View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade de Lisboa: Repositório.UL

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Real-Time Scheduling in Multicore Time- and Space-Partitioned Architectures

João Pedro Gonçalves Crespo Craveiro

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2013

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Real-Time Scheduling in Multicore Time- and Space-Partitioned Architectures

João Pedro Gonçalves Crespo Craveiro

Tese orientada pelo Prof. Doutor José Manuel de Sousa de Matos Rufino, especialmente elaborada para a obtenção do grau de DOUTOR em INFORMÁTICA, especialidade ENGENHARIA INFORMÁTICA

2013

This work was partially supported by

the European Space Agency Innovation (ESA) Triangle Initiative program
through ESTEC Contract 21217/07/NL/CB, Project AIR-II

.

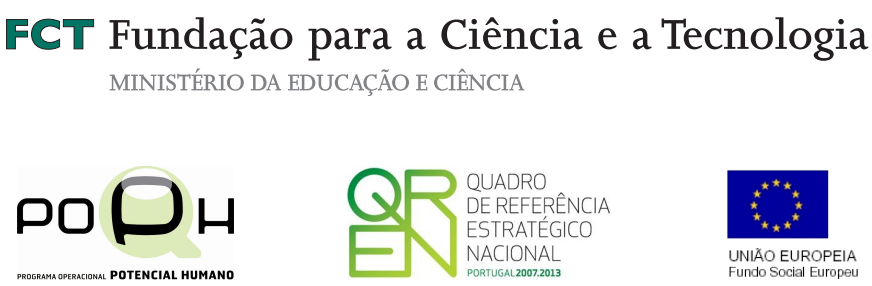
the European Commission Seventh Framework Programme (FP7)
through project KARYON (IST-FP7-STREP-288195)

.

Fundação para a Ciência e a Tecnologia (FCT), with Égide/CAMPUSFRANCE
through the PESSOA programme transnational cooperation project SAPIENT

.

Fundação para a Ciência e a Tecnologia (FCT)
through multiannual funding to the LaSIGE research unit, the CMU|Portugal programme,
the Individual Doctoral Grant to the author (SFRH/BD/60193/2009), and
project READAPT (PTDC/EEI-SCR/3200/2012)



Abstract

The evolution of computing systems to address size, weight and power consumption (SWaP) has led to the trend of integrating functions (otherwise provided by separate systems) as subsystems of a single system. To cope with the added complexity of developing and validating such a system, these functions are maintained and analyzed as components with clear boundaries and interfaces. In the case of real-time systems, the adopted component-based approach should maintain the timeliness properties of the function inside each individual component, regardless of the remaining components. One approach to this issue is time and space partitioning (TSP) — enforcing strict separation between components in the time and space domains. This allows heterogeneous components (different real-time requirements, criticality, developed by different teams and/or with different technologies) to safely coexist. The concepts of TSP have been adopted in the civil aviation, aerospace, and (to some extent) automotive industries. These industries are also embracing multiprocessor (or multicore) platforms, either with identical or non-identical processors, but are not taking full advantage thereof because of a lack of support in terms of verification and certification. Furthermore, due to the use of the TSP in those domains, compatibility between TSP and multiprocessor is highly desired. This is not the present case, as the reference TSP-related specifications in the aforementioned industries show limited support to multiprocessor. In this dissertation, we defend that the active exploitation of multiple (possibly non-identical) processor cores can augment the processing capacity of the time- and space-partitioned (TSP) systems, while maintaining a compromise with size, weight and power consumption (SWaP), and open room for supporting self-adaptive behavior. To allow applying our results to a more general class of systems, we analyze TSP systems as a special case of hierarchical scheduling and adopt a compositional analysis methodology.

Resumo

A evolução dos sistemas computacionais para endereçar questões de tamanho, peso e consumo energético conduziu à tendência de integrar funções (de outra forma fornecidas por sistemas separados) como subsistemas de um único sistema. Para lidar com a complexidade do desenvolvimento e validação de tal sistema, estas funções são mantidas e analisadas como componentes com fronteiras e interfaces bem definidos. No caso dos sistemas de tempo-real, a abordagem baseada em componentes adotada deve preservar as propriedades de pontualidade das funções dentro de cada componente, independentemente dos restantes. Uma abordagem para este problema é a compartimentação no espaço e no tempo (CET) — impor uma estrita separação entre os componentes nos domínios do tempo e do espaço. Tal permite que componentes heterogéneos (diferentes requisitos de tempo-real, criticidade, desenvolvidos por equipas diferentes e/ou com diferentes tecnologias) coexistam em segurança.

Os conceitos de CET têm sido adotados nas indústrias da aviação civil, aeroespacial, e (até certo ponto) automóvel. Estas indústrias também estão a adotar plataformas multiprocessador (ou multinúcleo), tanto com processadores idênticos e não-idênticos, mas não estão a tirar partido destas por falta de suporte em termos de verificação e certificação. Além disso, devido ao uso de CET nesses domínios, a compatibilidade entre CET e multiprocessador é altamente desejada. Este não é o estado atual, dado que as especificações relacionadas com CET usadas como referência nas indústrias referidas mostram suporte limitado para multiprocessador.

Nesta tese, defendemos que tirar partido de vários núcleos (possivelmente não-idênticos) de processador pode aumentar a capacidade de processamento dos sistemas CET (mantendo um compromisso com o tamanho, peso e consumo de energia) e abrir caminho para suportar comportamentos auto-adaptativos. Para permitir a aplicação dos nossos resultados a uma classe mais geral de sistemas, analisamos os sistemas CET como um caso particular de escalonamento hierárquico e adotamos uma metodologia de análise composicional.

Keywords

Compositional analysis
Hierarchical scheduling
Multiprocessor
Real-time systems
Time- and space-partitioned systems

Palavras Chave

Análise composicional
Escalonamento hierárquico
Multiprocessador
Sistemas tempo-real
Sistemas compartimentados no espaço e no tempo

Resumo Alargado

Um sistema computacional de *tempo-real* é aquele cujos resultados devem observar correção, não apenas lógica, mas também temporal. A relação entre a pontualidade (face a uma determinada meta temporal) com que o sistema garante o fornecimento de resultados e a sua utilidade permite a definição de diferentes classes de tempo-real — classicamente, tempo-real *estrito* (*hard*) e *lato* (*soft*). Um sistema de tempo-real estrito contém, pelo menos, uma tarefa de tempo-real estrito — uma tarefa cujo resultado tem de ser fornecido, sempre, dentro da sua meta temporal (caso contrário, o seu resultado não tem utilidade). Um sistema de tempo-real lato não contém qualquer tarefa de tempo-real estrito, mas contém pelo menos uma tarefa de tempo-real lato — uma tarefa que deve cumprir a sua meta temporal, mas que pode ocasionalmente falhá-la (em cujo caso a utilidade do seu resultado diminui com o tempo) (Kopetz, 1997; Verissimo & Rodrigues, 2001). A investigação sobre sistemas de tempo-real tem se focado no conjunto de algoritmos e técnicas de análise que permitam aos desenvolvedores de um sistema saberem, antes de colocarem este em produção, se é possível garantir o cumprimento dos seus requisitos de tempo-real (estrito ou lato).

Os sistemas computacionais têm vindo a evoluir para acomodar determinadas necessidades, incluindo preocupações com as suas dimensões, peso e consumo energético (e, consequentemente, custo). Esta evolução conduziu a uma tendência de integrar sistemas separados como subsistemas de um único sistema computacional, mais complexo. Esta integração pode incluir a coexistência de subsistemas com diferentes classes de requisitos de tempo-real (estrito e lato), ou subsistemas desenvolvidos por diferentes equipas e/ou com diferentes níveis de certificação. A complexidade acrescida do sistema aplica-se às atividades dos seus desenvolvimento, testes, validação e manutenção. A abordagem de desenhar estes sistemas em torno da noção de *componente*, permitindo assim uma *análise baseada em componentes*, traz vários benefícios, alguns dos quais específicos aos sistemas de tempo-real (Lipari *et al.*, 2005; Lorente *et al.*, 2006). No caso da coexistência de diferentes classes de tempo-real, as vantagens de manter as partes do sistema com requisitos de tempo-real estrito logicamente separadas das de tempo-real lato estão relacionadas com dois aspetos. Por um lado, a análise separada permite cumprir os requisitos de

tempo-real estrito dos respetivos componentes sem impor pessimismo desnecessário à análise dos componentes de tempo-real lato. Por outro lado, com considerações de desenho apropriadas, as falhas de pontualidade permitidas aos componentes de tempo-real lato não invalidam a pontualidade dos componentes de tempo-real estrito (Abeni & Buttazzo, 1998). Uma abordagem de desenho para este efeito é a *compartimentação no tempo e no espaço* (CET), na qual cada componente constitui uma unidade de separação lógica e contenção (*partição*).

Contexto. Um exemplo proeminente da aplicação da abordagem CET ao desenho de sistemas computacionais é a adoção das especificações ARINC 651 (AEEC, 1991) e ARINC 653 (AEEC, 1997) no domínio da aviação civil. A abordagem tradicional até então, denominada *aviónica federada*, faz uso de funções distribuídas alojadas em componentes dedicados. Tendo cada função os seus recursos computacionais dedicados (e, por vezes, fisicamente separados), que não podem ser reatribuídos em tempo de execução, o potencial para uma utilização ineficiente dos recursos surge como um contraponto à inerente independência de falhas (Audsley & Wellings, 1996; Sánchez-Puebla & Carretero, 2003). Por outro lado as arquiteturas de Aviónica Modular Integrada (AEEC, 1991) empregam um ambiente compartimentado de alta integridade que aloja múltiplas funções de aviónica com diferentes níveis de criticidade numa plataforma computacional partilhada. A identificação de requisitos semelhantes ao da indústria aeronáutica levou a indústria aeroespacial a exprimir interesse em aplicar os conceitos de compartimentação no tempo e no espaço da Aviónica Modular Integrada ao *software* a bordo de missões espaciais. Este interesse surgiu quer do lado de parceiros norte-americanos como a NASA (Black & Fletcher, 2006; Fletcher, 2009; Hodson & Ng, 2007; Rushby, 1999) quer de parceiros europeus como a Agência Especial Europeia (ESA), a agência espacial francesa (CNES), e as empresas Thales Alenia Space e EADS Astrium (Planche, 2008; Plancke & David, 2003; Windsor & Hjortnaes, 2009). Além destas, também a indústria automóvel está ativa na adopção de desenhos de sistema CET. A iniciativa AUTOSAR (AUTomotive Open System ARchitecture) para a especificação de uma arquitetura padrão de software para a indústria automóvel, na especificação dos requisitos de alto nível para um sistema operativo, inclui provisões que correspondem, até certo ponto, às noções de compartimentação no tempo e no espaço.

Motivação. O emprego de múltiplos núcleos de processador surgiu como resposta à necessidade de estagnação do aumento da velocidade de relógio dos processadores (dado o consequente aumento de dissipação de energia ter atingido os limites práticos dos sistemas de arrefecimento). Um processador que aloja múltiplos nícleos no mesmo chip é denominado um processador multinúcleo (ou *multicore*). Os núcleos de processador podem ter espaços de endereçamento privados ou partilhados, sendo o último caso o tipicamente empregue em processadores multinúcleo (Patterson & Hennessy, 2009).

Os processadores multinúcleo estão a ganhar terreno no domínio dos sistemas embebidos, nomeadamente sistemas de segurança crítica e com requisitos de tempo-real estrito — como os encontrados nas indústrias aeronáutica e aeroespacial. As últimas versões dos processadores SPARC LEON, amplamente usados pela Agência Espacial Europeia, suportam configurações multinúcleo (com núcleos idênticos ou não-idênticos) (Andersson *et al.*, 2010). Contudo, tais capacidades são rotineiramente desativadas e consequentemente não exploradas, dada a falta de suporte às mesmas em termos de verificação e certificação (Anderson *et al.*, 2009). Dado o uso e interesse prevalentes das indústrias aeronáutica, aeroespacial e automóvel no que toca aos conceitos de compartimentação no tempo e no espaço, a compatibilidade entre sistemas CET e plataformas com múltiplos núcleos de processador, quer idênticos quer não, é assim altamente desejada; esse não é, porém, o cenário atual.

Metodologia. Como foi referido, os sistemas CET empregam tipicamente um escalonamento hierárquico de dois níveis. Analisar os sistemas CET como um caso particular de escalonamento hierárquico permite reutilizar os resultados obtidos num conjunto mais geral de sistemas e aplicações. O escalonamento hierárquico é um tópico atual na comunidade de investigação em escalonamento de tempo-real, na tentativa de resolver problemas em cenários reais de aplicação de *software* embebido (Abeni & Buttazzo, 1998; Lackorzyński *et al.*, 2012; Mok & Feng, 2002; Santos *et al.*, 2011; Xi *et al.*, 2011). A possibilidade de desenvolvimento independente e de hierarquias com mais de dois níveis constituem motivação e vantagem para a aplicação de *análise composicional*. Composicionalidade é a propriedade de um sistema complexo que pode ser analisado avaliando algumas propriedades dos seus componentes (sem saber a estrutura ou hierarquia interna destes) e a forma como são

conjugados (Easwaran *et al.*, 2006; Hausmans *et al.*, 2012). Neste sentido, um componente compreende uma aplicação a executar (que pode ser constituída por tarefas e/ou subcomponentes), um escalonador, e a especificação da disponibilidade de recursos — nomeadamente, o(s) processador(es). Este componente pode ser abstraído utilizando uma *interface* que, por um lado, esconde as suas caraterísticas internas perante o restante sistema e, por outro lado, esconde perante o próprio componente as caraterísticas extrínsecas da disponibilidade de recursos. A análise composicional compreende três pontos principais (Shin & Lee, 2007):

1. *Análise de escalonabilidade ao nível local* Analisar a escalonabilidade da aplicação de um componente com o escalonador e a disponibilidade de recurso especificados.

2. *Abstração do componente* Derivar a interface do componente a partir das suas caraterísticas internas.

3. *Composição de interfaces* Derivar, a partir de um conjunto de interfaces que exprimem os requisitos individuais de disponibilidade de recurso de componentes, o requisito cumulativo de disponibildade de recurso para escalonar estes componentes de acordo com uma dada estratégia de escalonamento.

Tese e contribuições. Neste tese defendemos que tirar partido de múltiplos núcleos de processador (potencialmente não-idênticos) pode **(i)** aumentar a capacidade de processamento dos sistemas compartimentados no tempo e no espaço, mantendo um compromisso com as dimensões, peso e consumo energético do sistema, e **(ii)** abrir caminho ao suporte a comportamentos auto-adaptativos para lidar com mudanças imprevistas nas condições operacionais e ambientais. No âmbito desta tese, são apresentadas as contribuições que se enumeram e descrevem de seguida.

Arquitetura e modelo de sistema. Propomos uma arquitetura de referência melhorada para sistemas CET com suporte a multiprocessador. Esta arquitetura constitui uma aproximação mais flexível ao multiprocessador do que aquela preconizada no estado da prática (nomeadamente, ARINC 653 e AUTOSAR). A nossa proposta permite paralelismo entre partições, paralelismo dentro das partições, e comportamento auto-adaptativo.

Análise composicional em multiprocessadores uniformes. Propomos o primeiro modelo para interfaces de componentes que permite a definição de hierarquias composicionais em multiprocessadores uniformes (aqueles cujos processadores podem ser não-idênticos, mas apenas em termos de velocidade). Esta contribuição permite a análise formal de sistemas CET com paralelismo entre e/ou dentro das partições em multiprocessadores potencialmente não-identicos. A nossa contribuição abrange os três aspetos da análise composicional de estruturas de escalonamento hierárquico — análise de escalonabilidade ao nível local, abstração do componente, e composição de interfaces — aplicando e estendendo resultados anteriores de outros autores (Baruah & Goossens, 2008; Easwaran *et al.*, 2009b). Os diferentes aspetos desta contribuição são validados analiticamente e demonstrados experimentalmente com recurso a simulação.

Análise, geração e simulação de escalonamento. Apresentamos o desenho e concretização do *hsSim*, uma ferramenta orientada a objetos para simulação e análise de escalonamento e geração de tabelas estáticas de escalonamento. O hsSim foi cuidadosamente desenhado com atenção aos padrões de desenho de software aplicáveis, com objetivos de modularidade, extensibilidade e interoperabilidade. Esta abordagem cuidadosa não é habitualmente aplicada na concretização de utilitários de suporte a experiências académicas, sendo esta a principal razão pela qual desenhamos uma ferramenta de raíz em vez de modificarmos uma ferramenta existente. Este facto não preclude porém que algumas das contribuições apresentadas sejam importadas para o código de outras ferramentas, como o Cheddar — que apresenta já bastante maturidade no que respeita à análise e simulação de escalonamento não-hierárquico. Concretizamos o suporte a algoritmos de escalonamento tradicionais, e ainda incorporamos as nossas contribuições no âmbito da análise composicional (teste de escalonabilidade, abstração de componentes, e algoritmo de escalonamento de componentes).

Resultados preliminares sobre auto-adaptação em sistemas CET. Reportamos as experiências levadas a cabo, com um protótipo de sistema CET e através de simulação, para endereçar a segunda parte da tese defendida.

Conclusão e trabalho futuro Nesta tese, abordámos o problema do escalonamento tempo-real em sistemas compartimentados no tempo e no espaço assentes

sobre processadores multinúcleo. Acrescentámos ao estado da arte a primeira aproximação à análise composicional sobre multiprocessadores não-idênticos; os resultados formais que desenvolvemos analiticamente são corroborados pelos testes que efetuámos com conjuntos de tarefas gerados aleatoriamente, e são consentâneos com os resultados encontrados na literatura para plataformas multiprocessador dedicadas. Provámos analiticamente que o algoritmo de escalonamento gEDF não assegura a composicionalidade na presença de multiprocessadores não-idênticos; para este efeito, crucial quando se consideram as vantagens de desenvolvimento e verificação independentes trazidas por abordagens composicionais, propusémos um novo algoritmo de escalonamento, o umprEDF. Apresentámos também o desenho e desenvolvimento do hsSim, uma ferramenta orientada a objetos para análise, simulação e geração de tabelas de escalonamento; o hsSim será disponibilizado como ferramenta de código aberto para usufruto da comunidade de investigação em escalonamento tempo-real, e é uma prova de conceito para a inclusão de suporte a escalonamento hierárquico numa ferramenta mais madura. Usámos o hsSim para mostrar em acção os métodos formais que apresentámos.

Panorama de aplicabilidade. O modelo de sistema utilizado contém assunções relacionadas com o impacto temporal de fenómenos relacionados com o hardware (preempção e migração de tarefas, contenção no barramento, memória cache). Esta assunção é comum na investigação em escalonamento tempo-real, e não é totalmente díspar da realidade (Jalle *et al.*, 2013; Jean *et al.*, 2012); iremos porém, no futuro, olhar com mais detalhe para este impacto temporal.

Ao longo dos anos, houve vários projetos de investigação, financiados por entidades europeias, a empregar abordagens baseadas em componentes e/ou compartimentação no tempo e no espaço para atingir hibridização arquitetual ou lidar com sistemas de criticidade mista (exemplos: ACTORS, RECOMP, KARYON). Em alguns destes projetos, essencialmente contemporâneos com o trabalho desta tese, os processadores multinúcleo são abordados até certo ponto. O trabalho apresentado nesta tese é aplicável às arquiteturas consideradas em ou resultantes destes projetos, endereçando até aspetos deixados em aberto no final dos mesmos. Em particular, os nossos resultados transpõem a fasquia destes projetos no que respeita ao paralelismo entre componentes e ao uso de processadores não-idênticos. No domínio aeroespacial, os amplamente usados processadores SPARC LEON permitem configurações

com núcleos de processador não-idênticos, mas a falta de suporte de sistema operativo tem desencorajado o seu aproveitamento.

Além do inicialmente estabelecido foco em domínio de aplicação críticos, os nossos resultados também contribuem para a verificação formal de sistemas baseados emc omponentes assentes multiprocessadores não-idênticos como o Cell e o big.LITTLE. Embora estes não sejam tipicamente empregues em domínios críticos, aplicar uma abordagem baseada em componentes ao desenvolvimento de software complexo para executar sobre os mesmos permite reduzir a complexidade e custo deste processo, enquanto se assegura que os componentes individuais tem garantias mínimas de qualidade de serviço. Tal será particularmente verdade à medida que o suporte de sistema operativo ao escalonamento hierárquico aumenta (Abdullah *et al.*, 2013; Bini *et al.*, 2011b).

Trabalho futuro. As vias de trabalho futuro relacionado com a presente tese que identificamos desde já incluem **(i)** suporte de hardware e assunções do modelo de sistema; **(ii)** algoritmos de escalonamento; **(iii)** análise composicional com garantias temporais variadas, como sejam tempo-real lato e escalonamento de criticidade mista; e **(iv)** reconfiguração e tolerâncias a faltas proativa.

Acknowledgments

Although individually authored, this dissertation owes its completion to the support of quite some people. I now take the opportunity to thank them.

To my advisor José Rufino, for his full scientific, institutional and personal support. I particularly praise the attention to detail with which he scrutinized the papers we co-wrote (and the drafts of this dissertation), the concern he puts in providing adequate and fair conditions for his students to carry on their work, and how he strives to give his PhD students the proper training for independent research by assigning complementary responsibilities (such as reviewing papers, and actively participating in the elaboration of project funding proposals).

To FCUL Professors André Falcão, Isabel Nunes, Luís Correia, Mário Calha, Pedro M. Ferreira and Vasco T. Vasconcelos, and to IST Professor Leonel Sousa, for the insightful comments I received at the different stages of evaluation of my work.

To the SAPIENT team at Lab-STICC (Université de Bretagne Occidentale, Brest, France), especially Laurent Lemarchand, Vincent Gaudel, and Frank Singhoff. The collaboration and idea exchanges within the SAPIENT project had a great influence in the way some parts of this work were conducted.

To LaSIGE and the Department of Informatics, for enabling such a lively and stimulating environment for learning, researching and teaching. To the Navigators group in general, and more specifically to the TADS research line leaders Paulo Verissimo and António Casimiro and to the "Toutiçanos" with whom I most closely collaborated: Jeferson Souza, Joaquim Rosa, Kleomar Almeida, Pedro Nóbrega da Costa, Ricardo Pinto, Rui Pedro Caldeira and Rui Silveira.

For the less work-oriented companionship, to my regular LaSIGE office and lunch mates Luís Duarte, Nádia Fernandes, Tiago Gonçalves, and — finally and specially — José Baptista Coelho. More than a colleague, José has been true friend for all occasions, enabling me to behave more like a human being than I was used to.

To my mother Mafalda, my sister Joana and my "sister" Sophie, for having been always, since way back in time, a permanent source of comfort and support.

Last but not least, to Catarina, my love. For having endured my periods of absence, unavailability, frustration, impatience and mood swings, and for countering them with copious amounts of love, dedication, understanding and good moments, you definitely deserve the dedication of this dissertation.

João Pedro Gonçalves Crespo Craveiro

Lisbon, August 2013

Para a minha querida Catarina.
Para o Nós.

"Vogons!" snapped Ford, "we're under attack!"

Arthur gibbered.

"Well what are you doing? Let's get out of here!"

"Can't. Computer's jammed. [...] It says all its circuits are occupied." [...]

"Tell me ... did the computer say what was occupying it? I just ask out of interest ..."

[...] "I think a short while ago it was trying to work out how to ... make me some tea."

"That's right guys," the computer sang out suddenly, "just coping with that problem right now, and wow, it's a biggy. Be with you in a while." It lapsed back into a silence that was only matched for sheer intensity by the silence of the three people staring at Arthur Dent.

As if to relieve the tension, the Vogons chose that moment to start firing.

— Douglas Adams (1980). *The Restaurant at the End of the Universe*. Pan Books.

Contents

List of Figures

LIST OF FIGURES

List of Tables

List of Theorems

List of Acronyms

Acronym	Meaning
APEX	Application Executive
ARINC	Aeronautical Radio, Incorporated
AEEC	Airlines Electronic Engineering Committee
AUTOSAR	Automotive Open Systems Architecture
DBF	Demand Bound Function
EDF	Earliest Deadline First
EDZL	Earliest Deadline until Zero Laxity
ESA	European Space Agency
FCT	*Fundação para a Ciência e a Tecnologia*
gEDF	Global **EDF**
HM	Health Monitoring
HRT	Hard Real-Time
HSF	Hierarchical Scheduling Framework
IMA	Integrated Modular Avionics
LLF	Least Laxity First
MPR	Multiprocessor Periodic Resource (model, interface)
MTF	Major Time Frame
PAL	**POS** Adaptation Layer
PMK	Partition Management Kernel
POS	Partition Operating System
PST	Partition Scheduling Table
RM	Rate Monotonic
RTEMS	Real-Time Executive for Multiprocessor Systems

(continues on next page)

(continued from previous page)

Acronym	Meaning
SBF	Supply Bound Function
SRT	Soft Real-Time
TSP	Time and Space Partitioning
UML	Unified Modelling Language
UMPR	Uniform Multiprocessor Periodic Resource (model, interface)
WCET	Worst-Case Execution Time
XML	Extensible Markup Language

List of Symbols

Symbol	Meaning
C_i	worst-case execution requirement of task τ_i
D_i	relative deadline of task τ_i
$J_{i,j}$	jth job of task τ_i
MTF	major time frame
O_i	offset of time window ω_i relative to the beginning of the MTF
$S_\ell(\pi)$	total capacity of the ℓ fastest processors in platform π
$S_m(\pi)$	total capacity of platform π
T_i	minimum interarrival time (or period) of task τ_i
$a_{i,j}$	arrival time of job $J_{i,j}$
c_i	duration of time window ω_i
$d_{i,j}$	absolute deadline of job $J_{i,j}$
$e_{i,j}$	execution requirement of job $J_{i,j}$
i, j, k, ℓ, p	indices
m	number of processors
n	number of tasks
q	number of components
s_i	schedulable utilization of the ith fastest processor in π
t	time (context-specific indices are used)
u_i	utilization of task τ_i
$u_{\max}(\mathcal{T})$	maximum task utilization in task set $\mathcal{T}$
$u_{\text{sum}}(\mathcal{T})$	total utilization of task set $\mathcal{T}$
$\text{DBF}(\tau_i, t)$	demand bound function of τ_i for EDF/gEDF

(continues on next page)

(continued from previous page)

Symbol	Meaning
$\text{SBF}(\mathcal{R}, t)$	supply bound function of resource interface $\mathcal{R}$
$\mathcal{A}$	scheduling algorithm
$\mathcal{C}$	component
$\mathcal{R}$	an interface, expressed with some resource model
$\mathcal{S}$	schedule (in the sense of a job scheduling sequence)
$\mathcal{T}$	task set
$\mathcal{U}$	an interface, expressed with the UMPR model[1]
$\mathbb{R}$	the set of real numbers
$\mathbb{R}^+$	the set of positive real numbers
$\mathbb{N}$	the set of natural numbers (positive integers)
$\mathbb{N}_0$	the set of non-negative integers ($\mathbb{N} \cup \{0\}$)
Θ	interface budget
Π	interface period
α	interface bandwidth
δ_i	density of task τ_i
$\delta_{\max}(\mathcal{T})$	maximum task density in task set $\mathcal{T}$
$\delta_{\text{sum}}(\mathcal{T})$	total density of task set $\mathcal{T}$
$\lambda(\pi)$	lambda parameter of platform π
μ	an interface, expressed with the MPR model (Easwaran *et al.*, 2009b)
π	uniform multiprocessor platform
τ_i	ith task
ω_i	ith time window in a partition scheduling table

[1] Proposed in Chapter 4 of this thesis

Publications

The contributions of this thesis have been reported, partially and in preliminary versions, in the following publications.

Book chapters

Craveiro, J., Rufino, J. & Verissimo, P. (2010a). Architecting robustness and timeliness in a new generation of aerospace systems. In A. Casimiro, R. de Lemos & G. Gacek, eds., *Architecting Dependable Systems VII*, vol. 6420 of Lecture Notes in Computer Science, 146–170, Springer Berlin / Heidelberg.
DOI: 10.1007/978-3-642-17245-8_7

Journals

Craveiro, J., Rufino, J. & Singhoff, F. (2011a). Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems. *ACM SIGBED Review*, **8**(3):23–27, special issue of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11) Work-in-Progress session.
DOI: 10.1145/2038617.2038622.

Formal proceedings of international conferences

Craveiro, J. & Rufino, J. (2010b). Schedulability analysis in partitioned systems for aerospace avionics. In *15th Internacional Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, Bilbao, Spain.
DOI: 10.1109/ETFA.2010.5641243

Rufino, J. & Craveiro, J. & Verissimo, P. (2010b). Building a time- and space-partitioned architecture for the next generation of space vehicle avionics. In *8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2010)*, 179–190, Waidhofen an der Ybbs, Austria.
DOI: 10.1007/978-3-642-16256-5_18

Craveiro, J. & Rufino, J. (2010a). Adaptability support in time- and space-partitioned aerospace systems. In *2nd International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2010)*, 152–157, Lisbon, Portugal.
ISBN: 978-1-61208-109-0

Rosa, J., Craveiro, J. & Rufino, J. (2011). Safe online reconfiguration of time- and space-partitioned systems. In *9th IEEE International Conference on Industrial Informatics (INDIN 2011)*, Caparica, Lisbon, Portugal.
DOI: 10.1109/INDIN.2011.6034932

Informal proceedings, national conferences

Craveiro, J.P., Rosa, J. & Rufino, J. (2011b). Towards self-adaptive scheduling in time- and space-partitioned systems. In *32nd IEEE Real-Time Systems Symposium (RTSS 2011) Work-in-Progress session*, Vienna, Austria.

Craveiro, J.P., Silveira, R.O. & Rufino, J. (2012a). hsSim: an extensible interoperable object-oriented n-level hierarchical scheduling simulator. In *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2012)*, Pisa, Italy.

Craveiro, J.P., Souza, J.L.R., Rufino, J., Gaudel, V., Lemarchand, L., Plantec, A., Rubini, S. & Singhoff, F. (2012b). Scheduling analysis principles and tool for time- and space-partitioned systems. In *INFORUM 2012 - Simpósio de Informática*, Lisbon, Portugal.

Craveiro, J.P. & Rufino, J. (2013b). Uniform Multiprocessor Periodic Resource Model. In *4th International Real-Time Scheduling Open Problems Seminar (RTSOPS 2013)*, Paris, France.

Technical reports

Craveiro, J.P. & Rufino, J. (2012). Towards compositional hierarchical scheduling frameworks on uniform multiprocessors. Tech. Rep. TR-2012-08, University of Lisbon, DI–FCUL, revised January 2013.

Chapter 1

Introduction

A *real-time (computing) system* is a computing system such that its computations' correctness (or utility) is defined, not only in terms of the accuracy of the logical results, but also in terms of the time at which these results are provided. The relationship between the timeliness of result provision and their utility allows considering different classes of real-time. Real-time systems have been classically divided into *hard real-time* (HRT) systems and *soft real-time* (SRT) systems. An HRT system contains, at least, an HRT task — a task which *must always* meet its timeliness requirements (deadline); otherwise, the results of that task's computation have no utility. HRT systems are usually associated with applications where failure to meet temporal constraints may cause catastrophical effects, including the loss of human lives or harm thereto. An SRT system contains no HRT tasks, but contains at least an SRT task — a task which *should* meet its timeliness requirements, but may occasionally miss them (in which case the utility of the result degrades through time). SRT systems are usually associated with applications where the beneficial outcome from the observance of temporal constraints lies within the spectrum of user experience, or comfort. Research on real-time systems has focused on the set of algorithms and analysis techniques which allow system developers to know, prior to the system's deployment and execution, if it will be able to guarantee the fulfillment of its timeliness requirements (either HRT or SRT) (Kopetz, 1997; Verissimo & Rodrigues, 2001).

Computing systems have evolved throughout the years to meet various needs, including concerns about size, weight and power consumption (and, consequently,

cost). This led to a trend towards integrating separate systems as subsystems of a more complex mixed-criticality system on a common computing platform. Such system features the coexistence of different classes of real-time (SRT and HRT), and subsystems which may be developed by different teams and with different levels of assurance. The classical approach, often called *federated*, was to host each of these subsystems in separate communicating nodes with dedicated resources — with the consequent added weight and cost of computing hardware and cables. The added complexity of the system propagates to system's development, testing, validation and maintenance activities. Designing such complex systems around the notion of *component*, thus allowing *component-based analysis*, brings several benefits, some specific to real-time systems (Lipari *et al.*, 2005; Lorente *et al.*, 2006). In the case of different classes of real-time, the advantages of keeping the SRT and HRT parts of the system logically separated (and analyzing them as such) are twofold. On the one hand, the separate analysis allows fulfilling the HRT requirements of such components without imposing unnecessary pessimism on the analysis of the SRT components. On the other hand, with appropriate design considerations, the tardiness permitted to the SRT components shall not void the timeliness of the HRT components (Abeni & Buttazzo, 1998). One such design approach is *time and space partitioning* (TSP). Each component is hosted on a logical separation and containment unit — *partition*. In a TSP system, the various onboard functions are integrated in a shared computing platform, however being logically separated into partitions. Robust temporal and spatial partitioning means that partitions do not mutually interfere in terms of fulfillment of real-time and addressing space encapsulation requirements.

1.1 Context

1.1.1 Civil aviation

A prominent example of TSP system design is the adoption of the ARINC specifications 651 — Design Guidance for Integrated Modular Avionics (AEEC, 1991) — and 653 — Avionics Application Software Standard Interface (AEEC, 1997) — in the aviation and aerospace domains.

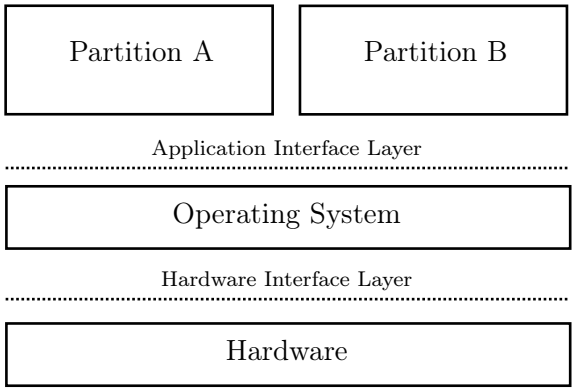


Figure 1.1: Basic architecture of an IMA computing module

The traditional approach, called *federated avionics*, makes use of distributed avionics functions packaged as self-contained units: Line Replaceable Units (LRU) and Line Replaceable Modules (LRM) (Watkins & Walter, 2007). An avionics system can be comprised of multiple LRUs or LRMs, potentially built by different contractors. What distinguishes LRUs from LRMs is that, while the former are potentially built according to independent specifications, the latter consummate a philosophy in which the use of a common specification is defended (Little, 1991). With each avionics function having its own dedicated (and sometimes physically apart) computer resources, which cannot be reallocated at runtime, inefficient resource utilization is a potential drawback from the inherent independence of faults (Audsley & Wellings, 1996; Sánchez-Puebla & Carretero, 2003).

On the other hand, Integrated Modular Avionics (IMA) architectures employ a high-integrity, partitioned environment that hosts multiple avionics functions of different criticalities on a shared computing platform. Figure 1.1 portrays a basic example of the layered architecture of a IMA module. IMA addresses the needs of modern systems, such as optimizing the allocation of computing resources, reducing size, weight and power consumption (a set of common needs in the area of avionics, which is commonly represented by the acronym SWaP), and consolidation development efforts (releasing the developer from focusing on the target platform, in favor of focusing on the software and easier development and certification processes) (Watkins & Walter, 2007).

The ARINC 653 specification (AEEC, 1997) is a fundamental block from the

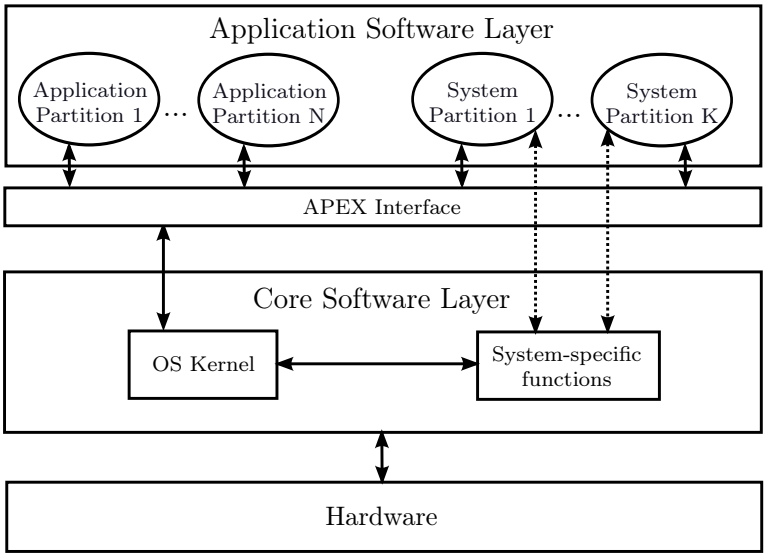


Figure 1.2: Standard ARINC 653 architecture — adapted from (AEEC, 1997)

IMA definition, where the partitioning concept emerges for protection and functional separation between applications, usually for fault containment and ease of validation, verification, and certification (AEEC, 1997; Rushby, 1999).

The architecture of a standard ARINC 653 system is sketched in Figure 1.2. At the application software layer, each application is executed in a confined context — a partition. The application software layer may include system partitions intended to manage interactions with specific hardware devices. Application partitions consist in general of one or more processes and can only use the services provided by a logical application executive (APEX) interface, as defined in the ARINC 653 specification (AEEC, 1997). System partitions may use also specific functions provided by the core software layer (e.g. hardware interfacing and device drivers), being allowed to bypass the standard APEX interface.

The ARINC 653 specification defines a standard interface between the software applications and the underlying operating system, known as application executive (APEX) interface. The first part of the specification (AEEC, 1997) describes a set of mandatory services, concerning partition management, process management, time management, intrapartition communication (i.e., between processes in the same partition), interpartition communication (i.e., between processes in different partitions), and health monitoring. The Part 2 of the ARINC 653 specification (AEEC, 2007)

adds, to the aforementioned mandatory services, optional services or extensions to the required services.

In ARINC 653 TSP systems, time partitioning is typically guaranteed by a two-level scheduler (AEEC, 1997). On the first level, partitions are selected to execute according to some schedule. When each partition is active according to such schedule, its tasks compete according to a local-level scheduler. This is a particular case of *hierarchical scheduling*, which is considered a good first building block for a component-based design and analysis approach (Lorente *et al.*, 2006).

1.1.2 Aerospace

The identification of similar requirements with the aviation industry led to the interest expressed from space industry partners in applying the time and space partitioning concepts of IMA and ARINC 653 to space missions onboard software.

North America The National Agency for Space Exploration (NASA) is one of the space industry players with documented interest in the concepts of TSP. Rushby (1999) analyzes the requisites and issues in providing time and space partitioning in IMA, which interfere with aspects of system design such as scheduling, communication, and fault tolerance. Formal methods are called for to be able to assure and certify safety-critical software for the deployment of an IMA system. Hodson & Ng (2007) at the NASA Software and Avionics Integration Office presented ideas for future avionics systems; one of the ideas consists of a modular, layered and partitioned software approach with support for ARINC 653 (AEEC, 1997) functionality. The presentation also highlights the need for tunable, scalable and reconfigurable avionics, and the problematic of power management. Black & Fletcher (2006) analyze the various aspects of the definition of an *open* system, in order to meet NASA's interest thereupon. The resulting recommendation for the design of a new system is to seriously consider employing widely used standards (e. g., for communications), non-proprietary hardware interfaces, and commercially available development tools. Fletcher (2009) picks up on these results, and documents the employment of IMA time and space partitioning concepts to create an open architecture solution which

addresses NASA's requirements, including cost savings and the avoidance of vendor lockin.

Europe In the European space industry domain, the TSP Working Group was established to cope with the issues of adopting TSP in space. This working group comprises representatives from the European Space Agency (ESA), the French government space agency (CNES, *Centre National d'Études Spatiales*), and from contractor companies Thales Alenia Space and EADS Astrium (a subsidiary of the European Aeronautic Defence and Space Company, dedicated to space transportation and satellite systems). Plancke & David (2003) proposed ensuring compatibility with ARINC 653/IMA as a future standardization action, so that the exchange of functional building blocks with the aeronautic industry (which had already adopted IMA) would be made possible. To manage the problem of how applications interface with the underlying operating system, ARINC 653 should be taken into account as an example, in order to define such an interface in a way that allows OS-independent software components. Windsor & Hjortnaes (2009) summarize the work of the TSP Working Group regarding the adoption of IMA-inspired time and space partitioning techniques into spacecraft avionics systems. The authors explain the principles of TSP, and both the benefits and the remaining technology gap to the intended adoption — to which no technological feasibility impairments were found. Planche (2008) establishes links between each aspect of the ARINC 653 specification and the relevant requirements and restrictions of its application in the space domain, in order to attain the applicability of each of those aspects.

1.1.3 Automotive industry

Besides the aviation and aerospace domains, the automotive industry has similar goals of temporal and spatial isolation. The AUTOSAR (AUTomotive Open System ARchitecture) is a joint initiative, established in 2003, involving automotive Original Equipment Manufacturers (OEM), their direct suppliers (the so-called Tier 1 suppliers), and other companies in various related industries (including electronics and software providers). From this cooperation stems the AUTOSAR specification of a standard software architecture for the automotive industry (AUTOSAR, 2006).

The top-level requirements for an AUTOSAR operating system include provisions that correspond, to some extent, to the notions of temporal and spatial isolation (AUTOSAR, 2013a, requirements SRS_Os_11008 and SRS_Os_11005, respectively). The specification of the operating system, however, does not prescribe the use of strict partitioned scheduling as a means to achieve temporal isolation among applications (AUTOSAR, 2013b).

1.2 Motivation

Over the years, processor manufacturers obtained performance improvements by increasing the clock rate of single processors. Such increase plateaued in the last decade, since the consequent increase in power dissipation reached the practical limits for cooling mechanisms. The trend in response was to take advantage of parallel (rather than faster) execution, by employing multiple processor cores. A processor that hosts multiple processor cores in the same chip is dubbed a *multicore* processor. The processor cores can have either private or shared memory addressing spaces; processor cores in the same multicore chip typically employ a shared memory addressing space (Patterson & Hennessy, 2009).

Multicore processors are paving their way into the realm of embedded systems (Mignolet & Wuyts, 2009), namely mixed-criticality systems which have subsystems with HRT requirements, such as those used in the civil aviation, aerospace, and automotive industries. Future avionics applications call for the application of multicore platforms to cope with increased performance requirements (Fuchsen, 2010). The latest versions of Aeroflex Gaisler's SPARC LEON processor, widely used by the European Space Agency, support multicore configurations, either with identical or non-identical processor cores (Andersson *et al.*, 2010). However, such capabilities are routinely not exploited, because of a lack of support thereto in terms of verification and certification (Anderson *et al.*, 2009). The use of multicore in safety-critical systems is still incipient and needs to be approached carefully (Parkinson, 2011; van Kampenhout & Hilbrich, 2013). This bottleneck extends to general computing as well: the step towards increasing the number of cores per microprocessor is being hindered by a lack of support from the application development side (Patterson, 2010). Due to the aviation, aerospace and automotive industries' prevalent

use of and interest in the concepts of time and space partitioning, compatibility between TSP and platforms with multiple processors, both identical and non-identical, is highly desired.

The ARINC 653 specification, a standard for TSP systems in aviation and aerospace, shows limited support thereto. The current approach to augmenting the processing capacity of safety-critical embedded systems is to have multiple uniprocessor nodes connected through some kind of bus — as in the classical federated approach. Besides the same SWaP implications as the latter, we identify a set of vectors of flexibility which are not taken advantage of. These include parallel execution, and reconfiguration of the binding between software and hardware parts (e.g., for fault tolerance purposes). This is particularly stringent when intervention on the system during its execution is not possible or desired, such as in planetary exploration robots, unmanned aerial vehicles (UAVs) or autonomous vehicles.

The top-level requirements for an AUTOSAR operating system have, contemporarily to this work[1], included some support to multicore. Tasks shall, however, be statically assigned to processor cores (AUTOSAR, 2013a, SRS_Os_80005). This requirement causes the specification of the AUTOSAR operating system to require that all tasks in the same application execute on the same core (AUTOSAR, 2013b, SWS_Os_00570).

1.3 Thesis statement

This work proposes the following research hypothesis:

> *The active exploitation of employing multiple (possibly non-identical) processor cores can*
>
> 1. *augment the processing capacity of the time- and space-partitioned (TSP) systems, while maintaining a compromise with size, weight and power consumption (SWaP); and*
> 2. *open room to supporting self-adaptive behavior to cope with unforeseen changes in operational and environmental conditions.*

[1] AUTOSAR Release 4.0, November 2011

The architecture we consider and improve (as a reference of TSP system design) in this dissertation was developed within activities sponsored the European Space Agency, subordinated to the adoption of TSP systems in the aerospace domain. However, by using a more general methodology (which we detail in the next section), we expect the present work to be of use upon other TSP system architectures, and other system architectures for safety-critical and mixed-criticality domains.

1.4 Methodology

As seen in Section 1.1.1, TSP systems typically employ a two-level hierarchical scheduler. Analyzing TSP systems as a special case of hierarchical scheduling allows reusing the obtained results in a more general class of systems and applications. Hierarchical scheduling is a current topic in the real-time scheduling, as an attempt to solve real problems in real application scenarios of embedded software. We can identify the roots of hierarchical scheduling in resource reservation frameworks, where an asymmetric hierarchy is employed to allow coexistence of HRT tasks and aperiodic SRT requests in multimedia applications (Abeni & Buttazzo, 1998). Hierarchical scheduling also sees application in the virtualization field (Lackorzyński *et al.*, 2012; Xi *et al.*, 2011) and in networked embedded systems (Santos *et al.*, 2011) — and the number of levels may go beyond two (Mok & Feng, 2002; Santos *et al.*, 2011).

The need for independent development and arbitrary number of levels are the main motivation and advantages of *compositional analysis*. Compositionality is the property of a complex system that can be analyzed by evaluating some properties of its components (without knowing their internal structure or hierarchy) and the way they are composed (Easwaran *et al.*, 2006; Hausmans *et al.*, 2012). Compositional analysis allows encompassing, under the same theoretical banner, resource reservation frameworks and hierarchical scheduling frameworks.

Within compositional analysis as applied to real-time scheduling, a component comprises a workload and a scheduler, and is abstracted by a resource interface. Each component's interface hides **(i)** from its parent component, the specific characteristics of its resource demand ; and **(ii)** from itself, the specific characteristics of the resource supply it receives from its parent component. Compositional analysis,

which we describe and survey in more detail in Section 2.4, comprises three main points (Shin & Lee, 2007).

1. *Local-level schedulability analysis* — analyzing the schedulability of a component's workload upon its scheduler and the resource supply expressed through the component's resource interface.

2. *Component abstraction* — obtaining the component's resource interface from its inner characteristics.

3. *Interface composition* — transforming the set of interfaces abstracting the real-time requirements of individual subcomponents into an interface abstracting the requirement of scheduling these subcomponent together according to a given *intercomponent scheduling* strategy.

1.5 Contributions

The contributions presented in this dissertation are as follows.

1. **System architecture and model**
   We propose an improved reference architecture for TSP systems with support for multiprocessor. This constitutes a more flexible approach to multiprocessor than that of interconnected uniprocessor nodes which is current practice. Our proposal enables (as we show in the subsequent contributions):

   - *Interpartition parallelism* — allowing more than one partition (application) to be active simultaneously (on distinct processors).
   - *Intrapartition parallelism* — allowing one partition (application) to use more than one processor to schedule its tasks.
   - *Adaptability and self-adaptability* — reconfiguring, in execution time, the binding between software and hardware parts (application tasks and processors) to adapt to different modes of operations, goals, or events.

2. **Compositional analysis on (non-)identical uniform multiprocessors**
   We propose the first interface model for the definition of compositional scheduling frameworks on uniform multiprocessors (those which may be non-identical, but only in terms of their speed). This contribution allows the formal analysis of TSP systems with interpartition and/or intrapartition parallelism on potentially non-identical multiprocessors. Our contribution encompasses the three aspects of compositional analysis of HSFs.

   - *Local-level schedulability analysis* — applying and extending previous results from other authors (Baruah & Goossens, 2008; Easwaran *et al.*, 2009b), we provide a sufficient local-level schedulability test which allows verifying if the application inside a component can fulfill its timeliness requirements with the resource provision specified by the component interface.
   - *Component abstraction* — we provide mechanisms to select the parameters of a components's interface which guarantee that the contained application fulfills its timeliness requirements.
   - *Interface composition* — we propose an algorithm to schedule components (given each one's interface) that specifically caters to the scenario where non-identical processors coexist. We also show how to derive the overall resource requirement to schedule these components.

3. **Simulation, analysis and schedule generation**
   We design and implement hsSim, an object-oriented tool for scheduling simulation, analysis, and generation. hsSim was carefully designed with attention to the applicable software design patterns, with the goal of modularity, extendability and interoperability. This careful approach is customarily not employed, which is the main reason why we design a tool from the ground up instead of modifying an existing tool. This does not however preclude the back port of some of our contributions into the code of other tools, such as Cheddar — which is already very mature with respect to non-hierarchical scheduling analysis and simulation.

- *Analysis* — we have incorporated our contributions on compositional analysis onto hsSim, so as to help derive the parameters that allow schedulability of the system.
- *Simulation* — we have implemented support to the simulation with many scheduling strategies, including global scheduling on multiprocessors (both identical and non-identical). We have also added our proposed intercomponent scheduling algorithm, so as to validate our claims. The simulation is logged to a file that allows visualization with an external tool.
- *Schedule generation* — we take advantage of hsSim's by-design extendability and implement a new logger which creates a partition scheduling table from the events in the simulation. The table is generated in a format inspired by the ARINC 653 XML format, so as to be used in the configuration of real TSP systems.

4. **Preliminary results on self-adaptation in TSP systems**
   We report the experiments we performed, both with a prototype implementation of a TSP system and through simulation, to address the second part of the research statement in Section 1.3.

1.6 Document outline

This dissertation is structured into 7 chapters (including this one).

Chapter 2 provides background notions and previous results.

Chapter 3 describes the first contribution — the improved reference architecture for TSP systems with support for multiprocessor, and respective formal model.

Chapter 4 describes the second contribution — compositional analysis of hierarchical scheduling frameworks on non-identical multiprocessors.

Chapter 5 describes the third contribution — tool-assisted simulation, analysis and schedule generation.

Chapter 6 describes the fourth contribution — towards self-adaptation in TSP systems.

Chapter 7 closes the document, with concluding remarks and future work directions.

Chapter 2

Background and Related Work

This chapter addresses the background concepts fundamental to this work, as well as previous related work. We start (Section 2.1) by introducing the reader to background concepts and definitions common to the whole spectrum of research on real-time scheduling; since the whole body of prior work in real-time scheduling theory suffers from inconsistent use of both nomenclatures and notations, we precisely establish in this section the exact meaning of terms and symbols used in this dissertation, following the most recent and common uses thereof. Then, in Section 2.2, we survey hard real-time schedulability analysis on dedicated platforms, i.e. upon a system model where all tasks are handled by one scheduler, which has a processing platform available at all times. In Section 2.3, we trace back the origin of hierarchical scheduling to the concepts of resource reservations and scheduling servers, and survey the existing approaches. We then (Section 2.4) present and survey compositional analysis from the point of view of a theoretical framework to perform verification of systems based on either resource reservations or hierarchical scheduling. In Section 2.5, we explore the state of practice with respect to TSP systems, namely operating system support and tools to support the verification of temporal properties in the integration phase of TSP system development.

2.1 Real-time scheduling background

Although TSP systems enable the safe coexistence of both HRT and SRT workloads, we will focus on *the timeliness aspects of scheduling HRT tasks*. As such, we now

present notions and results thereto pertaining.

Research on hard real-time scheduling dates back to the late 1960s (Liu, 1969). For that reason, it would be intractable and unnecessary to perform an exhaustive survey thereof in this dissertation. We will survey the results relevant to the present work, after introducing the notions and previous results necessary to their understanding. For a more in-depth analysis of previous work on real-time scheduling, the reader is directed to the surveys by Audsley *et al.* (1995), Sha *et al.* (2004), Carpenter *et al.* (2004), and Davis & Burns (2011), and to the books by Kopetz (1997, Chapter 11) and Buttazzo (1997).

2.1.1 Task models

A *task set* is a multiset[1] of n tasks, formally denoted as $\mathcal{T} \stackrel{\text{def}}{=} \{\tau_i\}_{i=1}^{n}$. We assume that each task τ_i is *independent* from the remaining ones, in the sense that they compete for no resource other than the processor. Furthermore, when choosing a task model, we have to choose in fact two models: the *activation* model, which specifies the amount of work the task has to perform and how it is distributed throughout time, and the *deadline* model, which specifies the temporal constrains for the tasks' activations — generally referred to as *jobs*.

2.1.1.1 Activation model

With respect to the distribution and amount of work throughout time, a task can be modeled as being either *aperiodic*, *periodic*, or *sporadic*. In the aperiodic task model, a task generates a stream of jobs whose arrival times are not known beforehand and whose execution requirement is only known, at the best, when the job arrives (some authors present results for aperiodic jobs whose execution requirement is only known when the job finishes). In the work presented in this dissertation, we do not consider this model. It is nevertheless relevant for this literature review, since the accommodation of aperiodic jobs motivated some approaches closely related to hierarchical scheduling.

[1] In set theory, a multiset is a generalization of a set in which multiple instances of identical elements may occur; in this case, it means that two identical tasks may coexist in the same task set (Bini *et al.*, 2009a).

Liu & Layland (1973) introduced the *periodic task model*. Under this model, a real-time task $\tau_i \stackrel{\text{def}}{=} (T_i, C_i)$ is characterized by its *period* T_i and *maximum execution requirement*[2] C_i. The deadline is implicit and identical to the task's period. Under this model, each task generates an unbounded sequence of *jobs* (or activations, or instances). The jth job generated by task τ_i, $J_{i,j} \stackrel{\text{def}}{=} (a_{i,j}, e_{i,j}, d_{i,j})$, is characterized by an *arrival time* a_i, an *execution requirement* $e_{i,j}$, and an *absolute deadline* $d_{i,j} = a_{i,j} + T_i$. The arrival times of two consecutive jobs of the same task are separated by *exactly* T_i time units.

Mok (1983) introduced the *sporadic task model* as a generalization of the periodic task model. Under this model, a real-time task $\tau_i \stackrel{\text{def}}{=} (T_i, C_i, D_i)$ is characterized by its *minimum interarrival time* T_i, maximum execution requirement C_i, and *relative deadline* D_i. Under this model, each task generates an unbounded sequence of jobs. The jth job generated by task τ_i, $J_{i,j} \stackrel{\text{def}}{=} (a_{i,j}, e_{i,j}, d_{i,j})$, is characterized by an *arrival time* a_i, an *execution requirement* $e_{i,j}$, and an *absolute deadline* $d_{i,j} = a_{i,j} + D_i$. The arrival times of two consecutive jobs of the same task are separated by *at least* T_i time units.

The derivation of the C_i parameter of a task constitutes a discipline of its own within real-time scheduling theory, with specific concepts and mechanisms which are out of the scope of this work; for a survey on the subject, the reader is referred to (Wilhelm *et al.*, 2008). We assume the maximum execution requirement of each task has been correctly derived with regard to the computing platform being considered. In Chapter 5, about scheduling analysis tools, we present these assumptions in more detail, explaining how we envision mapping worst-case execution times on a real hardware platform into worst-case execution requirements on a platform model (Section 2.1.2).

The length of the time interval between the instant a job arrives and the instant a job completes its execution is termed the job's *response time*. The maximum response time expected to be yielded by all jobs of a task is the task's *worst-case response time*. For a task to fulfill its temporal requirements, its worst-case response time must not be greater than its relative deadline.

[2]Historically called *worst-case execution time* (WCET), a name which becomes slightly inappropriate when dealing with different task execution rates.

In this work, we mainly consider workloads consisting of *constrained-deadline sporadic tasks*. When needed, we consider a periodic task as a special case of an implicit-deadline sporadic task.

2.1.1.2 Deadline model

As we have seen, sporadic tasks (including periodic tasks) are characterized by a relative deadline, which specifies the difference between the arrival of each of its jobs and the respective absolute deadline. The relative deadlines of sporadic tasks may be classified as:

- *implicit* — the task's deadline is equal to its minimum interarrival time;
- *constrained* — the task's deadline is not less than or equal to its minimum interarrival time ($D_i \leq T_i$); or
- *arbitrary* — there is no restriction on the relationship between the task's deadline and the minimum interarrival time.

In this dissertation, we focus on implicit and constrained deadlines. With both these models, all jobs must finish their execution before the arrival of the next job of the same task.

2.1.1.3 Additional notions

Before surveying schedulability analysis results for uniprocessor and global multiprocessor scheduling, let us introduce some notions which are recurrently applied regardless of the platform.

A notable notion related to the periodic and sporadic task models is that of *task utilization*. The utilization of task τ_i is represented as $u_i \stackrel{\text{def}}{=} C_i/T_i$. When dealing with sporadic tasks, we have to consider an additional notion, *task density*, defined as $\delta_i \stackrel{\text{def}}{=} C_i/\min\{T_i, D_i\}$. Since we do not deal with arbitrary-deadline tasks, we use the simplified definition $\delta_i \stackrel{\text{def}}{=} C_i/D_i$.

These two notions are also commonly applied to task sets, in the following ways:

- $u_{\text{sum}}(\mathcal{T}) \stackrel{\text{def}}{=} \sum_{\tau_i \in \mathcal{T}} u_i$ represents the total utilization of task set $\mathcal{T}$;

- $u_{\max}(\mathcal{T}) \stackrel{\text{def}}{=} \max_{\tau_i \in \mathcal{T}}\{u_i\}$ represents the maximum task utilization among all tasks in $\mathcal{T}$; and
- $\delta_{\max}(\mathcal{T}) \stackrel{\text{def}}{=} \max_{\tau_i \in \mathcal{T}}\{\delta_i\}$ represents the maximum task density among all tasks in $\mathcal{T}$.

2.1.2 Platform models

Multiprocessor platforms can be either identical, uniform or heterogeneous; in this dissertation, we focus only on the first two.

An *identical* multiprocessor platform is composed of m unit-*capacity* processors. In turn, a *uniform* multiprocessor platform is composed of processors with differing capacities. We represent a uniform multiprocessor platform as

$$\pi \stackrel{\text{def}}{=} \{s_i\}_{i=1}^{m} \ ,$$

where each s_i corresponds to each processor's capacity. A processor's capacity expresses its schedulable utilization; the semantics is that, if a task is scheduled to execute for one time unit on a processor with capacity s_i, its remaining execution requirement is decrements by s_i units. The capacity of a processor is a relative way of expressing its speed.

For convenience, we will use $S_\ell(\pi)$, with $\ell \leq m$, to represent the sum of the capacities of the ℓ fastest processors in π:

$$S_\ell(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{\ell} s_i \ ; \tag{2.1}$$

by convention, $S_0(\pi) = 0$, for any π. Hence, $S_m(\pi)$ represents the total capacity of π. We will also make use of the *lambda* parameter, which is defined as

$$\lambda(\pi) \stackrel{\text{def}}{=} \max_{\ell=1}^{m} \frac{S_m(\pi) - S_\ell(\pi)}{s_\ell} \tag{2.2}$$

and abstracts how different or similar to an identical multiprocessor platform is π. An m-processor identical multiprocessor platform can be seen as a special uniform multiprocessor platform with m processors, such that

- $s_i = 1.0$, for all $1 \leq i \leq m$;
- $S_\ell(\pi) = \ell$, for all $0 \leq i \leq m$; and
- $\lambda(\pi) = m - 1$.

2.1.3 Scheduling algorithm classification

In this dissertation, we focus on *preemptive priority-driven* scheduling algorithms. They work by assigning priorities to the active jobs, and selecting that (or those) with highest priority for execution. Since we deal with preemptive algorithms, a job is forced to suspend its execution (i.e., is *preempted*) if one or more higher priority jobs justifies it. To survey scheduling algorithms and schedulability analysis results, scheduling algorithms can be classified according to how priorities are assigned to jobs and, for multiprocessor, according to how jobs may be scheduled on the available processors.

2.1.3.1 Priority-based classification

Regarding priority assignment, we divide algorithms into three categories: *fixed task* priority, *fixed job* priority, and *dynamic* priority. This taxonomy is widely used in the literature, although the specific names of the categories vary (Davis & Burns, 2011).[3]

With *fixed task priority*, priority is a characteristic of the task. Therefore, all of a task's jobs are applied the same priority. Whenever two tasks have active jobs, the relationship between those two jobs' priorities will always be the same.

With *fixed job priority*, priority is a characteristic of the job. The jobs of each task may have different priorities, but each job has always the same priority since it becomes active until it finishes execution.

Finally, with *dynamic priority*, priority is a characteristic of the job which may change over time.

[3] In uniprocessor scheduling, due to some optimality results (Section 2.1.4.1), it is also common to merge the fixed job priority and dynamic priority algorithms (thus having a 2-category taxonomy of fixed vs. dynamic priority algorithms) (Carpenter *et al.*, 2004).

2.1.3.2 Migration-based classification

Regarding the degree of migration allowed when allocating the workload among the available processors, algorithms for multiprocessors can be categorized as either partitioned (no migration), restricted-migration global (task-level migration only) or unrestricted-migration global (full job-level migration).

In *partitioned* algorithms, tasks are assigned to processors and all of each task's jobs execute on the same processor. The scheduler maintains a separate job queue for each processor, and reduces the problem of scheduling to multiple instances of uniprocessor scheduling.

In *restricted-migration global* algorithms, the jobs of one same task may execute on different processor throughout time, but each job will always execute on the same processor. Conversely, in *unrestricted-migration global* algorithms, any job may be preempted in one processor and resume execution in another processor. Without loss of generality, we follow the assumption (common in real-time scheduling theory) that the cost of a job's migration from one processor to another is either negligible or already accounted for in its execution requirement. For this reason, regarding global multiprocessor scheduling, we will only consider unrestricted-migration global scheduling algorithms, to which we will refer to as simply "global".

2.1.3.3 Work-conserving algorithms

A scheduling algorithm is said to be *work-conserving* if it guarantees that there exists no instant at which some processing capacity is unused (idle) and there is an active job (ready to execute).

2.1.4 Schedulability analysis notions

2.1.4.1 Feasibility, schedulability, and optimality

A task set $\mathcal{T}$ is *feasible* if and only if there exists a sequence $\mathcal{S}$ according to which all jobs of all tasks $\tau_i \in \mathcal{T}$ fulfill their timeliness requirements. Regarding schedulability, let $\mathcal{A}$ be a scheduling algorithm. A task set $\mathcal{T}$ is $\mathcal{A}$-*schedulable* if and only if, when scheduled according to algorithm $\mathcal{A}$, all jobs of all tasks $\tau_i \in \mathcal{T}$ fulfill their timeliness requirements.

Furthermore algorithm $\mathcal{A}$ is said to be *optimal*, within some class of algorithms to which it belongs and some task model which it is able to schedule, if every task set (obeying to such task model) which is schedulable according to some algorithm in that class is also $\mathcal{A}$-schedulable. For example, the Deadline Monotonic scheduling algorithm is an optimal algorithm w.r.t. uniprocessor fixed task priority scheduling of constrained-deadline sporadic task sets.

2.1.4.2 Types of tests

As in logic in general, there are three types of tests in schedulability analysis: sufficient, necessary and exact (sufficient and necessary). With a *sufficient* schedulability test, we are able to assess if a task set is schedulable under given assumptions, but we are not able to asses if a task set is *not* schedulable — if the test does not give a positive result, the result is inconclusive and we can only say that the task set *may be* unschedulable. With an *exact* (necessary and sufficient) schedulability test we can surely assess if a task set is schedulable or not under given assumptions. The same applies for sufficient/exact feasibility tests.

2.2 Hard real-time scheduling on dedicated platforms

2.2.1 Scheduling on uniprocessor platforms

Scheduling is assumed to occur on a unit-capacity processor. This means that, within each unit of time, one unit of the execution requirement of the scheduled job is executed — in other terms, each processor has a schedulable utilization of one execution unit per time unit.

2.2.1.1 Fixed task priority

The seminal example of an algorithm using a fixed task priorities assignment is *Rate Monotonic* (RM), proposed by Liu & Layland (1973), in which higher priorities are assigned to tasks with shorter periods. RM assumes an implicit-deadline periodic

task model. In a classical result, Liu & Layland (1973)'s sufficient test states that a periodic task set $\mathcal{T}$ is RM-schedulable on a unit-speed processor if

$$u_{\text{sum}}(\mathcal{T}) \leq n(2^{1/n} - 1) \ .$$

Sufficient tests subsequently proposed by Lauzac *et al.* (1998) and by Bini *et al.* (2003) have been found to outperform it (Lupu *et al.*, 2010). Exact schedulability of implicit-deadline periodic tasks can be assessed through response time analysis (Audsley *et al.*, 1993).

Leung & Whitehead (1982) extended Liu & Layland (1973)'s analysis to the case of constrained deadlines, proposing the *Deadline Monotonic* algorithm, which assigns higher priorities to tasks in non-increasing order of their (relative) deadlines. Lehoczky (1990) in turn extended Leung & Whitehead (1982)'s analysis on the Deadline Monotonic scheduling to periodic task sets with arbitrary deadlines.

2.2.1.2 Fixed job priority

The *Earliest Deadline First* (EDF) algorithm (Liu & Layland, 1973) is the lead member of this category; in EDF, jobs are prioritized according to their absolute deadline time, which is constant. An implicit-deadline periodic task set $\mathcal{T}$ is EDF-schedulable on a unit-speed processor if and only if (Liu & Layland, 1973)

$$u_{\text{sum}} \leq 1 \ . \tag{2.3}$$

Dertouzos (1974) proved that EDF is optimal for the uniprocessor. This result is obtained by proving that any feasible schedule can be transformed into a schedule which follows the EDF strategy (Dertouzos & Mok, 1989).

For constrained-deadline sporadic task sets, the condition in Equation (2.3) is only sufficient. Baruah *et al.* (1990) proved an exact (sufficient and necessary) condition, incorporating the notion of *demand bound function*, which provides an upper bound on the maximum cumulative execution requirement by jobs of sporadic task τ_i which have both their arrival and deadline times within any time interval

with length t. For EDF scheduling, it is defined as

$$\text{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \max\left\{0, \left(\left\lfloor \frac{t - D_i}{T_i} + 1 \right\rfloor\right) \cdot C_i\right\} \quad . \tag{2.4}$$

They have derived the following exact condition based on the demand bound function: a constrained-deadline sporadic task set $\mathcal{T}$ is EDF-schedulable on a unit-speed processor if and only if $u_{\text{sum}} \leq 1$ (Equation (2.3)) and

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, t) \leq t, \text{ for all } t > 0 \quad . \tag{2.5}$$

The demand bound function is used to compute the *load* function of a sporadic task system:

$$\text{LOAD}(\mathcal{T}) \stackrel{\text{def}}{=} \max_{t>0} \frac{\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, t)}{t} \quad . \tag{2.6}$$

Using this function, the condition expressed in Equation (2.5) can be expressed as $\text{LOAD}(\mathcal{T}) \leq 1$. The load function is also used in multiprocessor schedulability tests, as we see in forthcoming sections.

2.2.1.3 Dynamic priority

An example of an algorithm with such priority assignment is *Least Laxity First* (LLF), which assigns priority according to each job's laxity (Mok, 1983) — which is the difference between the remaining time until its absolute deadline time and total time the job needs to be allowed to execute to finish its execution requirement.

LLF is found to be optimal for the uniprocessor case. According to Dertouzos & Mok (1989), such optimality can be proved in the same way as Dertouzos (1974) proves the optimality of EDF.

2.2.2 Partitioned scheduling on identical multiprocessors

Partitioned scheduling can be seen as consisting of two steps: *partitioning* tasks among processors (offline), and scheduling the tasks assigned to each processor using an uniprocessor algorithm (online) (Baker & Baruah, 2007).

Dhall & Liu (1978) described a phenomenon which remained known as "Dhall effect", which we describe in detail in Section 2.2.3.1, and which established for some years a generalized perception of superiority of partitioned scheduling over global scheduling.

Nevertheless, partitioning scheduling lends itself to non-work-conserving behavior; tasks with active jobs will not take advantage of idle capacity of processors other than the one to which the task is assigned. As such, we focus on global multiprocessor scheduling.

Partitioning tasks into processors is equivalent to a bin-packing problem, which is proven to be NP-hard in the strong sense. For this reason, heuristic-based strategies are employed to make the problem tractable. Such strategies are characterized by two aspects: in which order as the tasks considered (the *task sorting criterion*), and how is processor for each task selected (the *heuristic* itself). The possible task sorting criteria come from the set:

$$\{\text{Increasing}, \text{Decreasing}\} \times \{D_i, \delta_i, T_i, u_i\} \ .$$

After chosing a task sorting criterion, each task in order is assigned to a processor, employing one of the following heuristics:

- *First Fit* — select the first processor where the task fits (starting from the very first);
- *Next Fit* — select the first processor where the task fits (starting from current);
- *Best Fit* — select the processor where the task fits with less remaining processor capacity;
- *Worst Fit* — select the processor where task fits with most remaining processor capacity.

Verification of whether a task fits on a processor is done using uniprocessor schedulability tests.

According to Lupu *et al.* (2010), the choice of strategy depends on the total density of the task set, and on whether the goal is to minimize the number of processors

or to ensure that processors have some slack to accommodate, for instance, the admission of new tasks in runtime. The Decreasing δ_i sorting criterion is suitable for most cases (independently of the employed schedulability test). Regarding heuristics, if the goal is to minimize m, then Next Fit or Best Fit are preferred (depending on whether the total density of the task set is, respectively, $\leq 0.5 \cdot m$ or $> 0.5 \cdot m$); if the goal is to ensure execution slack, Worst Fit is preferred.

2.2.3 Global scheduling on identical multiprocessors

Global scheduling algorithms on identical multiprocessors are, by construction, work-conserving, since:

1. if there is an active job waiting to execute, no processor is idle; and
2. when the number of active jobs is less than the number of processors, all active jobs are executing.

This does not, however, make them superior to their partitioned counterparts, as we now see.

2.2.3.1 Fixed job priority

As mentioned, fixed job priority global scheduling is subject to the so-called "Dhall effect" (Dhall & Liu, 1978): some task sets with arbitrarily low total utilizations can be unschedulable regardless of the number of processors employed. The prototypical scenario of the "Dhall effect" (characterized by one high utilization task and several higher-priority low utilization tasks) can be solved using partitioned scheduling (to eliminate the contention between the high and low utilization tasks). This led to a perceived superiority of partitioned scheduling, which had impact of the focus of forthcoming real-time scheduling research. Such superiority was eventually disproved. Leung & Whitehead (1982) showed that there are some sporadic task sets which are scheduled by some partitioned fixed task priority algorithm but not by any global fixed task priority algorithm, as well as some sporadic task sets which are scheduled by some global fixed task priority algorithm but not by any partitioned fixed task priority algorithm. More succinctly, this result proves that global and

partitioned scheduling are *incomparable* with respect to fixed task priority scheduling algorithms. Baruah (2007) proved that global and partitioned scheduling are incomparable with regard to fixed job priority as well.

In the realm of global multiprocessor scheduling, fixed job priority algorithms such as EDF have received more attention than fixed task priority ones (Baker & Baruah, 2007). Buttazzo (2005) performed comparative experiments between RM and EDF, and point out that the real advantage of RM over EDF is implementation simplicity. In turn, EDF allows a more efficient utilization of the processor, which the authors point out has particular advantage when dealing with embedded systems and resource reservations (a particular case of hierarchical scheduling). Hence, in this dissertation, we focus mostly on global EDF (gEDF).

Goossens *et al.* (2003) studied gEDF scheduling of implicit-deadline sporadic task sets, providing a sufficient gEDF-schedulability test. Minor extensions to their proofs yield the so-called "density test" (Baruah, 2007) for constrained-deadline periodic task sets. According to this test, task set $\mathcal{T}$ is gEDF-schedulable on an m-processor identical multiprocessor platform if

$$\delta_{\text{sum}}(\mathcal{T}) \leq m - (m-1) \cdot \delta_{\text{max}}(\mathcal{T}) \ .$$

Baker (2003) proposed a sufficient gEDF-schedulability test for constrained-deadline periodic task sets on an m-processor identical multiprocessor platform. Before the test itself, we now highlight the structure of the proof by contrapositive developed by Baker (2003), which was used in several later works on schedulability of sporadic task sets.

1. Assume some job of a task $\tau_k \in \mathcal{T}$ misses its deadline at instant t_{d} (and it is the first job to miss a deadline).

2. Consider an interval $[t_0, t_{\text{d}}[$ (with $t_0 < t_{\text{d}}$), which is dubbed the *interference interval*. The exact definition of t_0 differs among works employing this proof structure.

3. Obtain an upper bound on the amount of work that gEDF is required (but fails) to over the interval $[t_0, t_{\text{d}}[$. Two sources contribute to this interference:

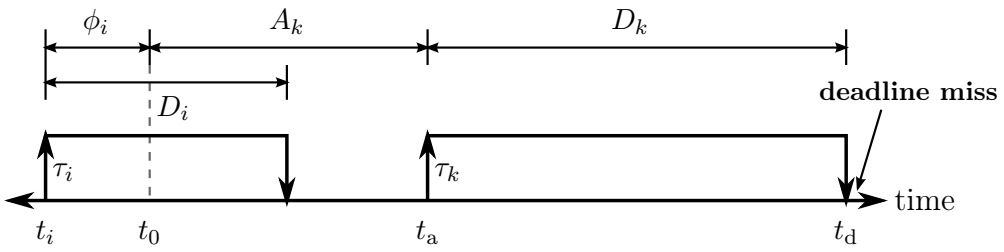


Figure 2.1: Execution pattern considered in Baker (2003)'s proof and thereon inspired works.

(i) jobs released in the interval; and **(ii)** jobs released before t_a which have not completed their execution up to that instant (*carry-in* jobs).

4. Derive a necessary condition for unschedulability — which will have the form *"If task set $\mathcal{T}$ is not gEDF-schedulable, then, for some task $\tau_k \in \mathcal{T}$, condition ψ holds"*.

5. Take the contrapositive of this condition (*"task set $\mathcal{T}$ is gEDF-schedulable if, for all tasks $\tau_k \in \mathcal{T}$, condition ψ does not hold"*), which yields a sufficient condition for schedulability.

The execution pattern underlying to this proof structure is shown in Figure 2.1. Besides the considered job of task τ_k, the figure illustrates a hypothetical job of task τ_i which is released prior to time instant t_0 and carries in some execution to the considered interference interval, $[t_0, t_d[$.

To derive their sufficient gEDF-schedulability tests, Baker (2003) considers t_0 to be the earliest time instant prior to the arrival of the deadline-missing job of τ_k ($t_a = t_d - D_k$) at which a certain condition is satisfied, whereas Bertogna *et al.* (2005) considered $t_0 = t_a = t_d - D_k$. Baruah (2007) analyzed the shortcomings of both Baker (2003)'s and Bertogna *et al.* (2005)'s tests, and derived a sufficient schedulability test by employing Baker (2003)'s strategy, considering t_0 to be the earliest time instant prior to the arrival of the deadline-missing job of τ_k at which some processor is idle. As pointed out before, since gEDF is a work-conserving scheduling algorithm, if some processor is idle, then all the active jobs must be executing on some processor. The sufficient schedulability condition obtained by

Baruah (2007) is that, for all tasks $\tau_k \in \mathcal{T}$ and for all $A_k \geq 0$,

$$\sum_{\tau_i \in \mathcal{T}} \hat{I}_i + \sum_{m-1 \text{ largest}} (\bar{I}_i - \hat{I}_i) \leq m(A_k + D_k - C_k) \ ,$$

where

$$\hat{I}_i \stackrel{\text{def}}{=} \begin{cases} \min\{\text{DBF}(\tau_i, A_k + D_k) - C_k, A_k\} & \text{if } i = k \\ \min\{\text{DBF}(\tau_i, A_k + D_k), A_k + D_k - C_k\} & \text{otherwise} \end{cases} , \tag{2.7}$$

$$\bar{I}_i \stackrel{\text{def}}{=} \begin{cases} \min\{\text{DBF}'(\tau_i, A_k + D_k) - C_k, A_k\} & \text{if } i = k \\ \min\{\text{DBF}'(\tau_i, A_k + D_k), A_k + D_k - C_k\} & \text{otherwise} \end{cases} , \text{ and}$$

$$\text{DBF}'(\tau_i, t) \stackrel{\text{def}}{=} \left\lfloor \frac{t}{T_i} \right\rfloor \cdot C_i + \min \left\{ C_i, t - \left\lceil \frac{t + T_i - D_i}{T_i} \right\rceil \cdot T_i \right\} \ .$$

$\hat{I}_i$ and $\bar{I}_i$ are upper bounds on the interference of τ_i, respectively if it does or does not have a job carrying in some execution to the $[t_0, t_d[$ time interval. By definition of the interval, at most $m - 1$ tasks have such carry-in jobs.

2.2.3.2 Hybrid variants of gEDF

To address the previously described "Dhall effect" (Dhall & Liu, 1978) without being subject to the higher number of preemptions characteristic of LLF, *hybrid* variants of gEDF were developed. EDZL (Earliest Deadline until Zero Laxity) (Lee, 1994) borrows the use of laxity from LLF, but only considers it for priority assignment purposes when needed. EDZL gives highest priority to jobs with zero laxity, assigning the priorities of the remaining jobs according to the classic EDF policy.

Other approaches involve applying fixed priorities to some tasks, while maintaining the fixed job priority aspect of EDF for the remaining tasks. EDF-US (Srinivasan & Baruah, 2002) gives a static highest priority to tasks with utilizations greater than $\frac{m}{2m-1}$ (with ties arbitrarily broken), and schedules the remaining tasks' jobs according to the classic EDF policy. fpEDF (Baruah, 2004) gives a static highest priority to at most $m - 1$ tasks with utilization greater than $\frac{1}{2}$, and schedules the remaining tasks' jobs according to the classic EDF policy.

2.2.4 Global scheduling on uniform multiprocessors

We already saw that global scheduling algorithms on identical multiprocessors are, by construction, work-conserving. The same does not automatically apply for uniform multiprocessors without specific considerations. Funk *et al.* (2001) introduced a gEDF scheduling strategy which is work-conserving on uniform multiprocessors. Such property is achieved by obeying to the following rules (cf. rules for global scheduling on identical multiprocessors, Section 2.2.3):

1. if there is an active job waiting to execute, no processor is idle;
2. when the number of active jobs is less than the number of processors, the jobs execute on the fastest processors (and only the slowest ones are idle); and
3. higher-priority (in the case of gEDF, earlier-deadline) jobs execute on faster processors.

Baruah & Goossens (2008) provide a sufficient gEDF-schedulability test for constrained-deadline sporadic task sets on uniform multiprocessors. They derive it using Baker (2003)'s technique: from an execution pattern where a job is the first to miss a deadline, they derive a necessary unschedulability condition; then, the contrapositive of this condition provides a sufficient condition for schedulability. In deriving the necessary unschedulability condition, Baruah & Goossens (2008) take special advantage of the second aforementioned rule of gEDF — if some processor is idle, then an active job is guaranteed to be executing on a processor at least as fast as that one. The authors thus prove that task set $\mathcal{T}$ is schedulable by gEDF on an m-processors uniform multiprocessor platform π if

$$\textsc{load}(\mathcal{T}) \leq S_m(\pi) - (\lambda(\pi) - \nu) \cdot \delta_{\max}(\mathcal{T}) \ , \tag{2.8}$$

where $\textsc{load}(\mathcal{T})$ is defined as seen in Equation (2.6), and

$$\nu \stackrel{\text{def}}{=} \max \{\ell : S_\ell(\pi) < S_m(\pi) - \lambda(\pi) \cdot \delta_{\max}(\mathcal{T})\} \ . \tag{2.9}$$

We refer back to this test in Chapter 4, since we extend Baruah & Goossens (2008)'s reasoning to the case when the uniform multiprocessor platform is not totally available at all times.

2.3 Scheduling approaches for mixed systems

In the previous section, we have seen related work focused on system models where one scheduler has to guarantee the HRT requirements of periodic or sporadic tasks. We now look into approaches which have been proposed in the literature to deal with systems that encompass tasks with mixed criticality, real-time requirements (HRT, SRT, non-real-time), activation models (periodic, sporadic, aperiodic), and/or origin (various development teams or providers).[4]

2.3.1 Resource reservation frameworks

In some domains of application, systems may have two coexisting types of workload:

1. a hard real-time (HRT) workload, whose temporal characteristics are known beforehand and which has strict deadlines; and

2. an aperiodic workload, whose temporal characteristics (e.g., execution requirement) may not be known beforehand, and which can have SRT or non-real-time requirements.

An efficient scheduling approach for such a system shall guarantee the fulfillment of the HRT tasks' deadlines and maximize the quality of service (QoS) of the SRT tasks (without compromising the former). There are several ways of characterizing the QoS of SRT tasks — for instance, average response time.

The simplest way of guaranteeing temporal isolation of HRT tasks is by relegating SRT and non-real-time tasks to the lowest priority, causing them to be scheduled only when there are no HRT tasks ready to execute. However, this leads to long response times for the SRT and non-real-time tasks; in the case of SRT tasks, this is incompatible with our goal of providing some QoS guarantee. This issue is addressed by employing *resource reservations*; the term was introduced by Mercer *et al.* (1993). Figure 2.2 shows an example resource reservation framework. Under this approach, fractions of the processor availability are reserved to a certain task or subset of tasks.

[4] Although we deal with models for mixed criticality *systems*, the paradigm of mixed criticality (or multi-criticality) *scheduling* proposed by Vestal (2007) is beyond the scope of this dissertation.

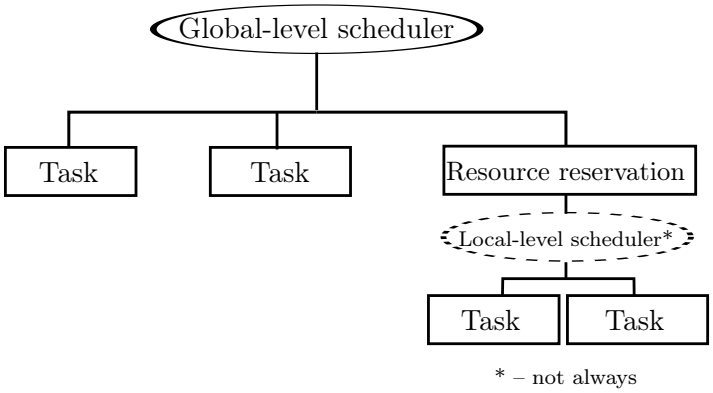


Figure 2.2: Resource reservation framework

The resource reservation then competes at the global level with the HRT tasks to given processing time to its subset of tasks.

Most resource reservations are implemented with a *scheduling server*, which is a special periodic task whose purpose is to service the jobs of its subset of tasks as soon as possible (Buttazzo, 1997). The existence of a proper scheduler associated with each server is optional; some implementations have the server insert the jobs of its subset of tasks into the queue of the global-level scheduler, and in some implementations the local-level scheduler is a simple First Come First Serve (FCFS) queue.[5] We now present an overview of the scheduling servers proposed by other authors, with only enough detail to integrate them into the context of this dissertation. For more specific details on the operation of these servers, the interested reader is directed to the corresponding papers, or to Chapters 5 and 6 of Buttazzo (1997)'s book.

2.3.1.1 Fixed task priority scheduling

Servers to work on fixed task priority algorithms include the Polling Server (Sha *et al.*, 1986), the Deferrable Server (Lehoczky *et al.*, 1987), and the Sporadic Server (Sprunt *et al.*, 1989). All these servers are characterized by a tuple (Π_s, Θ_s), where Π_s is the server's period and Θ_s is the server's budget; the ratio $\alpha_s = \frac{\Theta_s}{\Pi_s}$ is termed

[5]The literature usually employs the terms "local scheduler" and "global scheduler". To avoid confusion between the latter and global (as opposed to partitioned) multiprocessor scheduling, we opt for "local-level" and "global-level".

the server's *bandwidth*. The server is scheduled as a highest priority periodic task τ_i with period $T_i = \Pi_s$ and execution requirement $C_i = \Theta_s$.

2.3.1.2 Fixed job priority scheduling

To work with fixed job priority algorithms (such as EDF), Spuri & Buttazzo (1994) propose several server mechanisms, of which we highlight two. The Dynamic Sporadic Server is an extension of the Sporadic Server whereby the priority of the server is based on a deadline value, which is updated each time the server's budget is replenished. The Total Bandwidth Server is markedly different, in the following aspects:

- the Total Bandwidth Server is not characterized by a (Π_s, Θ_s) tuple, but only by a bandwidth α_s;
- instead of being scheduled to provide its budget to the jobs of SRT aperiodic tasks, the Total Bandwidth Server assigns deadlines to the latter and inserts them into the same queue as the jobs of HRT tasks, to be scheduled by EDF.

However, the Total Bandwidth Server is also subject to classical task schedulability analysis as the other servers. To assess the schedulability of the system composed of HRT tasks and one Total Bandwidth Server, the latter is considered as equivalent to a task with utilization $u_i = \alpha_s$. Abeni & Buttazzo (1998) propose the Constant Bandwidth Server (CBS), in a scheme with one server to handle each SRT task. This scheme guarantees temporal isolation between HRT and SRT tasks, and among each of the latter.

Baruah *et al.* (2002) present M-CBS, an extension of the CBS scheme for multiprocessors. For the purpose of this literature review, the main difference to be highlighted is that M-CBS employs, instead of EDF, a scheduling strategy based on the contemporarily proposed EDF-US (Srinivasan & Baruah, 2002): a subset of servers with highest utilization are considered *high-priority* servers (with their priority ties broken arbitrarily but consistently), whereas the remaining are considered *deadline-based* servers (and are scheduled according to EDF).

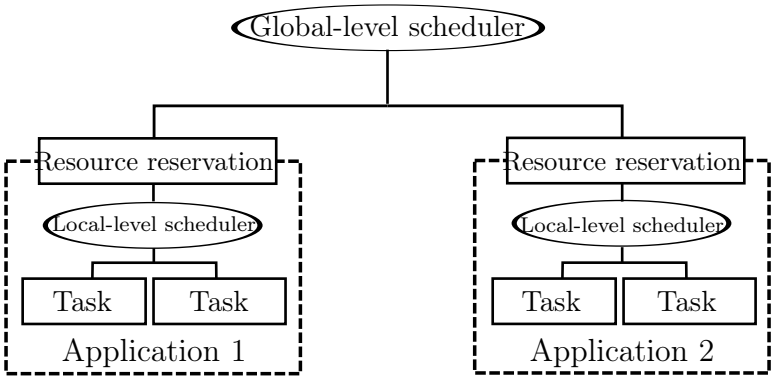


Figure 2.3: Hierarchical scheduling framework

2.3.2 Hierarchical scheduling frameworks (HSF)

Hierarchical scheduling frameworks (HSF) are an extension of resource reservation frameworks. Figure 2.3 illustrates an HSF. The main difference is that all tasks (and not only some of them) are divided into *applications* — subsets of tasks which are managed by a local-level scheduler. Each application is given a temporal guarantee through a resource reservation. Applications compete among themselves in a global-level scheduler.

2.3.2.1 Server-based HSFs

In what is considered a seminal paper regarding two-level hierarchical scheduling frameworks, Deng *et al.* (1997) propose a scheme in which each real-time application is handled by a Constant Utilization Server, with one more Constant Utilization Server to handle all non-real-time applications. Deng & Liu (1997) overcome some limitations, by establishing that applications employing preemptive scheduling should be handled by a Total Bandwidth Server (Spuri & Buttazzo, 1994) instead of a Constant Utilization Server. The authors provide sufficient tests, both for local-level schedulability of each application and global-level schedulability. The global-level schedulability test does, however, rely on knowledge of the internals of each application, including the tasks' relative deadlines.

Based on the premise that operating systems would not support the EDF scheduling algorithm very well, Kuo & Li (1999) extend the work of Deng & Liu (1997) by replacing EDF with RM as the global-level scheduler, which maintains and schedules

the servers, and using a Sporadic Server (Sprunt *et al.*, 1989) to service each application. Each application can use either RM or EDF as its local-level scheduler. Kuo *et al.* (2000) extend Kuo & Li (1999)'s framework for multiprocessors. Each server is, however, assumed to execute on only one processor. This hierarchical scheduling framework does, for this reason, only support parallelism between applications.

Lipari & Buttazzo (1999) propose the Bandwidth Sharing Server and a thereon based hierarchical scheduling framework. In this scheme, the global and local levels are not properly decoupled. Although each application is handled by a dedicated server, the latter assigns deadlines to the arriving jobs and inserts them into a global-level EDF queue, shared by all the servers (similarly to Spuri & Buttazzo (1994)'s Total Bandwidth Server). Lipari & Baruah (2000) extend the Bandwidth Sharing Server, proposing the BSS-I algorithm, and a thereon based hierarchical scheduling framework which improves this decoupling issue. Each application is handled by a server, which maintains a budget and a ready queue. The local-level scheduling policy, which dictates the ordering of the ready queue, may vary independently per application; the authors provide local-level schedulability tests for RM and EDF. At the global level, the servers as scheduled according to EDF, with the deadline of each server being that of the earliest-deadline job in its ready queue (which, depending on each local-level policy, may differ from the highest-priority job).

2.3.2.2 Time Partitioning

The ARINC 653 specification (AEEC, 1997) prescribes a two-level hierarchical scheduling framework to guarantee temporal isolation between the applications, each hosted in a logical containment unit — *partition*. On the first level, a cyclic global-level scheduler selects partitions according to a predefined partition scheduling table. When each partition is active according to such schedule, its tasks compete according to a local-level scheduler, which is specified to be preemptive and priority-based.

Audsley & Wellings (1996) proposed a general framework for schedulability analysis of applications running on ARINC 653-based TSP systems.[6] Their condition is

[6] Audsley & Wellings (1996)'s work precedes the first publication of the ARINC 653 specification (AEEC, 1997), which was at the time in draft. The considered aspects of the specification do, however, hold for the final specification.

based on response time analysis, and assumes the knowledge of (besides each task's timing characteristics as per the sporadic task model):

- the worst-case blocking time each task may be subject to by lower-priority tasks in the same partition;
- the worst-case delay each partition may experience as a consequence of the operating system entering a critical section;
- the worst-case delay between an event occurring and the corresponding task being placed into the scheduler's *ready* queue; and
- the maximum overhead that the operating system can cause within a certain time interval (as a consequence of interrupt service routines, context switching, etc.).

Some of these parameters cause the schedulability analysis of one application (partition) to depend on knowing characteristics of applications running in other partitions and the exact partition scheduling table. For this reason, Audsley & Wellings (1996)'s framework does not allow a usable independent analysis of each application, neither the computation of a partition schedule to guarantee schedulability.

Lee *et al.* (1998) proposed a sufficient test to assess the schedulability of an application (contained in a partition) composed of a task set scheduled according to some fixed task priority policy. Their test depends on the knowledge of the period according to which the partition is scheduled and of the fraction of this period during which the partition is scheduled — the bandwidth (cf. server bandwidth, Section 2.3.1). For the test to produce a conclusive positive result, two conditions must hold:

- the task set is schedulable — according to Lehoczky (1990)'s test — on a slower dedicated processor (whose speed corresponds to the partition's bandwidth); and
- the period according to which the partition is scheduled does not exceed a bound, calculated as a function of the bandwidth and the timing characteristics of the task set.

They also present an approach to derive the cyclic partition scheduling table, after having derived the period–bandwidth pairs which guarantee schedulability of each application. Compared to the approach of Audsley & Wellings (1996), Lee *et al.* (1998)'s work does allow independent verification and computation of each application's scheduling requirements; on the other hand, the aforementioned blocking and delay parameters are excluded.

2.4 Compositional analysis

The need for independent development and arbitrary number of levels are the main motivation and advantages of *compositional analysis*. Compositional analysis brings together, under the same theoretical banner, hybrid system models with resource reservation frameworks (Section 2.3.1) and HSFs (Section 2.3.2).

The compositional analysis paradigm consists of the possibility to **(i)** decompose a complex system into components; **(ii)** develop and analyze these components independently; and **(iii)** integrate (compose) the components in a way which preserves two main principles: *compositionality* and *composability* (Bini & Lipari, 2011). Both these principles are tied to the notion of formally analyzing a system using *abstractions* of its components. Compositionality is the possibility to analyze the system resulting from the composition of components by looking into the composition of components' abstractions. Composability, which is a condition for compositionality to hold, is observed if the composition of components does not void the validity of each component's abstraction as such. For a more formal and general definition of compositionality and composability, the reader is referred to the work of Julliand *et al.* (2007). Let us see these principles applied to real-time scheduling for a better understanding.

Within compositional analysis as applied to real-time scheduling, a component comprises a workload (tasks and/or subcomponents) and a scheduler, and is abstracted by a *resource interface*. Figure 2.4 illustrates such a *compositional scheduling framework*; component $\mathcal{C}_0$ is the *root component*, with subcomponents $\mathcal{C}_1$ and $\mathcal{C}_2$. Component $\mathcal{C}_0$ sees the interfaces $\mathcal{R}_1$ and $\mathcal{R}_2$, which abstract to $\mathcal{C}_0$ the resource demand by each of the subcomponents, hiding how it is composed (tasks and/or subsubcomponents and their characteristics). For each subcomponent $\mathcal{C}_1$ and $\mathcal{C}_2$, its

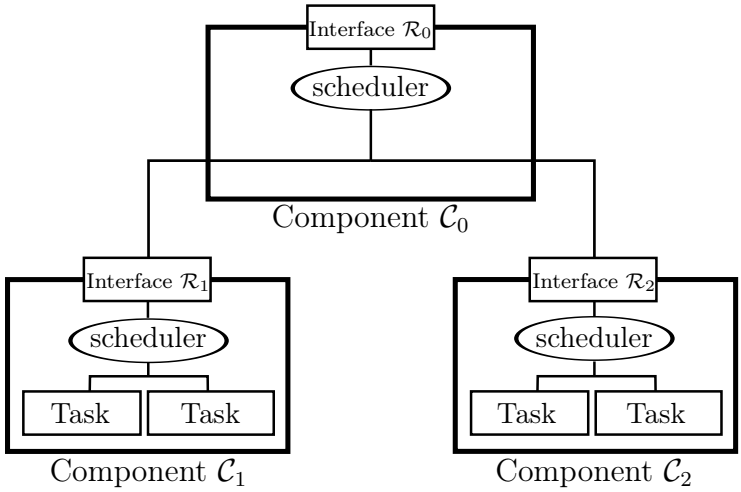


Figure 2.4: Compositional scheduling framework

interface abstracts the resource supply it is provided, hiding who provides it and how (physical platform vs. parent component). With respect to the two central notions of compositional analysis,

- compositionality lies in the fact that we can derive $\mathcal{R}_0$ solely with the knowledge of $\mathcal{R}_1$, $\mathcal{R}_2$, and how the components are put together — composed; and
- composability lies in the fact that composition — in this case, the way component $\mathcal{C}_0$'s scheduler handles the subcomponents — does not void the validity of $\mathcal{R}_1$ and $\mathcal{R}_2$ as abstractions of components $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively.

Compositional analysis comprises three main aspects (Shin & Lee, 2007).

1. *Local-level schedulability analysis* — analyzing the schedulability of a component $\mathcal{C}_i$'s workload upon its scheduler and the resource supply it receives (as abstracted by interface $\mathcal{R}_i$).
2. *Component abstraction* — obtaining a component $\mathcal{C}_i$'s resource interface, $\mathcal{R}_i$, so that is guarantees schedulability of $\mathcal{C}_i$'s workload.
3. *Interface composition* — transforming the set of interfaces abstracting the real-time requirements of individual subcomponents into an interface abstracting the requirement of scheduling these subcomponent together according to a

given *intercomponent scheduling* strategy. Formally, given a set of interfaces $\mathcal{R}_1$, $\mathcal{R}_2$, ..., derive an interface $\mathcal{R}_0$ which is compatible with $\mathcal{R}_1 \| \mathcal{R}_2 \| \ldots$ (where the composition operator, $\|$, models the intercomponent scheduling strategy). Observing composability and compositionality, we can say that $\mathcal{R}_0$ is compatible with $\mathcal{C}_0$ (i.e., $\mathcal{C}_1 \| \mathcal{C}_2 \| \ldots$).

In this section, we survey, under the light of these three aspects, the work that has been done in compositional analysis. First, let us introduce some additional notions and definitions which are common to all or most approaches thereto.

2.4.1 Common definitions

Mok *et al.* (2001) introduce, to support their static partition and bounded-delay resource models (which we review later on), some concepts which are common to subsequent works in the field. We now present those concepts, updating their terminology and notation to reflect present widespread use. For this purpose, we first introduce a function $\text{SUPPLY}(\mathcal{R}, t_1, t_2)$, as expressing the effective supply that is provided according to interface $\mathcal{R}$ over the time interval $[t_1, t_2[$. For some resource models, as we will see ahead, the exact knowledge of the effective supply is not feasible.

The *supply bound function* $\text{SBF}(\mathcal{R}, t)$ gives the minimum resource supply guaranteed by interface $\mathcal{R}$ over any interval of length t. Its main property is, consequently, that

$$\text{SBF}(\mathcal{R}, t) \leq \min_{o \geq 0} \text{SUPPLY}(\mathcal{R}, o, o + t) \ . \tag{2.10}$$

Since the supply bound function may be a step function (rather than a linear one), some authors also make use of a *linear lower bound* on the supply bound function. The characterizing property of this function is that

$$\text{LSBF}(\mathcal{R}, t) \leq \text{SBF}(\mathcal{R}, t), \text{ for all } t > 0 \ .$$

The *resource bandwidth* α (cf. server bandwidth, Section 2.3.1) represents the

fraction of resource that is provided over time. Formally, it is defined as follows:

$$\alpha \stackrel{\text{def}}{=} \lim_{t \to +\infty} \frac{\text{SUPPLY}(\mathcal{R}, 0, t)}{t} \quad . \tag{2.11}$$

The resource models we consider, being periodic in nature (at least to some extent), allow computing bandwidth in a way which is significantly less complex than the formal definition.

2.4.2 Uniprocessor

2.4.2.1 Static partition resource model

Mok *et al.* (2001) introduce a *static partition* resource model as an abstraction of a resource which is made available to a group of tasks only during specific intervals. This models very closely the resource supply mechanism present in ARINC 653 TSP systems (Section 2.3.2.2). A resource partition $\mathcal{P}$ is defined as a tuple (ω, Π), with $\omega = \{(S_i, E_i)\}_{i=1}^{N}$ and Π being the period of the partition. Each pair (S_i, E_i) represents a time window (respectively, its start and end) in terms of offset relative to the beginning of each period of length Π. Formally, this means that the processor is available to the task set executing on this partition only during the time intervals:

$$[k \cdot \Pi + S_i; k \cdot \Pi + E_i[, \text{ for all } i \in [1..N], \text{ and for all } k \in \mathbb{N}_0 \quad .$$

Since the allocation defined in ω repeats over a period of length Π, the bandwidth is calculated simply as

$$\alpha = \frac{\sum_{(S_i, E_i) \in \omega} (E_i - S_i)}{\Pi} \quad .$$

The computation of $\text{SUPPLY}(\mathcal{P}, t_1, t_2)$ is straightforward, and the definition of a supply bound function fulfilling the formal definition in Equation (2.10) follows directly: $\text{SBF}(\mathcal{P}, t) \stackrel{\text{def}}{=} \min_{o \geq 0} \text{SUPPLY}(\mathcal{R}, o, o + t)$ for all $t > 0$. The authors do not, however, present a means to compute it.

Local-level schedulability analysis Mok *et al.* (2001) show that the optimality of EDF on uniprocessor also holds for the static partition resource model; this means

that any task set that is feasible on a static partition $\mathcal{P}$ is also EDF-schedulable on $\mathcal{P}$. From there, an exact EDF-schedulability test for implicit-deadline periodic task sets on a static partition is derived. Periodic task set $\mathcal{T}$ is schedulable with EDF on a static partition $\mathcal{P}$ if and only if

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, t) \leq \text{SBF}(\mathcal{P}, t), \text{ for all } t > 0 \ .$$

Component abstraction / Interface composition Mok *et al.* (2001) point out that the static partition resource model is more adequate for scenarios where the resource (the processor) is already partitioned. For this reason, the authors do not approach component abstraction (and, consequently, interface composition) and propose a more appropriate model: the bounded-delay resource model.

2.4.2.2 Bounded-delay resource model

The *bounded-delay resource* model (Mok *et al.*, 2001) consists of a tuple $\Phi \stackrel{\text{def}}{=} (\alpha, \Delta)$, where α is the bandwidth (as per Equation (2.11)) and Δ is the delay bound — the maximum delay in the partition's resource supply, in relation to what it would receive when executing on a fully available processor with speed α (instead of a partially available processor of speed 1); the authors call this slower processor scenario the *normalized execution* of Φ. This is similar to the reasoning done by Lee *et al.* (1998), of confronting execution on a shared resource and execution on a slower resource.

Shin & Lee (2004) build upon Mok *et al.* (2001)'s work, introducing a compositional scheduling framework based on the bounded-delay resource model. The definition of the supply bound function for the bounded-delay resource model is very simple, due to the specific definition of Δ:

$$\text{SBF}(\Phi, t) \stackrel{\text{def}}{=} \begin{cases} \alpha(t - \Delta) & \text{if } t \geq \Delta, \\ 0 & \text{otherwise.} \end{cases} \tag{2.12}$$

Local-level schedulability analysis In (Mok *et al.*, 2001), a sufficient schedulability condition for task set $\mathcal{T}$ on $\Phi = (\alpha, \Delta)$ stems from this notion of normalized execution. If, when $\mathcal{T}$ is scheduled on the normalized execution of Φ, all jobs are

guaranteed to finish Δ time units before their deadlines, then $\mathcal{T}$ is schedulable on Φ.

In turn, based on the definition of a supply bound function, Shin & Lee (2004) provide an exact schedulability test for a periodic task set $\mathcal{T}$ scheduled with EDF upon Φ (cf. the test in Equation (2.5) for EDF upon a dedicated processor):

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, t) \leq \text{SBF}(\Phi, t), \text{ for all } 0 < t \leq \operatorname*{lcm}_{\tau_i \in \mathcal{T}}\{T_i\} + \max_{\tau_i \in \mathcal{T}}\{D_i\} \ .$$

The authors also provide a sufficient condition based on a utilization bound on the task set:

$$u_{\text{sum}}(\mathcal{T}) \leq \alpha \cdot \left(1 - \frac{\Delta}{\min_{\tau_i \in \mathcal{T}}\{T_i\}}\right) \ .$$

Component abstraction Shin & Lee (2004) approach the problem of component abstraction as being that of searching for values of α and Δ which guarantee schedulability of the task set. To search for optimal values, a search interval $[\Delta_{\min}, \Delta_{\max}]$ can be established; the objective is to find the $\Delta \in [\Delta_{\min}, \Delta_{\max}]$ for which schedulability is guaranteed with the least possible α.

Interface composition Feng & Mok (2002) propose a hierarchical real-time virtual resource model extending resource partitioning to potential multiple levels.

2.4.2.3 Periodic resource model

Shin & Lee (2003, 2008) propose the *periodic resource model* $\Gamma \stackrel{\text{def}}{=} (\Pi, \Theta)$ to characterize the periodic behavior of a partitioned resource; this model describes a partitioned resource which guarantees Θ units of execution over every Π time units period. The bandwidth of such an interface is simply $\alpha = \Theta/\Pi$. For the periodic resource model, the supply bound function is defined as

$$\text{SBF}(\Gamma, t) \stackrel{\text{def}}{=} \begin{cases} t - (k+1) \cdot (\Pi - \Theta) & \text{if } t \in [(k+1) \cdot \Pi - 2 \cdot \Theta, (k+1) \cdot \Pi - \Theta] \\ (k-1) \cdot \Theta & \text{otherwise} \end{cases}$$

where $k = \max\left(\left\lceil \frac{t-(\Pi-\Theta)}{\Pi} \right\rceil, 1\right)$.

Local-level schedulability analysis The authors prove that implicit-deadline periodic task set $\mathcal{T}$ is EDF-schedulable upon Γ if

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, t) \leq \text{SBF}(\Gamma, t), \text{ for all } 0 < t \leq \operatorname{lcm}_{\tau_i \in \mathcal{T}} \{T_i\} \ . \tag{2.13}$$

Component abstraction As with the bounded-delay resource model, component abstraction can be seen, on a first approach, as a search for optimal values. In the case of the periodic resource model, Shin & Lee (2003) propose an approach where Π is given and the minimum Θ which guarantees schedulability (according to Equation (2.13)) is searched for. To render this process more tractable, the authors rely on a linear lower bound on the supply bound function. Such linear bound is defined as follows for the periodic resource model:

$$\text{LSBF}(\Gamma, t) \stackrel{\text{def}}{=} \frac{\Theta}{\Pi} (t - 2 \cdot \Pi + 2 \cdot \Theta) \ .$$

Using this function, a non-optimal value for Θ which guarantees schedulability can be found

Interface composition Given the proximity between the periodic resource model and the periodic task model, the derivation of an interface $\Gamma_0 = (\Pi_0, \Theta_0)$ expressing the overall requirement of scheduling the subcomponents abstracted as $\Gamma_1, \Gamma_2, \ldots$, is essentially identical to applying the principles of component abstraction enunciated in the previous paragraph.

Further extensions Easwaran *et al.* (2007) extend Shin & Lee (2003)'s periodic resource model, proposing the *explicit-deadline* periodic (EDP) resource model. Easwaran *et al.* (2009a) extend and improve the periodic resource model and the explicit-deadline periodic resource model with techniques to take into account ARINC 653-specific issues, such as process communication (modeled as offset, jitter and constrained deadlines) and preemption/blocking overhead.

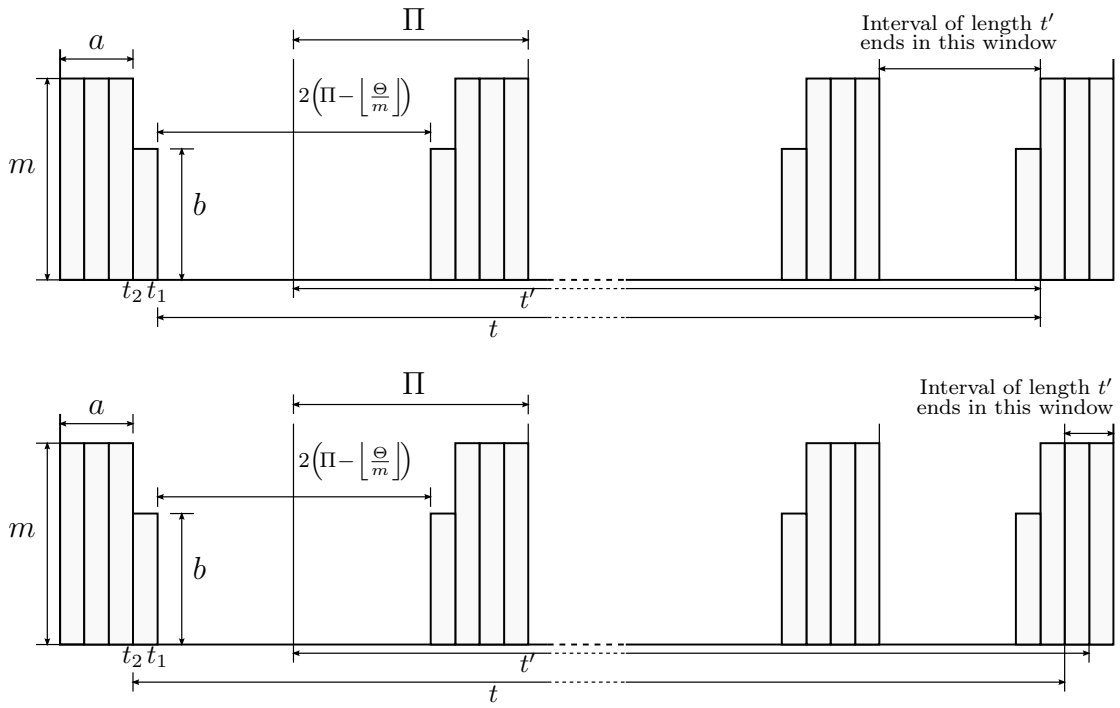


Figure 2.5: Minimum supply schedule for an MPR interface $\mu = (\Pi, \Theta, m)$ — adapted from (Easwaran *et al.*, 2009b).

2.4.3 Identical multiprocessor

2.4.3.1 Multiprocessor periodic resource model

Shin *et al.* (2008) propose the *multiprocessor periodic resource* (MPR) model. An MPR interface $\mu \stackrel{\text{def}}{=} (\Pi, \Theta, m)$ abstracts the provision of Θ processing units over every period with Π time units length over a virtual platform consisting of m identical unit-capacity processors. Easwaran *et al.* (2009b) correct and improve Shin *et al.* (2008)'s results. From this point on, all considerations regarding the MPR model rely on the improved results.

The supply bound function for the MPR is based on the scenario which maximizes the length of the longest time interval without supply. Such scenario is pictured in Figure 2.5, where $a \stackrel{\text{def}}{=} \left\lfloor \frac{\Theta}{m} \right\rfloor$ and $b \stackrel{\text{def}}{=} \Theta - m \cdot a$. Depending on the value of t, the interval of length t which yields the least supply may be of one of two types — each graph in the figure portrays one of these types. From this scenario,

Easwaran *et al.* (2009b) derive the supply bound function for the MPR:

$$\text{SBF}(\mu, t) \stackrel{\text{def}}{=} \begin{cases} 0, & r < 0 \\ w, & r \geq 0 \wedge x \in [1, y] \\ w - (m - b), & r \geq 0 \wedge x \notin [1, y] \end{cases} \tag{2.14}$$

where

$$\begin{aligned} w &\stackrel{\text{def}}{=} \left\lfloor \frac{r}{\Pi} \right\rfloor \cdot \Theta + \max\{0, m \cdot x - (m \cdot \Pi - \Theta)\} \ , \\ r &\stackrel{\text{def}}{=} t - \left(\Pi - \left\lceil \frac{\Theta}{m} \right\rceil\right) \ , \\ x &\stackrel{\text{def}}{=} \left(r - \Pi \cdot \left\lfloor \frac{r}{\Pi} \right\rfloor\right) \ , \\ y &\stackrel{\text{def}}{=} \Pi - \left\lfloor \frac{\Theta}{m} \right\rfloor \ , \\ \text{and } b &\stackrel{\text{def}}{=} \Theta - \left\lfloor \frac{\Theta}{m} \right\rfloor \cdot m \ . \end{aligned}$$

Local-level schedulability analysis For the uniprocessor case, the base schedulability tests rely solely on the condition that the upper bound on the demand by the task set does not exceed the lower bound on the supply by the resource interface. For multiprocessors, all analysis must take into account the limitation that a job may execute on only one processor at a time (even though more processors may be available and idle). The test presented by Easwaran *et al.* (2009b) is inspired by the works of Bertogna *et al.* (2005) and Baruah (2007) (who, in turn, use Baker (2003)'s framework, described in Section 2.2.3.1). A sporadic task set $\mathcal{T}$ is schedulable by gEDF on a resource provided according to an interface $\mu \stackrel{\text{def}}{=} (\Pi, \Theta, m)$ if, for all tasks $\tau_k \in \mathcal{T}$ and for all $A_k \geq 0$,

$$m \cdot C_k + \sum_{\tau_i \in \mathcal{T}} \hat{I}_i + \sum_{m-1 \text{ largest}} (\bar{I}_i - \hat{I}_i) \leq \text{SBF}(\mu, A_k + D_k) \ , \tag{2.15}$$

where $\hat{I}_i$ is defined as in Equation (2.7),

$$\bar{I}_i \stackrel{\text{def}}{=} \begin{cases} \min\{\text{DBF}(\tau_i, A_k + D_k) + \text{CI}(\tau_i, A_k + D_k) - C_k, A_k\} & \text{if } i = k \\ \min\{\text{DBF}(\tau_i, A_k + D_k) + \text{CI}(\tau_i, A_k + D_k), A_k + D_k - C_k\} & \text{otherwise} \end{cases} \text{, and}$$

$$\text{CI}(\tau_i, t) \stackrel{\text{def}}{=} \min\left\{C_i, \max\left\{0, t - \left\lceil \frac{t + T_i - D_i}{T_i} \right\rceil \cdot T_i \right\}\right\} \quad .$$

This test generalizes Baruah (2007)'s test (Section 2.2.3.1).

The authors also provide a strict upper bound on the values of A_k which have to be verified, by proving that if the schedulability condition is violated for some value of A_k, then it is also violated for a value of A_k such that

$$A_k < \frac{\sum_{m-1 \text{ largest}} C_i + m \cdot C_k - D_k \cdot \left(\frac{\Theta}{\Pi} - u_{\text{sum}}(\mathcal{T})\right) + U + B}{\frac{\Theta}{\Pi} - u_{\text{sum}}(\mathcal{T})}$$

where

$$U \stackrel{\text{def}}{=} \sum_{\tau_i \in \mathcal{T}} (T_i - D_i) \cdot \frac{C_i}{T_i} \quad , \qquad \text{and} \qquad B \stackrel{\text{def}}{=} \frac{\Theta}{\Pi} \cdot \left(2 + 2 \cdot \left(\Pi - \frac{\Theta}{m}\right)\right) \quad .$$

Component abstraction As in previous works, the process of deriving the interface which guarantees schedulability is facilitated through the use of a linear lower bound on the supply bound function; for interfaces expressed with the MPR model, it is given by

$$\text{LSBF}(\mu, t) \stackrel{\text{def}}{=} \frac{\Theta}{\Pi}\left(t - \left(2 \cdot \left(\Pi - \frac{\Theta}{m}\right) + 2\right)\right) \quad .$$

The component abstraction problem now includes obtaining both Θ and m, given Π, while minimizing the interface bandwidth $\alpha = \Theta/\Pi$. The approach proposed by Easwaran *et al.* (2009b) consists of performing a binary search for m in the interval $[\lceil u_{\text{sum}}(\mathcal{T}) \rceil, \frac{\sum_{\tau_i \in \mathcal{T}} C_i}{\min_{\tau_i \in \mathcal{T}}\{D_i - C_i\}} + n]$. The selected interface will have m as the smallest number of processors for which schedulability of the component is guaranteed with $\Theta \leq m \cdot \Pi$ — considering the condition in Equation (2.15) with $\text{SBF}(\mu, A_k + D_k)$ replaced with $\text{LSBF}(\mu, A_k + D_k)$).

Interface composition The derivation of an interface $\mu_0 = (\Pi_0, \Theta_0, m_0)$ expressing the overall requirement of scheduling the q subcomponents abstracted as μ_1, μ_2, ..., μ_q, is essentially identical to applying the principles of component abstraction enunciated in the previous paragraph to a task set obtained from the transformation of those MPR interfaces. Each MPR interface $\mu_p = (\Pi_p, \Theta_p, m_p)$ (with $p \leq q$)

is transformed into a task set $\mathcal{T}^{(\mu_p)} \stackrel{\text{def}}{=} \left\{\tau_i^{(\mu_p)}\right\}_{i=1}^{m}$ with total utilization Θ_p. With $b = \Theta_p - m_p \cdot \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor$ and $k = \lfloor b \rfloor$, each *interface task* $\tau_i^{(\mu_p)} \stackrel{\text{def}}{=} \left(T_i^{(\mu_p)}, C_i^{(\mu_p)}, D_i^{(\mu_p)}\right)$ is derived as follows:

- if $i \leq k_i$, $\tau_i^{(\mu_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor + 1, \Pi_p\right)$;
- if $i = k_i + 1$, $\tau_i^{(\mu_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor + b - k \cdot \left\lfloor \frac{b}{k} \right\rfloor, \Pi_p\right)$;
- otherwise, $\tau_i^{(\mu_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor, \Pi_p\right)$.

The interface μ_0 is derived so as to guarantee that the task set $\bigcup_{p=1}^{q} \mathcal{T}^{(\mu_p)}$ is schedulable; more specifically, m_0 will be the minimum number of processors found to allow deriving a feasible interface. By definition, m_0 lies within the interval $\left[\min_{p=1}^{q} m_p, \sum_{p=1}^{q} m_p\right]$.

2.4.3.2 Multi Supply Function

Bini *et al.* (2009b) point out that the advantage of the simplicity of the MPR approach, of having a common period and an aggregate contribution for all processors, has a pessimistic component abstraction as a drawback. The authors then propose the Multi Supply Function abstraction, which consists of a set of m supply bound functions $\{\text{SBF}(\mathcal{R}_k, t)\}_{k=1}^{m}$, where each function corresponds to the minimum amount of availability of each virtual processor — abstracted with an interface $\mathcal{R}_k$ which can be expressed using uniprocessor resource models (thus employing the respective supply bound functions). Based on this possibility, they propose the Multi-(α, Δ) abstraction, in which each virtual processor is expressed using (Mok *et al.*, 2001)'s bounded-delay resource model. It renders a special case of a Multi Supply Function which takes advantage of the simple definition of the supply bound function for the bounded-delay resource model (see Equation (2.12)).

However different the virtual processors may be, the work of Bini *et al.* (2009b) assumes that all of them correspond to partial availability of identical unit-capacity physical processors.

Local-level schedulability analysis Bini *et al.* (2009b) provide sufficient schedulability tests for constrained-deadline sporadic task sets, with variations to accommodate local-level scheduling (over a Multi Supply Function) according to **(i)** gEDF; **(ii)** any global fixed task priority algorithm; and **(iii)** any work-conserving algorithm.

Component abstraction / Interface composition Bini *et al.* (2009b) do not provide component abstraction neither interface composition results for the Multi Supply Function abstraction. In a related work, Buttazzo *et al.* (2010, 2011) propose a method for allocating a parallel real-time application, described as a set of tasks with time and precedence constraints, on a multicore platform. To achieve modularity and simplify portability of applications on different multicore platforms, the authors use the Multi Supply Function.

2.4.3.3 Parallel Supply Function

Bini *et al.* (2009a) propose the Parallel Supply Function to abstract resource provision to a component. Their abstraction consists of m functions $\text{PSF}_k(t)$, each one expressing the minimum amount of resource provided to the component, *with parallelism at most* k, within every time interval of length t. This allows lifting the assumption (used in the MPR and Parallel Supply Function models) that periods (and hence resource provisions) are synchronized among all processors.

Local-level schedulability analysis For the Parallel Supply Function, Bini *et al.* (2009a) present two sufficient schedulability tests considering gEDF as the local-level scheduler. The first employs the notion of *forced-forward demand bound function* proposed by Baruah *et al.* (2009). The second is, like Easwaran *et al.* (2009b)'s test for the MPR model, inspired by Bertogna *et al.* (2005).

Component abstraction / Interface composition Bini *et al.* (2009a) do not approach the issue of component abstraction (and, consequently, interface composition), which in this case corresponds to deriving the m level-j supply functions to guarantee schedulability. Lipari & Bini (2010) demonstrate that component abstraction with the Parallel Supply Function model is too complex.

2.4.3.4 Bounded-Delay Multipartition

(Lipari & Bini, 2010) propose the Bounded-Delay Multipartition, a new interface model that allows the designer to balance resource usage versus flexibility in selecting the virtual platform parameters. It borrows, on the one hand, from the bounded-delay resource model and, on the other hand, from the Parallel Supply Function. A Bounded-Delay Multipartition interface $\mathcal{I} \stackrel{\text{def}}{=} (m, \Delta, \{\beta_k\}_{k=1}^{m})$, with

- $\Delta \geq 0$;
- $0 \leq \beta_k - \beta_{k-1} \leq 1$, for all $1 \leq k \leq m$; and
- $\beta_k - \beta_{k-1} \leq \beta_{k+1} - \beta_k$, for all $1 \leq k \leq m$;

if, for all resource allocations compatible with it, the Parallel Supply Functions are

$$\text{PSF}_k(t) \geq \beta_k \cdot \max(0, t - \Delta), \text{ for all } 1 \leq k \leq m \ .$$

Each β_k expresses the cumulative bandwidth provided with parallelism at most k, whereas the Δ parameter is the interface delay (for the bounded-delay multipartition, this is length of the longest interval with no guaranteed resource supply).

Compared to the MPR model, the authors lift the assumption that the periods are synchronized across processors. Under this scenario, it is not possible to derive a worst-case resource allocation for the MPR; for the Bounded-Delay Multipartition the authors are able to prove that the worst-case resource allocation for a given $\mathcal{I} = (m, \Delta, \{\beta_k\}_{k=1}^{m})$ is the set of m bounded-delay resource interfaces $\{(\alpha_k, \Delta)\}_{k=1}^{m}$, with $\alpha_k = \beta_k - \beta_{k-1}$, for all $1 \leq k \leq m$.

Local-level schedulability analysis Since the definition of a Bounded-Delay Multipartition relies on properties attached to the Parallel Supply Function, schedulability analysis can be performed with the tests proposed by Bini *et al.* (2009a) for the latter. (Lipari & Bini, 2010) reformulate one of those tests, presenting schedulability conditions for both gEDF and global fixed task priority local-level schedulers.

Component abstraction Lipari & Bini (2010) present a definition of the schedulability region of a real-time application. The specific interface (namely, its Δ parameter) can be chosen so as to constitute a compromise between schedulability and overhead.

Interface composition To bind the virtual resource allocations on an identical multiprocessor physical platform. Lipari & Bini (2010) present an algorithm called Fluid Best-Fit. The algorithm can be summarized as consisting of a Best Fit partitioning (Section 2.2.2) of the worst-case allocation of each Bounded-Delay Multipartition interface, followed by bandwidth modifications.

2.5 Technological support to TSP

In this section we survey the existing technical support to TSP systems, with respect to both operating system support and tools to support the verification of temporal properties in the integration phase of TSP system development. Due to its tighter connection to industry practice, contains a significant amount of references to websites, whose locations are provided in footnotes and have been verified to be correct and available at the time of writing.[7]

2.5.1 Operating system support

The purpose of this subsection, where we survey the current state of practice, is twofold. On the one hand, the existing solutions have varying degrees of support to multicore and are, as such, candidates to improvement as per the architectural considerations done in Section 3.2. On the other hand, even without such improvements, systems based on the surveyed solutions can be represented using the system model we propose in Section 3.3, and are potential targets for the application of the principles and tools presented in this dissertation (Chapters 4 and 5, respectively).

[7] Last verification on July 31, 2013.

2.5.1.1 PikeOS

PikeOS[8] is a commercial microkernel-based real-time operating system (RTOS), developed by SYSGO. While initially implementing the L4 version 2.0 kernel application programming interface, the developers of PikeOS have identified issues with L4 version 2.0 and, while maintaining the base principles, have deeply modified the programming interface. Kaiser & Wagner (2007a) expose how PikeOS evolved, describing the aforementioned issues and how they were approached. In this survey, we will only focus on the characteristics which differentiate PikeOS from other TSP operating systems in terms of scheduling.

The PikeOS microkernel provides a more complex partition scheduling mechanism than the one defined in ARINC 653 (AEEC, 1997). At each instant, besides one of the application time partitions (with a task set $\mathcal{T}_i$), an additional time partition is always active (the background partition, running a task set $\mathcal{T}_0$). The scheduler will always select to be executed the highest priority task in $\mathcal{T}_i \cup \mathcal{T}_0$. In the scope of this scheduling scheme, task priority definition plays a major role in offering flexibility to handle the coexistence of time-triggered and event-triggered tasks (Kaiser & Wagner, 2007b).

SYSGO introduced, in PikeOS 3.1, the first DO-178B-compliant implementation of multicore support (McConnel, 2010). The current version, PikeOS 3.3, includes improvements and increased support to multicore, resulting of SYSGO's participation in the European project RECOMP (RECOMP, 2011b; SYSGO, 2013).

2.5.1.2 DEOS

DEOS[9] is a commercial RTOS for safety-critical applications in the military and aerospace domains. Originating in technology developed by Honeywell (under the acronym DEOS — Digital Engine Operating System), it is (since 2008) provided by DDC-I Inc. (as DEOS — DDC-I Embedded Operating System).

DEOS provides space partitioning at the process level, and time partitioning at the thread level. In this regard, it departs greatly from the notion of TSP associated

[8] http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/
[9] http://www.ddci.com/products_deos.php

with other solutions and approaches, including the IMA and ARINC 653 specifications (Binns, 2001a). Time partitioning is enforced by scheduling processes and threads according to a Rate Monotonic policy, enhanced with a patented (Binns, 2001b) *slack scheduling* extension. DEOS supports creating and deleting threads in runtime, with budget respectively being received from and returned to the main thread. Consequently, offline verification is intractable (requiring the simulation of all the expectable combinations of processes, threads, and budgets), and calculations such as schedulability analysis have to be done in execution time (Penix *et al.*, 2000).

We could not assess the exact extent and characteristics of DEOS's support to multicore processors, especially regarding scheduling. DDC-I (2012)'s DEOS fact sheet makes no mention thereto, and we could only find superficial mentions to DDC-I's view on multicore support in web articles (Howard, 2011; King, 2011, 2013). In an interview to Howard (2011), Tim King at DDC-I talks about the work being developed at the time with respect to adding multicore support to DEOS, namely the approach of partitioning caches that are shared by processor cores. King (2011) defends that reclaiming unused slack improves the performance of multicore processors, but does not describe how DEOS's slack scheduling operates over multiple processor cores. King (2013) reports experiments carried at DDC-I with respect to cache partitioning. Although tackling the multicore-related issue of cache partitioning, the experiments are based on running Deos on only one processor core, which no description of how Deos supports multicore.

2.5.1.3 XtratuM

Developed at the Polytechnical University of Valencia under a contract from CNES, XtratuM[10] first started as a nanokernel satisfying the requisites of a hard real-time system (Masmano *et al.*, 2005). XtratuM was since redesigned, and is now a hypervisor aiming to fulfill safety-critical real-time requirements. It features TSP capabilities, supported on the fixed cyclic scheduling of partitions (time partitioning) and on having partitions run in user mode without sharing any memory (space partitioning). Masmano *et al.* (2009) describe the core functionality of XtratuM

[10] http://www.xtratum.org/

and evaluate the overhead introduced by mechanisms such as context switch when commuting between partitions, resorting to a prototype implementation on a SPARC LEON2 processor platform without a memory management unit (MMU); the lack of an MMU did not allow the implementation of space partitioning capabilities in full.

Masmano *et al.* (2010) port XtratuM to a SPARC LEON3 processor platform with an MMU, thus allowing to implement full space partitioning. Albeit using ARINC 653 concepts and models, XtratuM does not follow the ARINC 653 (AEEC, 1997) interface.

Contemporarily to the work presented in this dissertation, the XtratuM architecture was included and extended in the MultiPARTES (Multi-cores Partitioning for Trusted Embedded Systems) project.[11] The MultiPARTES approach to multicore is similar to the one adopted in this dissertation (Craveiro *et al.*, 2011), taking advantage of multiprocessor for parallelism at both the local (intrapartition) and global (interpartition) level (Coronel & Crespo, 2012). In the scope of MultiPARTES, de la Puente *et al.* (2012) draw some considerations on scheduling partitioned systems on multicore platforms. The authors acknowledge the work of Shin *et al.* (2008) on compositional analysis on identical multiprocessors, but fail to acknowledge more recent works (Bini *et al.*, 2009a,b; Lipari & Bini, 2010). Non-identical multiprocessors (both uniform and heterogeneous) are mentioned, but not specifically dealt with.

2.5.1.4 VxWorks 653

VxWorks 653 Platform[12] is Wind River's commercial solution for ARINC 653/IMA systems (Wind River, 2010). It includes the VxWorks 653 operating system, which implements the complete Application Executive (APEX) interface specified in ARINC 653 (AEEC, 1997), including the optional service for multiple module schedules defined in Part 2 (AEEC, 2007). Time and space partitioning is supported by a two-level scheme to schedule MMU-protected partitions. Each partition runs an instance of the vThreads partition operating system. Applications can be developed to run on vThreads using the ARINC 653 APEX interface, POSIX, C, C++, Ada, or Java.

[11] http://www.multipartes.eu/.
[12] http://www.windriver.com/products/platforms/safety_critical_arinc_653/

In addition to the cyclic partition scheduling from ARINC 653, VxWorks 653 provides the *ARINC Plus Priority-Preempting Scheduling* (APPS) hybrid mode. In APPS, the system can change to priority-preemptive scheduling. Examples of situations when this might occur include the detection of idle time in a time slice attributed to a partition, or an idle partition time slice. Under priority-preemptive scheduling, the non-idle partition with the highest priority is scheduled to run during the detected idle time of the current time slice.

Wind River (2010)'s product note for the VxWorks 653 Platform makes no mention to multicore support. The only mention we could find is a statement by Joe Wlad from Wind River, in an interview to Howard (2011), that "VxWorks 653 ... supports multicore architectures".

2.5.2 Scheduling analysis and simulation tools

In this section, we restrict our attention to tools which provide some degree of support to hierarchical scheduling or TSP systems. With the exception of Grasp (which we describe first), these tools are presented here as solutions upon which this dissertation (namely Chapter 5) constitutes an improvement.

2.5.2.1 Grasp

Grasp[13] is a trace visualization tool set. It supports the visualization of multiprocessor hierarchical scheduling traces. Traces are recorded from the target system (by the Grasp Recorder or any other appropriate means) into Grasp's own script-like format, and displayed graphically by the Grasp Player (Holenderski *et al.*, 2013).

The Grasp tool set does not support simulation and supports only a two-level hierarchy. However, the Grasp Player reads traces in a simple text-based format which can be recorded by other tools. In Chapter 5, we take advantage of this feature, and use Grasp to visualize simulation results.

[13] http://www.win.tue.nl/san/grasp/

2.5.2.2 Cheddar

Cheddar[14] provides a set of scheduling algorithms and policies on which it is capable of performing feasibility tests and/or scheduling simulations for either uniprocessor or multiprocessor environments (Singhoff *et al.*, 2004). In its latest versions, Cheddar already supports schedulability analysis of ARINC 653-like time- and space-partitioned (TSP) systems to some extent (Singhoff *et al.*, 2009). However, Cheddar presents some limitations in its current support thereto. A partition scheduling table (PST) is defined as an array of durations; besides presenting less usability, the current implementation limits the PST to having only one time window per partition per major time frame (Craveiro *et al.*, 2011).

2.5.2.3 CARTS

CARTS[15] is an open-source compositional analysis tool for real-time systems, which automatically generates the components' resource interfaces; it does not perform simulation (Phan *et al.*, 2011). CARTS relies strongly upon some of the authors' theoretical results, namely Shin & Lee (2003)'s periodic resource model, Easwaran *et al.* (2007)'s explicit-deadline periodic resource model, and Easwaran *et al.* (2009a)'s ARINC 653-specific approach (reviewed in Section 2.4.2.3). Although implemented in Java, CARTS does not take advantage of the latter's object-oriented characteristics (inheritance, polymorphism, encapsulation — especially regarding the separation between domain and user interface). This makes it difficult to be extended and, as such, we chose to develop our proof-of-concept tool (Chapter 5) from scratch instead of modifying CARTS.

2.5.2.4 Schesim

The open-source Schesim scheduling simulator[16] supports two-level hierarchical scheduling on uniprocessor. Regarding task scheduling, Schesim supports fixed task priority and EDF algorithm, both in uniprocessor and partitioned multiprocessor settings (global multiprocessor scheduling is not supported).

[14] http://beru.univ-brest.fr/~singhoff/cheddar/
[15] http://rtg.cis.upenn.edu/carts/
[16] https://code.google.com/p/schesim/

Extending Schesim is not an easy task. In particular, adding a new scheduling algorithm requires obtaining, understanding and modifying Schesim's source code (Matsubara *et al.*, 2012). Other than Matsubara *et al.* (2012)'s paper, all documentation (including comments in the source code) is in Japanese language only.

2.5.2.5 Xoncrete

Within the scope of the XtratuM hypervisor (Section 2.5.1.3), Brocal *et al.* (2010) describe the Xoncrete[17] tool for scheduling analysis of ARINC 653-based TSP systems. The scheduling analysis performed by Xoncrete relies on timing requirement information, not from tasks themselves, but rather sequences of tasks with temporal attributes called End-To-End Flows. The notions of End-To-End Flow, task and partition are independent, in the sense that an End-To-End Flow can contain tasks from different partitions, and that one given task can appear as part of more than one End-To-End Flow. The tool generates a schedule (partition plan) from the End-To-End Flow parameters, using EDF as the base scheduling policy.

2.5.2.6 SymTA/S

SymTA/S [18] is Symtavision's model-based timing analysis and optimization commercial solution. Symtavision (2013)'s product note states support to ARINC 653, making no specific mention to any other hierarchical scheduling framework.

2.6 Summary

In this chapter, we have presented the fundamental background notions on real-time scheduling, followed by a survey of the state of art and of the state of practice. In the following chapters, we present the contributions of this dissertation, which are related as follows with the presented related works.

The TSP system architecture that we describe in Chapter 3, with improvements to support multicore processors (Section 3.2), can serve as the basis for improvements to the current operating system support to TSP Section 2.5.1). In Section 3.3, we

[17] http://www.fentiss.com/en/products/xoncrete.html
[18] https://symtavision.com/symtas.html

propose modeling our improved TSP system architecture as a compositional scheduling framework (Section 2.4). This system model can be used to model systems built around the existing TSP solutions (Section 2.5.1), as well as other systems based on resource reservation frameworks or hierarchical scheduling frameworks.

Consequently, the compositional analysis that we propose in Chapter 4 can be applied those systems. Our proposal extends results from classical real-time scheduling analysis (Section 2.2) and fills a void in state-of-the-art compositional analysis (Section 2.4).

The tool we developed in the scope of this dissertation (Chapter 5) is a proof of concept for improvements over the state-of-the-art tools we surveyed in Section 2.5.2.

Chapter 3

Architecture and Model for Multiprocessor Time- and Space-Partitioned Systems

In this chapter we present our first contribution: a system architecture definition and formal model for multiprocessor time- and space-partitioned (TSP) systems. We start (Section 3.1) by briefly presenting the reference architecture for uniprocessor TSP systems upon which we improve. Then, in Section 3.2, we discuss the gap to multiprocessor in the state of the art, and present an improved system architecture which tackles such issues. In Section 3.3 we present a system model which allows formal reasoning upon TSP systems which use multiprocessor platforms, with either identical or non-identical processors; our system model is directed towards compositional analysis, and is applicable to many of the state of the art approaches regarding resource reservations and hierarchical scheduling.

3.1 Architecture overview

The architecture design, improved in the early stages of this dissertation's work (Rufino *et al.*, 2010), evolved from a proof of feasibility for adding ARINC 653 support to the Real-Time Executive for Multiprocessor Systems (RTEMS) to a multi-OS (operating system) TSP architecture (Rufino *et al.*, 2007). Its modular design

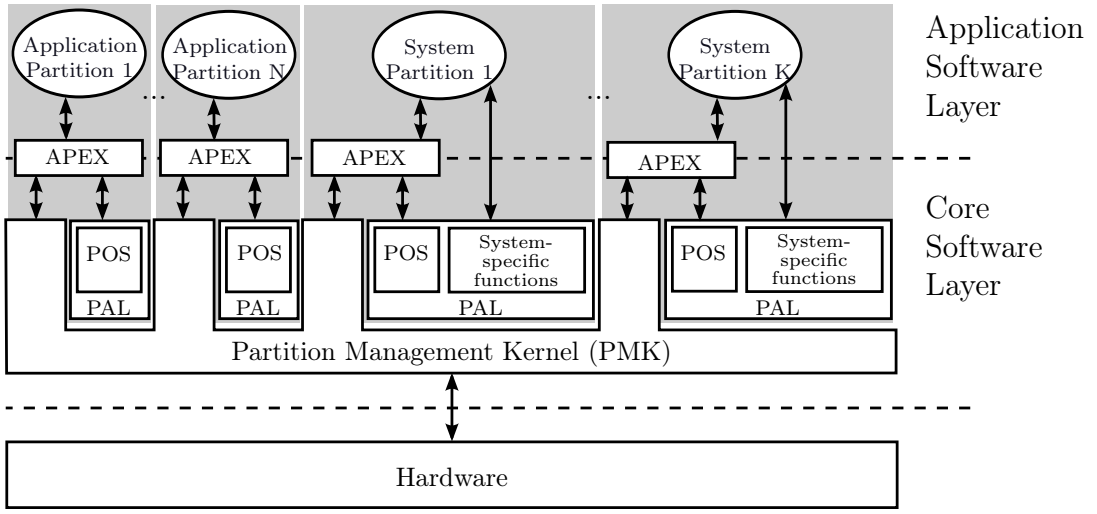


Figure 3.1: Architecture overview

aims at high levels of flexibility, hardware- and OS-independence, and independent component verification, validation and certification.

Each partition can host a different OS (the partition operating system, POS), which in turn can be either a real-time operating system (RTOS) or a generic non-real-time one. We will now describe the AIR architecture for TSP systems in enough detail for the scope of this dissertation. A more in-depth description of AIR can be found in (Rufino *et al.*, 2010).

The modular design of the AIR architecture is pictured in Figure 3.1. In this section, we describe its components and functionalities.

3.1.1 Architecture components

The *Partition Management Kernel (PMK)* is the basis of the Core Software Layer. The AIR PMK hosts crucial functionality such as partition scheduling and dispatching, low-level interrupt management, and interpartition communication support. The *POS Adaptation Layer (PAL)* encapsulates the POS (and, where appliable, system-specific functions) of each partition, providing an adequate POS-independent interface to the surrounding components. The *APEX Interface* component provides a standard programming interface derived from the ARINC 653

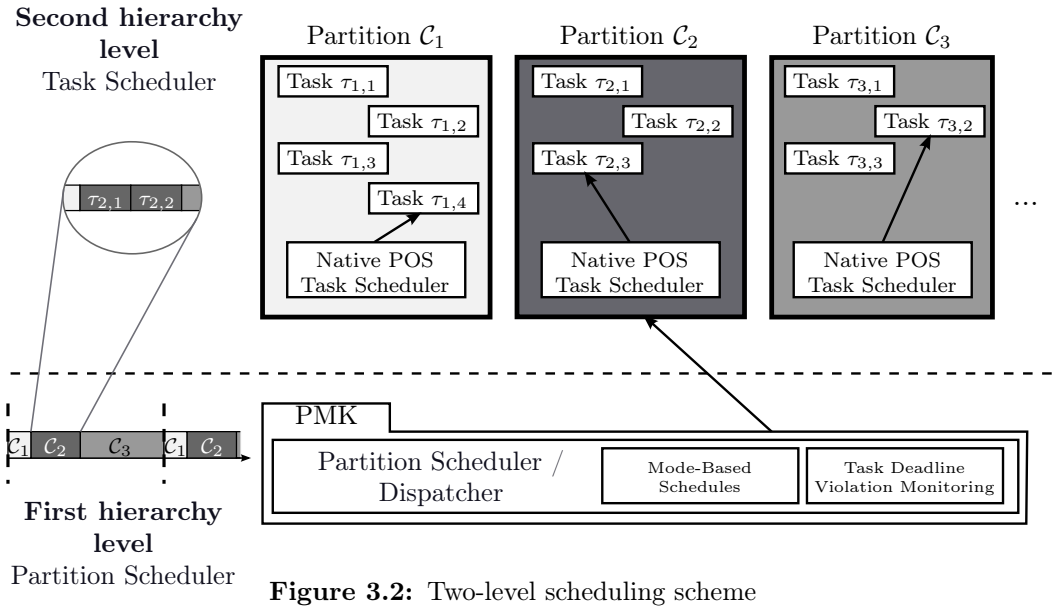


Figure 3.2: Two-level scheduling scheme

specification (AEEC, 1997). For certain partitions, this component can provide either only a subset of the APEX interface, or specific extensions (Rosa *et al.*, 2011).

Our system architecture also incorporates *Health Monitoring (HM)* functions to contain faults within their domains of occurrence and to provide the corresponding error handling capabilities. Support to these functions is spread throughout virtually all of the architecture's components.

We now describe how temporal partitioning is achieved in our system architecture. Since this dissertation focuses on temporal aspects, we omit the description of spatial partitioning mechanisms. The interested reader is refered to (Craveiro *et al.*, 2009; Craveiro, 2009) for details on the latter.

3.1.2 Achieving time partitioning

A two-level scheduling scheme, pictured in Figure 3.2, guarantees temporal isolation between partitions. The first level corresponds to partition scheduling and the second level to task scheduling. Partitions are scheduled on a cyclic basis, according to a partition scheduling table (PST) repeating over a major time frame (MTF). This table assigns execution time windows to partitions. Inside each partition's time

windows, its processes compete according to the POS's native process scheduler.[1]

Our scheduling scheme is enriched with temporal adaptation mechanisms, such as mode-based schedules and task deadline violation monitoring. We now cover only the essentials of these mechanisms in this section, so as to provide context for the contributions described in the following sections — evolution to multiprocessor and system model. We provide detailed description of the implementation and operation of our adaptation mechanisms in Chapter 6.

3.1.2.1 Mode-based schedules

Instead of one fixed PST, the system can be configured with multiple PSTs, which may differ in terms of the MTF duration, of which partitions are scheduled, and of how much processor time is assigned to them. The system can then switch between these PSTs; this is performed through a service call issued by an authorized and/or dedicated partition. To avoid violating temporal requirements, a PST switch request is only effectively granted at the end of the ongoing MTF.

3.1.2.2 Task deadline violation monitoring

During system execution, it may be the case that a task exceeds its deadline. This can be caused by a malfunction, by transient overload (e. g., due to abnormally high event signaling rates), or by the underestimation of its worst-case execution requirement at system configuration and integration time. Temporal partitioning ensures that this issue does not propagate to partitions other than the one to which the task belongs. Nevertheless, it may be necessary or beneficial to monitor and act upon such events. In the case of SRT applications, deadline misses does not directly constitute a fault, but should be monitored in light of the consequent QoS (or lack thereof). For HRT applications, deadline misses should be detected and mitigated as soon as possible to prevent serious consequences.

Our system architecture includes a lightweight mechanism for task deadline violation monitoring, which then communicates these events to the appropriate HM handler, defined by the application developer and/or by the system integrator.

[1] Partitions are denoted as $\mathcal{C}$, which stands for *component*, for coherence with the component-based approach used in this dissertation.

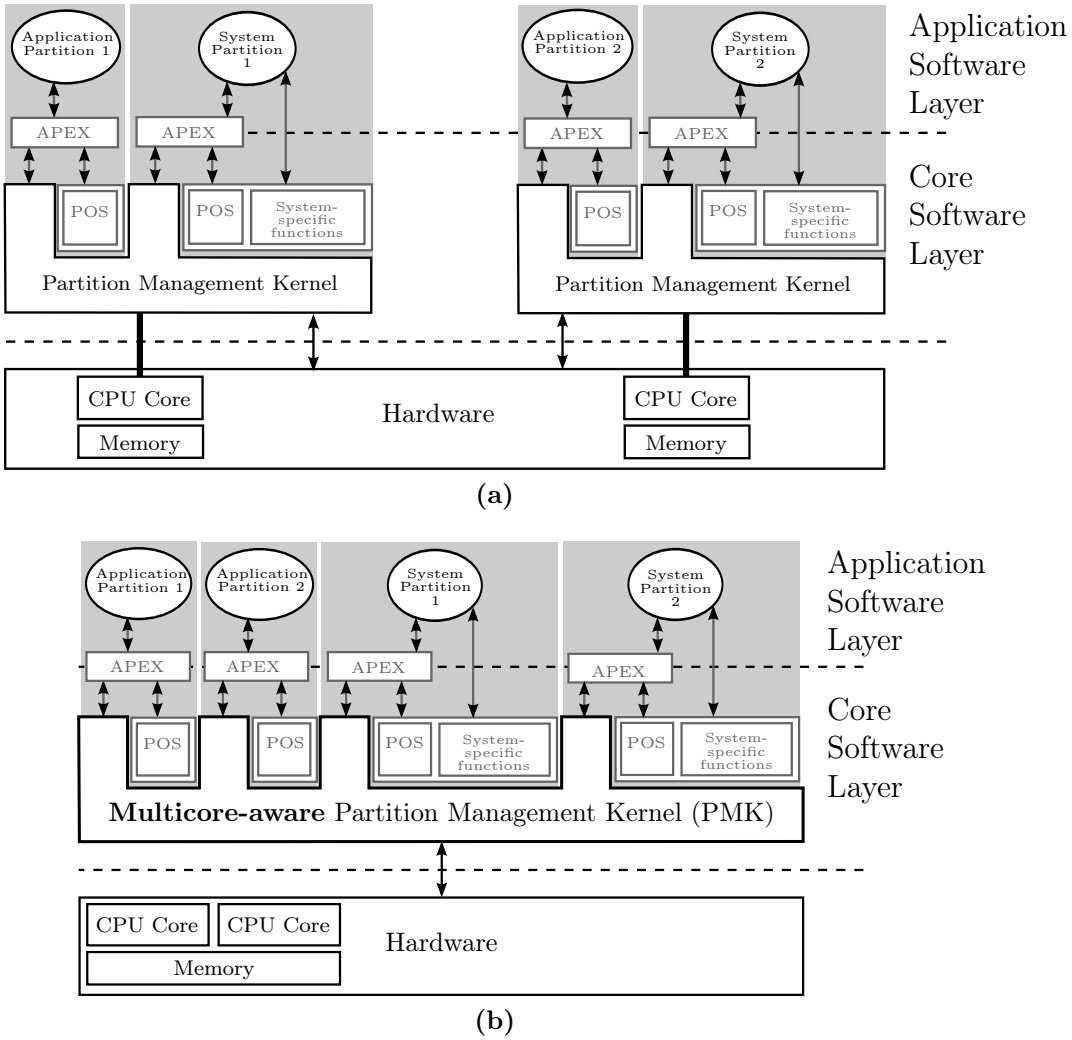


Figure 3.3: Example comparison between (a) a multiprocessor system implemented as interconnected uniprocessor TSP nodes, and (b) a multicore (or shared-memory multiprocessor) TSP system implemented with our proposal of an evolved architecture

3.2 Evolution for multiprocessor

Approaches targeting multiple processors, such as that suggested in ARINC 653 (AEEC, 1997), imply replicating the core software layer. In the case of our architecture, this implies replicating the PMK, as we see in Figure 3.3a. This configuration is inevitable when the memory adressing spaces are private to each processor and/or they are physically separated (Patterson & Hennessy, 2009). In some cases, this configuration may even be advantageous — for instance, when integration with

legacy systems is involved. However, the replicated instances are independently integrated and configured, and the partitions associated with each one would be bound to a strong affinity to a processor right from integration time. This limits the extent to which we can take advantage of shared-memory multiprocessor platforms in general, and multicore processors in particular. On the one hand, a partitioned scheduling approach (Section 2.2.2) is forced upon the system. As we have seen in Section 2.2.3.1, the partitioned and global scheduling approaches are incomparable (in the sense that none of them is strictly better than the other one), and it is desirable to have a flexible way to opt for one or the other depending on the specific system being configured. On the other hand, this approach limits the extent to which we can explore the adaptation mechanisms present in the AIR architecture, such as using redundancy and mode-based schedules to overcome a hardware failure.

To add both the desired capacity and flexibility, we propose an architectural evolution whereby the core software layer is enhanced to take advantage of an underlying shared-memory multiprocessor or multicore processor platform, as we show in Figure 3.3b. The association between partitions and cores is thus more volatile, as it can be expressed in configuration parameters common to all the cores and designed to change dynamically (e. g., through mode-based partition schedules).

Our advanced TSP system architecture may take advantage of a shared-memory multiprocessor or multicore processor platform in several ways, either in alternative or cumulatively, related with scheduling parallelism and fault tolerance. Regarding parallelism, upon which we focus in this dissertation, we propose

1. having multiple partitions simultaneously scheduling/dispatching their respective processes on distinct processor cores (*interpartition parallelism*); and/or

2. allowing some partitions to be assigned more than one processor core during their active time windows, so as to parallelly execute multiple processes (*intrapartition parallelism*).

Other advantages concern enhanced spatial partitioning, and self-adaptive behavior to cope with processor core failures.

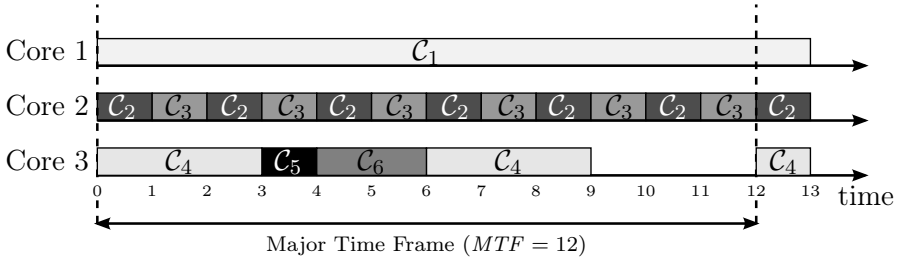


Figure 3.4: Interpartition parallelism example timeline

3.2.1 Interpartition parallelism

Interpartition parallelism is achieved by extending the first level of the hierarchical scheduling scheme presented in Section 3.1.2, as pictured in Figure 3.4. At every given moment, more than one partition can be simultaneously active (scheduling and dispatching its processes), as long as those partitions do so on different processor cores.

3.2.2 Intrapartition parallelism

Another point of view under which we defend we can take profit from multicore platforms is allowing some partitions to simultaneously use more than one processor core during their active time windows. As such, a partition may parallelly execute multiple processes.

A partition will seldom need to occupy all the available processor cores. As such, the reasonable approach is to combine both interpartition and intrapartitition parallelism. By doing so, partitions which do not use all the processor cores made available open room for the execution of processes from other partitions (interpartition parallelism). Figure 3.5 illustrates this with an example timeline. Partition $\mathcal{C}_2$ is the only partition to use more than one processor core. This way, it is able to run two processes simultaneously, as can be seen in the callout in Figure 3.5. Since $\mathcal{C}_2$ does not use all the three cores, it also accommodates the parallel execution of the processes pertaining to partition $\mathcal{C}_1$, which takes over core 3.

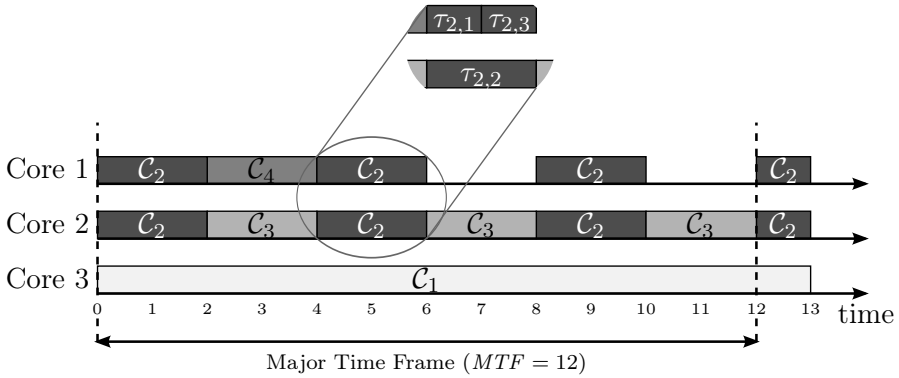


Figure 3.5: Example timeline with a combination of both inter- and intrapartition parallelism

3.2.3 Enhanced spatial partitioning

The temporal and spatial segregation enhancements we defend can be achieved through the attribution of dedicated cores to certain functions. Functions deemed adequate for these purposes include partition scheduling and management, timeliness control, and low-level device interface operations.

The deep investigation of this possibility is beyond the scope of our thesis, as stated in Section 1.3.

3.2.4 Self-adaptive fault tolerance

The proposed architecture shall allow self-adaptively tolerating hardware (e. g., processor core) faults, by adaptively coupling hardware redundancy with timeliness control mechanisms. Upon the detection, through health monitoring mechanisms, of a processor core fault, an action is triggered to begin scheduling partitions so that the duties of the faulty core are taken over by a backup core; this only covers faults of the processor core itself, and not for instance the memory. The simplest example of such self-adaptive behavior is shown in Figure 3.6. Unlike with the standard mode-based schedules functionality, the new scheduling scheme shall come into effect immediately.

The way in which this self-adaptive fault tolerance feature can be combined with the inter- and intrapartition parallelism is shown in Figure 3.7. The initial schedule has partition $\mathcal{C}_2$ executing its processes among both cores 1 and 2 during its time

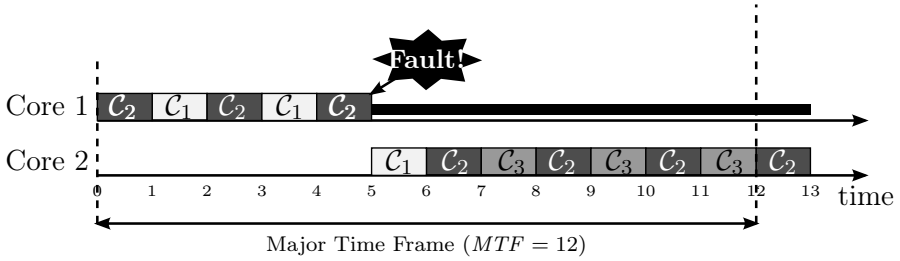


Figure 3.6: Fault tolerance example timeline

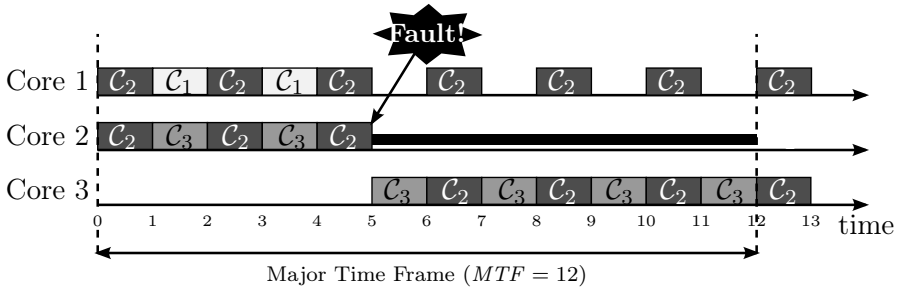


Figure 3.7: Example timeline showing a combination of fault tolerance with inter- and intra-partition parallelism

windows (intrapartition parallelism) and partition $\mathcal{C}_3$ executing on core 2, while $\mathcal{C}_1$ has some time windows of execution on core 1 (interpartition parallelism). After the fault detection and the respective adaptation, partition $\mathcal{C}_2$ will execute its processes among cores 1 and 3 during its time windows, and $\mathcal{C}_3$ will become assigned to core 2. Partition $\mathcal{C}_1$ does not have further time windows in this MTF, and will continue executing on core 1 when it is dispatched.

3.3 TSP system model

As we have stated in the introduction, we approach TSP systems as a special case of a hierarchical scheduling framework, and aim to formally analyze them in the light of compositional analysis. The system itself is modeled as a root component $\mathcal{C}_0$, with q children components $\mathcal{C}_p$ (with $1 \leq p \leq q$) representing the partitions.

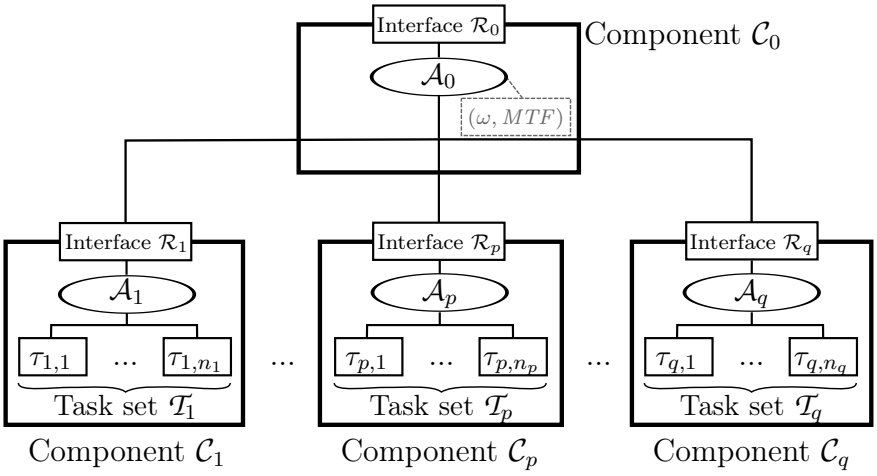


Figure 3.8: System model

3.3.1 Platform model

The physical platform upon which the system executes, is modeled as a *uniform multiprocessor platform* (Section 2.1.2) $\pi_0 \stackrel{\text{def}}{=} \{s_{0,j}\}_{j=1}^{m_0}$. It comprises m_0 processors, each of them characterized by a capacity $s_{0,j}$ (with $1 \leq j \leq m_0$). Without loss of generality, it is understood that $s_{0,1} = 1.0$, and $s_{0,j} \leq s_{0,j-1}$ for all $1 < j \leq m_0$.

The uniform multiprocessor platform model allows representing multiprocessor/multicore platforms where the processor core can be either identical or not regarding speed — being identical in all remaining aspects, such as instruction set. As a consequence, it is assumed that the execution time of tasks varies uniformly with processor capacity. This means that, for instance, the same amount of execution requirement takes twice as long to be completed on a processor with half the capacity. The effect of caches and bus contention is assumed to be either reduced to negligible or subsumed into execution requirement estimation by whichever means available (e.g., caches are disabled, non-shared, or partitioned).

3.3.2 Component model

Each component $\mathcal{C}_p$ (with $1 \leq p \leq q$) encompasses a task set $\mathcal{T}_p$, scheduled by an algorithm $\mathcal{A}_p$, and is abstracted with a resource interface $\mathcal{R}_p$. Let us analyze each of these aspects in more detail.

3.3.2.1 Task model

The task set of each component, $\mathcal{T}_p$, is composed of n_p constrained-deadline sporadic tasks (Section 2.1.1) $\tau_{p,i} \stackrel{\text{def}}{=} (T_{p,i}, C_{p,i}, D_{p,i})$. As we have seen in Section 2.1.1.1, a sporadic task represents an unbounded sequence of jobs, with execution requirement at most $C_{p,i}$, with two consecutive jobs being released at least $T_{p,i}$ time units apart. The worst-case execution requirement is assumed to have been correctly derived, and corresponds to the worst-case execution time on the fastest processor considered (i.e., the processor with schedulable utilizaton of 1.0 units of execution per unit of time).

3.3.2.2 Scheduling algorithm

Each component's scheduling algorithm $\mathcal{A}_p$ (for $p > 1$) is independent from the remaining components, and may come in the form of any scheduling algorithm capable of scheduling constrained-deadline sporadic tasks on the platform provided by the resource interface $\mathcal{R}_p$.

In this dissertation, we will focus on gEDF — global fixed job priority scheduling on a multiprocessor platform.

3.3.2.3 Resource model

The resource interface of each component, $\mathcal{R}_p$, has a dual role of abstraction:

1. to component $\mathcal{C}_p$, its resource interface $\mathcal{R}_p$ hides the details of how it receives resource into a virtual processing platform; it is upon this restricted view of the processing platform that scheduling algorithm $\mathcal{A}_p$ makes its decisions;

2. to the parent component $\mathcal{C}_0$, each resource interface $\mathcal{R}_p$ hides how the respective component $\mathcal{C}_p$ is internally structured; it is upon this restricted view of the children components of $\mathcal{C}_0$ that scheduling algorithm $\mathcal{A}_0$ makes its decisions.

The model used to express the resource interface may vary. In Chapter 4, we propose a model to express resource interfaces on uniform multiprocessor platforms. Since the latter generalize other platform models, we are as well able to express resource interfaces on uniprocessor and identical multiprocessor platforms.

3.3.3 Global-level scheduling

The global-level scheduler $\mathcal{A}_0$ is responsible for assigning the time windows during which the partitions will be able to execute their tasks on the processor cores. When $\mathcal{A}_0$ is a cyclic executive, it bases its scheduling decisions on a *partition scheduling table* (PST). At any moment, the scheduler uses *one* partition scheduling table (PST); more PSTs can be present in the system, but only one is used at a time.

To model the PST, we extend Mok *et al.* (2001)'s static partition resource model to acommodate the multiprocessor support that we are adding. The active PST is defined as a tuple (ω, MTF), where MTF is the major time frame, and ω is a set of time windows covering the whole of the MTF. Each time window is represented as $\omega_i \stackrel{\text{def}}{=} (O_i, c_i, P_i, K_i)$, with $1 \leq P_i \leq q$ and $1 \leq K_i \leq m_0$, and signifies that partition $\mathcal{C}_{P_i}$ has the K_ith processor of π_0 (with schedulable utilization s_{0,K_i}) available to schedule its tasks during every time interval $[k \cdot MTF + O_i, k \cdot MTF + O_i + c_i[$ $(k \geq 0)$. All PSTs observe the following validity properties:

- time windows are only defined for the duration of one MTF (subsequent scheduling decisions are taken by repeating the PST):

$$0 \leq O_i < MTF \wedge 0 < O_i + c_i \leq MTF, \text{ for all } \omega_i \in \omega \ ;$$

- each processor core can only be available to one partition at a time:

$$K_i = K_j \Rightarrow (O_i + c_i \leq O_j \vee O_j + c_j \leq O_i)\,, \text{ for all } \omega_i, \omega_j \in \omega \text{ with } i \neq j \ .$$

To generate the partition scheduling table, we can perform formal analysis assuming that the global-level scheduling algorithm, $\mathcal{A}_0$ takes its scheduling decisions at execution time — i.e., it is an *online* scheduling algorithm. We then obtain the partition preemption points from simulating the behavior of this algorithm.

3.4 Summary

In this chapter, we have presented a system architecture design for shared-memory multiprocessor TSP systems. The presented architecture improves on the current

state of art and practice by not demanding that the core software layer is replicated (with the consequent stronger affinity of partitions to processor cores). This improvement, which is applicable to multicore processors, enables not only inter-partition parallelism, but also intrapartition parallelism (i.e., parallelism between tasks in the same partitions) and adaptability/self-adaptability. Finally, we have presented a system model for multiprocessor TSP systems, generalized as hierarchical scheduling frameworks. This model allows us to perform formal reasoning on TSP systems, as we do in the next chapter. For the global-level scheduling, we will approach formal reasoning under the assumption that $\mathcal{A}_0$ is an online algorithm. In Chapter 5 we show how we obtain a partition scheduling table (to be used by a cyclic executive scheduler, as employed in safety-critical TSP systems) by simulating the behavior the such an online algorithm.

Chapter 4

Compositional Analysis on (Non-)Identical Uniform Multiprocessors

In this chapter, we provide analytical and experimental results with regard to compositional analysis on multiprocessors which may differ in terms of speed. We considered the system model we presented in Section 3.3, introducing a new model to express the resource interfaces of the components: the uniform multiprocessor periodic resource model (Section 4.1). We approach the three aspects of compositional analysis — local-level schedulability analysis (Section 4.2), component abstraction (Section 4.3), and intercomponent scheduling/interface composition (Section 4.4).

4.1 Resource model

To solve the described problem, we propose the *uniform multiprocessor periodic resource* (UMPR) model,

$$\mathcal{U} \stackrel{\text{def}}{=} (\Pi, \Theta, \pi) \ .$$

A resource interface expressed with the UMPR model specifies the provision of Θ units of resource over every period of length Π over a virtual uniform multiprocessor platform π (with $\Theta \in \mathbb{R}^+$ and $\Pi \in \mathbb{N}$). Platform π is defined, as we saw in

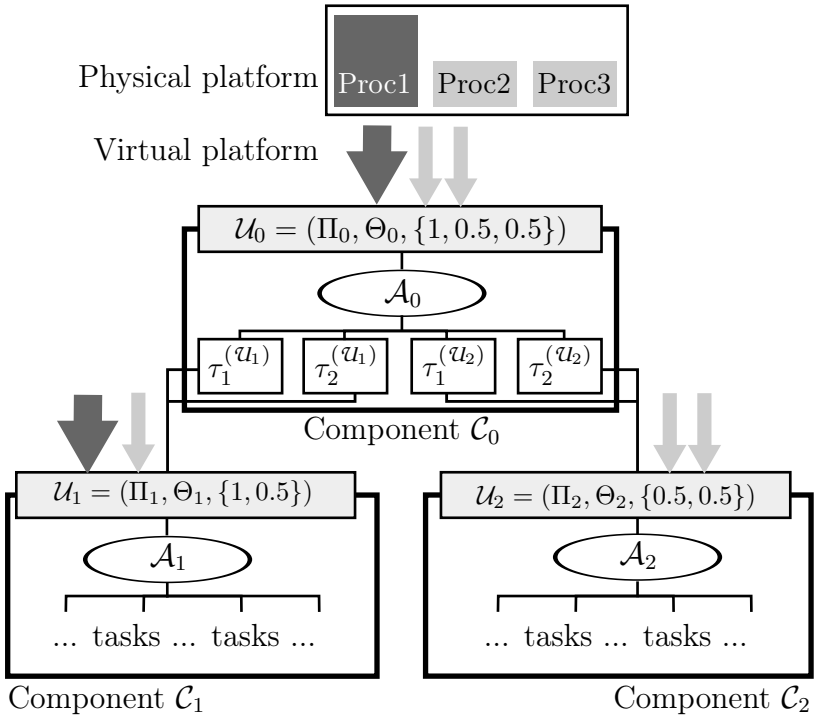


Figure 4.1: Compositional scheduling framework with the UMPR

Section 2.1.2, as

$$\pi \stackrel{\text{def}}{=} \{s_i\}_{i=1}^{m} \ ,$$

with $1.0 \geq s_{i-1} \geq s_i > 0$, for all $1 < i \leq m$.

The UMPR model extends and, more importantly, generalizes the MPR model. An MPR interface $\mu = (\Pi, \Theta, m)$ directly translates to a UMPR interface $\mathcal{U} \stackrel{\text{def}}{=} (\Pi, \Theta, \{s_i = 1.0\}_{i=1}^{m})$ without loss of representation space. For both the MPR and the UMPR models, the ratio Θ/Π is termed *bandwidth*.

Figure 4.1 provides a graphical representation of the kind of hierarchy we can build with the UMPR model. Root component $\mathcal{C}_0$ receives a virtual resource provision directly from the physical platform, whereas the remaining components receive their virtual resource provision from $\mathcal{C}_0$. The tasks in $\mathcal{C}_0$, called *interface tasks*, abstract the resource requirements of the subcomponents for use with classical schedulers and analysis (we describe these tasks in Section 4.4.1). This frameworw allows analyzing, among others, TSP systems supported upon multicore processors with

identical or non-identical cores.

4.1.1 Supply bound function

As we have seen in Section 2.4, previous works on compositional analysis use a supply bound function to express schedulability conditions and, in turn, to generate a component's resource interface. The supply bound function of a resource model is a lower bound on the amount of resource supply guaranteed to be given by it in *any* time interval of a given length. We now derive the supply bound function for UMPR interfaces. We do so by adapting the supply bound function for the MPR model, derived by Easwaran *et al.* (2009b) (Equation (2.14)).

With the MPR model, both the number of processors and the total capacity of the virtual platform are expressed in one single abstraction — m — and the role of m in the definition of the supply bound function for the MPR model is expressing the total capacity of the platform. In the UMPR model, these must be separate notions.

The top diagram in Figure 4.2 represents the schedule that yields the minimum supply in a time interval of length t (as defined by Easwaran *et al.* (2009b)) for a $\mathcal{U} = (\Pi, \Theta, \pi)$. The portrayed schedule for $\mathcal{U}$ can be described as a supply of the full capacity of π (that is $S_m(\pi)$, as defined in Equation (2.1)) for $a = \lfloor \frac{\Theta}{S_m(\pi)} \rfloor$ time units, up to one time unit with partial supply (b makes up for the eventual difference to Θ), and no supply for the rest of the period. When this extreme schedule is provided the furthest apart possible on two consecutive periods, we have the schedule that allows us to determine the minimum possible supply over any time interval of length $t > 0$. The bottom diagram represents the schedule that yields the minimum supply in the same time interval of length t, but for UMPR $\mathcal{U}' = (\Pi, \Theta, \pi')$, where $S_m(\pi') < S_m(\pi)$ (both π and π' have m processors). Since π' has less capacity than π, a provision of the same Θ units of execution capacity requires more time. Consequently, the maximum interval with no supply is shorter. If π (the platform in $\mathcal{U}$) is an identical unit-capacity multiprocessor platform, then $\mathcal{U}'$ illustrates a scenario which was not possible using the MPR model.

These two UMPR interfaces also reflect the definition of the supply bound function. The exact positioning of the t-long time interval that yields minimum supply

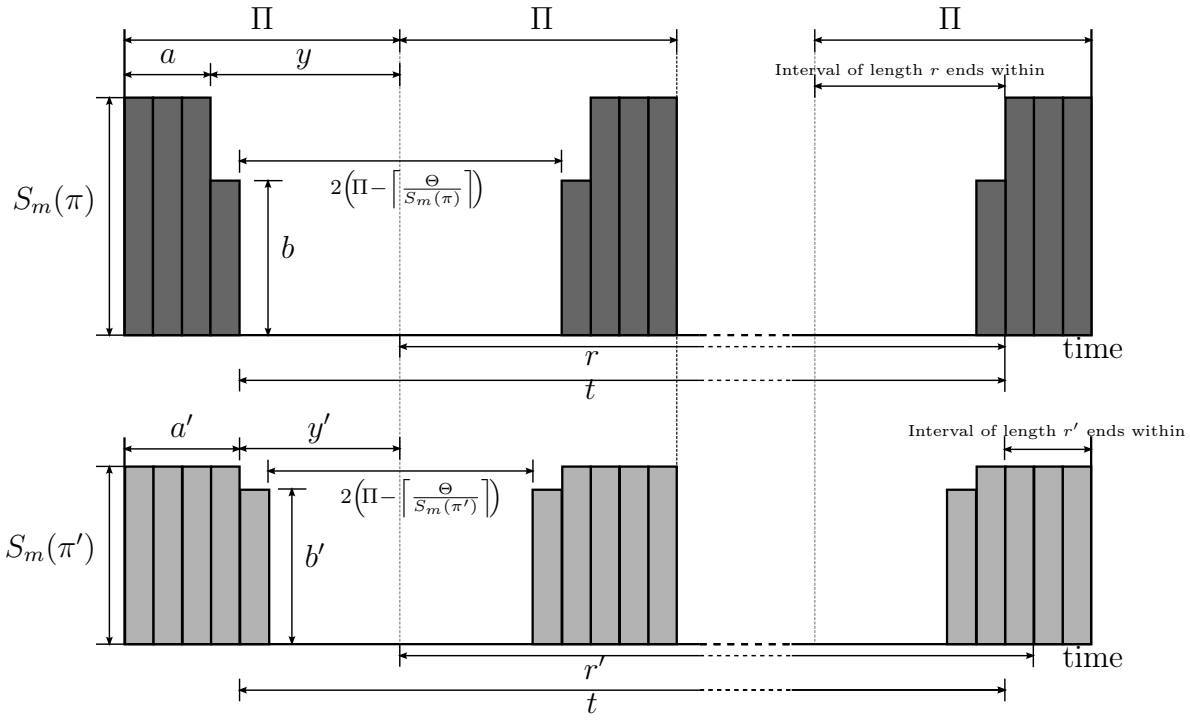


Figure 4.2: Minimum supply schedule for two UMPR interfaces: $\mathcal{U} = (\Pi, \Theta, \pi)$ (top) and $\mathcal{U}' = (\Pi, \Theta, \pi')$ (bottom), where $S_m(\pi) > S_m(\pi')$ (both with m processors) — adapted from (Easwaran *et al.*, 2009b)

may have two main variants, characterized by the positioning of an interval of length $r = t - \left(\Pi - \left\lceil \frac{\Theta}{S_m(\pi)} \right\rceil\right)$ whose beginning coincides with the beginning of a period. Each of the pictured schedules illustrates one of these variants; each of these variants corresponds to a specific branch of the supply bound function. With the aforementioned adaptation (due to the introduced separation between m and $S_m(\pi)$), we apply the supply bound function for the MPR to the UMPR; it is defined as

$$\text{SBF}(\mathcal{U}, t) \stackrel{\text{def}}{=} \begin{cases} 0, & r < 0 \\ w, & r \geq 0 \wedge x \in [1, y] \\ w - (S_m(\pi) - b), & r \geq 0 \wedge x \notin [1, y] \end{cases} \tag{4.1}$$

where

$$
\begin{aligned}
w &\stackrel{\text{def}}{=} \left\lfloor \frac{r}{\Pi} \right\rfloor \cdot \Theta + \max\left\{0, S_m(\pi) \cdot x - (S_m(\pi) \cdot \Pi - \Theta)\right\} \ , \\
r &\stackrel{\text{def}}{=} t - \left(\Pi - \left\lceil \frac{\Theta}{S_m(\pi)} \right\rceil\right) \ , \\
x &\stackrel{\text{def}}{=} \left(r - \Pi \cdot \left\lfloor \frac{r}{\Pi} \right\rfloor\right) \ , \\
y &\stackrel{\text{def}}{=} \Pi - \left\lfloor \frac{\Theta}{S_m(\pi)} \right\rfloor \ , \\
\text{and } b &\stackrel{\text{def}}{=} \Theta - \left\lfloor \frac{\Theta}{S_m(\pi)} \right\rfloor \cdot S_m(\pi) \ .
\end{aligned}
$$

When π is an identical multiprocessor platform, $m = S_m(\pi)$, and the supply bound function becomes identical to that of the equivalent MPR interface.

4.1.2 Linear lower bound on the supply bound function

To reduce the complexity of computations based on the supply bound function, it is usual to resort to a linear lower bound on the latter. We are also able to adapt for the UMPR the definition of the linear lower bound on the supply bound function, $\text{LSBF}(\mathcal{U}, t)$, that Easwaran *et al.* (2009b) defined for the MPR:

$$
\text{LSBF}(\mathcal{U}, t) \stackrel{\text{def}}{=} \frac{\Theta}{\Pi}\left(t - \left(2 \cdot \left(\Pi - \frac{\Theta}{S_m(\pi)}\right) + 2\right)\right) \ .
$$

An important property of the LSBF is that $\text{LSBF}(\mathcal{U}, t) \leq \text{SBF}(\mathcal{U}, t)$, for all $t > 0$ (Easwaran *et al.*, 2009b).

Figure 4.3 shows the plots of SBF and LSBF for the same two UMPR interfaces. In conformity with the aforementioned discussion, the range of lengths t for which there is no minimum guaranteed supply is greater for $\mathcal{U}$ than for $\mathcal{U}'$. Furthermore, we can also see that, after the range of lengths t for which there is no minimum guaranteed supply, the growth of the supply bound function repeats with a period of Π.

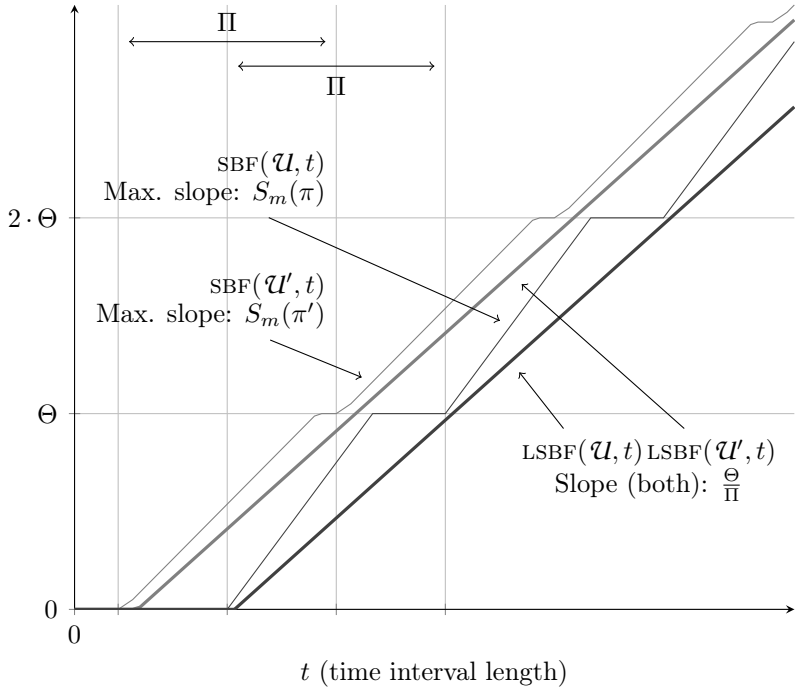


Figure 4.3: Plot of SBF and LSBF for $\mathcal{U}$ and $\mathcal{U}'$

4.2 Local-level schedulability analysis

We are considering hard real-time, so all tasks must be scheduled without violating their deadlines. A local-level schedulability test allows us to assess that, given a component $\mathcal{C}$ (which uses an algorithm $\mathcal{A}$ to schedule task set $\mathcal{T}$) and given that it is provided processing capacity in a manner abstracted as an interface (in our case, expressed using the UMPR model).[1] We now derive a sufficient schedulability condition, which means we can assess if $\mathcal{T}$ either *is schedulable* or *may be unschedulable*. Our approach extends the one followed for dedicated uniform multiprocessors by Baruah & Goossens (2008), which in turn extends the framework introduced by Baker (2003) (Section 2.2.3.1). Figure 4.4 shows the considered execution pattern.[2] We consider a job of a task τ_k, and suppose it is the first job to miss a deadline (at instant t_d) when component $\mathcal{C}$ schedules task set $\mathcal{T}$ under gEDF using the UMPR model — which corresponds to a time-shared supply of a uniform multiprocessor π.

[1] Since Sections 4.2 and 4.3 deal with each component in isolation, we simplify the notation (for a clearer reading) by omitting the subscript index pertaining to the component.

[2] Figure 4.4 is identical to Figure 2.1; we repeat it here for improved readability.

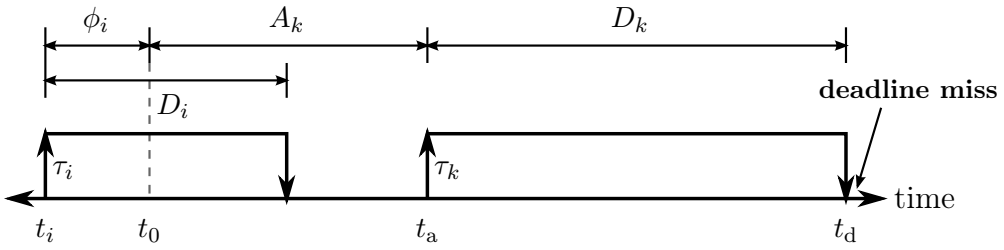


Figure 4.4: Considered execution pattern.

We now follow the strategy used by Baruah & Goossens (2008), with the due adaptation to the fact that we are dealing with a potentially non-dedicated resource supply. For this, we employ the function $\text{SUPPLY}(\mathcal{C}, t_1, t_2)$, which we have introduced in Section 2.4 and denotes the *effective* resource supply that component $\mathcal{C}$ receives over the *specific* time interval $[t_1, t_2[$. This is different from the supply bound function, which denotes the *minimum* guaranteed resource supply the component receives over *any* time interval with a given length, according to its interface. Although we do not have a tractable way to compute this quantity, we acknowledge its existence and will take advantage of its following properties:

- $\text{SUPPLY}(\mathcal{C}, t_1, t_2) + \text{SUPPLY}(\mathcal{C}, t_2, t_3) = \text{SUPPLY}(\mathcal{C}, t_1, t_3)$, for all values of t_1, t_2, t_3 such that $t_1 < t_2 < t_3$;
- $\text{SBF}(\mathcal{U}, t_2 - t_1) \leq \text{SUPPLY}(\mathcal{C}, t_1, t_2)$, for all values of t_1, t_2 such that $t_1 < t_2$; we will call upon this property when deriving our schedulability condition.

4.2.1 Interference interval

The time interval we need to consider using this execution pattern is $[t_0, t_d[$. Instant t_d is the absolute deadline that τ_k misses. We perform our analysis over a scenario in which only *one* job (of some task τ_k) misses a deadline at instant t_d; however, its reasoning and results still hold if this assumption is lifted.[3] Baruah & Goossens (2008) do not point out this issue in their paper, but the same applies: although

[3]Example: consider that jobs of both τ_1 and τ_2 miss their deadlines, and both deadlines are at t_d. After breaking the tie, one of the jobs will have a priority lower than the other one; whether the lower-priority job is that of τ_1 or τ_2, it will not interfere with the higher-priority one. As such, only the higher-priority late job is relevant.

they implicitly assume only one job misses its deadline at instant t_{d}, their analysis is still valid otherwise.

We now determine which specific instant is t_0. For this, let $W(t)$ (with $t \leq t_{\mathrm{d}}$) denote the cumulative job execution requirement for time interval $[t, t_{\mathrm{d}}[$; this is different from the demand bound function we saw in Section 2.2.1.2 (which is an upper bound on one of the sources of this cumulative job execution requirement).

Let $I_{\ell,c}$ denote the total duration, over $[t_{\mathrm{a}}, t_{\mathrm{d}}[$, for which exactly ℓ processors, *the slowest of which is* s_c, are busy scheduling jobs of component $\mathcal{C}$ (with $0 \leq \ell \leq c < m$), and at least one processor in $\{s_{c+1}, \ldots, s_m\}$ is available (thus, idle). By definition of gEDF:

1. since processor s_c is busy, no processor in $\{s_1, \ldots, s_c\}$ can be available to the component but idle; and

2. whenever there is an idle processor, τ_i's job must be executing on one of the processors in $\{s_1, \ldots, s_c\}$.

Unlike when π is a dedicated platform (Baruah & Goossens, 2008), we cannot guarantee that, at each instant, τ_i is executing on one of the fastest processors in π, but only that it is executing on one of the fastest processors *available* at that instant. So, for each duration $I_{\ell,c}$, the job is guaranteed to be executing on a processor of speed, at least, s_c. Therefore,

$$C_k > \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} s_c \cdot I_{\ell,c} \ . \tag{4.2}$$

The total amount of execution completed on $\mathcal{C}$ over the interval $[t_{\mathrm{a}}, t_{\mathrm{d}}[$ is given by the difference between the resource supply it receives — $\textsc{supply}(\mathcal{C}, t_{\mathrm{a}}, t_{\mathrm{d}})$ — and the total capacity that was idled although available. For each duration $I_{\ell,c}$, the available capacity that is idled is upper bounded by the capacity of the $m - c$ slowest processors; as we have seen, no processor in $\{s_1, \ldots, s_c\}$ can be available but idle. Since this amount is not sufficient for τ_k's job to meet its deadline at t_{d}, we

have

$$
\begin{aligned}
W(t_\mathrm{a}) &> \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} (S_m(\pi) - S_c(\pi)) \cdot I_{\ell,c} \\
&= \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} \frac{S_m(\pi) - S_c(\pi)}{s_c} \cdot s_c \cdot I_{\ell,c} \\
&\geq \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \lambda(\pi) \cdot \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} s_c \cdot I_{\ell,c} \\
&\quad \text{(by definition of } \lambda(\pi) \text{—Equation (2.2))} \\
&> \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \lambda(\pi) \cdot C_k \\
&\quad \text{(by Equation (4.2))} \\
&= \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \lambda(\pi) \cdot \delta_k \cdot D_k \\
&\geq \textsc{supply}(\mathcal{C}, t_\mathrm{a}, t_\mathrm{d}) - \lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot D_k \ .
\end{aligned}
$$

Now let us consider a particular instant t_0, which is formally defined as the smallest value $t \leq t_\mathrm{a}$ such that $W(t) \geq \textsc{supply}(\mathcal{C}, t, t_\mathrm{d}) - \lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot (t_\mathrm{d} - t)$. This means that t_0 is the earliest instant t so that, in the time interval $[t, t_\mathrm{a}[$, no resource capacity available to $\mathcal{C}$ was idled. As a consequence, we can say that, for an arbitrarily small positive number ϵ, some processor was available to $\mathcal{C}$ but idle in the time interval $[t_0 - \epsilon, t_0[$.

4.2.2 Component demand

We have defined the interval we need to consider—that is $[t_0, t_\mathrm{d}[$. Next, we determine the cumulative execution that gEDF needs (but fails) to execute over that interval, denoted by $W(t_0)$. As we have seen, two sources contribute to $W(t_0)$:

- jobs with arrival times within $[t_0, t_\mathrm{d}[$ and deadline times within $[t_0, t_\mathrm{d}]$—this contribution includes the execution requirement of τ_k's deadline-missing job, and is upper-bounded by $\sum_{\tau_i \in \mathcal{T}} \mathrm{DBF}(\tau_i, A_k + D_k)$ (where $A_k = t_\mathrm{a} - t_0$); and

- jobs that arrive before instant t_0 and carry in some execution to time interval $[t_0, t_\mathrm{d}[$—by construction of the execution pattern and by definition of the

gEDF algorithm, all the considered carry-in jobs have their deadlines at or before t_d and do not miss them.

Let us then obtain an upper bound on the contribution of the carry-in jobs. We start by proving that Lemmas 1 and 2 proven by Baruah & Goossens (2008) apply, with due adaptation, to our case.

Lemma 4.1. *Each carry-in job has strictly less than* $(A_k + D_k) \cdot \delta_{\max}(\mathcal{T})$ *remaining execution requirement at instant* t_0.

Proof. Although our lemma is equivalent to Baruah & Goossens (2008)'s Lemma 1, we must take into account the fact that the uniform multiprocessor platform is not dedicated to the component we are analyzing.

Let us consider a carry-in job of a task $\tau_i \in \mathcal{T}$, which arrives at instant $t_i < t_0$ and has not completed its execution by instant t_0 (Figure 4.4). From the definition of instant t_0, we can determine that

$$W(t_i) - W(t_0) < \text{SUPPLY}(\mathcal{C}, t_i, t_0) - \lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot \phi_i \ .$$

$W(t_i) - W(t_0)$ represents the amount of work done by gEDF on this component's schedule over $[t_i, t_0[$. Let $J_{\ell,c}$ denote the total duration, over $[t_i, t_0[$, for which exactly ℓ processors, the slowest of which is s_c, are busy scheduling jobs of this component (with $0 \leq \ell \leq c < m$), and at least one processor in $\{s_{c+1}, \ldots, s_m\}$ is available (thus, idle). By definition of gEDF, τ_i's job is guaranteed to be executing on a processor of speed at least s_c whenever there is an idle processor. Therefore, the amount of execution received by the carry-in job of τ_i over the time interval $[t_i, t_0[$ is C'_i, such that

$$C'_i \geq \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} s_c \cdot J_{\ell,c} \ . \tag{4.3}$$

The total amount of execution completed on $\mathcal{C}$ over the interval $[t_i, t_0[$ is given by the difference between the resource supply it receives — $\text{SUPPLY}(\mathcal{C}, t_i, t_0)$ — and the total capacity that was idled although available. For each duration $J_{\ell,c}$, the available capacity that is idled is upper bounded by the capacity of the $m - c$ slowest

processors. Since this amount is not sufficient for τ_k's job to meet its deadline at t_d:

$$W(t_i) - W(t_0) > \textsc{supply}(\mathcal{C}, t_i, t_0) - \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} (S_m(\pi) - S_c(\pi)) \cdot J_{\ell,c}$$

So we have

$$-\lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot \phi_i > - \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} (S_m(\pi) - S_c(\pi)) \cdot J_{\ell,c}$$

$$-\lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot \phi_i > - \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} \frac{S_m(\pi) - S_c(\pi)}{s_c} \cdot s_c \cdot J_{\ell,c}$$

$$-\lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot \phi_i > - \lambda(\pi) \cdot \sum_{\ell=1}^{m-1} \sum_{c=\ell}^{m} s_c \cdot J_{\ell,c}$$

(by definition of $\lambda(\pi)$ — Equation (2.2))

$$-\lambda(\pi) \cdot \delta_{\max}(\mathcal{T}) \cdot \phi_i > - \lambda(\pi) \cdot C'i$$

(by Equation (4.3))

$$C'_i > \delta_{\max}(\mathcal{T}) \cdot \phi_i$$

Since $C_i = \delta_i \cdot D_i \leq \delta_{\max}(\mathcal{T}) \cdot D_i$, we have that

$$\begin{aligned} C_i - C'_i &< \delta_{\max}(\mathcal{T}) \cdot D_i - \delta_{\max}(\mathcal{T}) \cdot \phi_i \\ &= \delta_{\max}(\mathcal{T}) \cdot (D_i - \phi_i) \ . \end{aligned} \tag{4.4}$$

By construction of the considered execution pattern, we have that the absolute deadline of this τ_i's job $(t_i + D_i)$ is not greater than t_d, so

$$\begin{aligned} t_i + D_i - t_0 &\leq t_d - t_0 \\ D_i - \phi_i &\leq A_k + D_k \end{aligned}$$

Replacing this in Equation (4.4) we prove the lemma. ■

Lemma 4.2. *There are, at most, $\nu = m - 1$ carry-in jobs.*

Proof. We use the same principle as Baruah & Goossens (2008), but are not able to reach an equally tight bound (cf. Equation (2.9)). Let ϵ denote an arbitrarily small

positive number; by definition, $W(t_0-\epsilon) < \text{SUPPLY}(\mathcal{C}, t_0-\epsilon, t_\text{d}) - \lambda(\pi)\cdot\delta_\text{max}(\mathcal{T})\cdot(A_k + D_k + \epsilon)$, whereas $W(t_0) \geq \text{SUPPLY}(\mathcal{C}, t_0, t_\text{d}) - \lambda(\pi)\cdot\delta_\text{max}(\mathcal{T})\cdot(A_k + D_k)$; hence:

$$W(t_0-\epsilon) - W(t_0) < \text{SUPPLY}(\mathcal{C}, t_0-\epsilon, t_0) - \lambda(\pi)\cdot\delta_\text{max}(\mathcal{T})\cdot\epsilon,$$

which shows some processor was idled (although available to $\mathcal{C}$) in $[t_0-\epsilon, t_0[$. By definition of gEDF, this means all active jobs were executing; hence, the number of carry-in jobs is, at most, one less than the number of available processors. Since we do not possess a way to know exactly how many processors were available to $\mathcal{C}$ in $[t_0-\epsilon, t_0[$, we are not able to tighten this bound further than the worst case (m processors available, $m-1$ carry-in jobs). ■

4.2.3 Sufficient local-level schedulability test

With the conditions proven in Lemmas 4.1 and 4.2 we are able to formulate a necessary condition for unschedulability of $\mathcal{C}$, from which we then derive the sufficient schedulability condition.

Lemma 4.3. *If a component $\mathcal{C}$, comprising a virtual uniform multiprocessor platform π and a sporadic task set $\mathcal{T}$, is not schedulable under gEDF using UMPR model $\mathcal{U} = (\Pi, \Theta, \pi)$, then for some $\tau_k \in \mathcal{T}$ and some $A_k \geq 0$*

$$\sum_{\tau_i\in\mathcal{T}} \text{DBF}(\tau_i, A_k+D_k) + (\nu+\lambda(\pi))\cdot(A_k+D_k)\cdot\delta_\text{max}(\mathcal{T}) > \text{SBF}(\mathcal{U}, A_k+D_k) \ , \quad (4.5)$$

Proof. We are now able to establish a strict upper bound on $W(t_0)$, based on the two sources that contribute to such execution requirement. Jobs that have arrival and deadline times inside the interference interval contribute with at most $\sum_{\tau_i\in\mathcal{T}} \text{DBF}(\tau_i, A_k + D_k)$, while jobs that arrive before instant t_0 and carry in some execution contribute with strictly less than $\nu\cdot(A_k+D_k)\cdot\delta_\text{max}(\mathcal{T})$. So we have that

$$W(t_0) < \sum_{\tau_i\in\mathcal{T}} \text{DBF}(\tau_i, A_k+D_k) + \nu\cdot(A_k+D_k)\cdot\delta_\text{max}(\mathcal{T}) \ .$$

From the definition of t_0, we have that $W(t_0) \geq \text{SUPPLY}(\mathcal{C}, t_0, t_\text{d}) - \lambda(\pi)\cdot(A_k +$

$D_k) \cdot \delta_{\max}(\mathcal{T})$. Applying that and rearranging we obtain the following inequation:

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) > \text{SUPPLY}(\mathcal{C}, t_0, t_d) \ .$$

Since we have no tractable way to compute $\text{SUPPLY}(\mathcal{C}, t_0, t_d)$, we take advantage from the fact that $\text{SUPPLY}(\mathcal{C}, t_0, t_d) \geq \text{SBF}(\mathcal{U}, A_k + D_k)$ and prove the lemma. ■

By taking the contrapositive of Lemma 4.3, we have the following sufficient schedulability condition.

Theorem 4.1 (Sufficient gEDF-schedulability test for the UMPR)**.** *A component $\mathcal{C}$ comprising a virtual uniform multiprocessor platform π and n sporadic tasks $\tau \stackrel{\text{def}}{=} \{\tau_i \stackrel{\text{def}}{=} (T_i, C_i, D_i)\}_{i=1}^n$ is schedulable under gEDF using a UMPR interface $\mathcal{U} = (\Pi, \Theta, \pi)$, if for all tasks $\tau_k \in \tau$ and all $A_k \geq 0$,*

$$\sum_{i=1}^{n} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi)) \cdot (A_k + D_k) \cdot \delta_{\max}(\tau) \leq \text{SBF}(\mathcal{U}, A_k + D_k) \ . \tag{4.6}$$

From Lemma 4.3 we can also derive a maximum value of A_k we have to consider when verifying the condition.

Corollary 4.1. *If the necessary condition in Equation (4.5) holds for some $\tau_k \in \mathcal{T}$ and some $A_k \geq 0$, then it also holds for a value of A_k such that*

$$A_k < \frac{U + B - D_k \cdot \left(\frac{\Theta}{\Pi} - u_{\text{sum}}(\mathcal{T}) - 2 \cdot (m-1) \cdot \delta_{\max}(\mathcal{T})\right)}{\frac{\Theta}{\Pi} - u_{\text{sum}}(\mathcal{T}) - 2 \cdot (m-1) \cdot \delta_{\max}(\mathcal{T})} \tag{4.7}$$

where

$$U \stackrel{\text{def}}{=} \sum_{i=1}^{n} (T_i - D_i) \cdot \frac{C_i}{T_i} \ , \qquad \text{and} \qquad B \stackrel{\text{def}}{=} \frac{\Theta}{\Pi} \cdot \left(2 + 2 \cdot \left(\Pi - \frac{\Theta}{S_m(\pi)}\right)\right) \ .$$

Proof. We can derive a necessary condition on A_k from **(i)** the necessary condition for unschedulability in Equation (4.5); **(ii)** the definitions of $\text{DBF}(\tau_i, t)$ (Equation (2.4)), ν (Lemma 4.2) and $\lambda(\pi)$ (Equation (2.2)); and **(iii)** the fact that

$\text{SBF}(\mathcal{U}, t) \geq \text{LSBF}(\mathcal{U}, t)$ (for all t).

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) > \text{SBF}(\mathcal{U}, A_k + D_k)$$

$$(A_k + D_k) \cdot u_{\text{sum}}(\mathcal{T}) + U + 2 \cdot (m-1) \cdot (A_k + D_k) > \frac{\Theta}{\Pi}(A_k + D_k) - B\,.$$

Rearranging we obtain the inequality in Equation (4.7). ■

Due to the pessimism we introduced when bounding the number of carry-in jobs (Lemma 4.2), our test does not generalize Baruah & Goossens (2008)'s test for the dedicated uniform multiprocessor case (cf. Equation (2.8)). However, when in the presence of a UMPR interface which corresponds to a dedicated supply ($\Theta = S_m(\pi) \cdot \Pi$), the latter may be employed instead.

4.3 Component abstraction

The technique presented by Easwaran *et al.* (2009b) to generate the MPR for $\mathcal{C}$ consists of:

1. assuming Π is specified by the system designer;
2. computing Θ and m so that $\mathcal{C}$ is schedulable with the least possible bandwidth (Θ/Π); component interfaces where bandwidth exceeds the platform capacity are *infeasible*.

For the computation of the schedulability test to become tractable, $\text{SBF}(\mathcal{U}, t)$ is replaced by $\text{LSBF}(\mathcal{U}, t)$.

For the UMPR model, component abstraction is not this simple, because the notions of number of processors and total capacity are no longer represented together as only m. Even restricting to uniform platforms which are in fact identical, each number of processors $m > 0$ corresponds to an infinite amount of uniform multiprocessor platforms with different total capacities $S_m(\pi) \leq m$, and each total capacity $S_m(\pi)$ may be achieved through any number of identical processors $m \geq S_m(\pi)$. Furthermore, the available physical platform may impose restrictions on this. Because of the evident added complexity, we assume both Π and π are specified, and

only Θ is computed to guarantee schedulability. We nevertheless provide important guidelines towards its solution. For this, let us introduce one additional definition.

4.3.1 Minimum resource interface

If a UMPR interface $\mathcal{U}$ guarantees schedulability of $\mathcal{C}$ (comprising task set $\mathcal{T}$) *with its smallest possible bandwidth*, then, for some $\tau_k \in \mathcal{T}$ and some $A_k \geq 0$,

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) = \text{LSBF}(\mathcal{U}, A_k + D_k) \ .$$

In this case, $\mathcal{U}$ is component $\mathcal{C}$'s minimum resource interface.

We now provide and prove guidelines regarding the relationship between

1. the bandwidth of minimum resource interface and how close to being identical the virtual platform is; and

2. the bandwidth of minimum resource interface and the number of processors.

In the second comparison, we vary the number of processors with a fixed total capacity; in the similar considerations done by Easwaran *et al.* (2009b) for the MPR model, from which we draw inspiration, this was not possible.

4.3.2 Uniform vs. identical multiprocessor platform

We start by showing that platforms with lower values of $\lambda(\pi)$ allow UMPRs with lower resource bandwidth, which leads to a more particular consideration about the relation between (non-identical) uniform multiprocessor platforms and identical ones.

Lemma 4.4. *Consider UMPR interfaces* $\mathcal{U}_1 = (\Pi, \Theta_1, \pi_1)$ *and* $\mathcal{U}_2 = (\Pi, \Theta_2, \pi_2)$*, such that* $m_1 = m_2 (= m)$*,* $S_m(\pi_1) = S_m(\pi_2)$*, and* $\lambda(\pi_1) < \lambda(\pi_2)$*. Suppose each interface guarantees schedulability of the same component* $\mathcal{C}$ *with its respective smallest possible bandwidth. Then,* $\mathcal{U}_2$ *has a higher resource bandwidth than* $\mathcal{U}_1$*, i.e.* $\Theta_1 < \Theta_2$*.*

Proof. For this lemma and the following ones, we follow the structure of the proof by contradiction of Easwaran *et al.* (2009b)'s Lemma 2. Let us then consider $\mathcal{U}'_2 = (\Pi, \Theta'_2, \pi_2)$ such that $\Theta'_2 \leq \Theta_1$, and suppose $\mathcal{U}'_2$ guarantees schedulability of $\mathcal{C}$ according to Theorem 4.1.

Let diff_d denote the difference in processor requirements of $\mathcal{C}$ on π_1 and π_2 for some interval of length $A_k + D_k$ — i.e., the difference between the left sides of Equation (4.6) for each case:

$$\begin{aligned}
\text{diff}_\text{d} = & \left(\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu_2 + \lambda(\pi_2)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) \right) \\
& - \left(\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu_1 + \lambda(\pi_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) \right) \\
& = (\nu_2 + \lambda(\pi_2) - \nu_1 - \lambda(\pi_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) \ .
\end{aligned}$$

By our definition of ν, $\nu_1 = \nu_2$; since $\lambda_1 < \lambda_2$, we have that $\text{diff}_\text{d} > 0$. Let us now obtain the counterpart diff_s, denoting the difference between the linear supply bound functions for $\mathcal{U}_1$ and $\mathcal{U}'_2$, for some interval of length $A_k + D_k$:

$$\begin{aligned}
\text{diff}_\text{s} &= \text{LSBF}(\mathcal{U}'_2, A_k + D_k) - \text{LSBF}(\mathcal{U}_1, A_k + D_k) \\
&= \frac{\Theta'_2}{\Pi} \left((A_k + D_k) - \left(2 \cdot \left(\Pi - \frac{\Theta'_2}{S_m(\pi_2)} \right) + 2 \right) \right) \\
&\quad - \frac{\Theta_1}{\Pi} \left((A_k + D_k) - \left(2 \cdot \left(\Pi - \frac{\Theta_1}{S_m(\pi_1)} \right) + 2 \right) \right) \\
&\leq \frac{\Theta_1}{\Pi} \left((A_k + D_k) - \left(2 \cdot \left(\Pi - \frac{\Theta_1}{S_m(\pi_2)} \right) + 2 \right) \right) \\
&\quad - \frac{\Theta_1}{\Pi} \left((A_k + D_k) - \left(2 \cdot \left(\Pi - \frac{\Theta_1}{S_m(\pi_1)} \right) + 2 \right) \right) \\
&\leq \frac{2 \cdot {\Theta_1}^2}{\Pi} \cdot \left(\frac{1}{S_m(\pi_2)} - \frac{1}{S_m(\pi_1)} \right) = 0 \ .
\end{aligned}$$

Thus, $\text{diff}_\text{s} \leq 0$. Since $\mathcal{U}_1$ guarantees $\mathcal{C}$ to be schedulable with its smallest possible bandwidth, $\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) \quad + (\nu_1 + \lambda(\pi_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) = \text{LSBF}(\mathcal{U}_1, A_k + D_k)$ for some $A_k + D_k$. Thus, for the same $A_k + D_k$, $\sum_{i=1}^{n} \text{DBF}(\tau_i, A_k +$

$D_k)$ $+(\nu_2 + \lambda(\pi_2)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) > \text{LSBF}(\mathcal{U}'_2, A_k + D_k)$. This means that $\mathcal{U}'_2$ *does not* guarantee schedulability of $\mathcal{C}$ according to Theorem 4.1, which contradicts our initial assumption. ■

Theorem 4.2 (Superiority of less identical platforms)**.** *For the same number of processors and total capacity, UMPRs with (non-identical) uniform multiprocessor virtual platforms are better than those with identical multiprocessor virtual platforms.*

Proof. An identical multiprocessor platform π_1 is a particular case of a uniform multiprocessor platform where all m_1 processors have the same speed, and $\lambda(\pi_1) = m_1 - 1$. For any other (non-identical) uniform multiprocessor platform π_2, $\lambda(\pi_2) < \lambda(\pi_1)$. Then, the theorem follows from Lemma 4.4 and its proof. ■

Lemma 4.4 and Theorem 4.2 are coherent with previous findings in the literature, which state the superiority of "less identical" uniform multiprocessor platforms to scheduling sporadic task sets (Baruah & Goossens, 2008).

4.3.3 Number of processors

We now compare platforms which provide the same total capacity through different number of processors. Since we only want to observe the effect of the number of processors, we need to compare platforms which are equally identical — and for this it does not suffice that they have identical values of $\lambda(\pi)$. We perform two comparisons, using one extreme case in each comparison: **(i)** both platforms are identical multiprocessor platforms; **(ii)** both platforms are the furthest possible from being an identical multiprocessor platform ($\lambda(\pi_1) = \lambda(\pi_2) = 0$).

Lemma 4.5. *Consider UMPR interfaces* $\mathcal{U}_1 = (\Pi, \Theta_1, \pi_1)$ *and* $\mathcal{U}_2 = (\Pi, \Theta_2, \pi_2)$*, such that* $m_2 = m_1 + 1$*,* $S_{m_1}(\pi_1) = S_{m_2}(\pi_2)$*,* $m_1 - 1 = \lambda(\pi_1) < \lambda(\pi_2) = m_2 - 1$*. Suppose each interface guarantees schedulability of the same component* $\mathcal{C}$ *with its respective smallest possible bandwidth. Then,* $\mathcal{U}_2$ *has a higher resource bandwidth than* $\mathcal{U}_1$*, i.e.* $\Theta_1 < \Theta_2$*.*

Proof. The proof is similar to that of Lemma 4.4, so let us consider again $\mathcal{U}'_2 = (\Pi, \Theta'_2, \pi_2)$ such that $\Theta'_2 \leq \Theta_1$, and suppose $\mathcal{U}'_2$ guarantees schedulability of $\mathcal{C}$ according to Theorem 4.1. The premises for the computation of necessary conditions on the

values of diff_d and diff_s are maintained and we thus have that $\text{diff}_\text{d} > 0$ and $\text{diff}_\text{s} \leq 0$. We are then able to reach the same contradiction and prove Lemma 4.5. ■

Lemma 4.6. *Consider UMPR interfaces* $\mathcal{U}_1 = (\Pi, \Theta_1, \pi_1)$ *and* $\mathcal{U}_2 = (\Pi, \Theta_2, \pi_2)$*, such that* $m_2 > m_1$*,* $S_{m_1}(\pi_1) = S_{m_2}(\pi_2)$*,* $\lambda(\pi_1) = \lambda(\pi_2) = 0$*. Suppose each interface guarantees schedulability of the same component* $\mathcal{C}$ *with its respective smallest possible bandwidth. Then,* $\mathcal{U}_2$ *has a higher resource bandwidth than* $\mathcal{U}_1$*, i.e.* $\Theta_1 < \Theta_2$*.*

Proof. The proof is also similar to that of Lemma 4.4; let us have $\mathcal{U}'_2 = (\Pi, \Theta'_2, \pi_2)$ such that $\Theta'_2 \leq \Theta_1$, and we suppose $\mathcal{U}'_2$ guarantees schedulability of $\mathcal{C}$ according to Theorem 4.1. The platforms for $\lambda(\pi_1) = \lambda(\pi_2) = 0$ correspond (Funk *et al.*, 2001) to the extreme cases where:

$$\pi_1 = \{S_{m_1}(\pi_1), \underbrace{0, \ldots, 0}_{m_1 - 1 \text{ processors}}\}, \qquad \pi_2 = \{S_{m_2}(\pi_2), \underbrace{0, \ldots, 0}_{m_2 - 1 \text{ processors}}\}.$$

In this case, $\lambda(\pi_1) = \lambda(\pi_2)$, but $\nu_2 > \nu_1$, so we still have $\text{diff}_\text{d} > 0$. In a similar vein as before, we find that $\text{diff}_\text{s} \leq 0$. We are then able to reach the same type of contradiction regarding the assumptions on $\mathcal{U}'_2$, and prove Lemma 4.6. ■

The following theorem then follows from Lemmas 4.5 and 4.6, and their proofs.

Theorem 4.3 (Superiority of platforms with less processors)**.** *UMPRs with virtual platforms providing the same total capacity with a lower number of faster processors are better than those with a greater number of slower processors, provided none of the platforms being compared is "less identical" than the other one.*

This is coherent with Easwaran *et al.* (2009b)'s Lemma 2 for the (identical) multiprocessor periodic resource model.

4.3.4 Simulation experiments

The experiments we now report were performed upon randomly generated task sets. To enable confronting the obtained results with a wide array of results in the literature, we obtain our samples using a method based on Davis & Burns (2009)'s UUnifast-Discard task set generation algorithm and associated parameter generation strategy. For each total capacity $S_m(\pi) \in \{2, 4, 6, 8\}$, and for each total utilization

($u_{\text{sum}}(\mathcal{T})$) between 0.025 and 0.975 (in steps of 0.025) times $S_m(\pi)$, we generated 2000 task sets as follows.

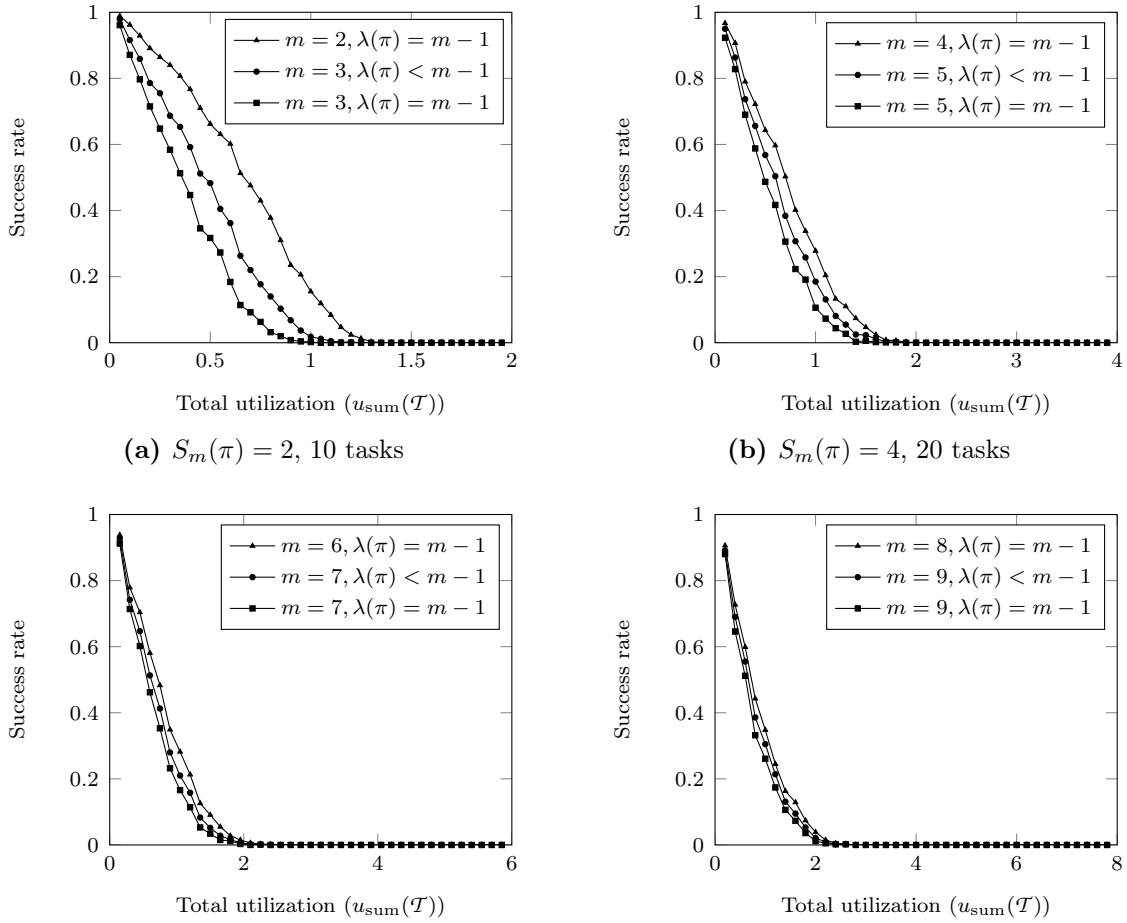
1. With UUnifast-Discard, we drew 2000 sets of $5 \cdot S_m(\pi)$ task utilizations totaling $u_{\text{sum}}(\mathcal{T})$.
2. From each utilization u_i, we derived a task $\tau_i \stackrel{\text{def}}{=} (T_i, C_i, D_i)$:
   (a) we drew an integer period T_i from a log-uniform distribution in the interval $[10, 1000[$ (this is achieved by deriving $T_i = \lfloor 10^{exp} \rfloor$, where exp is a real exponent drawn from a uniform distribution in $[1.0, 3.0[$);
   (b) the execution requirement is simply derived as $C_i = T_i \cdot u_i$; and
   (c) the relative deadline D_i is drawn from a uniform distribution in the interval $[\lfloor C_i + 1 \rfloor, T_i]$.

To derive the minimum resource interface for a component $\mathcal{C}$ comprising a randomly generated task set $\mathcal{T}$, we specify the period of the resource interface to be the minimum period among all tasks in $\mathcal{T}$. We first verify if $\Theta = S_m(\pi) \cdot \Pi$ satisfies the sufficient local-level schedulability condition with the supply bound function replaced by $S_m(\pi) \cdot t$. If it does not, then there is no feasible resource interface for the considered task set $\mathcal{T}$, platform π, and period Π. If it does, we perform a binary search on the interval $[\delta_{\text{sum}}(\mathcal{T}) \cdot \Pi, S_m(\pi) \cdot \Pi]$ for the minimum value of Θ which allows satisfying the sufficient local-level schedulability condition with the supply bound function replaced by its linear lower bound.

4.3.4.1 Component abstraction guidelines

To show the effects of the guidelines in Theorems 4.2 and 4.3, we vary $S_m(\pi)$ from 2 to 8 (in steps of 2) and randomly generate task set as described in the previous section. Then, for each task set, we generate the minimum UMPR interface that guarantees schedulability upon each of three platforms defined as follows:

- Identical 1: $m = S_m(\pi)$, and $s_i = 1.0$, for all $1 \leq i \leq m$;
- Identical 2: $m = S_m(\pi) + 1$, and $s_i = S_m(\pi)/m$, for all $1 \leq i \leq m$; for both Identical sets, $\lambda(\pi) = m - 1$;



(a) $S_m(\pi) = 2$, 10 tasks

(b) $S_m(\pi) = 4$, 20 tasks

(c) $S_m(\pi) = 6$, 30 tasks

(d) $S_m(\pi) = 8$, 40 tasks

Figure 4.5: Comparison between minimal bandwidth for UMPRs based on identical and non-identical uniform multiprocessors.

- Non-identical: $m = S_m(\pi) + 1$, $s_1 = 1.0$, and $s_i = \frac{S_m(\pi)-1}{m-1}$, for all $2 \leq i \leq m$; it can be easily shown that, for the Non-identical platform, $\lambda(\pi) = m-2 < m-1$.

For instance, for $S_m(\pi) = 2$, the platforms are respectively $\{1.0, 1.0\}$, $\{\frac{2}{3}, \frac{2}{3}, \frac{2}{3}\}$, and $\{1.0, 0.5, 0.5\}$.

Figure 4.5 shows the plots of the obtained results, in terms of the percentage of task sets, in each total utilization level, for which a feasible ($\Theta \leq S_m(\pi) \cdot \Pi$) resource interface could be derived. The overall shape of the plots, with success rate decreasing as total utilization increasing, is consistent with those seen in similar experiments in the literature. We see that, for the every total utilization level, increasing the

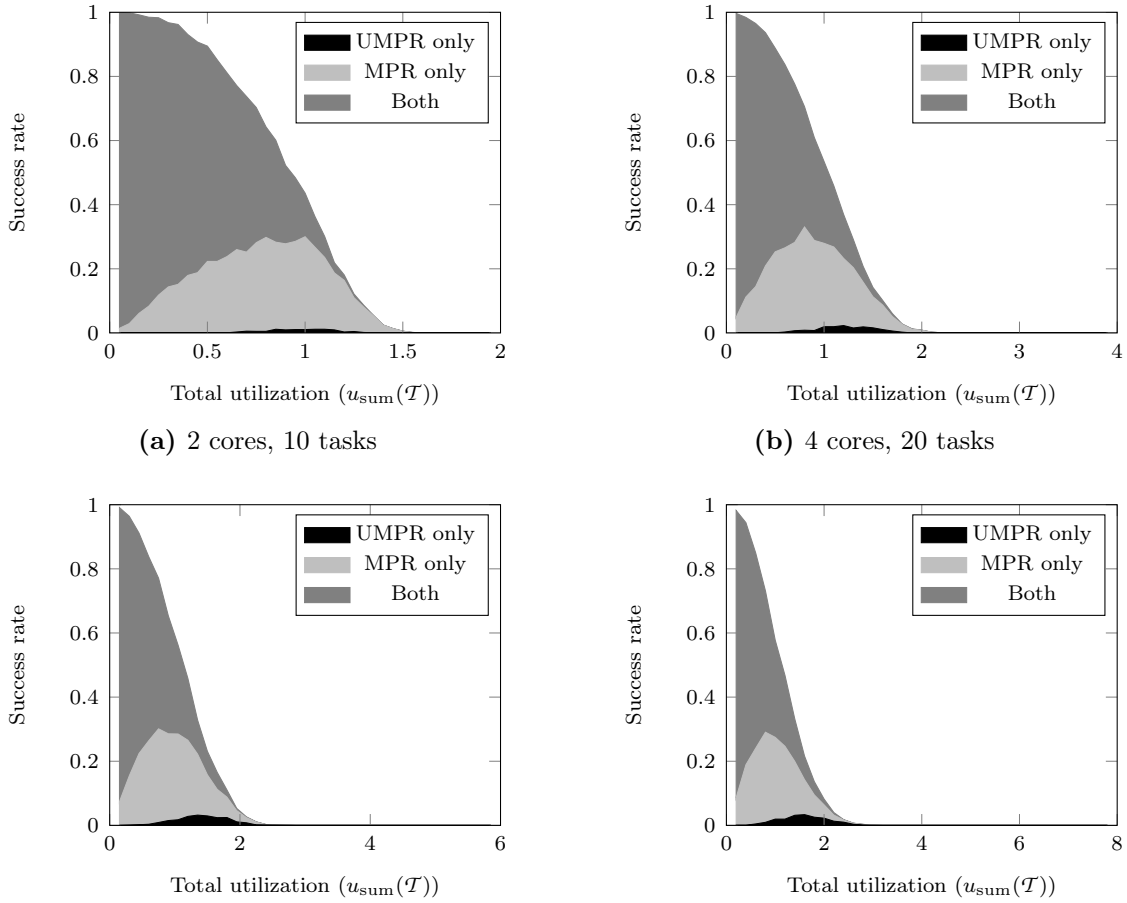
number of processors decreases the percentage of task sets (from the same group of 2000) for which a feasible UMPR interface can be derived. This confirms the guideline in Theorem 4.3, which states that, between two platforms with the same capacity and which are not less identical than one another, virtual platforms with less faster processors are better than those with more slower processors. We also see that, having the same capacity and number of processors, the Non-identical platform presents a higher success rate than the Identical 2 platform in every total utilization level. This confirms the guideline in Theorem 4.2, which states that less identical platforms (those with lower values of $\lambda(\pi)$) perform better.

4.3.4.2 UMPR vs. MPR

We performed another experiment, this time in order to compare our results with related work, namely with the work of Easwaran *et al.* (2009b) on the (identical) multiprocessor periodic resource model. We vary $S_m(\pi)$ from 2 to 8 (in steps of 2) and randomly generate task sets as described in the previous section. In this experiment, we consider only identical multiprocessor platforms with unit-capacity processors (i.e., $m = S_m(\pi)$) and derive the minimum UMPR and MPR interfaces, using, respectively, the schedulability test we propose in Section 4.2 and Easwaran *et al.* (2009b)'s schedulability test.

In Figure 4.6, we have a stacked area plot of the observed success rate. With the success rate divided as such, we can see that none of the schedulability tests fully dominates the other one, and that most of samples allowed derived both a feasible UMPR interface and a feasible MPR interface. However, it is visible that there are far more task sets that were schedulable only with an MPR interface than task sets that were schedulable only with a UMPR interface. This arises as a consequence of transitioning from identical to uniform, since we have to incur some pessimism when deriving Equations (4.2) and (4.3); this is also the case on dedicated uniform multiprocessors (Baruah & Goossens, 2008). Furthermore, we introduce additional pessimism in the number of carry-in jobs (Lemma 4.2).

Table 4.1 shows the percentage of samples, among those for which *some* resource interface was derived, for which only a UMPR interface, a MPR interface, or both was derived. In other words, the table provides, for each graph, the fraction of



(a) 2 cores, 10 tasks

(b) 4 cores, 20 tasks

(c) 6 cores, 30 tasks

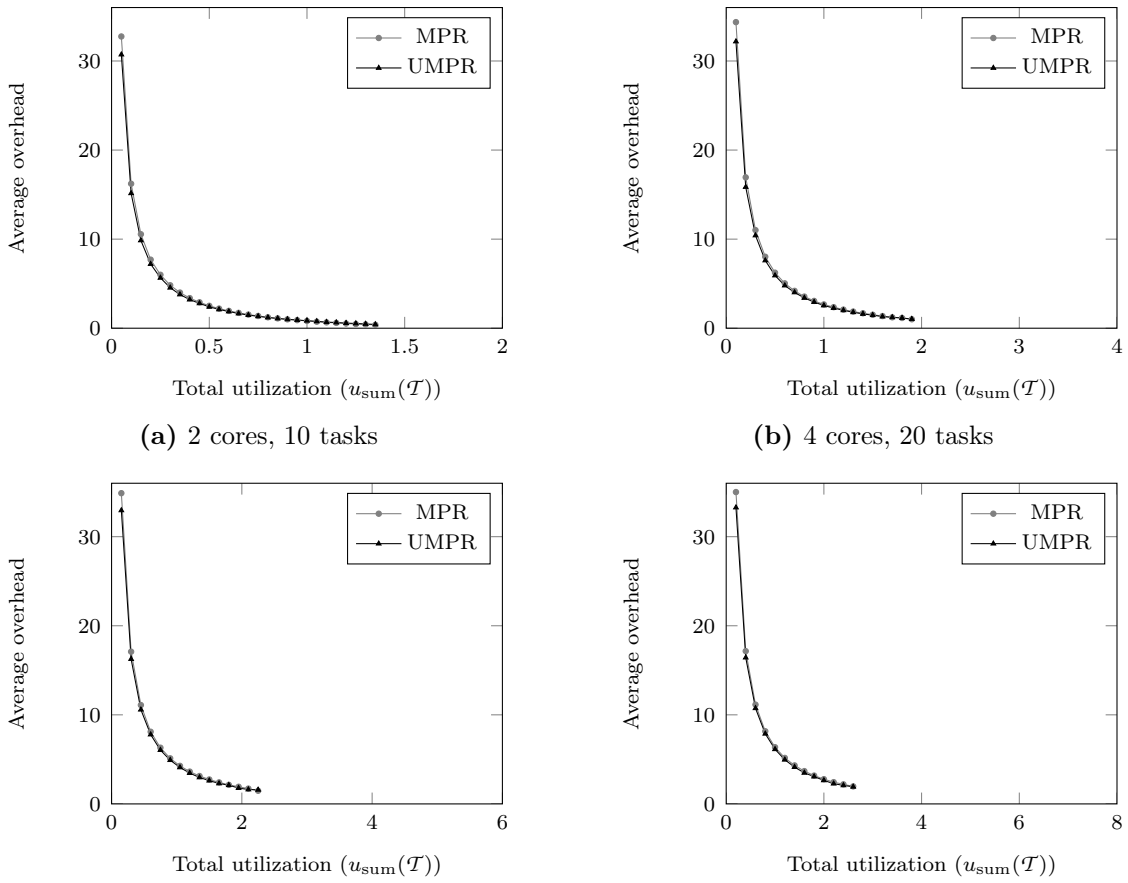
(d) 8 cores, 40 tasks

Figure 4.6: Success rates for MPR and UMPR interfaces with identical multiprocessor platforms.

Table 4.1: Success rates of MPR and UMPR interfaces with identical multiprocessor platforms (among feasible cases).

	2 cores	**4 cores**	**6 cores**	**8 cores**
UMPR only	0.60%	1.60%	2.94%	3.61%
MPR only	28.08%	32.46%	32.63%	31.44%
Both	71.32%	65.94%	64.43%	64.95%

the filled area corresponding to each case (thus, all columns add up to 100%. This highlights the fact that none of the schedulability tests dominates the other one completely.



(a) 2 cores, 10 tasks

(b) 4 cores, 20 tasks

(c) 6 cores, 30 tasks

(d) 8 cores, 40 tasks

Figure 4.7: Average overhead for MPR and UMPR interfaces with identical multiprocessor platforms.

The plots in Figure 4.7 show, for each considered value for total utilization, the average overhead among samples for which it was possible to derive both a feasible UMPR interface and a feasible MPR interface. The average overhead is calculated, for each utilization band, by dividing the average resource bandwidth (among samples for which we could derive both MPR and UMPR interfaces) by the task set utilization, and subtracting one. Since the analysis is based on the worst-case supply, the overhead is quite significant, with the lowest utilization bands reaching an average overhead of about 35 times the task set utilization. The lowest average overhead is observed in the experiment with 10 tasks on 2 cores, for the set of task sets with utilization 1.35; for this utilization band, the resource bandwidth for

MPR and UMPR interfaces is, respectively, 38% and 42% greater (on average) than the task set utilization. Contrarily to this particular data point, in most experiments and utilization values the UMPR interfaces presents less overhead that the MPR interfaces.

4.4 Intercomponent scheduling

We first approach the problem of interface composition assuming that $\mathcal{C}_0$ is the root component of the system (i.e., has access to a dedicated uniform multiprocessor platform). The resulting interface composition is also appliable to situations where $\mathcal{C}_0$ is not the root component, albeit resulting in a pessimistic resource interface $\mathcal{U}_0$. At the end of this section, we present improvements for this scenario.

We consider the system model we presented in Section 3.3, which we now briefly restate. Component $\mathcal{C}_0$ has q children components $\mathcal{C}_p$ $(1 \leq p \leq q)$, comprising a scheduling algorithm $\mathcal{A}_p$ = gEDF to schedule task set $\mathcal{T}_p \stackrel{\text{def}}{=} \{\tau_{p,i}\}_{i=1}^{n_p}$. Each component $\mathcal{C}_p$ has a UMPR resource interface $\mathcal{U}_p \stackrel{\text{def}}{=} (\Pi_p, \Theta_p, \pi_p)$, which abstracts:

- to $\mathcal{C}_0$, the overall resource demand $\mathcal{C}_p$ requires; and
- to $\mathcal{C}_p$, the resource supply that $\mathcal{C}_0$ provides.

The virtual platform in the resource interface, $\pi_p \stackrel{\text{def}}{=} \{s_{p,j}\}_{j=1}^{m_p}$, is a uniform multiprocessor platform.

4.4.1 Transforming components to interface tasks

As we have mentioned in Section 4.1, deriving a task set with an execution requirement equivalent to that of a component allows scheduling with a traditional scheduler (such as gEDF) and possibly analysis with classical results. We now present a method of transforming a UMPR interface $\mathcal{U}_p$ into a set of m_p periodic tasks. A periodic task is a particular case of a sporadic task whose jobs have their arrival times *exactly* separated by fixed time interval — the task's period.

Consider a UMPR interface $\mathcal{U}_p = (\Pi_p, \Theta_p, \pi_p)$. Let us have

$$b \stackrel{\text{def}}{=} \Theta_p - S_{m_p}(\pi_p) \cdot \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \quad \text{and} \quad k \stackrel{\text{def}}{=} \max\{\ell : S_\ell(\pi_p) \leq b\} \ .$$

We define the transformation from $\mathcal{U}_p$ to a periodic task set $\mathcal{T}^{(\mathcal{U}_p)}$ with m_p tasks $\tau_i^{(\mathcal{U}_p)} \stackrel{\text{def}}{=} (T_i^{(\mathcal{U}_p)}, C_i^{(\mathcal{U}_p)}, D_i^{(\mathcal{U}_p)})$ such that:

- if $i \leq k$, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left(\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor + 1\right) \cdot s_{p,i}, \Pi_p\right)$;
- if $i = k+1$, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} + b - S_k(\pi_p), \Pi_p\right)$;
- otherwise, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i}, \Pi_p\right)$.

Theorem 4.4 (Generalization of the task transformation for the MPR)**.** *Consider the UMPR $\mathcal{U}_p = (\Pi_p, \Theta_p, \pi_p)$. If π_p is in fact an identical multiprocessor platform, this transformation becomes equivalent to that by Easwaran et al. (2009b) for the MPR model.*

Proof. If π_p is an m_p-processor identical multiprocessor platform, then all processors have speed 1.0, and $S_\ell(\pi_p) = \ell$, for all $\ell \leq m_p$. Hence, $b \stackrel{\text{def}}{=} \Theta_p - S_{m_p}(\pi_p) \cdot \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor = \Theta_p - m_p \cdot \left\lfloor \frac{\Theta}{m_p} \right\rfloor$ and $k \stackrel{\text{def}}{=} \max\{\ell : S_\ell(\pi_p) \leq b\} = \max\{\ell \in \mathbb{N} : \ell \leq b\} = \lfloor b \rfloor$. These match the definitions by Easwaran *et al.* (2009b) (see Section 2.4.3.1). When we replace $S_{m_p}(\pi_p)$ with m_p and all $s_{p,i}$'s with 1 in the definition of our transformation, we have the following rules:

- if $i \leq k$, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor + 1), \Pi_p\right)$;
- if $i = k+1$, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor + b - S_k(\pi), \Pi_p\right) = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor + b - k \cdot \left\lfloor \frac{b}{k} \right\rfloor, \Pi_p\right)$;
- otherwise, $\tau_i^{(\mathcal{U}_p)} = \left(\Pi_p, \left\lfloor \frac{\Theta_p}{m_p} \right\rfloor, \Pi_p\right)$;

which match the rules of the transformation from MPR to tasks presented by Easwaran *et al.* (2009b). ■

Lemma 4.7. *The sum of the execution requirements of all the tasks in $\mathcal{T}^{(\mathcal{U}_p)}$ is equal to Θ_p.*

Proof. To prove this lemma, we consider three separate cases:

1. Θ_p is exactly divisible by $S_{m_p}(\pi_p)$;
2. Θ_p is not exactly divisible by $S_{m_p}(\pi_p)$, and $S_k(\pi_p) = b$; or

3. Θ_p is not exactly divisible by $S_{m_p}(\pi_p)$, and $S_k(\pi_p) < b$.

Case 1. If Θ_p is exactly divisible by $S_{m_p}(\pi_p)$, then $b = 0$ and $k = 0$. Hence, $\mathcal{T}^{(\mathcal{U}_p)}$ will have:

- one task $(\tau_1^{(\mathcal{U}_p)} \stackrel{\text{def}}{=} \tau_{k+1}^{(\mathcal{U}_p)})$ with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} + b - S_k(\pi_p) = \frac{\Theta_p}{S_{m_p}(\pi_p)} \cdot s_{p,i}$, with $i = 1$; and

- $m_p - 1$ tasks, each with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} = \frac{\Theta_p}{S_{m_p}(\pi_p)} \cdot s_{p,i}$, with $i \in [2..m_p]$.

The total execution requirement is given by

$$\begin{aligned}
\sum_{\tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)}} C_i^{(\mathcal{U}_p)} &= \sum_{i=1}^{m_p} \frac{\Theta_p}{S_{m_p}(\pi_p)} \cdot s_{p,i} \\
&= \frac{\Theta_p}{S_{m_p}(\pi_p)} \cdot \sum_{i=1}^{m_p} s_{p,i} \\
&= \frac{\Theta_p}{S_{m_p}(\pi_p)} \cdot S_{m_p}(\pi_p) \\
&= \Theta_p \ .
\end{aligned}$$

Case 2. If Θ_p is not exactly divisible by $S_{m_p}(\pi_p)$, then $b > 0$. Let us consider that k is such that $S_k(\pi_p) = b$ (therefore, $k > 0$). Hence, $\mathcal{T}^{(\mathcal{U}_p)}$ will have:

- k tasks with execution requirement $\left(\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor + 1 \right) \cdot s_{p,i}$, with $i \in [1..k]$;

- one task with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} + b - S_k(\pi_p) = \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i}$, with $i = k + 1$; and

- $m_p - (k + 1)$ tasks with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i}$, with $i \in [k + 2 \,..\, m_p]$.

The total execution requirement is given by

$$\begin{aligned}
\sum_{\tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)}} C_i^{(\mathcal{U}_p)} &= \left(\sum_{i=1}^{k} \left(\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor + 1 \right) \cdot s_{p,i} \right) \\
&\quad + \left(\sum_{i=k+1}^{m_p} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) \\
&= \left(\sum_{i=1}^{k} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) + \left(\sum_{i=1}^{k} s_{p,i} \right) \\
&\quad + \left(\sum_{i=k+1}^{m_p} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) \\
&= \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) + \underbrace{S_k(\pi_p)}_{b} \\
&= \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) + \Theta_p - \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) \\
&= \Theta_p \ .
\end{aligned}$$

Case 3. If Θ_p is not exactly divisible by $S_{m_p}(\pi_p)$, then $b > 0$. Let us consider that k is such that $S_k(\pi_p) < b$ (therefore, $k \geq 0$). Hence, $\mathcal{T}^{(\mathcal{U}_p)}$ will have:

- k tasks with execution requirement $\left(\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor + 1 \right) \cdot s_{p,i}$, with $i \in [1..k]$;
- one task with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} + b - S_k(\pi_p)$, with $i = k + 1$; and
- $m_p - (k + 1)$ tasks with execution requirement $\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i}$, with $i \in [k + 2 \,..\, m_p]$.

The total execution requirement is given by

$$\begin{aligned}
\sum_{\tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)}} C_i^{(\mathcal{U}_p)} &= \left(\sum_{i=1}^{k} \left(\left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor + 1 \right) \cdot s_{p,i} \right) \\
&\quad + \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,k+1} + b - S_k(\pi_p) \\
&\quad + \left(\sum_{i=k+2}^{m_p} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) \\
&= \left(\sum_{i=1}^{k} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) + \underbrace{\left(\sum_{i=1}^{k} s_{p,i} \right)}_{S_k(\pi_p)} \\
&\quad + \left\lfloor \frac{\Theta_p}{S_m(\pi_p)} \right\rfloor \cdot s_{p,k+1} + b - S_k(\pi_p) \\
&\quad + \left(\sum_{i=k+2}^{m_p} \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot s_{p,i} \right) \\
&= \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) + b \\
&= \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) + \Theta_p - \left\lfloor \frac{\Theta_p}{S_{m_p}(\pi_p)} \right\rfloor \cdot S_{m_p}(\pi_p) \\
&= \Theta_p \ .
\end{aligned}$$

Having proven our statement for the three cases that have to be considered, we prove the lemma. ■

Theorem 4.5 (Correctness of the transformation to interface tasks)**.** *If task set $\mathcal{T}^{(\mathcal{U}_p)}$ is schedulable by gEDF, then the supply to $\mathcal{C}_p$ is lower-bounded by* $\mathrm{SBF}(\mathcal{U}, t)$.

Proof. For each task $\tau_i^{(\mathcal{U}_p)} = (\Pi_p, C_i^{(\mathcal{U}_p)}, \Pi_p)$, not missing a deadline means that task $\tau_i^{(\mathcal{U}_p)}$ receives at least $C_i^{(\mathcal{U}_p)}$ execution units within every period of length Π. Overall, we have that tasks in task set $\mathcal{T}^{(\mathcal{U}_p)}$ receive at least $\sum_{\tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)}} C_i^{(\mathcal{U}_p)} = \Theta$ (Lemma 4.7) execution units within every period of length Π_p. This is exactly the definition of the supply provided according to a UMPR $\mathcal{U}_p = (\Pi_p, \Theta_p, \pi_p)$. ■

4.4.2 Compositionality with gEDF intercomponent scheduling

Let us have component $\mathcal{C}_1$, which is schedulable with the smallest possible bandwidth with UMPR interface $\mathcal{U}_1 = (\Pi_1, \Theta_1, \pi_1)$, where $\pi_1 = \{v, w, y\}$ such that $1.0 \geq v = w > y > 0.0$. Now let us assume that, with consideration to the remaining components, we employ a physical platform $\pi_0 = \{v, w, x, y\}$ such that $1.0 \geq v = w = x > y > 0.0$. To schedule $\mathcal{C}_1$, gEDF will schedule the respective interface tasks—$\tau_1^{(\mathcal{U}_1)}$, $\tau_2^{(\mathcal{U}_1)}$, and $\tau_3^{(\mathcal{U}_1)}$—on the available processors. The exact processors upon which $\mathcal{C}_1$'s interface tasks are scheduled depend on competition with the interface tasks of the remaining components. In the extreme case, $\mathcal{C}_1$'s interface tasks may be always scheduled on the three fastest processors in the platform. This means $\mathcal{C}_1$ ends up receiving resource according to a different UMPR interface, $\mathcal{U}_1' = (\Pi_1, \Theta_1, \pi_1')$, where $\pi_1' = \{v, w, x\}$ such that $1.0 \geq v = w = x > 0.0$. We now prove two lemmas to show the impact of this scenario on the schedulability of $\mathcal{C}_1$.

Lemma 4.8. *For any values of v, w, x, y such that $1.0 \geq v = w = x > y > 0.0$, $\lambda(\pi_1') > \lambda(\pi_1)$.*

Proof. We prove this by contradiction; assume $\lambda(\pi_1') \leq \lambda(\pi_1)$. By definition, this means

$$\max\left\{\frac{w+x}{v}, \frac{x}{w}\right\} \leq \max\left\{\frac{w+y}{v}, \frac{y}{w}\right\} \ .$$

Since $v = w$, we can replace all occurrences of v with w and rearrange, and have

$$\begin{aligned} \max\left\{\frac{x}{w}+1, \frac{x}{w}\right\} &\leq \max\left\{\frac{y}{w}+1, \frac{y}{w}\right\} \\ \frac{x}{w}+1 &\leq \frac{y}{w}+1 \\ x &\leq y \ , \end{aligned}$$

which contradicts the assumption that $x > y$. ■

Lemma 4.9. *If $\mathcal{C}_1$ receives resource according to $\mathcal{U}_1'$ (instead of $\mathcal{U}_1$), it is not guaranteed to be able schedule its tasks.*

Proof. Since $\mathcal{U}_1$ schedules $\mathcal{C}_1$ with the smallest possible bandwidth, then, by definition, for some task τ_k and some $A_k \geq 0$,

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) = \text{LSBF}(\mathcal{U}_1, A_k + D_k) \ .$$

For the same τ_k and A_k, since $\lambda(\pi'_1) > \lambda(\pi_1)$ (Lemma 4.8), we have

$$\begin{aligned} &\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi'_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) \\ &> \sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) \ . \end{aligned}$$

On the other hand, since $S_m(\pi'_1) > S_m(\pi)$, $\text{LSBF}(\mathcal{U}'_1, t) < \text{LSBF}(\mathcal{U}_1, t)$ (for all values of $t > 0$), i.e.,

$$\text{LSBF}(\mathcal{U}_1, A_k + D_k) > \text{LSBF}(\mathcal{U}'_1, A_k + D_k) \ .$$

Consequently,

$$\sum_{\tau_i \in \mathcal{T}} \text{DBF}(\tau_i, A_k + D_k) + (\nu + \lambda(\pi'_1)) \cdot (A_k + D_k) \cdot \delta_{\max}(\mathcal{T}) > \text{LSBF}(\mathcal{U}'_1, A_k + D_k) \ ,$$

which means $\mathcal{U}'_1$ cannot guarantee schedulability of the tasks in component $\mathcal{C}_1$. ■

Having proven these two lemmas, we can now prove an important theorem.

Theorem 4.6 (Inadequacy of gEDF for intercomponent scheduling)**.** *If $\mathcal{A}_0$ is gEDF, the considered scheduling framework (Figure 4.1) is not compositional.*

Proof. This follows from our example and Lemma 4.9. We cannot guarantee that the properties established and validated for components in isolation (namely, schedulability) hold once the components are integrated to form the system. ■

We have been able to show the inadequacy of gEDF as an intercomponent scheduling algorithm in the presence of different types of processors.

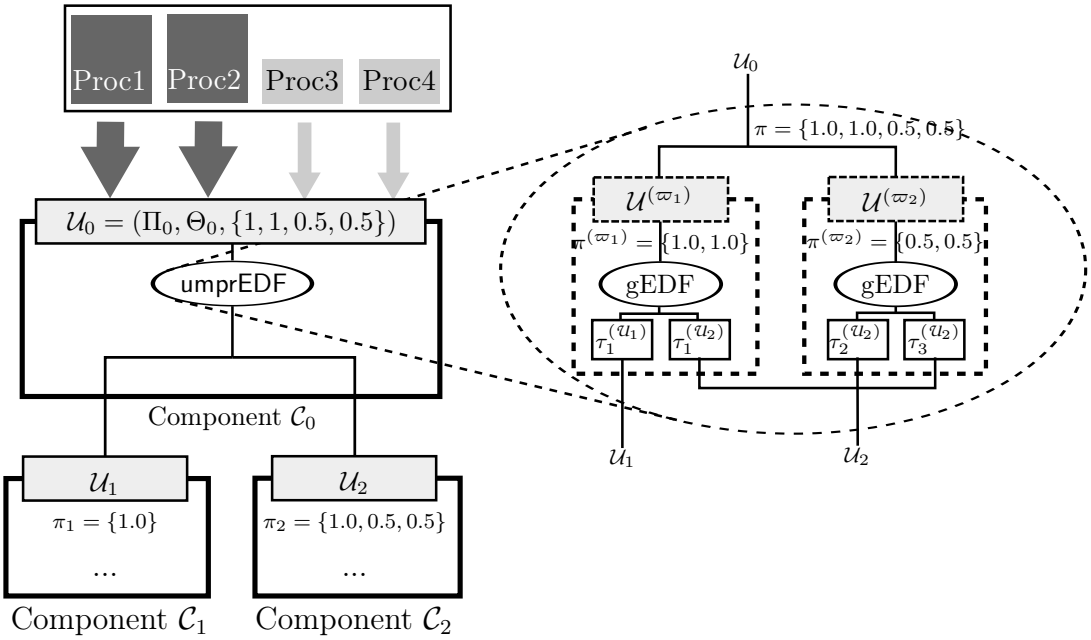


Figure 4.8: Formalization of umprEDF with pseudocomponents

4.4.3 The umprEDF algorithm for intercomponent scheduling

Scheduling all interface tasks (and consequently the components) with the completely work-conserving algorithm gEDF lends itself to scheduling anomalies (by allowing components to be supplied with processors which are not part of the virtual platform upon which their local-level analysis was done), thus voiding the compositionality property. We now present a non-work-conserving scheduling algorithm, which we call umprEDF, that prevents these anomalies by clustering interface tasks upon subsets of the platform's processors; these subsets are based on the processors' speeds.

To enable formal reasoning with familiar results, namely to pave the way to interface composition (i.e., computing the resource interface for the root component, $\mathcal{U}_0$), we model the operation of umprEDF as a set of *pseudocomponents*. Each pseudocomponent has a gEDF scheduler operating over a subset of platform π_0, as shown in Figure 4.8. We now describe formally and in more detail how the interface tasks of the children components and the processors in the platform are divided among these pseudocomponents.

Let ϖ denote the set of unique speeds found among all virtual platforms π_p (with

$1 \leq p \leq q$). For each $\varpi_\ell \in \varpi$, we assume a pseudocomponent $\mathcal{C}^{(\varpi_\ell)}$ aggregating all the tasks which model the contribution of processors with speed ϖ_ℓ. Formally:

$$\mathcal{T}^{(\varpi_\ell)} \stackrel{\text{def}}{=} \bigcup_{p \leq q} \left\{ \tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)} \mid s_{p,i} = \varpi_\ell \right\} \ . \tag{4.8}$$

Lemma 4.10. *If the set of interface tasks of component $\mathcal{C}_p$ (which is $\mathcal{T}^{(\mathcal{U}_p)}$, defined in Equation (4.8)) is schedulable with umprEDF, then the resource provided to $\mathcal{C}_p$ is guaranteed to comply with $\mathcal{U}_p$, both in terms of supply and platform.*

Proof. We can prove the compliance in terms of supply as we did for gEDF in Theorem 4.5. For each task $\tau_i^{(\mathcal{U}_p)} = (\Pi_p, C_i^{(\mathcal{U}_p)}, \Pi_p)$, not missing a deadline means that task $\tau_i^{(\mathcal{U}_p)}$ receives at least $C_i^{(\mathcal{U}_p)}$ execution units within every period of length Π. Overall, we have that tasks in task set $\mathcal{T}^{(\mathcal{U}_p)}$ receive at least $\sum_{\tau_i^{(\mathcal{U}_p)} \in \mathcal{T}^{(\mathcal{U}_p)}} C_i^{(\mathcal{U}_p)} = \Theta$ (Lemma 4.7) execution units within every period of length Π_p. This is exactly the definition of the supply provided according to a UMPR $\mathcal{U}_p = (\Pi_p, \Theta_p, \pi_p)$.

With respect to the platform, compliance with the component's resource interface comes from the division of the interface tasks in the different pseudocomponents, and from the fact that the interface tasks are paired with the processors considered in the virtual platform π_p. ■

The following theorem directly follows from Lemma 4.10.

Theorem 4.7 (Adequacy of umprEDF for intercomponent scheduling)**.** *If $\mathcal{A}_0$ is umprEDF, the considered scheduling framework (Figure 4.1) is compositional.*

4.4.4 Interface composition

We then derive the resource interface for each pseudocomponent $\mathcal{C}^{(\varpi_\ell)}$ — a UMPR interface $\mathcal{U}^{(\varpi_\ell)} \stackrel{\text{def}}{=} (\Pi_0, \Theta^{(\varpi_\ell)}, \pi^{(\varpi_\ell)})$. Since we are under the assumption that we are working with the physical platform, the processors in $\pi^{(\varpi_\ell)}$ are all fully available. We just need to compute how many of them we will need and $\Theta^{(\varpi_\ell)}$ is set to be equal to $\Pi_0 \cdot S_{m^{(\varpi_\ell)}}(\pi^{(\varpi_\ell)})$ (for an arbitrary Π_0). For this, we iterate on $m^{(\varpi_\ell)}$, considering each platform composed of $m^{(\varpi_\ell)}$ processors with speed ϖ_ℓ, and stopping at the minimum value of $m^{(\varpi_\ell)}$ for which $\mathcal{T}^{(\varpi_\ell)}$ is schedulable. Since the interface tasks are periodic, we can use schedulability tests thereto specific, which are less pessimistic.

From the resource interfaces of the pseudocomponents, we obtain the resource interface for component $\mathcal{C}_0$: $\mathcal{U}_0 = (\Pi_0, \Theta_0, \pi_0)$, where

$$\mathcal{U}_0 = \left(\Pi_0, \sum_{\varpi_\ell \in \varpi} \Theta^{(\varpi_\ell)}, \bigcup_{\varpi_\ell \in \varpi} \pi^{(\varpi_\ell)} \right) \ . \tag{4.9}$$

We now illustrate with a contrived example.

Example

Let us consider two components, $\mathcal{C}_1$ and $\mathcal{C}_2$. We start by abstracting each one with a UMPR interface.

Component $\mathcal{C}_1$ contains the sporadic task set

$$\mathcal{T}_1 = \{(21, 19, 21)\} \ .$$

If we consider the platform $\pi_1 = \{1.0\}$ and set Π_1 to the minimum period in the task set (which is 21), we find that the minimum Θ_1 which guarantees schedulability is 21. Transforming the UMPR interface $\mathcal{U}_1 = (21, 21, \{1.0\})$ into interface tasks, we get the periodic task set

$$\mathcal{T}^{(\mathcal{U}_1)} = \{(21, 21.0, 21)\} \ .$$

Component $\mathcal{C}_2$ contains the sporadic task set

$$\mathcal{T}_2 = \{(20, 3, 20), (20, 4, 20), (20, 4, 20), (20, 1, 20)\} \ .$$

If we consider the platform $\pi_2 = \{1.0, 0.5, 0.5\}$ and set Π_2 to the minimum period in the task set (which is 20), we find that the minimum Θ_2 which guarantees schedulability is 32.5. Transforming the UMPR interface $\mathcal{U}_2 = (20, 32.5, \{1.0, 0.5, 0.5\})$ into interface tasks, we get the periodic task set

$$\mathcal{T}^{(\mathcal{U}_2)} = \{(20, 16.5, 20), (20, 8.0, 20), (20, 8.0, 20)\} \ .$$

The set of unique speeds found among the virtual platforms is $\varpi = \{1.0, 0.5\}$, so we model the umprEDF scheduling of $\mathcal{C}_1$ and $\mathcal{C}_2$ with two pseudocomponents. Pseu-

docomponent $\mathcal{C}^{(\varpi_1)}$ contains the interface tasks $\tau_1^{(\mathcal{U}_1)} = (21, 21.0, 21)$ and $\tau_1^{(\mathcal{U}_2)} = (20, 16.5, 20)$. Conversely, pseudocomponent $\mathcal{C}^{(\varpi_2)}$ contains the interface tasks $\tau_2^{(\mathcal{U}_2)} = (20, 8.0, 20)$, and $\tau_3^{(\mathcal{U}_2)} = (20, 8.0, 20)$. Since these are *periodic* tasks being scheduled upon a dedicated uniform multiprocessor platform, we can use the exact test derived by Funk *et al.* (2001). According to this test, a periodic task set is gEDF-schedulable on a uniform multiprocessor platform with m processors if and only if

- the total utilization of the task set does not exceed the total capacity of the platform; and
- for all $k = 1, \ldots, m$, the sum of the k higher utilizations in the task set does not exceed the sum of the capacities of the k fastest processors in the platform.

With this test, we can assess that pseudocomponent $\mathcal{C}^{(\varpi_1)}$ needs two unit-capacity processors to guarantee schedulability, whereas pseudocomponent $\mathcal{C}^{(\varpi_2)}$ needs two 0.5-capacity processors.

If we select a value of $\Pi_0 = 20$, then the UMPR interfaces of the pseudocomponents are $\mathcal{U}^{(\varpi_1)} = (20, 40, \{1.0, 1.0\})$ and $\mathcal{U}^{(\varpi_2)} = (20, 20, \{0.5, 0.5\})$. Applying Equation (4.9), we obtain the UMPR interface for $\mathcal{C}_0$:

$$\mathcal{U}_0 = (20, 60, \{1.0, 1.0, 0.5, 0.5\}) \ ,$$

which corresponds to a dedicated availability of platform $\pi_0 = \{1.0, 1.0, 0.5, 0.5\}$. Although in this example the platform π_0 is identical to the union of all the virtual platforms, it is possible that it is a subset of this union (formally, $\pi_0 \subseteq \bigcup_{p=1}^{q} \pi_p$), as seen in Figure 4.1.

4.5 Summary

We approached the problem of compositional analysis upon uniform multiprocessor platforms, proposing the Uniform Multiprocessor Resource (UMPR) model to express component interfaces. We extended previous related works, providing results and intuitions which span over the three main points of compositional analysis:

- a sufficient local-level gEDF-schedulability test for sporadic task sets in components with the UMPR as a resource interface;

- guidelines for the complex problem of selecting the virtual platform when abstracting a component; and
- interface composition based on a new intercomponent scheduling algorithm, umprEDF, that solves the inadequacy of using gEDF to schedule components with non-identical multiprocessor virtual platforms.

We do still have room for improvement of these results. We envision providing a quantitative evaluation — with some resource augmentation-based (Phillips *et al.*, 2002) metric — of the UMPR and related theoretical results, as well as tightening the latter to reduce the incurred pessimism and overhead. Furthermore, we intend to assess the potential for other models built upon identical multiprocessors (e.g., Lipari & Bini (2010)'s) to be extended to deal with uniform multiprocessor platforms (albeit with a different compromise between simplicity and bandwidth optimality).

In the next chapter, we present the design, development and use of a proof-of-concept analysis and simulation tool where the formal methods proposed in this chapter can be seen in action. Our tool also allows generating a partition scheduling table to configure systems based on a cyclic partition scheduler, such as those based on the evolved TSP architecture presented in Chapter 3.

Chapter 5

Scheduling Analysis, Generation and Simulation Tool

In this chapter, we describe the design, development and use of *hsSim*, an extensible and interoperable tool that allows analyzing and simulating hierarchically organized multiprocessor real-time systems (such as hierarchical scheduling and resource reservation frameworks), as well as generating partition scheduling tables for the specific case of TSP systems. From the beginning, we imprinted into the development of hsSim some concerns that have echo in the embedded and real-time systems research community (Lipari, 2012). For this reason, we put a significant amount of emphasis on designing our tool to be easily extended and to interoperate with other tools. In Section 5.1, we describe the analysis and design process, focused on applying software design patterns to model specific structures and behaviors of real-time systems. In Section 5.2, we describe the implementation of this design. In Section 5.3, we demonstrate our implementations compositional analysis, scheduling simulation (with visualization through the Grasp Player), and scheduling table generation.

5.1 Object-oriented analysis and design

The implementation of a tool which we want to be flexible, extensible and more easily maintainable must be preceded by careful analysis and design. We will now document the main analysis and design steps and decisions taken, supported by Unified Modelling Language (UML) diagrams where deemed necessary.

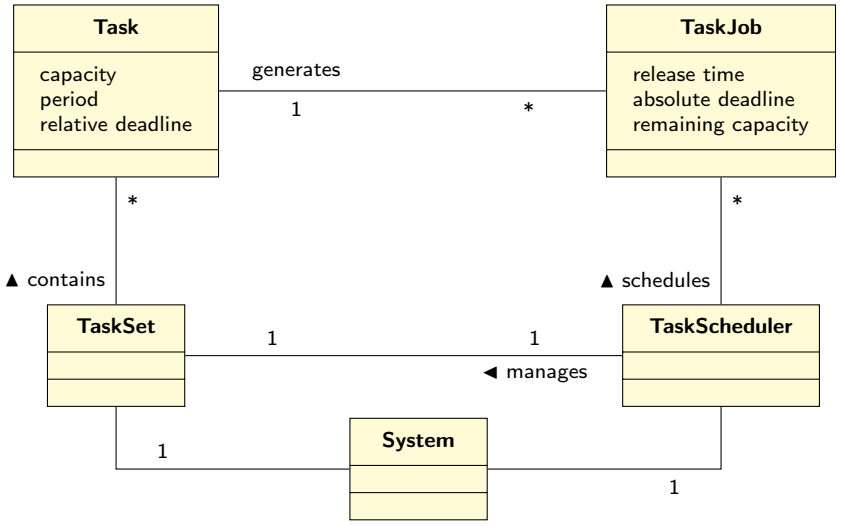


Figure 5.1: Traditional 1-level system domain model

5.1.1 Domain analysis

The traditional real-time system, as described and surveyed in Section 2.1, is flat — a one-level hierarchy. The UML diagram for such a system's domain is pictured in Figure 5.1. The system has a flat task set and a task scheduler.

A two-level hierarchical scheduling framework, such as those corresponding to TSP systems (Section 2.3.2.2), can be modelled as seen in Figure 5.2. The system has a set of partitions and a root scheduler coordinating which partition is active at each instant. Each partition then has a set of tasks and a local scheduler to schedule the latters' jobs. This domain model strategy has two main drawbacks:

1. it is hard-limited to two levels; and
2. it only allows homogeneous levels (i. e., partitions and tasks cannot coexist at the same level), and this is thus not applicable to similar system models such as resource reservation frameworks (Section 2.3.1)

5.1.2 n-level hierarchy: the Composite pattern

The *Composite* pattern is a design pattern that may be used when there is a need to represent part–whole hierarchies of objects, and allow clients to ignore the differences between composition of objects and individual objects Gamma *et al.* (1997).

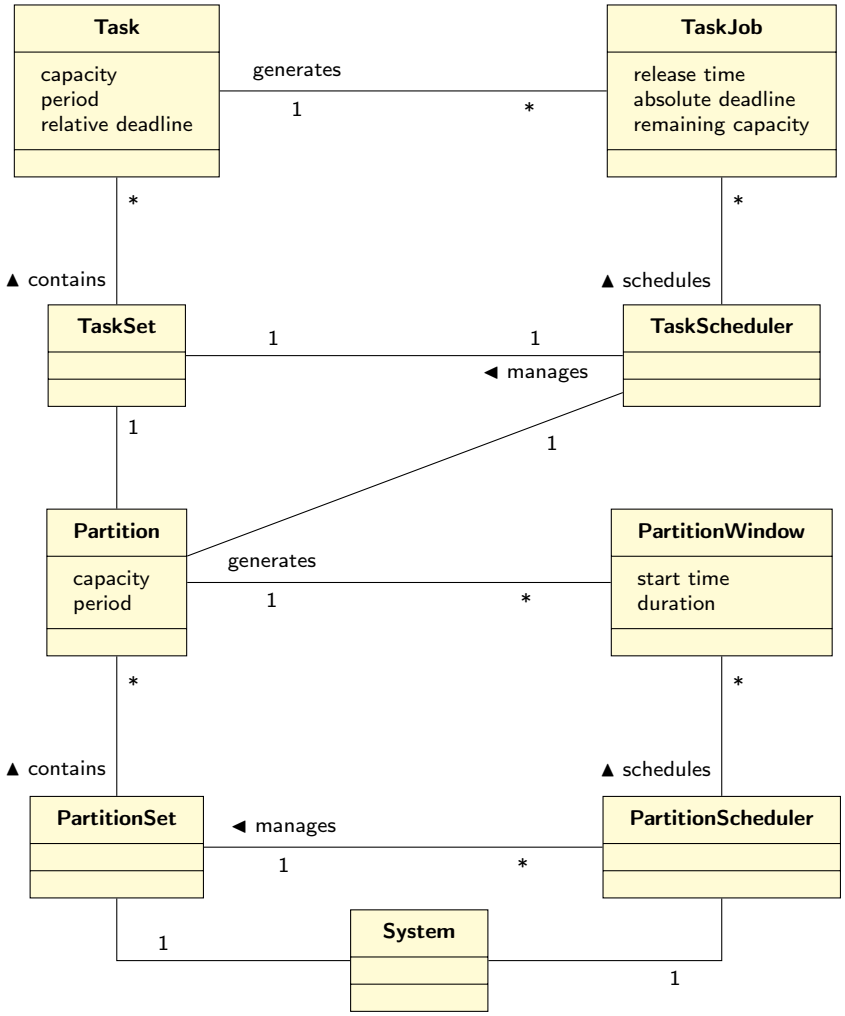


Figure 5.2: 2-level hierarchical scheduling system domain model

Clients manipulate objects in the composition through a component interface, which abstracts individual objects and compositions.

Figure 5.3 shows the UML representation of the Composite pattern as applied to our domain. Applying this pattern to our model of a hierarchical scheduling framework allows breaking two limitations: the fixed number of levels in the hierarchy (Easwaran *et al.*, 2007; Shin & Lee, 2003), and the need for the hierarchy to be balanced (Abeni & Buttazzo, 1998). The clients of the IAbsSchedulable interface include schedulers, which will be able to schedule both tasks and partitions through a common interface, reducing implementation efforts and allowing extensions through

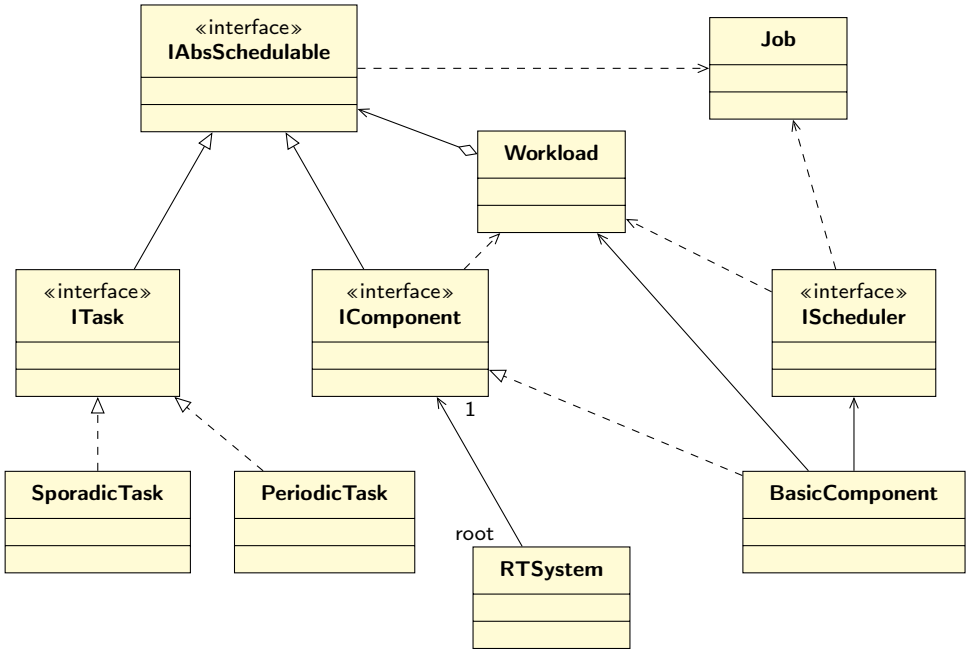


Figure 5.3: *n*-level hierarchical scheduling system using the Composite pattern

new schedulable entities (e. g., servers). The application of this pattern triggered further refinements, such as making the TaskSet the System's and Partitions' IAbsSchedulables container, and merge task and partition activations under a generic Job abstraction.

5.1.3 Scheduling algorithm encapsulation: the Strategy pattern

The *Strategy* (or *Policy*) pattern is an appropriate solution to when we want to define a family of algorithms which should be interchangeable from the point of view of their clients Gamma *et al.* (1997). In designing hsSim, we apply the Strategy pattern to encapsulate the different scheduling algorithms, as seen in Figure 5.4. In the Scheduler abstract class, although we use a scheduling policy to initialize the JobQueue, we leave the obtention of the scheduling policy (the getPolicy() method) to the concrete scheduler classes; this is supported on another well-known design pattern, the Template Method Gamma *et al.* (1997).

The SchedulingPolicy interface extends Java's Comparator interface; this way, an

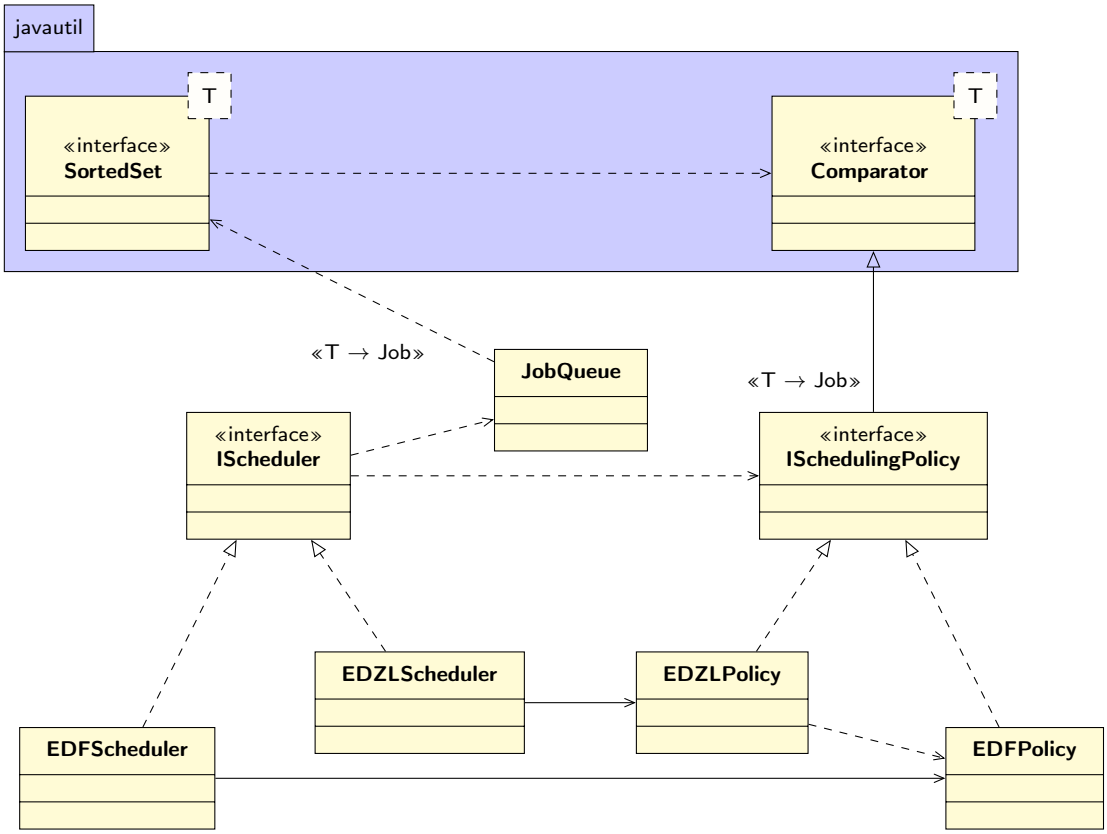


Figure 5.4: Scheduling algorithm encapsulation with the Strategy pattern

instance of a subclass of SchedulingPolicy can be used to maintain the scheduler's job queue ordered in the manner appropriate for the scheduling algorithm being implemented.

The available strategies (scheduler types) are stored in a catalog, and more strategies can be loaded in runtime (provided the user interface gives a means to it). This is made possible by Java's native reflection capabilities.[1]

5.1.4 n -level hierarchy and polymorphism

Due to the design decisions regarding the Composite and Strategy patterns, most operations can implemented without having to account for which scheduler (or schedulers) are present, or for the structure and/or size of the hierarchy (partitions and

[1]Since we anticipated using the Java to implement hsSim, this and the following design decisions take explicit advantage from facilities provided by the Java libraries.

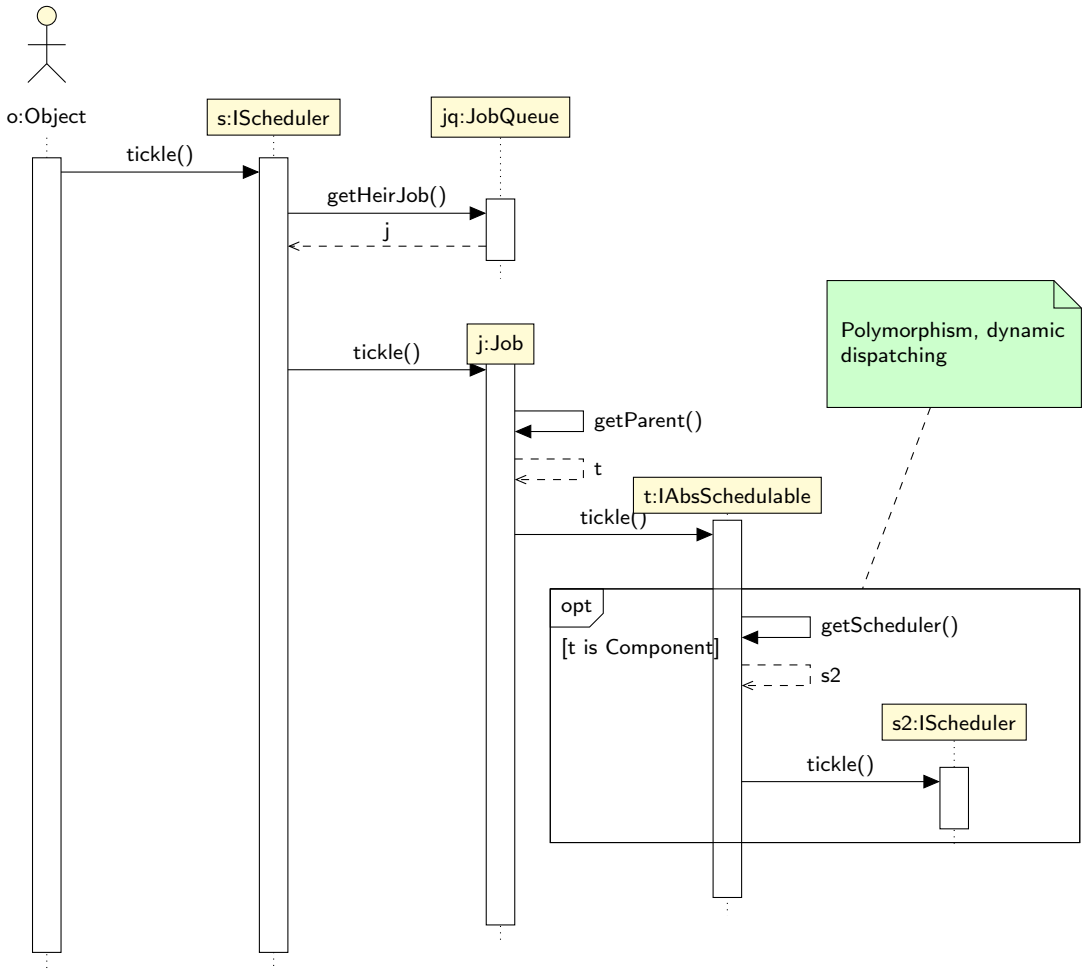


Figure 5.5: Sequence diagram for the scheduler tickle operation

tasks). Taking advantage of subtype polymorphism, we can invoke methods on Scheduler and AbstractTask references instances without knowing of which specific subtype thereof the instances are.

Let us see how this works with the scheduler tickle operation, which simulates the advance of system execution by one time unit. Currently, we implement hsSim as a cycle-step execution simulator; an event-driven approach (Nikolic *et al.*, 2011) is planned for future work. The UML sequence diagram modeling the interactions between objects in this operation is shown in Figure 5.5. The hierarchical tickle process is started by invoking the tickle operation on the root scheduler without specific regard for what subtype of Scheduler it is; the right job to execute will be

obtained because the scheduler's job queue is maintained accordingly ordered by an instance of an unknown SchedulingPolicy subtype. This job is then tickled, and in turns tickles its parent AbstractTask without knowing if it is a Task or a Partition. It is the job's parent's responsibility to invoke the right behaviour according to its type. If it is a Partition instance, this involves tickling its scheduler; this will cause an identical chain of polymorphic invocations to take place.

5.1.5 Decoupling the simulation from the simulated domain using the Observer and Visitor patterns

In hsSim, we want to decouple the simulation aspects (such as running the simulation and logging its occurrences) from the simulated domain itself. On the one hand, we want changes in the simulated domain (a system with partitions, tasks, jobs) to be externally known of, namely by one or more loggers, without the domain objects making specific assumptions about these loggers behaviour or interfaces. On the other hand, we want to be able to create new loggers without tightly coupling them to the domain objects or having to modify the latter. We found the Observer and Visitor patterns to be most appropriate to solve this specific problem. In few words, the Observer pattern allows loggers to register themselves as interested in receiving events, and the Visitor pattern helps each logger define what to do with each kind of event. Let us now see the application of these patterns in detail.

The *Observer* pattern defines a publisher–subscriber dependency between objects, so that observers (subscribers) are notified automatically of the state changes of the subjects they have subscribed to. The subjects only have to disseminate their state changes to a vaguely known set of observers, in a way that is totally independent of how many observers there are and who they are — in the form of events. We take advantage from the simple Observer implementation provided by Java, with the Logger interface extending the Observer interface, as pictured in Figure 5.6. This way, classes implementing the Logger interface (the concrete loggers) must provide the method to be called to notify the respective logger of an event.

The *Visitor* pattern defines a way to represent an operation to be performed on an object hierarchy independently from the latter Gamma *et al.* (1997). The Logger interface also extends our EventVisitor interface, which defines methods to process

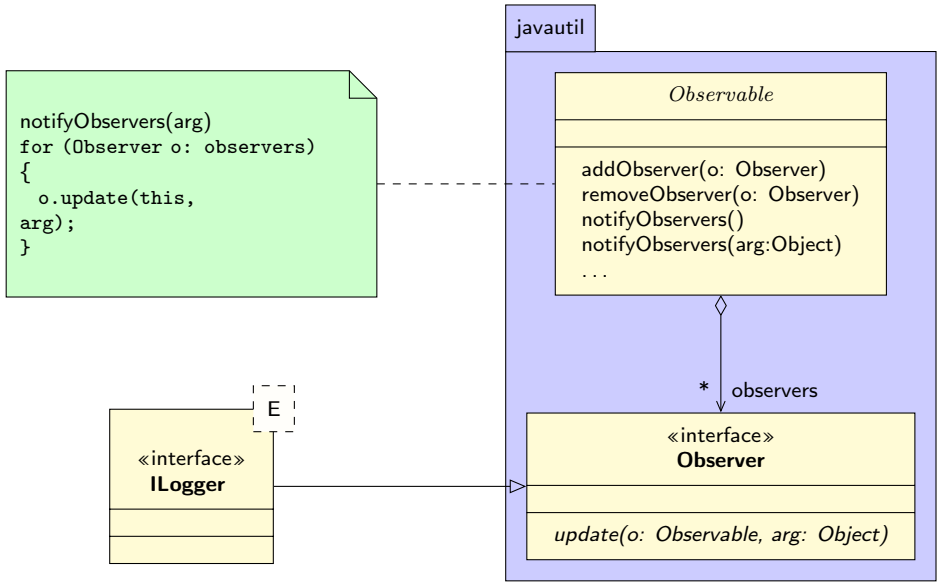


Figure 5.6: Application of the Observer pattern for loggers

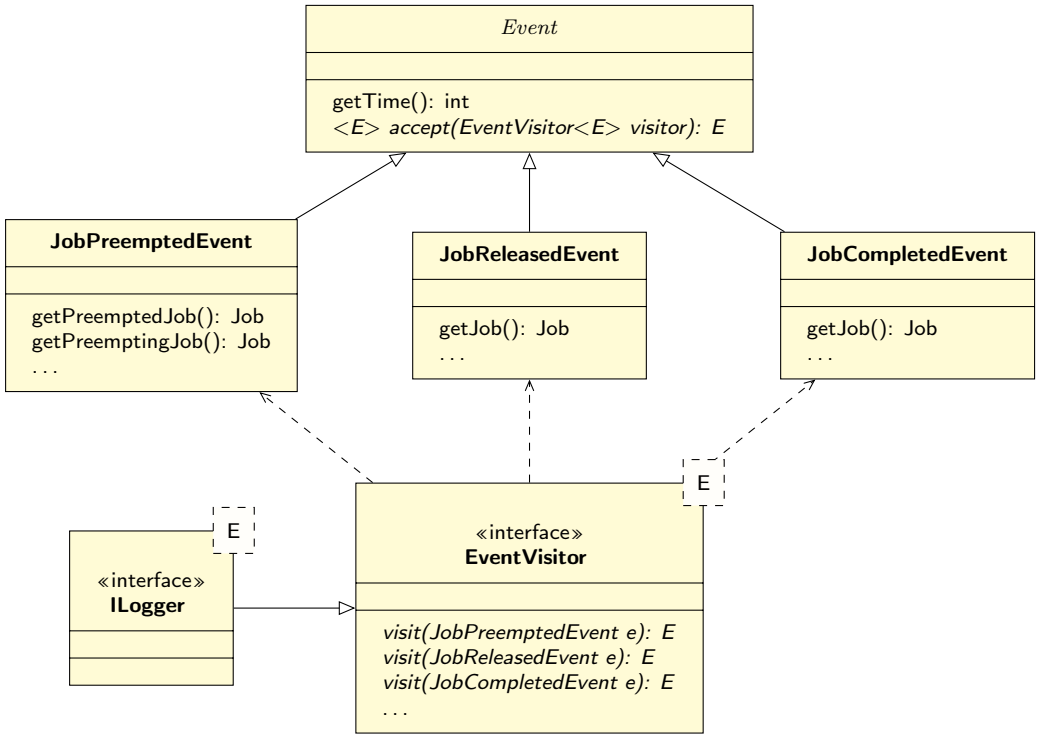


Figure 5.7: Application of the Visitor pattern for loggers

each type of event. The visit methods are overloaded and every Event subclass

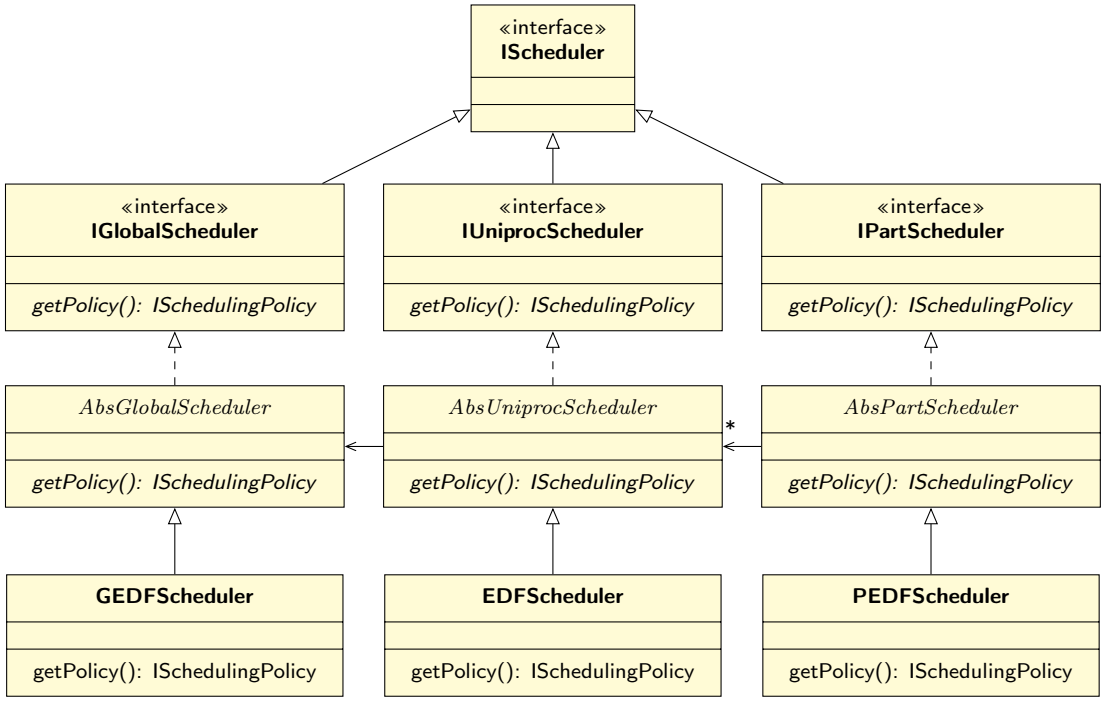


Figure 5.8: Multiprocessor schedulers

provides the following method:

```
<E> E accept(EventVisitor<E> visitor) {
    visitor.visit(this);
}
```

This way, when receiving an event `e`, a logger only has to invoke

```
((Event) e).accept(this)
```

to have the right visit method called.

We also apply the Observer pattern to establish a dependency between the system clock and the schedulers, so that the latter become aware of when to released their tasks' jobs.

5.1.6 Multiprocessor schedulers

As we have seen in Section 2.1.3, the degree of migration that a multiprocessor scheduling algorithm permits to the jobs it schedules allows categorizing it as either partitioned or global. Uniprocessor schedulers can be seen as a particular case

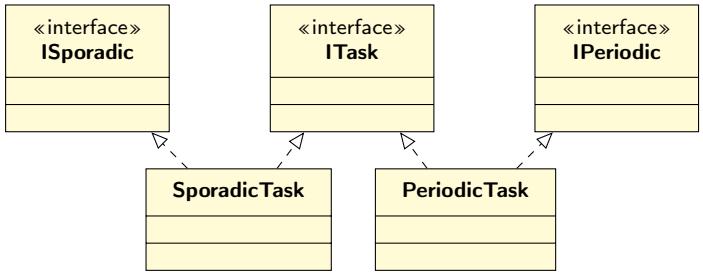


Figure 5.9: Interfaces implemented by the periodic and sporadic task classes.

of either one operating on a platform with one unit-capacity processor. We have already seen that the problem of partitioned multiprocessor scheduling reduces, after partitioning tasks into the m available processors, to m uniprocessor scheduling problems. Considering these properties, we can reduce the complexity of implementing scheduling algorithms upon hsSim by implementing

1. uniprocessor schedulers at the expense of global schedulers; and
2. partitioned schedulers at the expense of a collection of uniprocessor schedulers.

This way, we concentrate most of the complexity on the global schedulers. We implement a global work-conserving scheduler on uniform multiprocessor platforms as the abstract class AbsGlobalScheduler and rely on instances of this class to implement the AbsUniprocScheduler and AbsPartScheduler abstract classes. Concrete schedulers can then be derived by extending the proper abstract class and implementing the method which returns the scheduling policy.

5.1.7 Interfaces aiding scheduling analysis

Classes representing schedulable entities, such as tasks and components, implement the respective interfaces that dictate the minimum set of methods to be provided. However, they can also implement other interfaces, which may or may not require the classes to provide additional methods. Instead, these accessory interfaces can serve to express certain properties of the schedulable entity being implemented. In Figure 5.9, we provide an example where we show the hierarchy of interfaces involved in how we implement, in hsSim, periodic and sporadic tasks. Taking the periodic task as an example, the refactoring of its "is periodic" and "is task" aspects into two

separate interfaces allows having tasks which are not periodic (e. g., sporadic tasks) and periodic entities which are not tasks (namely, periodic components).

The rationale for the introduction of these interfaces is the potential to aid the selection of the appropriate scheduling analysis methdology — for instance, verifying if a task set being verified for schedulability complies with the system model the schedulability test assumes (Gaudel *et al.*, 2013). With these interfaces, coupled with the multiprocessor scheduler taxonomy that we described in Section 5.1.6, these aids can be implemented resorting to Java's instanceof primitive, applied to the schedulable entities (e. g., tasks) and to the scheduler(s) of the target system. We have implemented so far only compositional analysis techniques, so the potential of this approach is not yet fully noticeable.

5.1.8 Compositional analysis with the Decorator pattern

The Decorator pattern (Gamma *et al.*, 1997) allows adding responsibilities to objects of a given class without extending it. There are many valid reasons to avoid relying on inheritance, especially when there is no full access to the class being augmented. Without full knowledge of the implementation of the latter, we may be unknowingly overriding methods which are used by other methods (thus modifying their behavior). It may also be the case that we want to add responsibilities to a class which cannot be extended (final modifier, in Java). Finally, adding responsibilities through decoration instead of inheritance also allows circumventing the absence of multiple inheritance in most object-oriented programming languages.

We employ the Decorator pattern to add compositional analysis aspects to the notion of component, as pictured in Figure 5.10. The CompositionalComponent class implements the IComponent interface, and decorates an instance of a class that also implements the IComponent interface (in our case, the BasicComponent class). Instances of the CompositionalComponent class are not instances of the BasicComponent class, but rather hold a reference to an instance of the BasicComponent class to preserve and reuse the latter's behavior when possible.

The additional responsibilities added by the CompositionalComponent class are those required by the ICompositional interface, which depend on the use of a resource model. The UMPR class implements the IResourceModel interface, and each instance

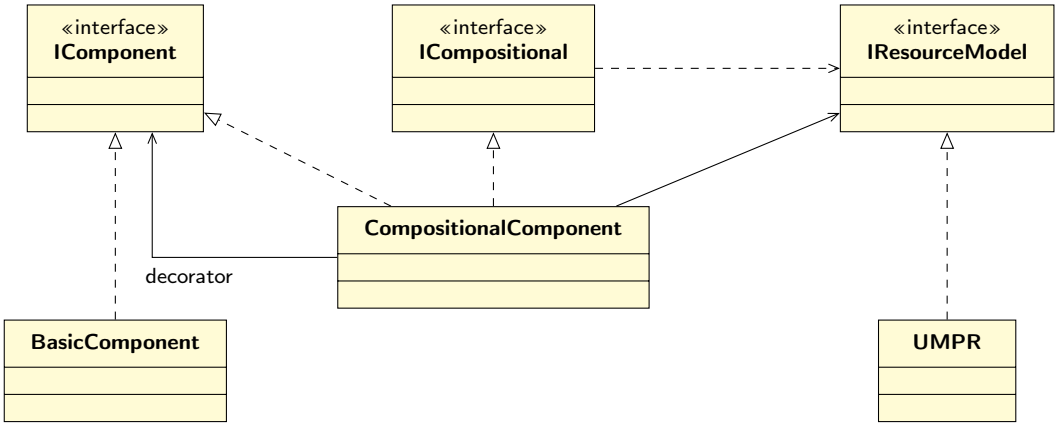


Figure 5.10: Support for compositional analysis with the Decorator pattern

of it represents a resource interface expressed with the *uniform multiprocessor periodic resource* model we proposed in Section 4.1.

5.2 Implementation

A first iteration, where the core design principles described in Sections 5.1.1 to 5.1.5 are implemented, was produced in colaboration with Silveira (2012). We then refactored such implementation to add support to multiprocessor and compositional analysis (Sections 5.1.6 to 5.1.8).

We now highlight the implementation of extensions to the core of hsSim, geared towards interoperability. These take advantage of the careful design of the hsSim core, and we did implement them in separate packages (to emulate the development by a third-party with access to the hsSim as a library).

5.2.1 Extensions

5.2.1.1 umprEDF Scheduler

In Section 4.4.3, we have proposed the umprEDF algorithm intercomponent scheduling, to overcome the anomalies incurred by gEDF when scheduling components over uniform multiprocessor platforms. We have implemented it upon the hsSim core, with a ready queue for each considered unique speed.

Table 5.1: Mapping between hsSim events and Grasp trace content.

Event	hsSim class	Grasp log
Job arrives	JobReleasedEvent	If job of a task: jobArrived If job of a component: serverReplenished
Job obtains processor	JobPreemptionEvent[a]	If job of a task: jobResumed If job of a component: serverResumed
Job preempted by another	JobPreemptionEvent	If job of a task: jobPreempted[b] If job of a component: serverPreempted[b]
Job completes execution	JobCompletedEvent	If job of a task: jobCompleted If job of a component: serverDepleted

[a] With a *null* preempted event.

[b] Plus the appropriate jobResumed/serverResumed event.

5.2.1.2 Grasp logger

Because of our low coupling design for the event logging, it is straightforward to trace the simulation to the format interpreted by Grasp Player (Holenderski *et al.*, 2013). When we started this work, Grasp Player did not support uniform multiprocessors, namely with respect to tracing the component's budget consumption according to rates different from one unit of budget per unit of time; we have implemented this functionality on Grasp Player.[2]

As said, the Grasp logger is implemented exactly as if it were developed by an external team, who could even only have access to the hsSim core as a library. We implement the Logger generic interface, instantiating its type variable with the String type, since we want the processing (visit) of events to return the text to be added to the Grasp trace. The visit methods are invoked when an event notification is received, via the update method. Invocation is done indirectly through the accept method, so the right visit method is automatically selected and called.

Along the visit methods, we implement the mapping between hsSim events and Grasp trace content shown in Table 5.1. Listing 5.1 shows an example excerpt of a trace generated by hsSim's Grasp Logger.

[2]We provided our patch to the developers of Grasp, and it was merged in version 1.8.

Listing 5.1: Grasp trace excerpt

```
...
plot 0 jobArrived job_task11_1 task11
plot 0 serverReplenished C1 21.0
plot 0 jobArrived job_task21_1 task21
plot 0 jobArrived job_task22_1 task22
plot 0 jobArrived job_task23_1 task23
plot 0 jobArrived job_task24_1 task24
plot 0 serverReplenished C2 32.5
plot 0 serverReplenished C2 32.5
plot 0 serverReplenished C2 32.5
plot 0 serverResumed C2 -rate 0.5
plot 0 serverResumed C2 -rate 0.5
plot 0 serverResumed C1 -rate 1.0
plot 0 serverResumed C2 -rate 1.0
plot 0 jobResumed job_task11_1 -processor coreproc2
plot 0 jobResumed job_task23_1 -processor coreproc4
plot 0 jobResumed job_task22_1 -processor coreproc3
plot 0 jobResumed job_task21_1 -processor coreproc1
plot 3 jobCompleted job_task21_1 -processor coreproc1
plot 3 jobPreempted job_task22_1 -target job_task24_1 -processor coreproc3
plot 3 jobResumed job_task24_1 -processor coreproc3
plot 3 jobResumed job_task22_1 -processor coreproc1
plot 5 jobCompleted job_task24_1 -processor coreproc3
...
```

5.2.1.3 ARINC 653 logger

We implemented the generation of ARINC 653-inspired partition scheduling tables in a similar manner. There are nevertheless two major differences. The first is that our ARINC 653 Logger is only concerned with events pertaining to components (i.e., partitions), and not tasks. The second difference is that, unlike the Grasp Logger (where events are immediately logged), we have to keep state, at each moment, of the currently active windows' — so that, when an event signals that this window is over, the start and end of the window can be coupled and logged.

5.2.1.4 Worst-Case Response Time logger

Finally, we implemented a simple logger which only considers the job completion events, and uses them to maintain a record of the worst-case response times of the tasks. At the end of the simulation, it provides a table with the values.

5.3 Example use case

We now use an example use case to provide a deeper explanation of the scheduling analysis, simulation and schedule generation functionalities that we have implemented. This example also serves the purpose of demonstrating the formal methods proposed and results described in Chapter 4. For this reason, let us consider the same example that we saw in Section 4.4.4: two components, $\mathcal{C}_1$ and $\mathcal{C}_2$, containing gEDF-scheduled sporadic task sets

$$\mathcal{T}_1 = \{(21, 19, 21)\} \text{ and } \mathcal{T}_2 = \{(20, 3, 20), (20, 4, 20), (20, 4, 20), (20, 1, 20)\} \ ,$$

respectively.

5.3.1 Scheduling analysis

With the period and virtual platform for each component's resource interface being provided by the system designer, hsSim calculates (in the case of the UMPR model) the minimum value for Θ that guarantees schedulability of the enclosed task set. As before, we will use the least minimum interarrival time among each task set as the period for the respective component. With those values, we obtain the same resource interfaces as we saw in Section 4.4.4: $\mathcal{U}_1 = (21, 21, \{1.0\})$ and $\mathcal{U}_2 = (20, 32.5, \{1.0, 0.5, 0.5\})$. hsSim automatically generates the corresponding interface tasks, which will then be used to release jobs of the component.

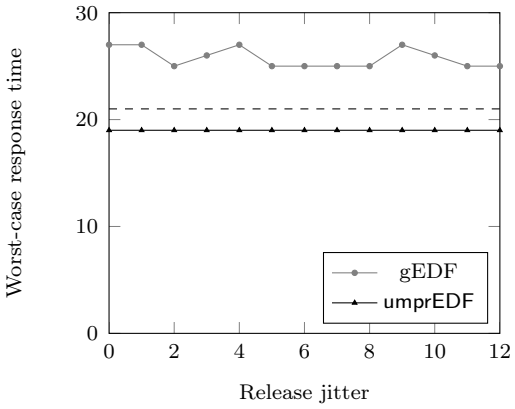
5.3.2 Scheduling simulation and visualization

We have simulated the execution of the system comprising components $\mathcal{C}_1$ and $\mathcal{C}_2$ with both gEDF and umprEDF intercomponent scheduling algorithms. The results of the simulation are provided by the considered loggers. In our case, we employed loggers to register worst-case response times, to create a trace to be visualized with Grasp, and to generate an ARINC 653-like partition scheduling table. We now analyze the information provided by the worst-case response time and Grasp loggers; we analyze the partition scheduling table generation in the next section.

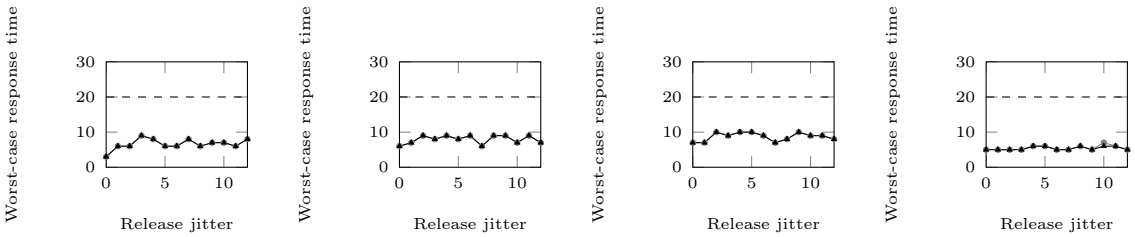
Sporadic tasks release their jobs with a minimum separation, whereas periodic tasks release their jobs with a constant separation. For this reason, to have results closer to the reality of sporadic tasks, we have introduced a fixed release *jitter*. We have performed simulations with jitters between 0 and 12. The semantics of this fixed jitter (let us denote it as j) is that every job of a task $\tau_{p,i}$ is released $T_{p,i} + j'$ time units after the previous job of the same task, where j' is a random integer number in the interval $[0, j]$. The values of j' are generated by per-task random number generators, which are initialized with a task-specific seed when the task is created; this is to allow the job releases to happen at the same time instants when simulating with each of the intercomponent scheduling algorithms (as thus allow the results to be comparable).

In Figure 5.11, we can see the worst-case response times of the five tasks in the system; we highlight task $\tau_{1,1}$, since the remaining tasks have the same worst-case response time with gEDF and with umprEDF. The dashed horizontal line in each graph represents the respective task's relative deadline. We see that, when using gEDF for intercomponent scheduling, task $\tau_{1,1}$ yielded a worst-case response time longer than its relative deadline, which means that one or more of its jobs missed their deadlines.

What we see here is the consequence of the phenomenon we predicted analytically in Section 4.4.2. Although the components are abstracted with resource interfaces derived to guarantee schedulability, gEDF does not guarantee that the components have the resource supply therein expressed (namely in terms of virtual platform). We can verify this by visualizing, in Grasp, the trace recorded for this simulation, shown in Figure 5.12). The execution of the task in component $\mathcal{C}_1$ is traced in dark gray bars, whereas that of tasks in component $\mathcal{C}_2$ is traced in light gray bars. At the bottom of the figure, we have traced the remaining budget of each component, which corresponds to the total remaining execution time of the interface tasks of its resource interface. We can see, right at time instant 0, that the tasks in $\mathcal{C}_2$ take over the three fastest processors, including two 1.0-speed processors, although the virtual platform considered for it had only one 1.0-speed processor. Component $\mathcal{C}_1$ is relegated to the fourth processor, with speed $s_{0,4} = 0.5$, although the resource interface $\mathcal{U}_1$ only considered one processor of speed 1.0. This anomaly has an effect on the resource provision given to component $\mathcal{C}_1$. As can be seen in the budget traces



(a) Task $\tau_{1,1}$



(b) Task $\tau_{2,1}$ **(c)** Task $\tau_{2,2}$ **(d)** Task $\tau_{2,3}$ **(e)** Task $\tau_{2,4}$

Figure 5.11: Worst-case response time

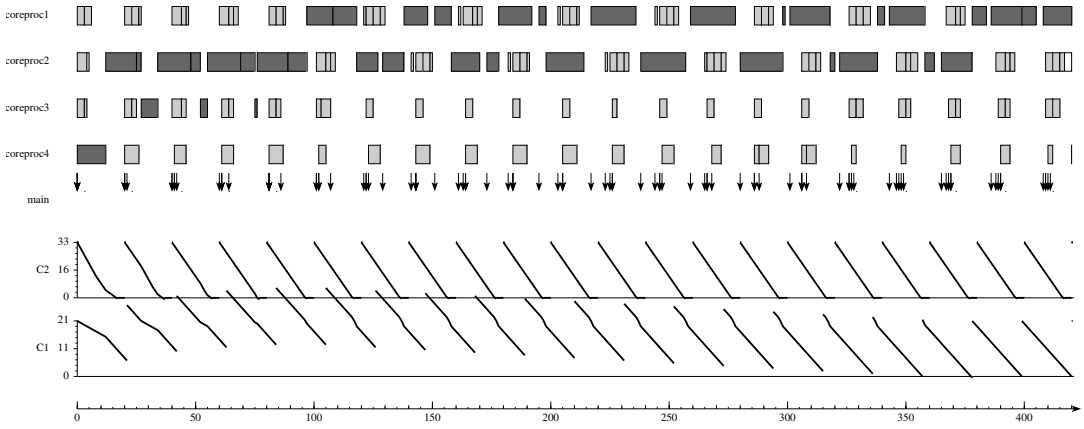


Figure 5.12: Grasp trace of the simulation with gEDF global-level scheduling (jitter = 1).

at the bottom of the figure, does not reach zero every 21 time units, which means $\mathcal{C}_1$ is not receiving the periodic resource supply specified in UMPR interface $\mathcal{U}_1$. Eventually, task $\tau_{1,1}$ misses its deadline at time instant 21 (its first job only finishes

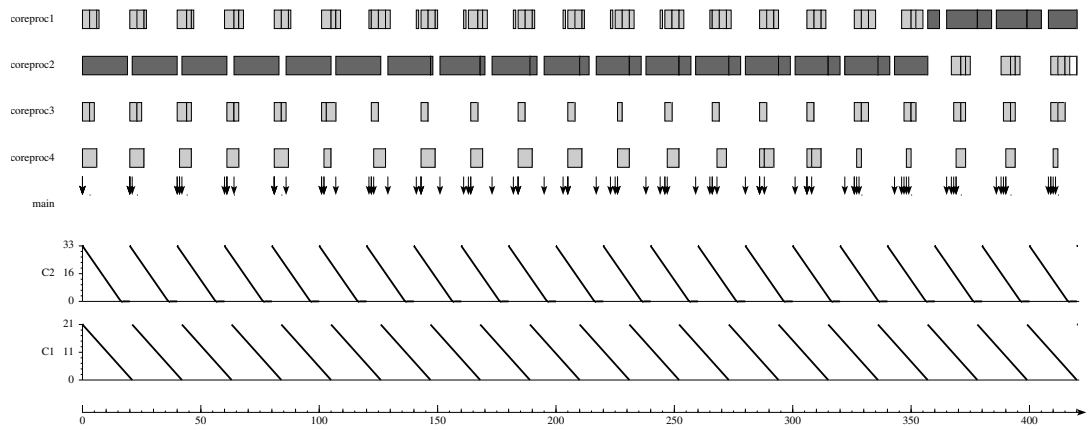


Figure 5.13: Grasp trace of the simulation with umprEDF global-level scheduling (jitter = 1).

execution at time instant 25); the second job begins execution right afterwards, but component $\mathcal{C}_1$ is again relegated, for some time, to a slower processor, and this job also misses its deadline (yielding a response time of 27 time units).

When executing with umprEDF as a global-level intercomponent scheduler, as shown in the trace in Figure 5.13, we see that the virtual platforms of the components' resource interfaces are kept. At each instant, component $\mathcal{C}_2$ never schedules its tasks on a virtual platform other than one 1.0-speed processor and two 0.5-speed processors. As a consequence, $\mathcal{C}_1$ is never relegated to a 0.5-speed processor, and receives the periodic resource supply specified in its UMPR interface.

We should point out that if, we schedule the tasks of $\mathcal{T}_1$ and $\mathcal{T}_2$ together with gEDF on the physical platform π_0, we may have deadline misses, despite the considerable difference between the tasks' total utilization (which is less than 1.51) and the platform's total capacity (that is 3.0). In Figure 5.14 we show the Grasp trace of the execution sequence that occurs when the first job of all tasks arrives at time instant 0. In a clear expression of the Dhall effect we have reviewed in Section 2.2.2 (Dhall & Liu, 1978), the job of task $\tau_{1,1}$ is pushed forward by the remaining tasks, and ends up missing its deadline at time instant 21. This highlights one of the advantages of hierarchical scheduling frameworks: separation, for the purpose of temporal containment, of tasks which may differ in one or more aspects, including their total utilization.

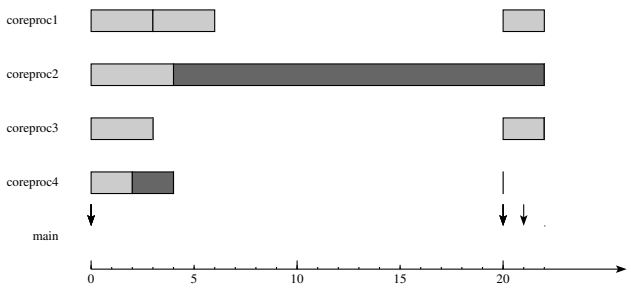


Figure 5.14: Grasp trace for the simulation of task set $\mathcal{T}_1 \cup \mathcal{T}_2$ being scheduled with gEDF directly on the physical platform.

5.3.3 Schedule generation

The type of XML file generated by hsSim is illustrated in Listing 5.2. As in the format prescribed by the ARINC 653 specification, the Module_Schedule node contains the partition scheduling table itself, whereas as the remaining nodes contain configuration information; in our case, the latter includes a new node, Module_Platform, pertaining to the processors.

In the ARINC 653 format, the Module_Schedule node contains the partition scheduling table divided in Partition_Schedule sub-nodes. We retain this structure, since it is convenient for verification purpose; the set of scheduling windows can be analyzed to assess if it adds up to the partition's periodic requirement (expressed in the PeriodExecutionRequirement and Period attributes). We do however add an alternative representation of the partition scheduling table, tailored at the online scheduling process itself. For the generation of this representation, periods where a processor was not (in the logged simulation) assigned to any partition are assigned to the last partition to which the processor was assigned. This allows simplifying the scheduling decisions taken in execution time and drastically reducing the number of partition preemptions.

5.4 Summary

In this chapter, we have described the design, development and use of *hsSim*, a scheduling analysis, generation and simulation tool with a focus on hierachical scheduling, extensibility and interoperability. The two latter aspects were dealt

Listing 5.2: Sample

```
<Module>
  <Module_Schedule MajorFrame="420.0">
    <Processor_Schedule ProcessorIdentifier="proc1">
      <Window_Schedule WindowDuration="357.0" WindowPartitionIdentifier="C2"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="63.0" WindowPartitionIdentifier="C1"
          WindowStart="357.0"/>
    </Processor_Schedule>
    <Processor_Schedule ProcessorIdentifier="proc2">
      <Window_Schedule WindowDuration="360.0" WindowPartitionIdentifier="C1"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="60.0" WindowPartitionIdentifier="C2"
          WindowStart="360.0"/>
    </Processor_Schedule>
    ...
    <Partition_Schedule PartitionIdentifier="C2" PartitionName="C2" Period="20.0"
        PeriodExecutionRequirement="32.5">
      <Window_Schedule WindowDuration="16.0" WindowProcessorIdentifier="proc3"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="16.0" WindowProcessorIdentifier="proc4"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="17.0" WindowProcessorIdentifier="proc1"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="16.0" WindowProcessorIdentifier="proc3"
          WindowStart="20.0"/>
      ...
    </Partition_Schedule>
    <Partition_Schedule PartitionIdentifier="C1" PartitionName="C1" Period="21.0"
        PeriodExecutionRequirement="21.0">
      <Window_Schedule WindowDuration="21.0" WindowProcessorIdentifier="proc2"
          WindowStart="0.0"/>
      <Window_Schedule WindowDuration="21.0" WindowProcessorIdentifier="proc2"
          WindowStart="21.0"/>
      ...
    </Partition_Schedule>
  </Module_Schedule>
  <Partition PartitionIdentifier="C2" PartitionName="C2"/>
  <Partition PartitionIdentifier="C1" PartitionName="C1"/>
  <Module_Platform>
    <Processor ProcessorIdentifier="proc1" ProcessorSchedulableUtilization="1.0"/>
    <Processor ProcessorIdentifier="proc2" ProcessorSchedulableUtilization="1.0"/>
    <Processor ProcessorIdentifier="proc3" ProcessorSchedulableUtilization="0.5"/>
    <Processor ProcessorIdentifier="proc4" ProcessorSchedulableUtilization="0.5"/>
  </Module_Platform>
</Module>
```

with through the careful application of software design patterns. We have implemented support for compositional analysis, namely the formal methods we proposed in Chapter 4, and extensions to interoperate with the Grasp trace player (to graphically visualize the timeline of the simulation) and with ARINC 653-like TSP systems (through the generation of an XML file containing the partition scheduling table). Finally, we have shown all these capabilities in action through an example

use case, where we also highlight the observance of the effects that we had analytically predicted in Chapter 4 — which motivated the proposal of umprEDF as an intercomponent scheduling algorithm.

The improvements we foresee for hsSim are mostly related to increasing its user-friendliness. We aim to implement a graphical user interface and to consolidate the ability (not described in this dissertation) of importing systems to simulate from a file. With these improvements, we expect to soon release a first public version of hsSim, under a free license, on the Google Code project already created for the effect.[3] This way, researchers on real-time scheduling can profit from an environment where new algorithms and formal methods can be easily prototyped and evaluated.

[3] https://code.google.com/p/hssim

Chapter 6

Towards Self-Adaptation in Time- and Space-Partitioned Systems

In this chapter, we present preliminary results on self-adaptation in TSP systems. These results serve the purpose of sustaining our statement that the active exploitation of employing multiple processor cores can open room for supporting self-adaptive behavior to cope with unforeseen changes in operational and environmental conditions (Section 1.3).

We begin, in Section 6.1, by describing with more detail the monitoring and adaptation mechanisms in the AIR architecture (which we presented in Section 3.1.2).[1] We also perform an evaluation of these mechanisms on a prototype implementation of an AIR system. Then we propose how to take advantage of these mechanisms to achieve self-adaptive behavior in TSP systems, and evaluate our proposal through simulation experiments (Section 6.2). In Section 6.3, we discuss extensions which can improve the coverage of the described self-adaptation approach, based on multiprocessor platforms, reconfiguration, and proactive fault tolerance.

6.1 Monitoring and adaptation mechanisms

We now describe AIR's monitoring (deadline violation detection) and adaptation (mode-based schedules) mechanisms, and their relation to ARINC 653-related as-

[1] These mechanisms were first introduced in the scope of the author's Master thesis (Craveiro, 2009) and consolidated in the scope of the hereby described work.

Algorithm 1 Deadline verification at the AIR PAL level

```
1: PAL_ClockTickAnnounce(elapsedTicks)
2: for all (τ_i, d_i) ∈ PAL_deadlines do
3:     if d_i >PAL_GetCurrentTime( ) then
4:         break
5:     end if
6:     HM_DeadlineViolated(τ_i)
7:     PAL_RemoveDeadline(τ_i)
8: end for
```

pects of the AIR architecture, such as Health Monitoring and the APEX interface.

6.1.1 Task deadline violation monitoring

During system execution, it may be the case that a task exceeds its deadline. This can be caused by a malfunction, by transient overload (e. g., due to abnormally high event signalling rates), or by the underestimation of a task's worst case execution requirement at system configuration and integration time. Factors related to faulty system planning (such as the time windows not satisfying the partitions' timing requirements) could, in principle, also cause deadline violations; however, such issues should be predicted and avoided using offline tools that verify the fulfilment of timing requirements. In addition, it is also possible that a task exceeds a deadline while the partition in which it executes is inactive. This violation will only be detected when the partition is being dispatched, just before invoking the task scheduler.

Deadline verification, which is invoked for the currently active partition immediately following the announcement of elapsed system clock ticks, obeys to Algorithm 1. In the absence of a violation, only the earliest deadline is checked. Subsequent deadlines may be verified until one has not been missed. This methodology is optimal with respect to deadline violation detection latency; the upper bound for this latency, for each partition, is the maximum interval between two consecutive time windows.

Figure 6.1 provides a use-case scenario for the task deadline violation monitoring functionality. The exemplified task τ_i has a relative deadline of $t_3 - t_1$. When it becomes ready at time instant t_1 via the START primitive, its absolute deadline time is set to t_3. At time instant t_2, the REPLENISH primitive is called, requesting a deadline time replenishment, so that the new absolute deadline time is t_4. When the time instant t_4 arrives and the task τ_i has not yet finished its execution (e. g.,

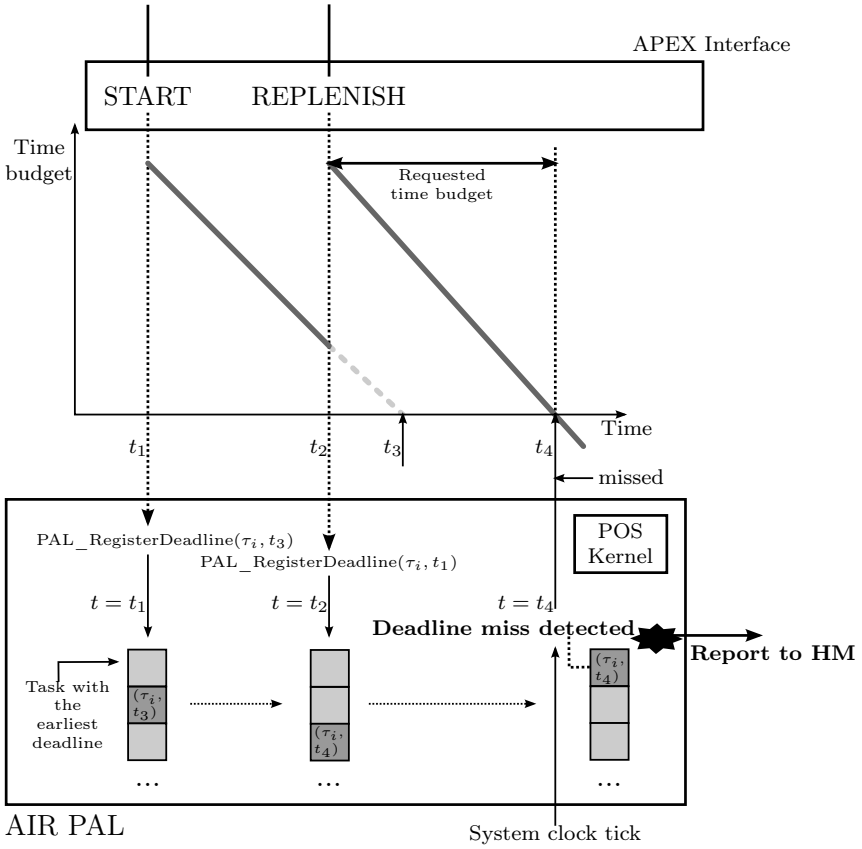


Figure 6.1: Deadline violation monitoring example

having called the PERIODIC_WAIT primitive to suspend itself until its next release point), its deadline will be the earliest registered in the respective AIR PAL data structure. The violation is then detected, and reported to the Health Monitor.

Impact on APEX services

Task deadline monitoring calls for the adaptation of task management services, encapsulated by appropriated AIR PAL functions. Table 6.1 shows the APEX primitives which need to register deadline information (either updating the deadline information for the task, or inserting it if it does not exist yet) or unregister deadline information (removing any information on the respective task from the AIR PAL deadline verification data structures). The APEX primitive RAISE_APPLICATION_ERROR, described in Table 6.2, is used to report a task deadline violation to the HM and trigger the defined error handler.

Table 6.1: APEX services in need of modifications to support task deadline violation monitoring

Primitive	Short description
	Need to register/update deadline
[DELAYED_]START	Start a task [with a given delay]
PERIODIC_WAIT	Suspend execution of a (periodic) task until the next release point
REPLENISH	Postpone a task's deadline time
	Need to unregister deadline
STOP[_SELF]	Stop a task [itself]

Table 6.2: Essential APEX services for Health Monitoring

Primitive	Short description
RAISE_APPLICATION_ERROR	Notify the error handler for a specific error type

Our task deadline violation monitoring mechanisms does not need particular algorithmic modifications to be used in multiprocessor settings, since there will be only one AIR PAL instance per partition (regardless of the number of processors to which the partition can have access). However, because of the potential for concurrent modifications through the APEX interface (Table 6.1), the data structures in the AIR PAL should be protected using appropriate synchronization mechanisms (e.g., readers–writer lock).

Relation to Health Monitor and error handling

In the context of fault detection and isolation, ARINC 653 classifies deadline violation as a "process level error" — an error that impacts one or more processes (tasks) in the partition, or the entire partition (AEEC, 1997). The action to be performed in the event of an error is defined by the application programmer through an appropriate error handler.

The error handler may an application task tailored for partition-wide error processing. In the case of spacecraft, the occurrence of task-/partition-level errors may be signalled through interpartition communication to a (system partition) process

Algorithm 2 AIR Partition Scheduler featuring mode-based schedules

```
1: ticks ← ticks + 1                                   ▷ ticks is the global system clock tick counter
2: if schedules_currentSchedule.table_tableIterator.tick = (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf
   then
3:     if currentSchedule ≠ nextSchedule ∧ (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf = 0 then
4:         currentSchedule ← nextSchedule
5:         lastScheduleSwitch ← ticks
6:         tableIterator ← 0
7:     end if
8:     heirPartition ← schedules_currentSchedule.table_tableIterator.partition
9:     tableIterator ← (tableIterator + 1) mod schedules_currentSchedule.table.size
10: end if
```

performing a Fault Detection, Isolation and Recovery (FDIR) function (Fortescue *et al.*, 2003); a system-wide reconfigurability logic should be included in FDIR.

6.1.2 Mode-based schedules

One of the possible actions that can be configured to be taken in reaction to detected timing faults is changing the partition scheduling. The AIR architecture for TSP systems approaches this with the notion of *mode-based schedules*, inspired by the optional service defined in ARINC 653 Part 2 (AEEC, 2007). Instead of one fixed PST, the system can be configured with multiple PSTs, which may differ in terms of **(i)** major time frame duration; **(ii)** which partitions are scheduled; and **(iii)** how much processor time is assigned to each of them. The different PSTs may even take advantage of the fact that the generation of minimum-requirement partition scheduling tables lends itself to the existence of time intervals that are not necessarily attributed to one specific partition (see Listing 5.2, where a gap between two consecutive windows on the same processor can be seen in the schedule for partition $\mathcal{C}_2$).

The system can then switch between the existing PSTs; this is performed through a service call issued by an authorized and/or dedicated partition. To avoid violating temporal requirements, a PST switch request is only effectively granted at the end of the ongoing major time frame.

The AIR Partition Scheduler, with the addition of mode-based schedules, functions as described in Algorithm 2.

The first verification to be made is whether the current instant is a partition preemption point (Line 2). In case it is not, the execution of the partition scheduler

Algorithm 3 AIR Partition Scheduler featuring mode-based schedules and interpartition parallelism on multiprocessor

```
1: ticks ← ticks + 1                                   ▷ ticks is the global system clock tick counter
2: for all processor j do
3:     if schedules_currentSchedule.table_j,tableIterator_j.tick =
       (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf then
4:         if currentSchedule ≠ nextSchedule ∧
           (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf = 0 then
5:             currentSchedule ← nextSchedule
6:             lastScheduleSwitch ← ticks
7:             tableIterator_j ← 0
8:         end if
9:         heirPartition_j ← schedules_currentSchedule.table_j,tableIterator_j.partition
10:        tableIterator_j ← (tableIterator_j + 1) mod schedules_currentSchedule.table_j.size
11:    end if
12: end for
```

is over; this is both the best case and the most frequent one. If it is a partition preemption point, a verification is made (Line 3) as to whether there is a pending scheduling switch to be applied and the current instant is the end of the MTF. If these conditions apply, then a different PST will be used henceforth (Line 4). The partition which will hold the processing resources until the next preemption point, dubbed the heir partition, is obtained from the PST in use (Line 8) and the AIR Partition Scheduler will now be set to expect the next partition preemption point (Line 9).

Due to the way we have extended to multiprocessor the ARINC 653 XML format for partition scheduling tables, modifying the AIR Partition Scheduler to support interpartition parallelism is straighforward. As can be seen in Algorithm 3, the modification consists essentially of wrapping most of Algorithm 2 in a for loop to go over each processor schedule (cf. Listing 5.2).

Generation of different partition schedules can be aided by a tool that applies rules and formulas to the temporal requirements of the constituent task sets of the necessary partitions. In Chapter 5, we have described a proof of concept for this kind of tool.

Impact on APEX services

To allow application programmers to use the mode-based schedules functionality, the APEX interface is extended with additional primitives presented in Table 6.3.

Table 6.3: Essential APEX services to support mode-based schedules

Primitive	Short description
SET_SCHEDULE	Request for a new PST to be adopted at the end of the current MTF
GET_SCHEDULE_STATUS	Obtain current schedule, next schedule (same as current schedule if no PST change is pending), and time of the last schedule switch

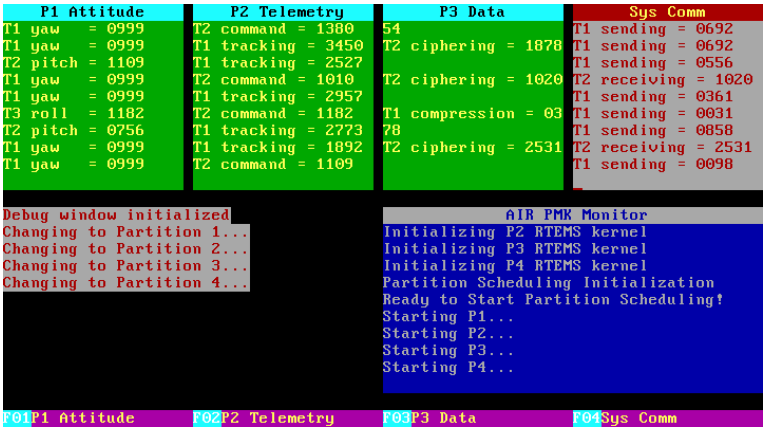


Figure 6.2: Screenshot of the Intel IA-32 prototype of the AIR architecture.

6.1.3 Prototype implementation

To demonstrate the advanced timeliness control features, we have developed prototype implementations of an AIR-based system. Each partition executes an RTEMS-based (Real-Time Executive for Multiprocessor Systems) mockup application representative of typical functions present in a satellite system. The demonstrations were implemented for Intel IA-32 and SPARC LEON targets.[2] To allow visualization and interaction, the Intel prototype includes VITRAL, a text-mode windows manager for RTEMS developed by Coutinho *et al.* (2006); its visual appearance can be seen in Figure 6.2. There is one window for each partition, where its output can be seen, and also two more windows which allow observation of the behavior of AIR components. VITRAL also supports keyboard interaction, which is used, for demonstration purposes, to allow switching to a given partition scheduling table at

[2]The SPARC LEON prototype was implemented jointly with the team at GMV (Rufino *et al.*, 2011), and is not addressed in this dissertation.

Table 6.4: Logical SLOC and cyclomatic complexity (CC) for the AIR Partition Scheduler with mode-based schedules

	Logical SLOC	CC
AIR Partition Scheduler [a]	13	4
Underlying clock tick ISR	>190 [b]	>20

[a] Algorithm 2
[b] C code only; plus >182 assembly instructions

the end of the present major time frame.[3]

6.1.4 Evaluation

We now evaluate the implementation of the monitoring and adaptation mechanisms into the Intel prototype of the AIR architecture. This evaluation is limited to the implementation for uniprocessor.

6.1.4.1 Code complexity

Critical software, namely that developed to go aboard a space vehicle, goes through a strict process of verification, validation and certification. Code complexity increases the effort of this process and the probability of there being bugs.

One metric for code complexity is its size, in lines of source code. Since equivalent code can be arranged in ways which account for different lines of code counts, standardized accounting methods must be used. We employ the *logical source lines of code (logical SLOC)* metric of the Unified CodeCount tool (Nguyen *et al.*, 2007). Another useful metric is *cyclomatic complexity*, which gives an upper bound for the number of test cases needed for full branch coverage and a lower bound for those needed for full path coverage.

The C implementation of Algorithm 2 is accounted for in Table 6.4, which shows its logical SLOC count and cyclomatic complexity. Code introduced at the AIR PAL level to achieve task deadline violation monitoring is accounted for in Table 6.5. The

[3] Although out of the scope of this dissertation, the keyboard interaction we support in the Intel prototype also allows activating a task that deliberately accesses memory from another partition — so as to demonstrate spatial partitioning mechanisms.

Table 6.5: Logical SLOC and cyclomatic complexity (CC) for the implementation of deadline violation monitoring in AIR PAL

	Logical SLOC	CC
Register deadline	34	6
Unregister deadline	12	3
Verify deadlines [a]	16	2

[a] Algorithm 1

total complexity added in terms of code executed during a clock tick ISR is a small fraction of that already present in the underlying ISR code.

6.1.4.2 Computational complexity analysis

Since the verifications of deadlines and of the need to apply a new PST are executed inside the system clock tick ISR, they must have a bounded execution time; computational complexity is a good indicator thereof.

In the AIR Partition Scheduler (Algorithm 2), all instructions are $\mathcal{O}(1)$. Accesses to multielement structures are made by index, and thus their complexity does not depend on the number of elements or the position of the desired element.

Linear complexity is easily achievable for the majority of the executions of the task deadline verification at the AIR PAL level (Algorithm 1). By placing deadlines in a linked list in ascending order of deadline times, the earliest deadline is retrieved in constant time. The removal of a violated deadline from the data structure can also be made in constant time, since we already hold a pointer for the said deadline. In case of deadline violation, the next deadline(s) will successively be verified until reaching one that has not expired. Thus, the worst case yields $\mathcal{O}(n)$, where n is the number of tasks in the partition. However, by design, the number of tasks is bounded, and these worst cases are highly exceptional and mean the total and complete failure of the partition, which should be signalled to the HM.

6.1.4.3 Basic execution metrics

AIR Partition Scheduler execution time has been measured, resulting in the basic metrics shown in Table 6.6. The second row shows the minimum, maximum and

Table 6.6: AIR Partition Scheduler (with mode-based schedules) execution time — basic metrics

	Minimum (ns)	Maximum (ns)	Average (ns)
ns	32	186	36
% (50 µs tick)	0.06%	0.37%	0.07%

average impact of each execution of the AIR Partition Scheduler; these occur with each clock tick, which is triggered every 50 microseconds. These values were obtained executing the AIR prototype demonstrator on a native machine equipped with an Intel IA-32 CPU with a clock of 2833 MHz. Time readings are obtained from the CPU Time Stamp Counter (TSC) 64-bit register, being rounded to 1 ns. Measurements showed negligible variation between sample executions.

Though encouraging, these results should be enriched. Further work will compare them with the execution of the whole clock tick ISR, dissect the execution time by subcomponents, and identify trends in execution time variation.

6.2 Self-adaptation upon temporal faults

The self-adaptation mechanisms we hereby propose aim for the ability to cope with timing faults in TSP systems, without abandonding the two-level hierarchical scheme (which brings benefits in terms of certification, verification and validation). It profits from AIR's mode-based schedules and task deadline violation monitoring we described.

In our approach, the system integrator includes several PSTs that fulfil the same set of temporal requirements for the partitions, but distribute additional spare time inside the MTF among these partitions in different ways. This set of PSTs, coupled with the monitoring and adaptation mechanisms described in Section 6.1, can be used to temporarily or permanently assign additional processing time to a partition hosting an application which has repeatedly observed to be missing deadlines.

6.2.1 Evaluation

This approach with static precomputed PSTs is a first step towards a lightweight, safe, and certifiable dynamic approach to self-adaptability in TSP systems. We now evaluate this initial proof of concept. The next steps to enhance the coverage and efficacy of our proposal are already planned and are described at the end of this chapter.

6.2.1.1 Sample generation

We ran our experiments on a set of 200 samples, each of which corresponding to a TSP system comprising 3 partitions. Each partition in a sample system was generated by **(i)** drawing a task set from a pool of 500 random task sets with total utilizations between 0.19 and 0.25; tasks' periods were randomly drawn from the harmonic set {125, 250, 500, 1000, 2000}; these values are in milliseconds, and correspond to typical periods found in real TSP systems; **(ii)** computing the partition's timing requirements from the timing characteristics of the task set, using Shin & Lee (2003)'s periodic resource model with the partition cycle from the set $\{25, 50, 75, 100\}$ which yielded the lowest bandwidth. The sample's MTF is set to the least common multiple of the partitions' cycles.

6.2.1.2 Simulation process

We implemented a simulation of the described two-level scheduling scheme, with AIR's deadline violation monitoring and mode-based scheduling mechanisms described in Section 6.1, and supporting the proposed self-adaptive schedule switch mechanisms.

For each sample, the first step of the simulation is to generate PSTs by emulating the behavior of rate monotonic scheduling, assigning available time slots on a cycle basis to fulfil the minimum duration requirements. In the end, there are typically unassigned time slots in the MTF covered by the PST. The extension we implemented for these experiments was to derive *three PSTs* from the one provided by our algorithm: χ_1, where all the unassigned time is given to partition P_2, and χ_2, where all the unassigned time is given to partition P_1, and χ_3, where all the

unassigned time is given to partition P_3.[4] We simulated the execution of each of the samples for 1 minute (60000 ms), with the following varying conditions:

Task temporal faultiness The percentage by which the jobs of the tasks in partition P_1 exceed the parent task's worst-case execution requirement estimate was varied between 0% and 100%.

Self-adaptation mechanisms For each considered task temporal faultiness, simulations were performed with our proposed self-adaptive scheduling mechanisms respectively disabled and enabled. In both, the initial PST by which partitions are scheduled is χ_1. With self-adaptive scheduling disabled, this PST is used through the entire simulation. Otherwise, when a task deadline miss is notified to the Health Monitor, a flag is activated, which will cause the system to switch, at the end of the current MTF, from PST χ_1 to χ_2.

We choose to limit **(i)** the timing fault injection to P_1, and **(ii)** the execution time trade to happen between P_2 and P_1; to see how the mechanism behaves in presence of *one* timing fault on a known partition. This simplification shall be lifted in future experiments, where we will analyze the behavior when faults are multiple or can occur in any partition.

6.2.1.3 Example

To help illustrate our approach and our simulation, let us look at an example sample, comprising a set of three partitions, with task sets:

- $\mathcal{T}_1 = \{(500, 44, 500), (1000, 62, 1000), (2000, 58, 2000)\}$;
- $\mathcal{T}_2 = \{(250, 29, 250), (1000, 28, 1000), (1000, 50, 1000), (2000, 89, 2000)\}$; and
- $\mathcal{T}_3 = \{(250, 35, 250), (2000, 99, 2000)\}$.

From each of the task sets, we derived the following partition timing requirements for the generation of PSTs χ_1, χ_2 and χ_3:

- $\Gamma_1 = (100, 19)$

[4] PST χ_3 is, however, not taken advantage of in this experiment.

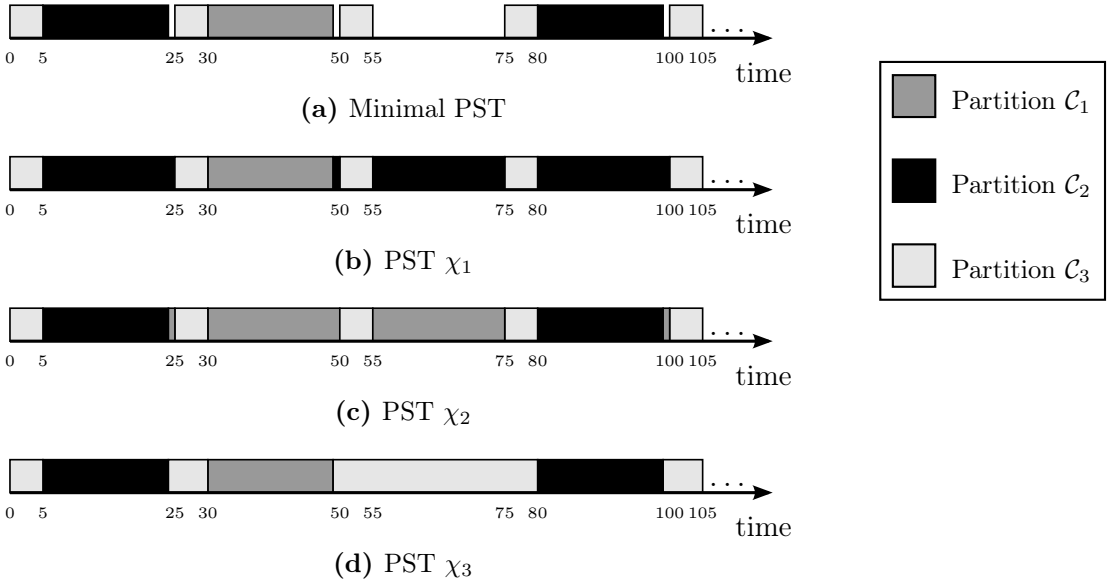


Figure 6.3: Example partition scheduling tables

- $\Gamma_2 = (75, 19)$
- $\Gamma_3 = (25, 5)$

An excerpt of the minimal PST to serve as the basis of PSTs χ_1, χ_2 and χ_3 is pictured in Figure 6.3a. The analogous excerpts of PSTs χ_1 and χ_2, with unassigned time given to, respectively, partitions P_2 and P_1, are shown in Figures 6.3b and 6.3c.

6.2.1.4 Results

We selected two deadline miss measures to evaluate the efficacy of our approach. First we measured the percentage of all jobs generated for the considered interval (60000 ms) which missed their deadline — the *deadline miss rate*. The graph in Figure 6.4a shows the average deadline miss rate (over all samples) for each value of injected temporal faultiness, both without and with our self-adaptation mechanisms. As expected, the deadline miss rate grows approximately linearly with the injected faultiness. However, when self-adaptation based on the detected deadline misses is employed, the growth of this rate is controlled. Even when the jobs of the tasks in partition $\mathcal{C}_1$ are exceeding their worst-case execution requirement estimate by 100%,

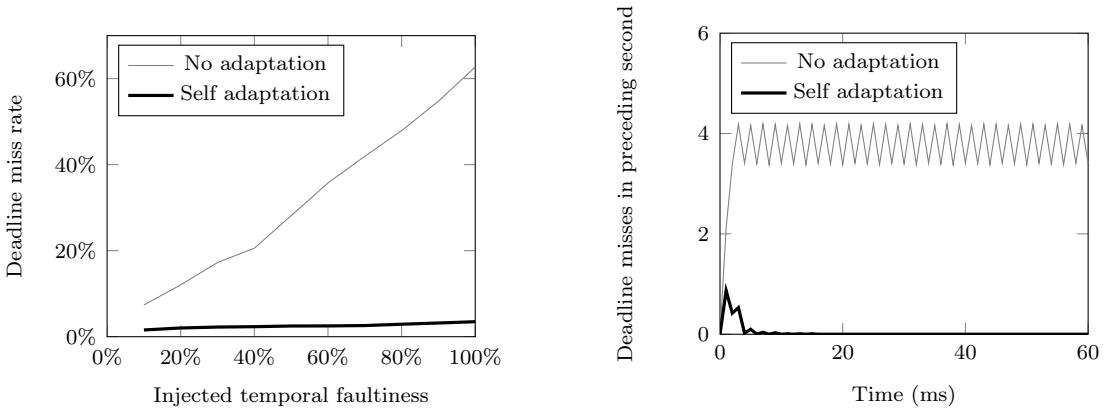


Figure 6.4: Simulation results: deadline miss rate; deadline misses over time

only less than 5% of the released jobs miss their deadlines, whereas the deadline miss rate amounts to more than 60% if self-adaptation is disabled.

The second measure equates to how long the timing fault's effects take to wear off (i. e., deadline misses cease to occur) in a system with 100% faultiness. The graph in Figure 6.4b plots the average number (over all samples) of deadline misses detected in each 1000 ms interval. When self-adaptation is not used, the timing fault persists over time, with about 4 misses detected each second. With our self-adaptation mechanism, P_1's tasks cease to have their jobs miss deadlines after, at most, 15 seconds.

6.3 Improvements discussion

The results presented in the previous section are encouraging and strongly suggest the efficacy of our self-adaptation mechanisms. However, this outcome so far only sees direct application to soft real-time systems, since it still allows for some deadline misses to occur. Furthermore, the scope of our experiments is limited in terms of fault coverage. Although solving this still requires some future work, let us look at already foreseen extensions to self-adaptation mechanisms which allow better and more widely applicable improvements.

6.3.1 Multiprocessor

The results we have obtained are restricted to uniprocessor platforms. As we have stated in Section 1.3, taking advantage of multiprocessor platforms (with identical or non-identical processors) opens room for supporting self-adaptive behavior to cope with unforeseen changes in operational and environmental conditions. We have described and illustrated in Section 3.2.4 the approach that we envision. The overall idea is to detect that the root cause of a temporal fault lies with a specific processor core, and perform a schedule switch that reassigns the activities of that core to another core; the latter, prior to that moment, can be either idle or attributed to a function which can tolerate a performance degradation for the sake of safety (e.g., a non-critical function).

6.3.2 Reconfiguration

As shown in the results, the use of self-adaptation mechanisms significantly decreases the number of deadline misses, when comparing to the same system execution without the use of such mechanisms. However, besides the good contributions to mitigate this problem, these self-adaptation mechanisms do not cover all the possible cases, neither ensure perpetual stability of the system. For example, temporal faultiness may further increase, due to the additional execution time provided to the faulty partition not being enough. Furthermore, this problem can be even more serious if the system encounters deadline misses in more than one partition.

As it is impossible to foresee all the cases, i.e., all the suitable PSTs associated to different temporal requirements, we need to provide ways to reconfigure the system, during its execution, when facing this kind of problems. This reconfiguration can be done replacing the original set of PSTs by a redefined and updated set that covers the new temporal requirements and encounters issues. Self-adaptability can, in such more serious scenario, provide a temporary mitigation to allow time for a more targeted solution through online reconfiguration which enables mission survival.

We have collaborated with Rosa *et al.* (2011) in approaching reconfiguration in a study which, despite contemporary and related, is outside the scope of this dissertation.

6.3.3 Proactivity

The results we presented in Section 6.2.1.4, although pointing towards the advantages of self-adaptive behavior, are for now only directly applicable to soft real-time systems — since they still allow for some deadline misses to occur before taking action. To apply this approach to safety-critical systems with hard real-time workloads, action should be taken to *avoid* temporal faults (Almeida *et al.*, 2013). For this, it is necessary to identify patterns which indicate, with a given degree of confidence, that the system is behaving towards a temporal fault.

6.4 Summary

In this chapter, we have present preliminary results on self-adaptive behavior in TSP systems, based on the monitoring and adaptation mechanisms present in the AIR architecture. We have proposed coping with temporal faults in TSP systems with a set of feasible and specially crafted partition scheduling tables, among which the system shall switch upon detecting temporal faults within a partition. The added self-adaptability does not void the certification advantages brought by the use of precomputed partition schedules. The results of our experimental evaluation show the self-adaptation to timing faults we propose achieves a degree of temporal fault control suitable for soft real-time guarantees. Future work includes a more complete evaluation with a wider fault coverage, namely faults in more than one partition. Furthermore, this preliminary approach shall evolve towards a dynamic approach that does not compromise the mission's safety and the system's certifiability. Other enhancements include taking profit from multiprocessor and proactively avoid deadline misses. The latter should lead us closer to a self-adaptation policy for hard real-time.

Chapter 7

Conclusion and Future Work

In this dissertation, we have approached the problem of real-time scheduling on multicore time- and space-partitioned (TSP) system archictures. We have proposed (Chapter 3) a system architecture (and respective formal model) for TSP systems support parallelism at both the application (between tasks) and system (between applications) level.

We then augmented the state of the art with a formal approach, based on compositional analysis, to obtain valid parameters to ensure schedulability of applications (Chapter 4); the presented approach is the first addressing the compositional analysis problem upon a non-identical multiprocessor platform. In Section 4.3.2, we have analytically proven that non-identical multiprocessor platforms provide gains over identical multiprocessor platforms with the same overall capacity, in terms of component-level schedulability in systems modeled as compositional scheduling frameworks; the tests we performed with randomly generated task sets (Section 4.3.4.1) confirm this result, which is coherent with results found in the literature for dedicated multiprocessor platforms (Baruah & Goossens, 2008). In Section 4.4.2, we have analytically proven that gEDF, when used as a global-level intercomponent scheduler, does not ensure that compositionality is maintained in the presence of non-identical platforms. We have then proposed a new scheduler, umprEDF, which guarantees that the resource interface of each component is respected and ensured, and compositionality is maintained; this is a crucial point when considering the advantage of independent development and verification brought by component-based approaches.

Next (Chapter 5), we presented the design and development of hsSim, an object-oriented scheduling analysis, simulation and schedule generation tool; hsSim shall be made available as an open-source tool for the real-time scheduling research community, and is a proof of concept for the inclusion of support to hierarchical scheduling in a more mature tool.[1] We have used hsSim to show the formal methods we presented in Chapter 4 in action.

Finally, in Chapter 6, we have presented preliminary results pertaining to self-adaptive behavior. Although preliminary, and limited in terms of assumptions, our results allow us to believe that, besides parallelism, the use of multiprocessor also enables self-adaptive behavior in TSP systems.

Given the above, we can say that the overall goals have been met: to show that multiprocessor platforms, in particular with non-identical processors, provide new capabilities to time- and space-partitioned systems with respect to parallelism, and open room for self-adaptability.

7.1 Applicability perspective

In this dissertation, we employ a system model relying on assumptions regarding the temporal impact of hardware-related phenomena (such as bus contention, or cache memory). More specifically, we assume that this temporal impact is either negligible, already accounted for in the worst-case execution requirement estimations, and/or at least partially eliminated by the choice of cache managament or bus contention policies. This is a common assumption in real-time scheduling research, and not so far from reality as it may seem. For instance, Jalle *et al.* (2013) experimentally show that the two most-used policies to manage bus contention (Time-Division Multiple Access and Interference-Aware Bus Arbitrer) observe time composability — that is, worst-case execution requirement estimations obtained for one task in isolation (taking into account the bus contention control policy) hold when it is put to contend for bus access with other tasks (regardless of the characteristics of the latter). In a study commissioned by the European Aviation Safety Agency, Jean *et al.* (2012) recommend that the approach to multicore processors in airborne systems uses hardware-based cache partitioning mechanisms.

[1] https://code.google.com/p/hssim.

Over the years, there were research projects funded by European entities (most notably the European Commission and the European Space Agency) employing component-based approaches and/or time and space partitioning to achieve architectural hybridization or to cope with mixed-criticality systems. In some of these projects, mostly contemporary with the hereby presented work, multicore processors have been approached to some extent.

- The ACTORS project[2] (2008–2011) employed reservation-based scheduling to enforce dependability and predictability over multicore embedded systems. Bini *et al.* (2011b) describe the approach (based on compositional analysis with Bini *et al.* (2009b)'s Multi Supply Function abstraction) and implementation into the Linux kernel (based on Abeni & Buttazzo (1998)'s Constant Bandwidth Server); their approach is limited to partitioned local-level scheduling. Bini *et al.* (2011a) describe an approach to use with global local-level scheduling algorithms (based on compositional analysis with Bini *et al.* (2009a)'s Parallel Supply Function), which was however not successfully demonstrated.

- The RECOMP project[3] (2010–2013) aimed at enabling component-based development of mixed-criticality systems on multicore platforms. Low experience with hierarchical scheduling is pointed out as one of the obstacles to certifying safety-critical applications on multicores, and the work of Easwaran *et al.* (2009b) (which we have extended towards non-identical multiprocessors) is described as fulfilling several of the project's requirements (RECOMP, 2011a). The approach to multicore followed in the RECOMP project is that of having multiple interconnected nodes, each of them working on one single processor (akin to the approach pictured in Figure 3.3a) (RECOMP, 2011b, 2013).

- In the KARYON project[4] (2011—), architectural hybridization is proposed to cope with the risks of cooperating in an uncertain environment, and time and space partitioning is proposed as a way of implementing such architecture (Nóbrega da Costa *et al.*, 2013).

[2] Adaptivity and Control of Resources in Embedded Systems, http://www.actors-project.eu/

[3] Reduced Certification Costs Using Trusted Multi-core Platforms, http://atcproyectos.ugr.es/recomp/

[4] Kernel-Based ARchitecture for safetY-critical cONtrol, http://www.karyon-project.eu/

The work presented in this dissertation is applicable to the architectures considered in or resulting from the aforementioned projects, even addressing issues left open at the end of some of them. Most notably, our results go beyond the achievements of these projects with respect to intrapartiton parallelism and to the use non-identical processors. We believe this is the first step towards breaking the vicious cycle between the hardware and software support to non-identical processors. In the aerospace domain, the widely used SPARC LEON processors allow heterogeneous multicore configurations, but the lack of operating system support thereto disencourages its use.[5]

Besides the initially established main focus on safety-critical application domains, our results also contribute towards the formal verification of component-based systems supported on non-identical multiprocessors platforms such as Cell (Gschwind *et al.*, 2006) and big.LITTLE — namely the big.LITTLE MP use model (Greenhalgh, 2011). Although these multiprocessors are not typically employed in safety-critical domains, applying a component-based approach to the development of complex software to run on them allows reducing the complexity and cost of the latter, while securing that individual components achieve a minimum quality of service. This shall be particularly true as operating system support to hierarchical scheduling increases; for instance, Abdullah *et al.* (2013) are starting to implement virtual-clustered multiprocessor scheduling (Easwaran *et al.*, 2009b) in the Linux kernel.

7.2 Future work

Although our goals have been met, the work presented here is only a fraction of a wider work concerning the industrial adoption of systems based on the architecture design we consider. Most notably, the various constraints inherent to the scope of a PhD prevented improving the analysis by lifting system model assumptions that separate it from real systems (namely aspects related to multiprocessor hardware) and validating the work with a prototype implementation of a multiprocessor TSP system on real hardware. The lack of real or realistic case studies, endemic to the real-time systems and cyber-physical systems community (Di Natale *et al.*, 2013),

[5] http://comments.gmane.org/gmane.comp.hardware.opencores.leon-sparc/14873.

has also been an obstacle to the full expression of this work, circumvented to the possible extent through the use of synthetic workloads.

The project READAPT (*Reconfigurability and Adaptability in Safe and Secure Multicore Architectures for Mixed-Criticality Applications*) was approved for funding by *Fundação para a Ciência e a Tecnologia* (FCT, the Portuguese national science foundation), and began near the conclusion of the work described in this dissertation.[6] In READAPT, the LaSIGE team collaborates with the industrial partner GMV. This project will hopefully provide the necessary means to bring the hereby presented results into real practice, and to pursue other related work. Collaborations with international partners from both academia and industry have been proposed, and some more are being prepared at the time of this writing. We are for this reason optimistic that we will have great opportunities to continue, improve and expand the work presented here. We now list some of the open issues and topics to address.

7.2.1 Hardware support and system model assumptions

As mentioned before, our a system model is limited by assumptions regarding the temporal impact of hardware-related phenomena (such as bus contention, or cache memory). Although finding echo in real requirements, this surely limits the applicability of our results. In the future, we aim to take these phenomena into specific account when analyzing TSP systems.

Another hardware-related aspect worth investigating in the future is that of evaluating the power consumption trade-off between TSP systems based on identical multiprocessors and those bases on non-identical multiprocessors.

7.2.2 Scheduling algorithms

In Chapter 4, we have analyzed compositional scheduling frameworks assuming a gEDF local-level scheduling algorithm. Since the divisions of tasks into components may be functional or institutional in nature, it is not guaranteed that the "Dhall & Liu (1978) effect" of which gEDF suffers is eliminated. As we have seen in Section 2.2.3, hybrid variants of gEDF have been proposed to cope with this effect,

[6] http://www.navigators.di.fc.ul.pt/wiki/Project:READAPT.

without incurring the typically higher number of preemptions of the LLF algorithm; these variants include EDZL (Lee, 1994).

However, we have found (Craveiro & Rufino, 2013) that, when there are time intervals at which the processing platform is partially or totally unavailable to the scheduler, EDZL is not appropriate. The reason for this is that the scheduler compares the laxity of the jobs to a constant value (zero) without regard for the expected availability of the platform. We have shown a proof of concept for the general idea of a possible solution, using a simple resource model and comparing the laxity of the jobs against a non-zero value obtained from the resource interface; our proof of concept shares some characteristics with Davis & Kato (2012)'s Fixed Priority until Static Laxity algorithm.

Our intuition is that the solution for this problem may be based on an adapted notion of laxity that considers the difference between how much execution capacity one single job can have available until its deadline (according to the resource interface) and its remaining execution. Open questions with respect to this problem include the dynamics of this laxity and schedulability analysis when scheduling occurs upon a multiprocessor reservation according to the resource models proposed in the literature (Bini *et al.*, 2009a; Burmyakov *et al.*, 2012; Craveiro & Rufino, 2012; Lipari & Bini, 2010; Shin *et al.*, 2008).

7.2.3 Compositional analysis

Although compositional analysis is applied to systems which are motivated by the coexistence of heterogeneous workloads, the vast majority of works on compositional analysis (including the work presented in Chapter 4 of this dissertation) assume hard real-time tasks only. In the near future, we aim to approach compositional analysis on systems models contemplating soft real-time and mixed-criticality. To the best of our knowledge, only Leontyev & Anderson (2009) approach compositional analysis on a hierarchical scheduling framework where components may have soft real-time workloads, on identical multiprocessors; however, the resource provision is hard (i.e., it must be guaranteed). We aim to provide soft resource provisions for soft real-time workloads.

7.2.4 Reconfiguration and proactivity

As seen in Section 6.3.3, improving the survivability of space vehicles requires not only reacting to abnormal events but also proactively tolerating their occurrence (Almeida *et al.*, 2013). Performing proactive fault-tolerance actions implies forecasting and preventing future uncontrolled abnormal events. With respect to proactive fault tolerance in TSP systems, a set of statistically significant patterns of interest should be usable as key indicators for forecasting the system's behavior. Such indicators shall assist the detection of a potentially anomalous situation, i.e. an imminent failure.

The use of multiprocessor platforms is TSP sytstems that we defend in our thesis brings added value to proactive fault tolerance. For instance, we can have one or more spare processor cores that take over the duties of cores which present evidence of an imminent failure — thus avoiding more severe faults before they happen.

References

Abdullah, S.M.J., Khalilzad, N.M., Behnam, M. & Nolte, T. (2013). Towards implementation of virtual-clustered multiprocessor scheduling in Linux. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013) Work-in-Progress session*, Porto, Portugal. xix, 148

Abeni, L. & Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. In *19th IEEE Real-Time Systems Symposium (RTSS '98)*, 4–13, Madrid, Spain. xiv, xv, 2, 9, 31, 109, 147

AEEC (1991). Design guidance for Integrated Modular Avionics. ARINC Report 651, Airlines Electronic Engineering Committee (AEEC), initial release (current revision: 651-1, November 1997). xiv, 2

AEEC (1997). Avionics application software standard interface. ARINC Specification 653, Airlines Electronic Engineering Committee (AEEC), initial release (current revision: 653P1-3, November 2010). xiv, 2, 3, 4, 5, 33, 49, 51, 59, 61, 132

AEEC (2007). Avionics application software standard interface. ARINC Specification 653 Part 2 (Extended Services), Airlines Electronic Engineering Committee (AEEC), initial release (current revision: 653P2-3, June 2012). 4, 51, 133

Almeida, K., Pinto, R.C. & Rufino, J. (2013). Fault detection in time- and space-partitioned systems. In *INFORUM 2013 — Simpósio de Informótica*, Évora, Portugal. 144, 151

Anderson, J., Baruah, S. & Brandenburg, B. (2009). Multicore operating-system support for mixed criticality. In *CPS Week 2009 Workshop on Mixed Criticality*, San Francisco, CA, USA. xv, 7

Andersson, J., Gaisler, J. & Weigand, R. (2010). Next generation multipurpose microprocessor. In *DASIA 2010 "DAta Systems In Aerospace" Conference*, Budapest, Hungary. xv, 7

Audsley, N. & Wellings, A. (1996). Analysing APEX applications. In *17th IEEE Real-Time Systems Symposium (RTSS '96)*, 39–44, Washington, DC, USA. xiv, 3, 33, 34, 35

Audsley, N., Burns, A., Richardson, M., Tindell, K. & Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, **8**(5):284–292. 21

Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W. & Wellings, A.J. (1995). Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, **8**(2):173–198. 14

AUTOSAR (2006). Technical overview, v2.0.1. 6

AUTOSAR (2013a). Requirements on operating system, v3.1.0, release 4.1, revision 1. 7, 8

AUTOSAR (2013b). Specification of operating system, v5.1.0, release 4.1, revision 1. 7, 8

Baker, T.P. (2003). Multiprocessor EDF and Deadline Monotonic schedulability analysis. In *24th IEEE Real-Time Systems Symposium (RTSS '03)*, Cancun, Mexico. v, 25, 26, 28, 43, 76

Baker, T.P. & Baruah, S.K. (2007). Schedulability analysis of multiprocessor sporadic task systems. In I. Lee, J.Y.T. Leung & S.H. Son, eds., *Handbook of Real-Time and Embedded Systems*, chap. 3, Chapman & Hall/CRC. 22, 25

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *28th IEEE Real-Time Systems Symposium (RTSS '07)*, Tucson, AZ, USA. 25, 26, 27, 43, 44

Baruah, S. & Goossens, J. (2008). The EDF scheduling of sporadic task systems on uniform multiprocessors. In *29th IEEE Real-Time Systems Symposium (RTSS '08)*, Barcelona, Spain. xvii, 11, 28, 76, 77, 78, 80, 81, 84, 87, 91, 145

Baruah, S., Goossens, J. & Lipari, G. (2002). Implementing constant-bandwidth servers upon multiprocessor platforms. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '02)*, 154–163, San Jose, CA, USA. 31

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A. & Stiller, S. (2009). Implementation of a speedup-optimal global EDF schedulability test. In *21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, Dublin, Ireland. 46

Baruah, S.K. (2004). Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, **53**(6):781–784. 27

Baruah, S.K., Mok, A.K. & Rosier, L.E. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th Real-Time Systems Symposium (RTSS '90)*, Lake Buena Vista, Florida, USA. 21

Bertogna, M., Cirinei, M. & Lipari, G. (2005). Improved schedulability analysis of EDF on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems (ECRTS '05)*, 209–218, Palma de Mallorca, Spain. 26, 43, 46

Bini, E. & Lipari, G. (2011). Introduction to the special issue. *ACM SIGBED Review*, **8**(1), Special Issue on 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2010). 35

Bini, E., Buttazzo, G. & Buttazzo, G. (2003). Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, **52**(7):933 – 942. 21

Bini, E., Bertogna, M. & Baruah, S. (2009a). Virtual multiprocessor platforms: Specification and use. In *30th IEEE Real-Time Systems Symposium (RTSS '09)*, 437–446, Washington, DC, USA. 14, 46, 47, 51, 147, 150

Bini, E., Buttazzo, G. & Bertogna, M. (2009b). The multi supply function abstraction for multiprocessors. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, 294–302, Beijing, China. 45, 46, 51, 147

Bini, E., Buttazzo, G. & Bertogna, M. (2011a). Multicore. Deliverable D4d, ACTORS project. 147

Bini, E., Buttazzo, G., Eker, J., Schorr, S., Guerra, R., Fohler, G., Årzén, K.E., Romero, V. & Scordino, C. (2011b). Resource management on multi-core systems: the ACTORS approach. *IEEE Micro*, **31**(3):72–81. xix, 147

Binns, P. (2001a). A robust high-performance time partitioning algorithm: the Digital Engine Operating System (DEOS) approach. In *20th IEEE/AIAA Digital Avionics Systems Conference (DASC 2001)*, Daytona Beach, FL, USA. 50

Binns, P.A. (2001b). Slack scheduling for improved response times of period transformed processes. Patent, US 6189022. 50

Black, R. & Fletcher, M. (2006). Open Systems Architecture - Both Boon and Bane. In *25th IEEE/AIAA Digital Avionics Systems Conference (DASC 2006)*, 1 –7. xiv, 5

Brocal, V., Masmano, M., Ripoll, I., Crespo, A., Balbastre, P. & Metge, J.J. (2010). Xoncrete: a scheduling tool for partitioned real-time systems. In *Embedded and Real Time Software and Systems (ERTS^2 2010)*, Toulouse, France. 54

Burmyakov, A., Bini, E. & Tovar, E. (2012). The Generalized Multiprocessor Periodic Resource interface model for hierarchical multiprocessor scheduling. In *RTNS '12*, Pont à Mousson, France. 150

Buttazzo, G., Bini, E. & Wu, Y. (2010). Partitioning parallel applications on multiprocessor reservations. In *22nd Euromicro Conference on Real-Time Systems (ECRTS '10)*, 24–33, Brussels, Belgium. 46

Buttazzo, G., Bini, E. & Wu, Y. (2011). Partitioning real-time applications over multicore reservations. *IEEE Transactions on Industrial Informatics*, **7**(2):302–315. 46

Buttazzo, G.C. (1997). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems, Kluwer Academic Publishers, Boston / Dordrecht / London. 14, 30

Buttazzo, G.C. (2005). Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, **29**:5–26, 10.1023/B:TIME.0000048932.30002.d9. 25

Carpenter, J., Funk, S., Holman, P., Anderson, J. & Baruah, S. (2004). A categorization of real-time multiprocessor scheduling problems and algorithms. In J.Y.T. Leung, ed., *Handbook on Scheduling: Algorithms, Methods, and Models*, Chapman & Hall/CRC. 14, 18

Coronel, J. & Crespo, A. (2012). Design of the generic virtualization layer. Deliverable D3.3, MultiPARTES project. 51

Coutinho, M., Almeida, C. & Rufino, J. (2006). VITRAL - a text mode window manager for real-time embedded kernels. In *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2006)*, 1254–1260, Prague, Czech Republic. 135

Craveiro, J., Rufino, J., Schoofs, T. & Windsor, J. (2009). Flexible operating system integration in partitioned aerospace systems. In *INForum 2009 — Simpósio de Informática*, 49–60, Lisbon, Portugal. 59

Craveiro, J., Rufino, J. & Singhoff, F. (2011). Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems. *ACM SIGBED Review*, **8**(3):23–27, special issue of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11) Work-in-Progress session. 51, 53

Craveiro, J.P. & Rufino, J. (2012). Towards compositional hierarchical scheduling frameworks on uniform multiprocessors. Tech. Rep. TR-2012-08, University of Lisbon, DI–FCUL, revised January 2013. 150

Craveiro, J.P. & Rufino, J. (2013). Global laxity-based scheduling on multiprocessor resource reservations. In *4th International Real-Time Scheduling Open Problems Seminar (RTSOPS 2013)*, Paris, France. 150

Craveiro, J.P.G.C. (2009). *Integration of generic operating systems in partitioned architectures*. Master's thesis, Faculty of Sciences, University of Lisbon, Lisbon, Portugal. 59, 129

Davis, R.I. & Burns, A. (2009). Priority assignment for global fixed priority preemptive scheduling in multiprocessor real-time systems. In *30th IEEE Real-Time Systems Symposium (RTSS '09)*, 398–409, Washington, DC, USA. 88

Davis, R.I. & Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, **43**(4):35:1–35:44. 14, 18

Davis, R.I. & Kato, S. (2012). FPSL, FPCL and FPZL schedulability analysis. *Real-Time Systems*, **48**(6):750–788. 150

DDC-I (2012). Deos: A Time & Space Partitioned DO-178B Level A certifiable RTOS. Fact sheet, DDC-I, http://www.ddci.com/product_fact_sheets/Deos.pdf. Retrieved on July 31, 2013. 50

de la Puente, J.A., Balbastre, P. & Garrido, J. (2012). Scheduling partitioned systems on multicore platforms. Deliverable D3.2, MultiPARTES project. 51

Deng, Z. & Liu, J.W.S. (1997). Scheduling real-time applications in an open environment. In *18th IEEE Real-Time Systems Symposium (RTSS '97)*, 308–319. 32

Deng, Z., Liu, J.S. & Sun, J. (1997). A scheme for scheduling hard real-time applications in open system environment. In *9th Euromicro Workshop on Real-Time Systems (RTS 1997)*, 191–199. 32

Dertouzos, M.L. (1974). Control robotics: The procedural control of physical processes. In *Information Processing 74: Proceedings of IFIP Congress 74*, 807–813, Stockholm, Sweden. 21, 22

Dertouzos, M.L. & Mok, A.K. (1989). Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, **15**(12):1497–1506. 21, 22

Dhall, S.K. & Liu, C.L. (1978). On a real-time scheduling problem. *Operations Research*, **26**(1):127–140. 22, 24, 27, 124, 149

Di Natale, M., Dong, C. & Zeng, H. (2013). Reality check: the need for benchmarking in RTS and CPS. In *4th International Real-Time Scheduling Open Problems Seminar (RTSOPS 2013)*, Paris, France. 148

Easwaran, A., Shin, I., Sokolsky, O. & Lee, I. (2006). Incremental schedulability analysis of hieararchical real-time components. In *6th International Conference on Embedded Software (EMSOFT '06)*, Seoul, Korea. xvi, 9

Easwaran, A., Anand, M. & Lee, I. (2007). Compositional analysis framework using EDP resource models. In *28th IEEE Real-Time Systems Symposium (RTSS '07)*, Tucson, AZ, USA. 41, 53, 109

Easwaran, A., Lee, I., Sokolsky, O. & Vestal, S. (2009a). A compositional scheduling framework for digital avionics systems. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, Beijing, China. 41, 53

Easwaran, A., Shin, I. & Lee, I. (2009b). Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, **43**:25–59. xvii, xiv, 11, 42, 43, 44, 46, 73, 74, 75, 84, 85, 86, 88, 91, 95, 147, 148

Feng, X.A. & Mok, A.K. (2002). A model of hierarchical real-time virtual resources. In *23rd IEEE Real-Time Systems Symposium (RTSS '02)*, 26–35, Austin, TX, USA. 40

Fletcher, M. (2009). Progression of an open architecture: from Orion to Altair and LSS. Tech. rep., Honeywell International. xiv, 5

Fortescue, P.W., Stark, J.P.W. & Swinerd, G., eds. (2003). *Spacecraft Systems Engineering*. Wiley, 3rd edn. 133

Fuchsen, R. (2010). How to address certification for multi-core based IMA platforms: current status and potential solutions. In *29th IEEE/AIAA Digital Avionics Systems Conference (DASC 2010)*, 5.E.3–1–5.E.3–11, Salt Lake City, UT. 7

Funk, S., Goossens, J. & Baruah, S. (2001). On-line scheduling on uniform multiprocessors. In *22nd IEEE Real-Time Systems Symposium (RTSS '01)*, London, UK. 28, 88, 104

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1997). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 108, 110, 113, 117

Gaudel, V., Singhoff, F., Plantec, A., Dissaux, P. & Legrand, J. (2013). Composition of design patterns : from the modeling of RTOS synchronization tools to schedulability analysis. In *The 3rd Embedded Operating Systems Workshop (EWiLi'13)*, Toulouse, France. 117

Goossens, J., Funk, S. & Baruah, S. (2003). Priority-driven scheduling of periodic task systems on multiprocessors. *Real Time Systems*, **25**(2–3):187–205. 25

Greenhalgh, P. (2011). big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. White paper, ARM Ltd. 148

Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y. & Yamazaki, T. (2006). Synergistic processing in Cell's multicore architecture. *IEEE Micro*, **26**(2):10–24. 148

Hausmans, J.P.H.M., Geuns, S.J., Wiggers, M.H. & Bekooij, M.J.G. (2012). Compositional temporal analysis model for incremental hard real-time system design. In *12th International Conference on Embedded Software (EMSOFT '12)*, Tampere, Finland. xvi, 9

Hodson, R. & Ng, T. (2007). Avionics for exploration. In *NASA Technology Exchange Conference*, Galveston, TX, USA. xiv, 5

Holenderski, M., Bril, R.J. & Lukkien, J.J. (2013). Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems. *Journal of Systems Architecture*, **59**(6):307–314, available online: July 17, 2012. 52, 119

Howard, C.E. (2011). RTOS for a software-driven world. *Military & Aerospace Electronics*, **22**(3), http://www.militaryaerospace.com/articles/print/volume-22/issue-30/technology-focus/rtos-for-a-software-driven-world.html. Retrieved on July 31, 2013. 50, 52

Jalle, J., Abella, J., Quiñones, E., Fossati, L., Zulianello, M. & Cazorla, F.J. (2013). Deconstructing bus access control policies for real-time multicores. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, Porto, Portugal. xviii, 146

Jean, X., Gatti, M., Berthon, G. & Fumey, M. (2012). MULCORS - the use of multicore procesors in airborne systems. Dossier CCC/12/006898 - Rev. 07, Thales Avionics / European Aviation Safety Agency. xviii, 146

Julliand, J., Mountassir, H. & Oudot, E. (2007). Composability, compatibility, compositionality: automatic preservation of timed properties during incremental development. Research Report 2007–01, Laboratoire d'Informatique de l'Université de Franche–Comté, Besançon, France. 35

Kaiser, R. & Wagner, S. (2007a). Evolution of the PikeOS microkernel. In I. Kuz & S.M. Petters, eds., *1st Int. Workshop on Microkernels for Embedded Systems (MIKES 2007)*, 50–57, Sydney, Australia. 49

Kaiser, R. & Wagner, S. (2007b). The PikeOS concept: History and design. White paper, SYSGO, http://www.sysgo.com/nc/en/products/document-center/whitepapers/the-pikeos-concept-history-and-design/. Retrieved on July 31, 2013. 49

King, T. (2011). Slack scheduling enhances multicore performance in safety-critical applications. *EDN*, http://www.edn.com/design/systems-design/4368338/Slack-scheduling-enhances-multicore-performance-in-safety-critical-applications/. Retrieved on July 31, 2013. 50

King, T. (2013). Cache partitioning increases CPU utilization for safety-critical multicore applications. *Military Embedded Systems*, http://mil-embedded.com/articles/cache-utilization-safety-critical-multicore-applications/. Retrieved on July 31, 2013. 50

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems, Kluwer Academic Publishers, Boston / Dordrecht / London. xiii, 1, 14

Kuo, T.W. & Li, C.H. (1999). A fixed-priority-driven open environment for real-time applications. In *20th IEEE Real-Time Systems Symposium (RTSS '99)*, 256 –267. 32, 33

Kuo, T.W., Lin, K.J. & Wang, Y.C. (2000). An open real-time environment for parallel and distributed systems. In *20th International Conference on Distributed Computing Systems (ICDCS 2000)*, 206 –213. 33

Lackorzyński, A., Warg, A., Völp, M. & Härtig, H. (2012). Flattening hierarchical scheduling. In *12th International Conference on Embedded Software (EMSOFT '12)*, Tampere, Finland. xv, 9

Lauzac, S., Melhem, R. & Mossé, D. (1998). An efficient RMS admission control and its application to multiprocessor scheduling. In *12th Int. Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*, 511–518. 21

Lee, S.K. (1994). On-line multiprocessor scheduling algorithms for real-time tasks. In *IEEE Region 10's Ninth Annual International Conference (TENCON'94)*, 607–611. 27, 150

Lee, Y., Kim, D., Younis, M. & Zhou, J. (1998). Partition scheduling in APEX runtime environment for embedded avionics software. In *5th International Conference on Real-Time Computing Systems and Applications (RTCSA 1998)*, 103–109, Hiroshima, Japan. 34, 35, 39

Lehoczky, J. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th Real-Time Systems Symposium (RTSS '90)*, 201–209. 21, 34

Lehoczky, J.P., Sha, L. & Strosnider, J.K. (1987). Enhanced aperiodic responsiveness in hard real-time environments. In *8th Real-Time Systems Symposium (RTSS '87)*, 261–270, San Jose, CA, USA. 30

Leontyev, H. & Anderson, J. (2009). A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, **43**:60–92. 150

Leung, J.Y.T. & Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, **2**(4):237–250. 21, 24

Lipari, G. (2012). Software and academic research: are we going in the right direction? Panel presentation at the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2012), Pisa, Italy, http://retis.sssup.it/waters2012/accepted/panel.pdf. Retrieved on July 31, 2013. 107

Lipari, G. & Baruah, S.K. (2000). Efficient scheduling of real-time multi-task applications in dynamic systems. In *6th IEEE Real-Time Technology and Applications Symposium (RTAS '00)*, 166–175, Washington , DC, USA. 33

Lipari, G. & Bini, E. (2010). A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *31st IEEE Real-Time Systems Symposium (RTSS '10)*, 249 –258. 46, 47, 48, 51, 105, 150

Lipari, G. & Buttazzo, G.C. (1999). Scheduling real-time multi-task applications in an open system. In *11th Euromicro Workshop on Real-Time Systems (RTS 1999)*, York, UK. 33

Lipari, G., Gai, P., Trimarchi, M., Guidi, G. & Ancilotti, P. (2005). A hierarchical framework for component-based real-time systems. *Electronic Notes in Theoretical Computer Science*, **116**:253–266, International Workshop on Test and Analysis of Component Based Systems (TACoS 2004). xiii, 2

Little, R. (1991). Advanced avionics for military needs. *Computing and Control Engineering Journal*, **2**(1):29–34. 3

Liu, C.L. (1969). Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, **37**(60):28–31. 14

Liu, C.L. & Layland, J.W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, **20**(1):46–61. 14, 20, 21

Lorente, J.L., Lipari, G. & Bini, E. (2006). A hierarchical scheduling model for component-based real-time systems. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece. xiii, 2, 5

Lupu, I., Courbin, P., George, L. & Goossens, J. (2010). Multi-criteria evaluation of partitioning schemes for real-time systems. In *15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, Bilbao, Spain. 21, 23

Masmano, M., Ripoll, I. & Crespo, A. (2005). An overview of the XtratuM nanokernel. In *1st International Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT 2005)*, Palma de Mallorca, Spain. 50

Masmano, M., Ripoll, I., Crespo, A., Arberet, P. & Metge, J.J. (2009). XtratuM: an open source hypervisor for TSP embedded systems in aerospace. In *DASIA 2009 "DAta Systems In Aerospace" Conference*, Istanbul, Turkey. 50

Masmano, M., Ripoll, I., Crespo, A. & Peiro, S. (2010). XtratuM for LEON3: an open source hypervisor for high integrity systems. In *Embedded and Real Time Software and Systems (ERTS^2 2010)*, Toulouse, France. 51

Matsubara, Y., Sano, Y., Honda, S. & Takada, H. (2012). An open-source flexible scheduling simulator for real-time applications. In *15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2012)*, Shenzhen, China. 54

McConnel, T. (2010). Embedded World — PikeOS 3.1 supports multicore. *EE Times*, http://www.eetimes.com/electronics-news/4136859/EMBEDDED-WORLD--PikeOS-3-1-supports-multicore. Retrieved on July 31, 2013. 49

Mercer, C.W., Savage, S. & Tokuda, H. (1993). Processor capacity reserves for multimedia operating systems. Tech. Rep. CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA. 29

Mignolet, J.Y. & Wuyts, R. (2009). Embedded multiprocessor systems-on-chip programming. *IEEE Software*, **26**(3):34–41. 7

Mok, A. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA. 15, 22

Mok, A.K. & Feng, A.X. (2002). Real-time virtual resource: A timely abstraction for embedded systems. In A.L. Sangiovanni-Vincentelli & J. Sifakis, eds., *2nd International Conference on Embedded Software (EMSOFT'02)*, vol. 2491 of *Lecture Notes In Computer Science*, 182–196, Springer-Verlag, London, UK. xv, 9

Mok, A.K., Feng, X.A. & Chen, D. (2001). Resource partition for real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, 75–84, Taiwan, ROC. 37, 38, 39, 45, 68

Nguyen, V., Deeds-Rubin, S. & Tan, T. (2007). A SLOC counting standard. In *The 22nd Int. Ann. Forum on COCOMO and Systems/Software Cost Modeling*, Los Angeles, USA. 136

Nikolic, B., Awan, M.A. & Petters, S.M. (2011). SPARTS: Simulator for Power Aware and Real-Time Systems. In *8th IEEE International Conference on Embedded Software and Systems (ICESS-11)*, Changsha, China. 112

Nóbrega da Costa, P., Craveiro, J.P., Casimiro, A. & Rufino, J. (2013). Safety kernel for cooperative sensor-based systems. In *Safecomp 2013 Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France. 147

Parkinson, P. (2011). Safety, security and multicore. In C. Dale & T. Anderson, eds., *Advances in Systems Safety*, 215–232, Springer London. 7

Patterson, D. (2010). The trouble with multi-core. *IEEE Spectrum*, **47**(7):24–29. 7

Patterson, D.A. & Hennessy, J.L. (2009). *Computer Orgnaization and Design: the Hardware/Software Interface*. Morgan Kaufmann, Burlington, MA, USA, 4th edn. xv, 7, 61

Penix, J., Visser, W., Engstrom, E., Larson, A. & Weininger, N. (2000). Verification of time partitioning in the DEOS scheduler kernel. In *22nd International Conference on Software engineering*, 488–497, Limerick, Ireland. 50

Phan, L.T.X., Lee, J., Easwaran, A., Ramaswamy, V., Lee, I. & Sokolsky, O. (2011). CARTS: A tool for compositional analysis of real-time systems. *ACM SIGBED Review*, **8**(1), Special Issue on 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2010). 53

Phillips, C.A., Stein, C., Torng, E., & Wein, J. (2002). Optimal time-critical scheduling via resource augmentation. *Algorithmica*, **32**(2):163–200. 105

Planche, L. (2008). Analysis of the application of the ARINC653 standard to space systems. Tech. Rep. ASG7.TN.12311.ASTR, EADS Astrium. xiv, 6

Plancke, P. & David, P. (2003). Technical note on on-board computer and data systems. European Space Technology Harmonisation, Technical Dossier on Mapping, Technical Note TOS-ES/651.03/PP, ESA. xiv, 6

RECOMP (2011a). Component model specification. Deliverable D2.1, RECOMP project. 147

RECOMP (2011b). Operating system support for safe multi-core integration: Architecture, implementation, evaluation. Deliverable D3.3, RECOMP project. 49, 147

RECOMP (2013). HW support for operating systems, applications and monitoring. Deliverable D3.4, RECOMP project. 147

Rosa, J., Craveiro, J. & Rufino, J. (2011). Safe online reconfiguration of time- and space-partitioned systems. In *9th IEEE International Conference on Industrial Informatics (INDIN 2011)*, Caparica, Lisbon, Portugal. 59, 143

Rufino, J., Filipe, S., Coutinho, M., Santos, S. & Windsor, J. (2007). ARINC 653 interface in RTEMS. In *DASIA 2007 "DAta Systems In Aerospace" Conference*, Naples, Italy. 57

Rufino, J., Craveiro, J. & Verissimo, P. (2010). Architecting robustness and timeliness in a new generation of aerospace systems. In A. Casimiro, R. de Lemos & C. Gacek, eds., *Architecting Dependable Systems VII*, vol. 6420 of *Lecture Notes in Computer Science*, 146–170, Springer Berlin / Heidelberg. 57, 58

Rufino, J., Craveiro, J., Schoofs, T., Cristóvão, J. & Tatibana, C. (2011). AIR: Technology innovation for future spacecraft onboard computing systems. In *International Conference on Computer as a Tool (EUROCON 2011)*, Lisbon, Portugal. 135

Rushby, J. (1999). Partitioning in avionics architectures: Requirements, mechanisms and assurance. NASA Contractor Report CR-1999-209347, SRI International, California, USA. xiv, 4, 5

Sánchez-Puebla, M.A. & Carretero, J. (2003). A new approach for distributed computing in avionics systems. In *1st International Symposium on Information and Communication Technologies (ISICT 2003)*, 579–584, Trinity College Dublin, Dublin, Ireland. xiv, 3

Santos, R., Behnam, M., Nolte, T., Pedreiras, P. & Almeida, L. (2011). Multi-level hierarchical scheduling in ethernet switches. In *11th International Conference on Embedded Software (EMSOFT '11)*, Taipei, Taiwan. xv, 9

Sha, L., Lehoczky, J. & Rajkumar, R. (1986). Priority inheritance protocols: an approach to real-time synchronization. In *Real-Time Systems Symposium (RTSS '86)*, New Orleans, LA, USA. 30

Sha, L., Abdelzaher, T., Årzén, K., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J. & Mok, A.K. (2004). Real time scheduling theory: A historical perspective. *Real-Time Systems*, **28**(2):101–155. 14

Shin, I. & Lee, I. (2003). Periodic resource model for compositional real-time guarantees. In *24th IEEE Real-Time Systems Symposium (RTSS '03)*, Cancun, Mexico. 40, 41, 53, 109, 139

Shin, I. & Lee, I. (2004). Compositional real-time scheduling framework. In *25th IEEE Real-Time Systems Symposium (RTSS '04)*, 57–67, Lisbon, Portugal. 39, 40

Shin, I. & Lee, I. (2007). Compositional real-time schedulability analysis. In I. Lee, J.Y.T. Leung & S.H. Son, eds., *Handbook of Real-Time and Embedded Systems*, chap. 5, Chapman & Hall/CRC. xvi, 10, 36

Shin, I. & Lee, I. (2008). Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, **7**(3):30:1–30:39. 40

Shin, I., Easwaran, A. & Lee, I. (2008). Hierarchical scheduling framework for virtual clustering of multiprocessors. In *20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, Prague, Czech Republic. 42, 51, 150

Silveira, R.P.O. (2012). *Design and implementation of a modular scheduling simulator for aerospace applications*. Master's project report, Faculty of Sciences, University of Lisbon, Lisbon, Portugal. 118

Singhoff, F., Legrand, J., Nana, L. & Marcé, L. (2004). Cheddar: a flexible real time scheduling framework. *Ada Letters*, **XXIV**(4):1–8. 53

Singhoff, F., Plantec, A., Dissaux, P. & Legrand, J. (2009). Investigating the usability of real-time scheduling theory with the Cheddar project. *Real-Time Systems*, **43**(3):259–295. 53

Sprunt, B., Sha, L. & Lehoczky, J. (1989). Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, **1**:27–60, 10.1007/BF02341920. 30, 33

Spuri, M. & Buttazzo, G. (1994). Efficient aperiodic service under earliest deadline scheduling. In *15th IEEE Real-Time Systems Symposium (RTSS '94)*, 2–11, San Juan, Puerto Rico. 31, 32, 33

Srinivasan, A. & Baruah, S. (2002). Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, **84**(2):93–98. 27, 31

Symtavision (2013). SymTA/S 3.3. Product note, Symtavision, http://www.symtavision.com/downloads/Flyer/Symtavision_SymTA-S_33_overview_EN_1302.pdf. Retrieved on July 31, 2013. 54

SYSGO (2013). PikeOS: Safe and Secure Virtualization. Pikeos 3.3 product datasheet, SYSGO, http://www.sysgo.com/nc/products/pikeos-rtos-and-virtualization-concept/. Retrieved on July 31, 2013. 49

van Kampenhout, J.R. & Hilbrich, R. (2013). Model-based deployment of mission-critical spacecraft applications on multicore processors. In *Reliable Software Technologies – Ada-Europe 2013*, vol. 7896 of *Lecture Notes in Computer Science*, 35–50, Springer Berlin Heidelberg. 7

Verissimo, P. & Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers. xiii, 1

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE Real-Time Systems Symposium (RTSS '07)*, 239–243, Tucson, AZ, USA. 29

Watkins, C. & Walter, R. (2007). Transitioning from federated avionics architectures to Integrated Modular Avionics. In *26th IEEE/AIAA Digital Avionics Systems Conference (DASC 2007)*, Dallas, TX, USA. 3

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. & Stenström, P.

(2008). The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, **7**(3):36:1–36:53. 15

Wind River (2010). Wind River VxWorks 653 Platform 2.3. Product note, Wind River, http://www.windriver.com/products/product-notes/PN_VE_653_Platform2_3_0410.pdf. Retrieved on July 31, 2013. 51, 52

Windsor, J. & Hjortnaes, K. (2009). Time and space partitioning in spacecraft avionics. In *3rd IEEE International Conference on Space Mission Challenges for Information Technology*, 13–20, Pasadena, CA, USA. xiv, 6

Xi, S., Wilson, J., Lu, C. & Gill, C. (2011). RT-Xen: Towards real-time hypervisor scheduling in Xen. In *11th International Conference on Embedded Software (EMSOFT '11)*, Taipei, Taiwan. xv, 9