are never lost even if the frequency with which interrupts are generated is higher than the frequency with which interrupts are serviced. In such a case, however, in order to prevent data loss (in the case there is data generation associated with the interrupt occurrence) a larger buffer and/or a ring buffer strategy should be employed.

The `update()` method of `Concurrent_Observer` depicted in Figure 28, invokes the "dual call operator" of its handler. For the `Semaphore_Handler`, the operator is `v()` and its dual call operator is `p()`. From the point of view of the Concurrent Observer pattern, the call operator is the one used by the publisher to notify a change and the dual call operator is the one used by the subscriber to wait for a change.

It should be noted that the use of semaphores in Concurrent Observer does not introduce priority inversion problems since they are not being used to guard a critical section. They are used only to synchronize the ISR with the corresponding IST, much in a Producer/Consumer way.

A.6 I/O DEVICES

Following a ukernel approach, I/O operations are not handled by the kernel but for a privileged domain named "Domain 0" as in Xen (CHISNALL, 2007) terminology. Therefore, the other domains perform I/O operations sending messages addressed to "Domain 0".

The interrupts generated by devices are received by the hardware mediator that abstract that device and follow the approach of concurrent interrupt handling describe in Section A.5.

Every device is abstracted by a hardware mediator, and the Domain 0 handles I/O hardware mediators. A task running in a VM uses the IDC mechanism to use the hardware mediator of a device (e.g. writing on a UART).

Inside the Domain 0, the *Concurrent Observer* design pattern is

employed for receiving and processing interrupts associated with each device.

An alternative to IDC to be used in the case of intensive I/O operations is to "give" the control of the hardware device to the domain that will interact with it. In such a case, the domain shall have the proper capability, and the memory-mapped I/O registers can be exported by attaching the segment that contains it into the address space of the domain. An example of the use of this approach is in the case of the video device. Since the video frame buffer is mapped on memory no supervisor mode instructions are required to control the device.

A.7 DISCUSSION

Table 17 compare the proposed hypervisor HyperEPOS with other hypervisors found in literature.

HyperEPOS focus on supporting HRT tasks while also supporting for Best-effort (BE) tasks running in other domains. It employs a synchronous IDC strategy. HyperEPOS uses real-time scheduling and page coloring to provide temporal isolation between distinct domains (each domain is assigned to a set of distinct colors). It also employs concurrent interrupt handling that bounds the handling time of an interrupt in a domain. Inter domain interrupt handling is also kept bounded while using IDC with messages of a fixed maximum size.

Table 17: Comparison of hypervisors, including the proposed one (HyperEPOS).

Hypervisor	**RT**	**IDC**	**Mem.**	**Interrupt**	**Time Isolation**
RT Xen 2.0	HRT	Sync.	None	Dom0+Sync IDC	RT sched.
CoMik	HRT	NM	Memory Tiles	NM	Cycle-accurate context switch
OKL4	SRT	Async.	None	vIRQ+Async IDC	RR+priorities
uC/OS-MMU	FRT	NM	None	Interposing	TDM sched. + Int. Interposing
HyperEPOS	HRT	Sync.	Cache-coloring	Concurrent Int. + Sync IDC	RT sched., Page col. Conc. Int.

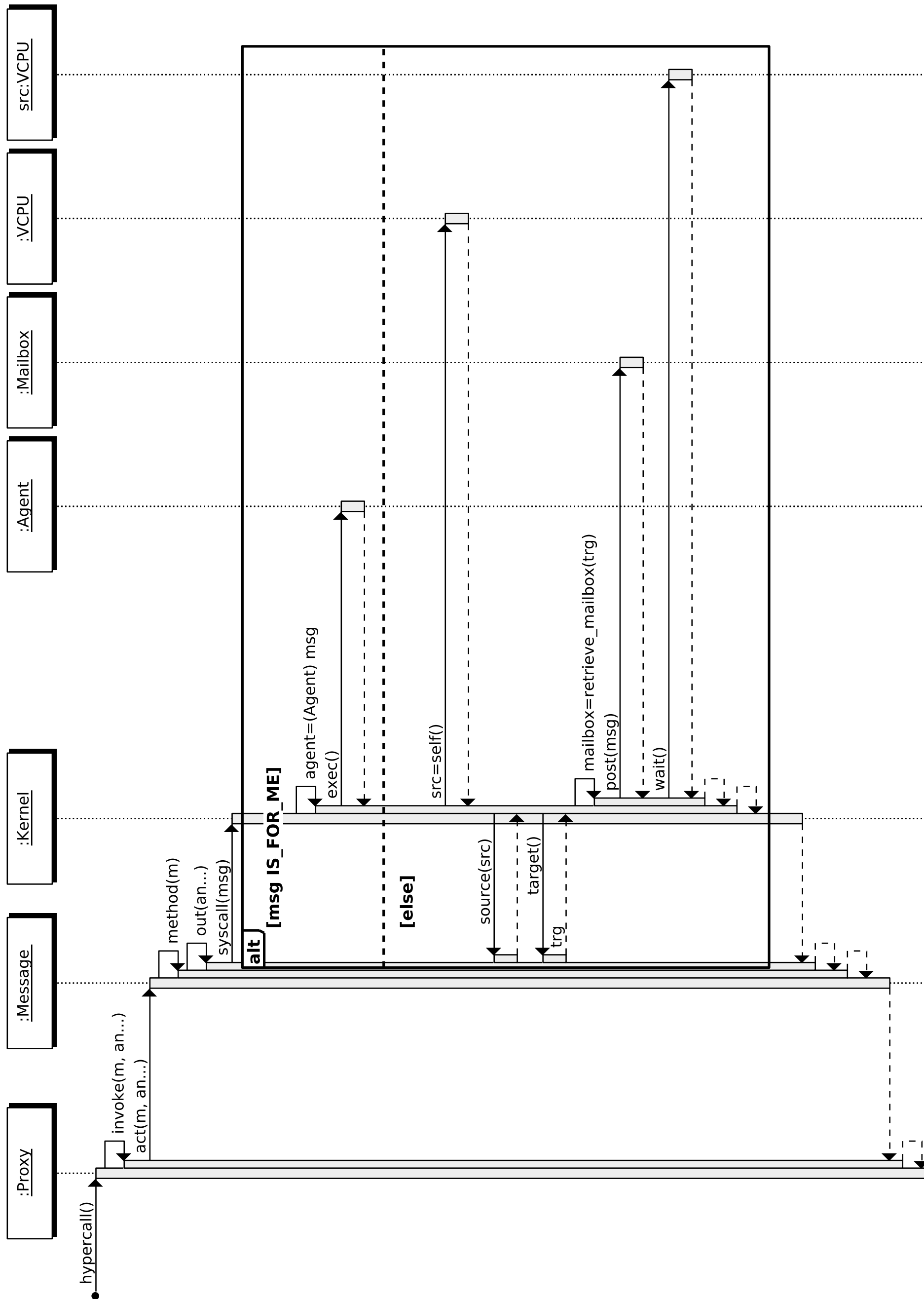


Figure 24: Hypercall invocation.

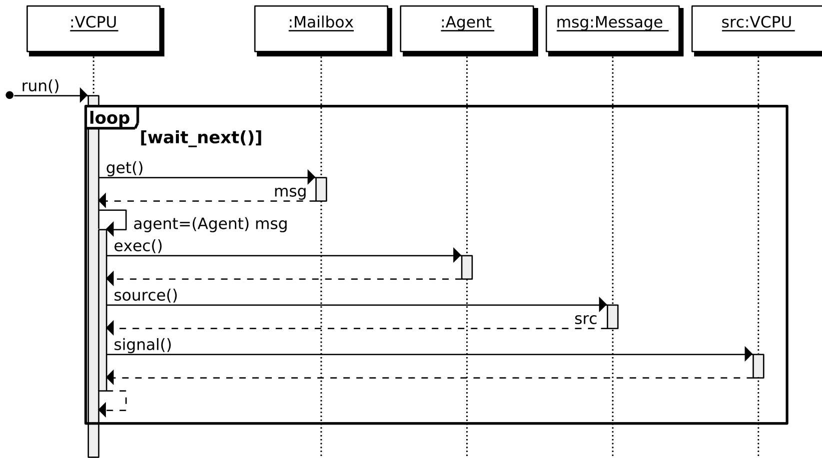


Figure 25: VCPU processing messages.

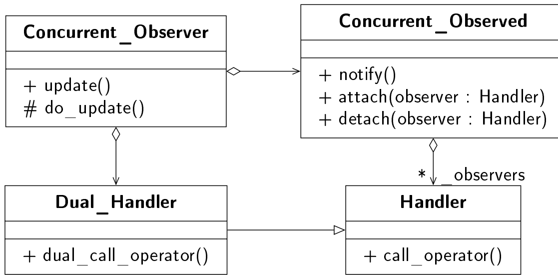


Figure 26: Concurrent Observer.

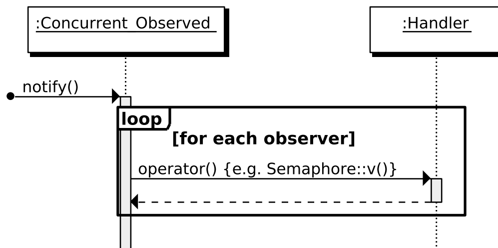


Figure 27: *Concurrent_Observed::notify*.

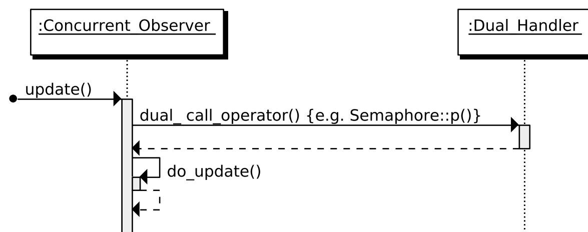


Figure 28: *Concurrent_Observer::update.*