PhD Thesis

Real-Time Software Transactional Memory

António Barros

CISTER-TR-180504

2018/04/29

Real-Time Software Transactional Memory

António Barros

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: amb@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

The current trend in the development of recent real-time embedded systems is driven by (i) a shift from single-core to multi-core platform architectures at the hardware level; (ii) a shift from sequential to parallel programming paradigms at the software level; and finally (iii) the ever increasing demand of new functionalities (e.g. additional tasks with specific timing requirements).These trends taken together increase the complexity of the system as a whole, and have a significant impact on the type of mechanisms that are adopted in order to guarantee both the functional and non-functional correctness of the system.This holds true especially in the case where these mechanisms have to maintain the correctness of data shared between different tasks executing concurrently in parallel.

The access to shared resources (e.g. main memory) on single-core systems has traditionally relied on lock-based mechanisms.At any time instant, a single task is granted an exclusive access to each shared resource.However, assuming the new settings, i.e. multi-core architectures executing a set of potentially parallel tasks sharing data, the big picture changes.Tasks executing in parallel on different cores and sharing the same data may have to compete before completing the execution.It has been proven that lock-based synchronisation approaches, which were sound in single-core context, do not to scale to multi-cores and, furthermore, they hinder the composability of the system, unfortunately.

On the path to solving these issues, Software Transactional Memory (STM) based approaches have been proposed as promising candidates.By using these alternative techniques, the underlying STM service would solve the conflicts between contending tasks while maintaining data consistency, and critical sections would be executed speculatively -- i.e. they are executed but if the result of the computation harms the system correctness, then changes made by the computation are reverted and the results are ignored.This way, the details on how to synchronise shared data would be hidden from the programmer, thus representing a significant advantage as compared to lock-based synchronisation techniques regarding the functional correctness of the system.Regarding the non-functional correctness instead, the use of STM based approaches in real-time systems also requires the tasks timing constraints to be met.This is due to the fact that each transaction aborting and repeating multiple times before its eventual commit incurs a timing overhead that might not be negligible and, therefore, must be taken into account to prevent deadline misses at runtime.

This work considers a set of potentially parallel real-time tasks sharing data and executed on a multi-core platform.Assuming this setting, first it proposes a complete framework where an STM service is associated to a set of fully partitioned scheduling algorithms in order to improve the predictability of the system as well as guaranteeing that the timing constraints are met for all the tasks.Then, it proposes the corresponding schedulability analysis for each pair of STM and scheduling algorithms.Finally, it proposes a lightweight syntax to enrich the original Ada programming language in order to support STM for concurrent real-time applications.

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time Software Transactional Memory

António Manuel de Sousa Barros

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Programa Doutoral em Engenharia Electrotécnica e de Computadores

Supervisor: Luís Miguel Rosário da Silva Pinho

May 29, 2018

Real-Time Software Transactional Memory

António Manuel de Sousa Barros

Programa Doutoral em Engenharia Electrotécnica e de Computadores

Abstract

The current trend in the development of recent real-time embedded systems is driven by (i) a shift from single-core to multi-core platform architectures at the hardware level; (ii) a shift from sequential to parallel programming paradigms at the software level; and finally (iii) the ever increasing demand of new functionalities (e.g. additional tasks with specific timing requirements). These trends taken together increase the complexity of the system as a whole, and have a significant impact on the type of mechanisms that are adopted in order to guarantee both the functional and non-functional correctness of the system. This holds true especially in the case where these mechanisms have to maintain the correctness of data shared between different tasks executing concurrently in parallel.

The access to shared resources (e.g. main memory) on single-core systems has traditionally relied on lock-based mechanisms. At any time instant, a single task is granted an exclusive access to each shared resource. However, assuming the new settings, i.e. multi-core architectures executing a set of potentially parallel tasks sharing data, the big picture changes. Tasks executing in parallel on different cores and sharing the same data may have to compete before completing the execution. It has been proven that lock-based synchronisation approaches, which were sound in single-core context, do not to scale to multi-cores and, furthermore, they hinder the composability of the system, unfortunately.

On the path to solving these issues, Software Transactional Memory (STM) based approaches have been proposed as promising candidates. By using these alternative techniques, the underlying STM service would solve the conflicts between contending tasks while maintaining data consistency, and critical sections would be executed speculatively – i.e. they are executed but if the result of the computation harms the system correctness, then changes made by the computation are reverted and the results are ignored. This way, the details on *how* to synchronise shared data would be hidden from the programmer, thus representing a significant advantage as compared to lock-based synchronisation techniques regarding the functional correctness of the system. Regarding the non-functional correctness instead, the use of STM based approaches in real-time systems also requires the tasks timing constraints to be met. This is due to the fact that each transaction aborting and repeating multiple times before its eventual commit incurs a timing overhead that might not be negligible and, therefore, must be taken into account to prevent deadline misses at runtime.

This work considers a set of potentially parallel real-time tasks sharing data and executed on a multi-core platform. Assuming this setting, first it proposes a complete framework where an STM service is associated to a set of fully partitioned scheduling algorithms in order to improve the predictability of the system as well as guaranteeing that the timing constraints are met for all the tasks. Then, it proposes the corresponding schedulability analysis for each pair of STM and scheduling algorithms. Finally, it proposes a lightweight syntax to enrich the original Ada programming language in order to support STM for concurrent real-time applications.

Sumário

A tendência actual no desenvolvimento de sistemas de tempo-real embutidos é caracterizada por (i) ao nível do *hardware*, a transição de arquitecturas baseadas em um único processador para múltiplos processadores num circuito integrado; (ii) ao nível do *software* pela transição de paradigmas de programação sequenciais para paralelos; e, finalmente, (iii) pela sempre crescente necessidade de novas funcionalidades (por exemplo, tarefas adicionais com requisitos temporais). Estes factores tomados em conjunto aumentam a complexidade do sistema de um ponto de vista geral, tendo um impacto significativo nos mecanismos que são adoptados para garantir a correcção funcional e não-funcional do sistema. Isto é especialmente verdadeiro no caso em que tais mecanismos têm que manter a correcção dos dados partilhados entre diferentes tarefas concorrentes executadas em paralelo.

O acesso a recursos partilhados (por exemplo, a memória principal) em sistemas com um único processador tem sido tradicionalmente gerido por mecanismos de bloqueio (habitualmente denominados de *locks*), i.e. estruturas que permitem bloquear temporariamente o acesso a esses recursos a pedido das tarefas. Em qualquer instante, o acesso a um recurso partilhado é apenas permitido em exclusivo a uma única tarefa. No entanto, assumindo o novo cenário de arquitecturas baseadas em múltiplos processadores num circuito integrado a executar um conjunto de tarefas potencialmente em paralelo, acedendo a dados partilhados em memória modifica o panorama em que assenta o desenvolvimento de sistemas. A execução de tarefas em paralelo em diferentes processadores que acedem aos mesmos dados poderão ter que competir entre elas até finalmente terminarem a sua execução. Infelizmente, a sincronização de tarefas baseada em mecanismos de bloqueio que são eficazes em sistemas mono-processador não são eficientes em arquitecturas com múltiplos processadores e, além do mais, têm um impacto negativo na composição do sistema.

As abordagens baseadas no conceito da *Software Transactional Memory* (STM) foram propostas como possíveis candidatos para resolver os problemas previamente mencionados. A utilização destas técnicas alternativas implicam um serviço que resolve conflitos entre tarefas concorrentes de forma a manter a consistência dos dados, enquanto as secções críticas das tarefas são executadas especulativamente – i.e. o resultado da execução da secção crítica pode ser descartado se comprometer a consistência dos dados. Desta forma, os detalhes sobre *como* sincronizar dados partilhados tornam-se transparentes para o programador, representando uma vantagem significativa sobre os mecanismos de bloqueio, do ponto de vista da correcção funcional do sistema. Mas no que toca à correcção não-funcional, é necessário que a utilização de abordagens baseadas em STM garantam que as restrições temporais das tarefas sejam respeitadas. Isto deve-se ao facto de que abortar e repetir uma transação múltiplas vezes até finalmente concluir com sucesso implica custos adicionais em tempo de execução que poderão ser significativos, que poderão traduzir-se em tarefas a ultrapassar os seus prazos em tempo de execução.

Esta tese considera um conjunto de tarefas com características de tempo-real, potencialmente paralelas, que partilham dados em memória, executadas numa plataforma com múltiplos processadores num circuito integrado. Assumindo este cenário, em primeiro lugar propõe uma estrutura completa em que um serviço de STM é associado a um conjunto de algoritmos de escalonamento particionado, de forma a incrementar a previsibilidade do sistema e garantir que as restrições temporais de todas as tarefas são respeitadas. Depois, propõe a respectiva análise de escalonabilidade para cada associação de STM vs. algoritmo de escalonamento. Por fim, propõe uma sintaxe para aumentar a linguagem de programação Ada, de forma a suportar STM no desenvolvimento de aplicações concorrentes de tempo-real.

Acknowledgements

First, I would like to express my deepest gratitude to my supervisor, Prof. Luís Miguel Pinho, for giving me the opportunity to work with him and undertake this research work. His guidance, support, motivation and care were crucial to overcome the many obstacles that we faced along the way, and complete this personal endeavour. I always felt comfortable to discuss any topics and issues with Miguel, and I am aware and grateful for all I have learned due to this kind openness. I must stress that Miguel demands no less than high standards, which occasionally made me quite nervous, thinking on how to properly handle the task in hand. But I must stress even more how he always encouraged me and pushed me to carry on when work seemed overwhelming.

I am immensely grateful to Patrick Meumeu Yomsi, who is in fact acting as a co-supervisor. We started working together only in 2015 but yet, much of him is in this thesis, as he helped me in so many, many ways. Always with a smile, he seriously motivated me to do things *the right way* – *"Keep pushing, man!"* – and this is reflected in this manuscript. I learned a lot from Patrick, and I certainly improved myself a lot with him.

I am very lucky to have these two great men as friends.

Thanks to my colleagues and friends at CISTER. I would like to acknowledge Eduardo Tovar for creating such a dynamic and challenging and yet, friendly and cosy environment. I would like to thank my colleagues for all the great ideas we discussed, but also for all the coffee break chit-chats that so many times help us to resume work with a much better state of mind, especially Ricardo Severino, Gurulingesh Raravi, Dakshina Dasari, Hossein Fotouhi, Maryam Vahabi, Ricardo Garibay-Martínez, Muhammad Ali Awan, Hazem Ali and Cláudio Maia. A special thanks for Paulo Baltarejo Sousa for all the help dealing with the linux kernel: I can't pay you all those days we spent programming, debugging, recompiling the kernel.

I also want to express my deepest gratitude to ISEP for all the institutional support to this PhD work, and to the Department of Informatics Engineering for doing everything to alleviate me as much as it was possible from my teaching-related labour, so I could pursue this endeavour. I deeply acknowledge my colleagues for their support and comprehension.

Finally I want express my infinite gratitude to my family. To my parents for their unconditional love, and who have always encouraged me to put my best on all the things I do. To my parents and parents-in-law for loving my daughters Núria and Alícia so much, and helping me with al those "small" daily tasks that represent so much. And specially to my wife, Dulce, who always believed in me, even when I was not so sure I could complete this thesis. I am aware of all the sacrifices she endured these last years. There are no words to thank for all the encouragement and companionship I received from Dulce.

António Barros

List of publications

Journal papers included in this thesis

António Barros, Luis Miguel Pinho, Patrick Meumeu Yomsi. Non-preemptive and SRP-based fully-preemptive scheduling of real-time Software Transactional Memory. *Journal of Systems Architecture (JSA)*, 61(10):553–566, Nov 2015.

Conference or Workshop papers included in this thesis

António Barros, Patrick Meumeu Yomsi, Luis Miguel Pinho. *Response time analysis of hard real-time tasks sharing software transactional memory data under fully partitioned scheduling*. In *Proceedings of the 11th IEEE International Symposium on Industrial Embedded Systems (SIES 2016)*, Krakow, Poland, May 2016.

António Barros, Luis Miguel Pinho. *Non-preemptive scheduling of Real-Time Software Transactional Memory*. In *Proceedings of the Conference on Architecture of Computing Systems (ARCS 2014)*, Lübeck, Germany, February 2014.

António Barros, Luis Miguel Pinho. *Revisiting Transactions in Ada*. In *Proceedings of the 15th International Real-Time Ada Workshop (IRTAW-15)*, pages 84-92, Fuente Dé, Spain, September 2011.

António Barros, Luis Miguel Pinho. *Software transactional memory as a building block for parallel embedded real-time systems*. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011)*, pages 251–255, Oulu, Finland, September 2011.

António Barros, Luis Miguel Pinho. *Managing contention of software transactional memory in real-time systems*. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010), Work-In-Progress Session*, San Diego, U.S.A., December 2010.

To Núria and Alícia.

Contents

List of Figures

List of Tables

Acronyms and Symbols

List of acronyms

ASTM	Adaptive STM
BF	Best Fit
BFD	Best Fit Decreasing
BHP	Bounded direct blocking with High Parallelism
CAS	Compare-And-Swap
CM	Contention Manager
DDR	Double Data Rate synchronous dynamic random-access memory
DP	Dynamic Priority scheduler
DS	Data set
DSTM	Dynamic STM
ECM	EDF Contention Manager
EDF	Earliest Deadline First
FBLT	First Bounded, Last Timestamp contention manager
FIFO	First In, First Out
FIFO-CRT	First In First Out Contention manager for Real-Time systems
FF	First Fit
FFD	First Fit Decreasing
FP	Fixed Priorities
FJP	Fixed-Job Priority scheduler
FMLP	Flexible Multiprocessor Locking Protocol
FTP	Fixed-Task Priority scheduler
G-EDF	Global Earliest Deadline First
L1	Level-1 cache
L2	Level-2 cache
L3	Level-3 cache
LCM	Length-based Contention Manager
LLF	Least Laxity First
McRT-STM	Multi-core RunTime STM
MIC	Many Integrated Core architecture
MPB	Message Passing Buffer
M-PCP	Multiprocessor Priority Ceiling Protocol
M-SRP	Multiprocessor Stack Resource Protocol
MWCAS	Multi-Word Compare-And-Swap
NoC	Network on a Chip
NPDA	Non-Preemptive During Attempt
NPUC	Non-Preemptive Until Commit

NUMA	Non-Uniform Memory Access
OMLP	O(m) Locking Protocol
OSTM	Object-based STM
PCP	Priority Ceiling Protocol
PD	Pseudo-Deadline Pfair algorithm
PD^2	PD algorithm with simplified tie-breaking mechanism
P-EDF	Partitioned Earliest Deadline First
PF	Proportionate Fair
Pfair	Proportionate fair scheduling
P-FP	Partitioned Fixed Priority
PIP	Priority Inheritance Protocol
PNF	Priority contention manager with Negative values and First access
P-PCP	Parallel Priority Ceiling Protocol
QPA	Quick convergence Processor-demand Analysis
RCM	RMA Contention Manager
RM	Rate Monotonic
RNLP	Real-time Nested Locking Protocol
RR	Round-Robin
RS	Read set
R/W RNLP	Multiple reads and mutually exclusive write Real-time Nested Locking Protocol
SCC	Single-chip Cloud Computer
SRP	Stack Resource Protocol
SRPTM	SRP for Transactional Memory
STM	Software Transactional Memory
TL2	Transactional Locking 2
U-EDF	Unfair but Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks
UMA	Uniform Memory Access
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
WF	Worst Fit
WFD	Worst Fit Decreasing
WFI	Worst Fit Increasing
WS	Write set

List of symbols

Task set parameters.

τ	A task set.
n	The number of tasks of τ.
τ_i	The i^{th} task of τ.
C_i	The worst-case execution time of τ_i.
T_i	The minimum inter-arrival time of τ_i.
D_i	The relative deadline of τ_i.
R_i	The worst-case response time of τ_i.
U_i	Task utilisation of τ_i.
U_τ	Task set utilisation of τ.
P	Hyper-period of the task set.

Job parameters.

$\tau_{i,j}$	The j^{th} job of τ_i.
$r_{i,j}$	The release time of $\tau_{i,j}$.
$d_{i,j}$	The absolute deadline of $\tau_{i,j}$.
$f_{i,j}$	The time instant at which $\tau_{i,j}$ finishes executing.
$rt_{i,j}$	The response time of $\tau_{i,j}$.
$W_{i,j}$	Transaction overhead of job $\tau_{i,j}$.
$A_{i,j}$	Number of aborts of transaction ω_i executed by job $\tau_{i,j}$.

Hardware platform parameters.

π	The set of cores.
m	The number of cores of π.
π_k	The k^{th} core of π.
$\sigma(\tau_i)$	Function that returns the core to which τ_i is assigned.

Transaction parameters.

ω_i	Transaction executed by task τ_i.
C_{ω_i}	The maximum execution time required to execute the sequential code of transaction ω_i.
$C_{\text{a}-\omega_i}$	Execution time required by the non-transactional section of τ_i executed before transaction ω_i.
$C_{\text{p}-\omega_i}$	Execution time required by the non-transactional section of τ_i executed after transaction ω_i.
DS_i	Data set of ω_i, i.e. the collection of transactional objects accessed by ω_i.
RS_i	Read set of ω_i, i.e. the collection of transactional objects accessed exclusively for reading by ω_i.
WS_i	Write set of ω_i, i.e. the collection of transactional objects accessed modified by ω_i.
$\lvert\text{DS}_i\rvert$	Size of the data set of ω_i.
$\lvert\text{RS}_i\rvert$	Size of the read set of ω_i.
$\lvert\text{WS}_i\rvert$	Size of the write set of ω_i.

STM parameters.

O	Set of STM objects.
o_j	The j_{th} object of O.
p	The number of objects of O.
$\|o_j\|$	The number of transactions that access o_j.

Contention group parameters.

G	A contention graph, in which transactions are represented as vertices. Any two transactions that have intersecting data sets are connected by an edge.
Ω_a	A contention group, i.e. a group of transactions that share intersecting data sets.
Π_a	Subset of cores allocated to contention group Ω_a
m_a	The number of cores in Π_a.

SRPTM.

λ_i	Preemption level of task τ_i.
λ_{ω_i}	Preemption level of transaction ω_i.
Λ_k	Ceiling of core π_k.
$ceil(o_k)$	Ceiling of transactional object o_k.
$ceil(\Omega_g)$	Ceiling of contention group Ω_g.

NPUC analysis.

$\upsilon(k)$	Transaction ω_i on the k^{th} position in a given sequence of transactions ordered by their release times.
$R^{(k)}$	WCRT of a transaction ω_i on the k^{th} position in a given sequence of transactions ordered by their release times.
$\mathscr{S}_i$	Set of all possible simple paths that converge to transaction ω_i.
$\mathscr{S}_{i,k}$	Subset of all simple paths on length k that converge to transaction ω_i.
R_{ω_i}	Worst-case response time of transaction ω_i.
C_{Ω_g,π_ℓ}	Longest execution time of all the transactions in contention group Ω_g that are assigned to core π_ℓ.
I_{Ω_g,π_k}	Maximum inter-core interference that any transaction in contention group Ω_g assigned to core π_k can experience.
$L_{\text{a}-\omega_i}$	Longest busy-period that occurs until transaction ω_i is released.
B_i	Blocking term associated to lower priority tasks.

SRPTM analysis.

$R_{\text{a}-\omega_i}$	Worst-case response time of the non-transactional section of code executed before transaction ω_i.
$R^*_{\omega_i}$	Worst-case response time of the last two attempts of transaction ω_i.
DB_i	The longest direct blocking term that task τ_i can experience.
IB_i	The longest indirect blocking term that task τ_i can experience.

Chapter 1

Introduction

Real-time systems are defined as those for which the correctness depends not only on the logical result of the computation, but also the time at which it is produced (Stankovic, 1988). When embedded in some form to control its environment, such a system consists of a set of tasks that interact and must process inputs from the environment in order to provide the adequate outputs within a pre-defined time window. This pre-defined time window is dictated by the requirements of the controlled system – i.e. the system's timing constraints. Examples of real-time systems include industrial computer-controlled systems, avionics applications such as airplane control, train braking systems, or real-time video processing such as the detection of people in front of a moving vehicle.

The timing constraints of a real-time system can be classified based on the consequences of meeting or not the timing requirements of each task (Burns, 1991). For a given system, when a task deadline miss may lead to a total failure of the entire system, then this task is referred to as a *hard real-time task*. When the integrity of the system is not at stake, that is the system can occasionally accommodate a few task deadline misses, but the benefit of this flexibility is drawn to zero after the deadline, i.e., the result produced is not useful, then the task is referred to as a *firm real-time task*. In the case the consequence of a task deadline miss is limited to a potential degradation of the Quality of Service (QoS) of the system, then the task is referred to as a *soft real-time task*. Obviously, a system may consist of a combination of all these three types of tasks. However, as soon as a single task has hard real-time requirements, then the whole system is stamped as *hard real-time*.

Traditional design of embedded real-time systems relies on the following assumptions about the external closed environment: (1) only a limited set of input sources is admissible, (2) only a limited set of functionalities is allowed and (3) only a limited set of output actions can be taken. With these assumptions, the design techniques often adopted for the allocation and management of the system resources in order to supply the pre-defined service requirements tend to be based on static rather than dynamic approaches (Klein et al., 1993). Following the trend described by

Hansen et al. (2001), contemporary embedded real-time systems increasingly combine the requirements of traditional closed systems and the challenges of an open, heterogeneous, and dynamic environment. These systems have to cope with the dynamic evolution of the environment while maintaining a predefined quality of service.

Embedded architectures are following a major evolution. We have witnessed a paradigm shift from single core to multi-core platforms, due to thermal and capacitive limitations in Moore's Law prediction. The use of multi-core architectures in the embedded domain offers the ability to execute various tasks in parallel. However, the number of available cores significantly impacts the way these systems are implemented. Supporting the concurrency and the parallel interaction between tasks and the inherent use of data sharing mechanisms are tremendous challenges to account for in this case. These challenges exacerbate even more when it is necessary to guarantee the non-functional requirements of the application (Anderson et al., 2006). Therefore, it is essential to provide innovative and efficient mechanisms that allow developing dynamic applications while guaranteeing their reliability.

Instead of developing sequential code functions that are then made concurrent through operating system calls, it is well accepted that there are several advantages in writing concurrent programs with the semantics of concurrency in the programming language (Burns and Wellings, 2009). Real-time embedded applications interact with the external world (people, cars, robots, conveyor belts, planes, etc.) in situations that are inherently parallel and distributed. Their programs have to deal with various components like interrupt handlers (which react to internal/external events), clock handlers (which react to clock ticks) and managers (which monitors system state, task states and modes of operation). They also have to execute tasks corresponding to specific functionalities while taking into account real-world properties. If the concurrent nature of the system is explicitly supported by the semantics of the programming language, then the program becomes more readable, maintainable and reliable (Sutter and Larus, 2005). However, this feature comes along with a number of new problems, the language models and mechanisms for concurrent programming must be assessed in the same manner as the application functional and non-functional requirements are scrutinized.

In summary, the actual context for real-time embedded applications is characterised by the growing demands on functional requirements that results in an increase in the number of software components; and the shift from single-core to multi-core platforms which increases the complexity of the interactions between those components. Current approaches try to ease the development process by providing means to build correct applications through the selection and integration of correct components (composability) while ensuring the timing requirements. This approach hides from the programmer the interactions between components by applying mechanisms that maintain the consistency of shared data in a deterministic way. This can be achieved through programming languages and operating systems that expose the appropriate mechanisms for concurrent interaction and data sharing.

1.1 Problem definition

On single-core platforms, task interaction and access to shared resources traditionally rely on locking mechanisms (Anderson et al., 1997) such as semaphores (Dijkstra, 1965) and monitors (Hoare, 1974). These mechanisms became commonplace as means to provide sections of code, also referred to as a *critical section*, an exclusive access to shared resources. However, locking mechanisms come along with at least two challenges. The first challenge is the lack of composability (Sutter and Larus, 2005), i.e. correct components can be selected and when they are assembled together, they form a non-correct application; and the second challenge is the exposure to priority inversion in priority-based preemptive scheduling (Sha et al., 1990; Baker, 1991).

On multiprocessor systems, although additional computing resources are available on the platform, further issues arise. Locks can be classified according to their granularity. As such, we distinguish between: (1) the *coarse-grained locks* where a single lock controls the access to a large fraction of the available shared resources, and (2) the *fine-grained locks* where a single lock controls the access to a single or a small fraction of the available resources. Coarse-grained locks impede the progress of non-conflicting tasks that request access to the same lock, thus degrading the system throughput. On the other hand, fine-grained locks increase the complexity of system design because the same rule for acquiring locks must be applied across the whole system, thus hindering composability. Another issue is the *convoy effect* (Bershad, 1993). Here, a task that is holding a lock may have its execution paused or even halted (e.g. due to a preemption by a higher-priority task, an interruption by a event handler, or a core shut-down), thus delaying the completion time of all blocked tasks.

Currently, two directions are considered in order to tackle these issues. The first direction consists in adapting previous and/or devise new locking mechanisms for parallel systems, as proposed by Gai et al. (2001), Block et al. (2007), Easwaran and Andersson (2009a,b). The second direction consists in devising non-blocking mechanisms, as proposed by Tsigas and Zhang (1999), Anderson and Holman (2000), Brandenburg et al. (2008), Sarni et al. (2009), Cotard (2013) and El-Shambakey and Ravindran (2013b).

Adapting and/or devising new locking mechanisms is not a safe path to follow as they do not allow us to fully circumvent the hurdles previously discussed. In contrast, non-blocking mechanisms appear to be a suitable alternative as long as it is possible to include them in real-time systems. They are already used in distributed and highly-parallel systems with satisfactory results. They rely on the concept of *non-blocking data object*. Here, conflicting accesses are managed by an underlying mechanism that is responsible of maintaining the consistency of the shared data object. Every piece of code is written without critical sections, and as such a task will not block when it needs to access a resource.

On multiprocessor systems, non-blocking objects present strong conceptual advantages (Tsigas and Zhang, 1999). They have proven to perform better than lock-based solutions in several scenarios (Brandenburg et al., 2008). Priority inversion and deadlock are eliminated because accesses to the object proceed in parallel and concurrent accesses can always be resolved in favour of the higher-priority tasks. The convoy effect is also eliminated because no task will block, waiting for another task that is failing or prevented from executing. These advantages provided by non-blocking objects are somehow balanced by the complexity of the mechanisms that are required to maintain the consistency of each shared object. This complexity depends on the guarantee of progress provided by the non-blocking data object. This guarantee of progress can be classified in the following three categories: *wait-free*, *lock-free* or *obstruction-free* (Herlihy and Shavit, 2008), given with the following interpretation. A wait-free object ensures that every call will finish in a finite number of steps independently of the pace of execution of the other tasks. In this category, all tasks are ensured to progress. A lock-free object relaxes the progress condition, so that the system is broadly able to progress, although some tasks may suffer from starvation. In this category, any update of a data object will only take effect if no conflict occurs; otherwise, only one access is granted to complete, while the remaining contenders will fail and will be deemed to retry. An obstruction-free object ensures that one task will eventually progress if it executes in isolation relative to other contending tasks. Thus, this category provides the most relaxed guarantee of progress.

Despite the interesting features of non-blocking objects, critical sections usually provide atomicity to a composition of operations. A section of code that operates shared memory space in an atomic manner and with no explicit locks is referred to as a *transaction*. Herlihy and Moss (1993) described how to add extensions in the cache-coherence protocol of a multiprocessor architecture to implement the concept of *transactional memory*. In contrast to the approach proposed by Herlihy and Moss (1993) at the hardware level, Shavit and Touitou (1995) adapted the concept of transactional memory in order to implement all the synchronisation operations at the software level. They named it Software Transactional Memory (STM). The STM presents a huge advantage over the hardware-based approach: it is portable in the sense that it can be implemented seamlessly to multiple architectures; and it is hardware agnostic in the sense that there is no need to modify the circuitry of the chip. For this reason, to manage the access to shared memory regions we opted for STM-based techniques in this work.

The very first STM-based approach was proposed by Shavit and Touitou (1995) and only supported static transactions, i.e. transactions and memory locations are predefined before runtime. Later, other researchers proposed implementations of STM that support dynamical memory usage and dynamical transactions, i.e. each transaction can decide which addresses to access based on values read at runtime. However, the underlying synchronisation mechanisms are diverse. For example, Fraser's STM (Fraser, 2003) applies lock-free mechanisms. The Dynamic STM (DSTM) (Herlihy et al., 2003) uses obstruction-free mechanisms. Ennals' STM (Ennals, 2005) applies an optimistic concurrency control for reads, but uses locks to protect objects that will be written by a transaction. This STM approach is implemented along with a mechanism that avoids deadlock when deciding in favour of a determined transaction.

In general, a transaction is executed in isolation and must complete either by *committing* (i.e. when it succeeds) or by *aborting* (i.e. when it failed). This holds true irrespective of other transactions executing in parallel. In case of an abort, the transaction is deemed to retry. Data consistency is maintained because transactions operate on private copies of shared data, so conflicting transactions will not race over the same data. Before completion, all accessed locations are checked for conflicting updates that may have occurred during the execution of the transaction. If no conflict is detected, then data is consistent and updates become effective. Otherwise, when a conflict has been detected, a contention policy is applied in order to allow, at least, one transaction to commit.

In order to achieve the intended guarantee of progress, every contention policy dictates how to solve conflicts among transactions. Typically, the contention policy must select the transaction to commit based on a criteria that meets the expected behaviour of the STM (Herlihy et al., 2003). It is also important that the contention policy avoids livelock, i.e. when two transactions are in conflict, they should not be able to abort each other indefinitely.

From the previous discussion, Software Transactional Memory is an interesting approach to solve the scalability and composability problems of lock-based approaches. Nevertheless, its use in real-time systems entails the support of appropriate mechanisms that would guarantee that all the timing requirements are met. The adopted contention policy must not only be functionally correct, but also its timing behaviour must be analysable. Furthermore, it should be possible to couple the programming model with the underlying implementation of these algorithms.

This thesis focuses on the problem of sharing data and synchronisation in concurrent real-time software executing on multi-core platforms.

1.2 Relevance of the subject

Because the traditional solution to increase the computing power of processors (i.e. reducing the size of components and increasing the clock frequency) reached physical limits (thermal and capacitive), chip manufacturers opted for packing multiple cores on a single chip. The current trend to push further in this direction is leading to *massive multi-core* or *many-core* processors. That is chips including tens to hundreds of cores, interconnected with switched networks. A few examples include the Tilera Tile64Pro which features 64 cores and the Tilera TILE-Gx8072 which features 72 cores (Tilera, 2012, 2015); the Intel SCC which features 48 cores (Intel, 2010); the Intel Xeon Phi which features 60 cores (Intel, 2012); the Kalray MPPA featuring 256 cores (Kalray, 2015); and the Adapteva Epiphany featuring 1024 cores (Adapteva, 2012). These many-core architectures allow multiple applications to be deployed on the same chip, thus maximizing the hardware utilisation, and reducing the cost, size, weight, and power requirements. They also allow the designers to improve the performance of the applications through parallelism. Nevertheless, all this computing power raises a number of challenges, including the core-to-core and the core-to-memory communications. In addition, cache coherency becomes a cornerstone especially for platforms encompassing a large number cores (Choi et al., 2011). Furthermore, platforms can either be homogeneous, with symmetric or asymmetric multiprocessing; or heterogeneous, with different core types. This also substantially impacts in the way applications share data.

Focusing on multi-core platforms, multiple solutions have been proposed exposing or not coherency between caches. One of the first examples of non-cache coherent approaches was provided by the experimental Intel Single-chip Cloud Computer (SCC) (Intel, 2010). This chip contained 24 tiles with two Pentium cores each, connected with a 4x6 2D-mesh, and a shared message passing buffer (MPB). Another platform exhibiting non-cache coherency between cores is the Kalray MPPA (Kalray, 2015). This platform consists of 16 clusters of 16 computing cores each, plus 32 cores that are dedicated to the management of the computing clusters, the I/O and the interconnect. Another architecture proposed by Intel is the Many Integrated Core architecture (MIC) (Intel, 2012). In this family, the Xeon Phi integrates 60 (Pentium-based) cores, but connected through a dual ring-bus. It features software-based coherency between caches. The Tilera (Tilera, 2012) architecture offers a different cache coherency solution. Each tile consists of a single core, with its own private cache and it is connected to the other tiles through several parallel NoCs (iMesh). On this platform, the tiles can be aggregated and form separate domains, with separate cache-coherent areas. The Epiphany platform (Adapteva, 2012) presents a similar organisation.

Recent architectures influence the way applications should be implemented in order to fully exploit the parallel computing power provided by these platforms. A promising candidate to this implementation paradigm is the thread-level parallel programming approach. By using this implementation methodology, the concurrency model of the tasks is prone to simultaneous requests of tasks running on different cores to shared resources, such as the main memory. Hence, the adopted synchronisation mechanism to manage the access to the shared resources and thus solve conflicts among transactions must be sharp, precise and well thought.

In the arena of admissible solutions, we adopt an STM-based approach as STM provides all the features to help us benefiting from the advantages of a concurrent programming paradigm, while limiting their potential disadvantages. More precisely, STM has proven to scale well with an increase of the number of cores available on a single processor chip (Dragojević et al., 2011). It delivers a higher throughput in comparison to coarse-grained locks and does not increase design complexity as compared to fine-grained locks (Rossbach et al., 2010). Last but not least, it is worth noticing that STM-based approaches present very good performances in terms of transaction abort ratio for systems with a low ratio of context switching during the execution of the transactions and with a predominance of (1) read-only transactions; and (2) transactions with a short execution time (Maldonado et al., 2010).

In this context, i.e., opting for an STM-based approach, the programmer writes sequential code as usual, but he has to specify through annotations the portions of the code that are to be executed as transactions. Although many optimisation techniques can be applied at the moment of breaking the ties between multiple transactions in conflicts (either a *read-write* or a *write-write* conflict), the role of the contention policy is to guarantee a meaningful and efficient serialisation of the access of the contending transactions to the shared resource. In this process, one transaction will always be granted the access, and thus the right to execute and commit, while the other contending transactions will be fated to abort, and consequently, will be deemed to repeat. The synchronisation details are seamlessly handled by an underlying mechanism that maintains the consistency of the shared data objects located at the *transactional memory*.

In the parallel computing domain, the focus of STM-based approaches is limited to the system throughput, i.e., the number of commit per time unit. Here, the contention management policy is designed in such a manner that livelocks are avoided and starvation (i.e., any transaction constantly being aborted by the contenders) is minimized.

In the real-time systems domain, in addition to ensure that every transaction will eventually commit as this is the case in the parallel computing domain, the timing requirements of every task must also be satisfied. This means that we must also pay a special attention to the time by which every commit occurs. Consequently, when designing an STM-based approach for real-time systems, we must have two objectives in mind: (1) guarantee that the number of aborts of every transaction is upper-bounded and (2) make sure that the deadline of every task is met. To this end, a sound and convincing schedulability analysis must be conducted. This is one of the main contributions of this work.

1.3 Main thesis preposition

Considering the context presented in Section 1.2, the main objective of this thesis is to support the use of Software Transactional Memory in real-time embedded systems. The central preposition of this dissertation is:

> ***The use of software transactional memory improves the composability of applications for multi-core embedded systems with real-time requirements.***

This can be accomplished by designing a contention management policy for the execution and management of the transactions, associated to a scheduling algorithm for the execution of the tasks. These two components carefully developed and put together will result in a very limited number of aborts for every transaction as a guarantee of the timing requirements of every task. The adopted programming model to go along with these components will also need to be augmented with the required mechanisms to enable the correct use of this type of data sharing approach.

1.4 Thesis contributions

Considering the previous main thesis preposition, the main contributions of this work are as follows:

C1: *Design and implementation of a fair and predictable contention management algorithm.*

The contention management policy for real-time applications (FIFO-CRT) proposed in this thesis serialises the transactions in such a manner that their timing attributes are considered. Unlike in lock-based synchronisation where a job suspends its execution while the desired resource is not free (thus yielding the processor to other ready tasks), here, a job holds the processor during the time it attempts to commit the transaction in execution. The designed policy provides a fair opportunity to commit to every transaction and is agnostic to the underlying task scheduling policy. In addition, it avoids deadlocks by making it possible for a later released transaction to abort an older released and preempted transaction. Along this line, we propose a set of rules on the scheduler associated to FIFO-CRT so as to prevent multiple simultaneous active transactions on each core, as this would result in an improvement of the responsiveness for each task.

C2: *Design, implementation and analysis of three scheduling algorithms.*

Associated to the FIFO-CRT policy, we designed and analysed three fully-partitioned scheduling algorithms, denoted as Non-Preemptive Until Commit (NPUC), Non-Preemtive During Attempt (NPDA) and Stack Resource Protocol for Transactional Memory (SRPTM). These three scheduling algorithms are based on the classical Partitioned EDF (P-EDF) scheduler. Specifically, they take the same scheduling decisions as P-EDF when there is no transaction in progress; otherwise the scheduling decisions are adjusted accordingly by following predefined sets of metrics. This results in at most one active transaction per core, thus improving the predictability of the FIFO-CRT policy.

In addition, we implemented these three scheduling algorithms together with FIFO-CRT on a multi-core based computer hosting a PREEMPT-RT-patched Linux kernel version and tested with a large group of synthetic task sets.

C3: *Specification of the language support in Ada.*

The use of transactional code must be as transparent as possible for the programmer. Based on previous work on transactions for fault-tolerant systems, we propose in this thesis a syntax for the Ada programming language to support software transactional memory for concurrent real-time applications.

1.5 Outline

This thesis is structured as follows.

Chapter 2 (Background on real-time embedded systems) provides an insight on the state-of-the-art on real-time embedded systems and real-time scheduling. Chapter 3 (Background on synchronisation mechanisms) presents an overview of synchronisation mechanisms for architectures based on multi-processor systems. Chapter 4 (System model) sets the system model and the assumptions adopted in this manuscript. It introduces all the parameters that will be used throughout this research work by distinguishing between three levels of abstraction, namely: the task specifications, the platform and scheduler specifications, and finally the STM specifications.

Chapter 5 (FIFO-CRT: a predictable STM contention management) formalises the FIFO-based contention management algorithm, referred to as First-In-First-Out Contention manager for Real-Time systems (FIFO-CRT), designed to provide predictability and prevent transaction starvation.

Chapter 6 (Scheduling tasks and transactions under FIFO-CRT) presents three scheduling policies – named (1) Non-Preemptive Until Commit (NPUC); (2) Non-Preemptive During Attempt (NPDA); and (3) SRP for Software Transactional Memory (SRPTM) – to address specific issues that impact on the predictability of FIFO-CRT. Chapter 7 (Schedulability analysis of tasks under NPDA, NPUC and SRPTM) reports on the schedulability analyses associated to these scheduling policies.

Chapter 8 (Evaluation) presents the simulation testbed that has been developed to compare the performance of our STM-based approaches (i.e. the FIFO-CRT contention manager associated to NPUC, NPDA and SRPTM scheduling policies) against the state-of-the-art lock-based synchronisation mechanism named Flexible Multiprocessor Locking Protocol (FMLP). This performance comparison is conducted by using the throughput of atomic sections executed by concurrent tasks and the number of deadline misses as the main metrics and a qualitative evaluation between all these approaches is also proposed. In order to analyse the performance of our synchronisation approaches and the precision of the theoretical analyses from a practical viewpoint, we implemented a minimalistic STM system with the FIFO-CRT contention manager as a Linux user space service, and the three scheduling policies in the Linux kernel. The results of these experiments are presented in Chapter 9 (Implementation).

Chapter 10 (Conclusions) summarises the contributions of this work and provides future research directions that can build upon the results of this thesis.

Appendix A (Ada language support for transactions) elaborates on the specifications of the programming language mechanisms that we proposed to support the execution of transactions. To this end, we discuss how the syntax of the Ada programming language can be adapted so as to support STM transactions.

Finally, Appendix B (Bounded-memory multi-version STM for real-time systems) elaborates on how to execute read-only transactions free of aborts by tuning a multi-version STM. Unlike for general-purpose STM systems, the timing characteristics of real-time task sets offer the opportunity to determine the exact number of versions for each data object.

Chapter 2

Background on real-time embedded systems

An *embedded system* is a computer that interacts with its surrounding physical elements and is implemented for a specific purpose. Figure 2.1 illustrates the general building blocks of such a system[1]. An embedded system is meant to control a larger system in which the computer is inserted. It perceives and receives stimuli from the physical system through *sensors*, and makes use of *actuators* to modify the state of the controlled physical system. Embedded systems have an innumerable range of applications such as smartphones, vehicle electronic stability control, industrial process control or rocket guidance systems, to name a few. Unlike a general purpose computer that performs a wide range of unforeseen tasks, the hardware design of an embedded system addresses the requested functionality. As such, the hardware components are selected to fit the requirements in order to reduce the final unit cost, which is very important in large scale applications or in mass-production commercial products.

Most embedded systems have to deal with real-time constraints and are as such called *real-time embedded systems*. These special embedded systems are those in which the correctness of the system depends not only on the correctness of the logical results of the computations, but also on the time at which these results are produced (Stankovic, 1988). Traditionally, the functionality of a real-time system is divided into less complex parts that are referred to as *tasks*. A task is a sub-program that is responsible of a particular part of the system. It is characterised by specific timing requirements and can be invoked recurrently for a potentially infinite number of times. Every invocation is also known as a *job* of the task. Each job is triggered either by time or an event, and must complete within a given time frame, defined by a deadline. The consequences of a job not being able to finish within the time frame depend on the criticality of the task. In some cases, violating the temporal constraints can lead to catastrophic consequences. For example,

[1]An embedded system communicates with other systems via dedicated *communication* technologies, and with a human operator via a *user interface*.

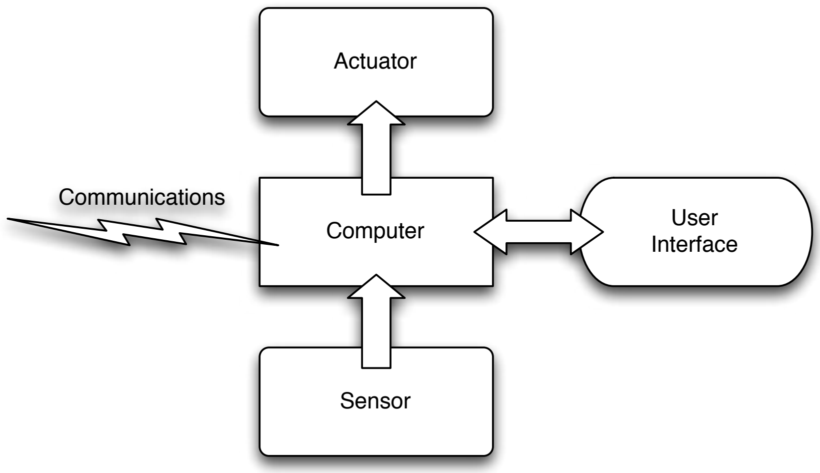


Figure 2.1: Typical embedded system building blocks.

if the thrust reverser systems accidentally take too long to activate after an aircraft touches the ground, the aircraft may not decelerate rapidly enough to avoid overrunning the runway. In other cases, the consequences can be experienced as a degradation in the quality of service, without posing any catastrophic risk. For example, if a runner's wrist GPS watch occasionally loses the fix on satellites, it may be annoying (the runner may temporarily not know his pace) but it will not have harmful consequences on the runner's integrity. Therefore, there is a special concern on how to schedule the set of tasks with sufficient resources to ensure that critical deadlines are met. A *schedulabilty analysis* allows to determine before runtime if a given scheduling algorithm will be able to meet the timing requirements of all the tasks on a target computing platform. This analysis must be rigorous for (critical) systems, ensuring that *no temporal constrains will be violated*; and must ensure that the timing requirements will be observed such that *the quality-of-service provided by the system is within the expected* for systems that are not critical.

This chapter presents an overview of the real-time scheduling of tasks for multi-core systems, and discusses relevant works published on this topic.

2.1 Modelling real-time systems

We recall that a real-time system distinguishes itself by the temporal constraints that augment the functional requirements of its constituent tasks. This section describes how the functionality and workload of a real-time application is modelled by the concepts of *task* and *job*, according to the classic periodic/sporadic task model, as formalised by Mok (1983).

Definition 1 (Task)**.** A *task* is an executable entity of workload that defines one part of the functionality provided by the system. A task is denoted as τ_i, where $i \in \mathbb{N}^+$ is the task index.

Each task τ_i is usually characterised by three parameters — C_i, D_i and T_i — which are given with the following interpretation:

- a *worst case execution time (WCET) — C_i*. This parameter defines the maximum processor time without interruption required by any job of task τ_i.
- a *relative deadline — D_i*. This parameter defines the maximum time window required by each job of task τ_i to complete its execution.
- an *activation interval — T_i*. This parameter defines the time span between two consecutive jobs of task τ_i. If T_i is constant, i.e., every job is released as soon as it is legally permitted to do so, then τ_i is called *periodic*; else if T_i is a minimum, then τ_i is called *sporadic*; otherwise, τ_i is called *aperiodic*.

Each task τ_i is further characterised by the relationship between its relative deadline D_i and its activation interval (also referred to as period) T_i.

- *Implicit deadlines.* Task τ_i is referred to as implicit deadline if $D_i = T_i$.
- *Constrained deadlines.* Task τ_i is referred to as constrained deadline if $D_i \leq T_i$.
- *Arbitrary deadlines.* Task τ_i is referred to as arbitrary deadline if D_i and T_i are independent.

Definition 2 (Task utilisation)**.** The *task utilisation* is the maximum ratio of the processor capacity used by the task. For task τ_i, it is denoted as U_i and is formally defined as $U_i = C_i/T_i$.

Definition 3 (Task set)**.** The *task set* is the collection of all tasks that compose the system under analysis. A task set which consists of n tasks is denoted as τ and defined as $\tau = \{\tau_1, \ldots, \tau_n\}$.

Definition 4 (Task set utilisation)**.** The *task set utilisation* is the maximum ratio of the processor capacity used by the whole task set. For the task set τ, it is denoted as U_τ and is formally defined as the sum of the utilisation of all individual task utilisations, i.e., $U_\tau = \sum_{i=1}^{n} U_i$.

This research considers sporadic implicit deadline task sets.

The sporadic implicit deadline task model of execution assumes that every task consists of a potentially infinite sequence of jobs such that (*i*) every two consecutive jobs are separated by at least the task's period and (*ii*) the relative deadline of the task is equal to the task's period, i.e., the execution of every job must finish before the following job is released. From now onward, we refer to the j^{th} job of task τ_i as $\tau_{i,j}$. Each job $\tau_{i,j}$ is characterised by a set of timing attributes — $r_{i,j}$, $s_{i,j}$, $d_{i,j}$, $f_{i,j}$, $rt_{i,j}$ — that are relative to the instant at which the job is released.

- The *release time — $r_{i,j}$*. This attribute corresponds to the time at which the job is released and becomes ready for execution.
- The *start time — $s_{i,j}$*. This attribute corresponds to the time at which the job actually starts executing. The job may not be scheduled upon its release, so $s_{i,j} \geq r_{i,j}$.
- The *absolute deadline — $d_{i,j}$*. This attribute defines the time by which the job must have completed its execution. Formally, $d_{i,j} = r_{i,j} + D_i$.

- The *finish time* — $f_{i,j}$. This attribute defines the time at which the job completes its execution. Ideally, we should have $f_{i,j} \leq d_{i,j}$, otherwise the job misses its deadline.
- The *response time* — $rt_{i,j}$. This attribute defines the time elapsed between the job release time and the job finish time. Formally, $rt_{i,j} = f_{i,j} - r_{i,j}$.

A task set can be categorised according to the time at which jobs of every task are released. Hence, we can distinguish between synchronous and asynchronous task sets.

- *Synchronous task set,* when there exists a time instant such that all the tasks release a job at that time instant, i.e., $\exists t \geq 0$ such that $\forall i \in \{1, \dots, n\}$, $\exists j \geq 1$ and $r_{i,j} = t$. Without any loss of generality, we can always assume that $j = 1$ (i.e., the first job) and $t = 0$ in this case.
- *Asynchronous task set,* when the task set in not synchronous. In contrast to the previous case, we can always assume that this is when $\exists i, j \in \{1, \dots, n\}$ such that $i \neq j$ and $r_{i,1} \neq r_{j,1}$.

This research considers synchronous task sets.

The response time may vary from one job to another for the same task, due to the fluctuations of the system workload at the time each job is released. In a real-time system domain, it is desirable that the response time of all the jobs of a task meets the deadline constraint. For a sound schedulabitlity analysis of a system, the maximum response time that all the jobs of a task may experience is very important. Therefore we define the worst-case response time of a task as follows.

Definition 5 (Worst-case response time of a task)**.** The *worst-case response time* of a task, say τ_i, denoted as R_i, is defined as the longest response time of all jobs of task τ_i. Formally speaking, $R_i = \max_{j \geq 1}(rt_{i,j})$.

The strictness of the deadline of each real-time task allows us to distinguish between two categories of tasks (Stankovic and Ramamritham, 1990).

- *Hard real-time tasks.* In this category, all the jobs of every task, say τ_i, must complete their execution before their corresponding deadline, otherwise the job is valueless and constitutes a failure of the system with potentially catastrophic consequences. Consequently, for such a task it must hold true that $R_i \leq D_i$.
- *Soft real-time tasks.* In this category, the execution of some jobs of such a task, say τ_j, are allowed to exceed their corresponding deadline, without posing any risk to the integrity of the system. In this case, a deadline miss can be seen as a degradation of the quality-of-service of the system. Consequently, for such a task the following condition is acceptable within statistical boundaries: $\exists k \geq 1, \;\; rt_{j,k} > D_j$.

The strictness of the deadlines across tasks of a task set allows us to categorise it as *hard* or *soft*.

Definition 6 (Hard real-time system)**.** A *hard real-time system* is one which consists *only* of hard real-time tasks.

Definition 7 (Soft real-time system)**.** A *soft real-time system* is one which contains *at least* one soft real-time task.

From Definition 6 and Definition 7, it follows that hard real-time systems are those that must be scrutinised in order to guarantee that no deadlines will ever be missed, otherwise the system will fail. In contrast, soft real-time systems must be analysed to guarantee that the probability of deadline misses leads to an acceptable quality-of-service.

This research focuses on hard real-time systems.

2.2 Modelling the computing platform

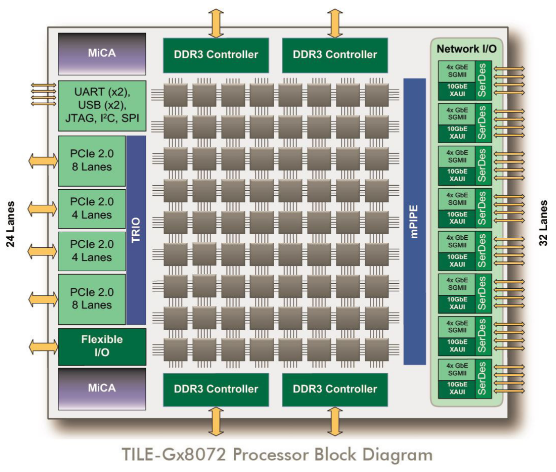
Real-time embedded systems can be built upon a wide range of computer platforms (e.g., architectures ranging from 4 to 64-bit), depending on the field of applications for which the system is designed for. Before the advent of multi-core chips, there were two approaches to allocate the tasks to the computing platform.

- In *uniprocessor systems*, the task set is entirely executed by one processor which is the only resource and must be shared among all tasks along time. Thus the challenge reduces to a scheduling problem. In this case, the processor computing power must be sufficient to guarantee all the task timing requirements.

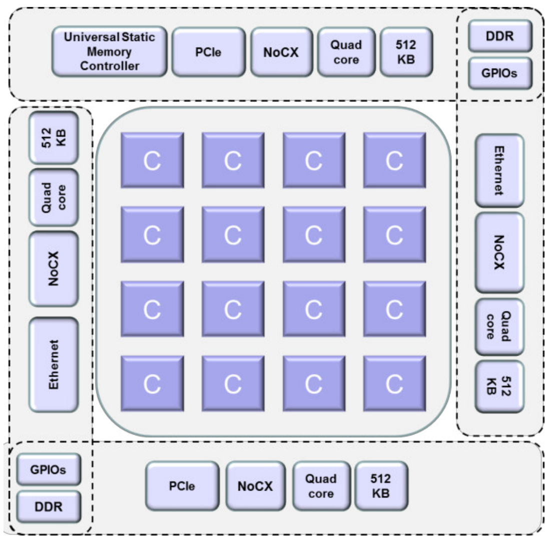
- In *multiprocessor systems* the task set is executed by more than one processor and thus parallel executions among processors are possible. Thus, in addition to the scheduling problem on each core, a task-to-processor mapping problem adds to the challenge. In this case, the timing requirements must be guaranteed by the processors computing power and by the reliability of the computing interconnect between the processors.

Due to the increases in the transistor scale integration, multi-core chips allows multiple processors on the same die. The miniaturisation process in the semiconductor technology reached the stage where further processing power enhancements related to uniprocessor systems are no longer affordable. Nowadays, the computing power required to support the requirements of many embedded applications cannot be provided by uniprocessor chips and, consequently, they are being replaced by multi-core chips. This paradigm shift has a relevant impact on the design of embedded software that besides being concurrent, it must now address parallelism as a performance factor, and specially in scheduling and mapping the tasks.

Multi-core is a platform in which multiple processing cores with the capability of operating independently share the same chip and the platform resources, e.g., the physical memory space and the I/O devices. This is generic enough to accommodate some design diversity, including the core



(a) The Tilera TILE-Gx8072. Source: EZchip Semiconductor, Inc.



(b) The Kalray MPPA-256. Source: KALRAY Corporation.

Figure 2.2: Two multi-core processors for embedded applications.

architectures, the core interconnect, the caches and the memory accesses. Figure 2.2 illustrates this diversity. Figure 2.2a illustrates the EZchip (former Tilera) TILE-Gx8072 (Tilera, 2015) and Figure 2.2b illustrates the Kalray MPPA-256 Kalray (2015).

- *Core architectures.* A multi-core chip can hold a collection of cores with the same or different architectures. Taking in account the abundance of multi-core designs, this multiprocessors can be classified according to the diversity of computing architectures in the chip.

  - *Identical multi-cores.* These architectures consist of identical cores, with the interpretation that all the cores have the same computing capability and are interchangeable.
  - *Uniform multi-cores.* For these architectures, each core, say π_i, is characterised by its own speed or computing capacity s_i, with the interpretation that a job that executes on π_i for t time units completes $s_i \cdot t$ units of execution.
  - *Unrelated multi-cores.* For these architectures, there is an execution rate $s^k_{i,j}$ associated to each job-core pair, say $(\tau_{i,j}, \pi_k)$, such that $\tau_{i,j}$ completes $s^k_{i,j} \cdot t$ units of execution when it executes on core π_k for t time units.

- *Core interconnect.* Previously, multi-core designs used a shared bus as core interconnect. Lately, as the number of cores on a single chip has drastically increased for performance reasons, the contention and consequently the core-to-core and core-to-memory latencies have followed the same trend in terms of data transfer. To mitigate this issue, chip manufacturers are putting considerable efforts in implementing switched networks on chip (NoC). This aim aligns neatly with the desired "wish-list" of most embedded systems.

- *Caches.* The speed of the memory lags behind the speed of current multi-core platforms. Cache is a component that replicates the a small portion of the memory allowing faster data accesses. Caches can be shared between a group of cores, or each core may have exclusive access to its private cache. A *cache hit* occurs when the running operation can be performed by using data available in the cache; otherwise, the outcome is a *cache miss*, i.e. when the data or part of the data is not available in the cache. Multi-core platform are implemented with either cache coherency or non-cache coherence mechanisms. A cache coherency protocol ensures that all cache line replicas among all the cores are equal. This cost circuits and time. Thus, non-coherent caches are cheaper (in terms of circuits) and faster as they produce fewer stalls due to cache synchronisation and especially false sharing, i.e. an efficient implementation in a core will not be hampered by memory operations initiated in other cores. However, they require synchronisation support.
- *Memory accesses.* The access to the main memory differs between various platform designs. We distinguish between *uniform memory access* (UMA) and *non-uniform memory access* (NUMA). UMA denotes systems in which the time to access data in the main memory is independent from the memory address and the core that requests the memory operation, whereas NUMA denotes systems in which the time to access main memory depends on the memory address and/or on the core that requests the operation.

This research focuses on identical multi-core chips with NoC-based interconnects, non-coherent caches and is agnostic about the memory access mechanism.

In Figure 2.2, both chips consist of identical processing cores: seventy two 64-bit cores in the TILE-Gx8072, and 256 32-bit cores in the Kalray MPPA-256. Note that the processing cores in the Kalray MMPA-256 are organised in 16 clusters of 16 cores, which in turn are interconnected via a NoC; whereas the cores are directly interconnected through a 2-D mesh on the Tile-Gx8072. Furthermore, the MPPA-256 is scalable as it uses a NoC external interface that allows to connect multiple chips on the same board. This is not the case for the TILE-Gx8072. On another front, each core of the TILE-Gx8072 has private L1 and L2 caches and a shared L3 cache, which are all coherent across the chip. In contrast, each core in the Kalray MPPA-256 has only a private L1 cache, and no cache coherency mechanism implemented. The TILE-Gx8072 chip includes four DDR3 memory controllers for main memory operations, whereas each cluster of the Kalray MPPA-256 contains a multibank shared memory of 2 MBytes and an additional memory can be accessed through two DDR3 channels.

2.3 Real-time scheduling paradigms

A very important aspect of hard real-time systems consists of scheduling tasks in such a way that all the deadlines are met. The *scheduling algorithm* is responsible for this operation, i.e., to sequence the tasks onto the processors for execution by the operating system. This operation is performed by allocating the ready jobs so that the processing capacity is used in an efficient manner. In general, operating systems use scheduler(s) to sequence the ready jobs for execution and each scheduler is characterised by the scheduling algorithm it runs. In this context, a bunch of very important results, techniques and intuitions on scheduling algorithms have been developed so far, especially for single core architectures. Hence, it would not be reasonable to discuss them exhaustively. However, it is important to note that a task scheduler does not need to run continuously, it is activated by the operating system only at the *scheduling points*[2]. In summary, real-time embedded systems employ either *clock-driven*, *event-driven* or *hybrid* schedulers. This classification is given with the following interpretation.

Definition 8 (Clock-driven scheduler)**.** In a *clock-driven scheduler*, the scheduling points are defined at the time instants marked by clock interrupts generated by a periodic timer.

Definition 9 (Event-driven scheduler)**.** In an *event-driven scheduler*, the scheduling points are defined by the occurrence of certain events which preclude clock interrupts.

Definition 10 (Hybrid scheduler)**.** In an *hybrid scheduler*, the scheduling points are defined by a combination of clock interrupts and event occurrences.

$\triangleright$ Examples of clock-driven schedulers include “table-driven” and “cyclic” schedulers. These schedulers are also called “off-line schedulers” as they fix the schelule before runtime, i.e., before the system starts to run. To this end, a fixed length of time is divided into slots and each slot is assigned to a task for execution. At runtime, the pre-defined schedule repeats over and over. These schedulers are simple, efficient and incurs of very small overheads at runtime, which aligns with the requirements of embedded systems with limited resources and a small number of tasks. However, they do not handle sporadic and aperiodic tasks per se since the exact time of occurrence of these tasks cannot be predicted.

$\triangleright$ Examples of event-driven schedulers include “priority-driven” schedulers such as Rate Monotonic (RM) and Earliest Deadline First (EDF) (Liu and Layland, 1973). These schedulers are more adaptive to dynamic solicitations of the tasks. In constrast to table-driven schedulers, the scheduling decisions are taken at runtime based on the *priority*[3] of the ready jobs. Given a task, say τ_i, the priority assigned to each job, say $\tau_{i,j}$ with $j \geq 1$, by following RM is inversely proportional to T_i, the period of τ_i. This priority in EDF is defined by the absolute deadline of the job ($d_{i,j} = r_{i,j} + D_i$, where $r_{i,j}$ and D_i are the release time and the relative deadline of τ_i, respectively). Priority-driven schedulers can be classified according to the complexity of the priority policy scheme they use. As such, we can distinguish between *Fixed-Task Priority* (FTP), *Fixed-Job Priority* (FJP) and *Dynamic Priority* (DP) schedulers. These three classes of schedulers are given with the following interpretation.

[2]Scheduling points of a scheduler are the points on the time line at which the scheduler makes decisions regarding which task is to be executed next.

[3]The priority is a value that is assigned to each job by following a priority assignment strategy.

- *FTP schedulers*. Given a task, the priority assigned to each job is constant and equal to the ones assigned to all other jobs of the task. This priority is based on a parameter which does not vary with time. RM (Liu and Layland, 1973) is an example of such a scheduler.
- *FJP schedulers*. Given a task, the priority assigned to each job is constant, but may differ from one job to another of the task. According to this statement, it follows that FTP schedulers are a special case of FJP schedulers. EDF (Liu and Layland, 1973) is an example of such a scheduler.
- *DP schedulers*. Given a task, the priority assigned to each job is allowed to vary with time. Least Laxity First (LLF) (Mok, 1983) is an example of such a scheduler.

▷ Examples of hybrid schedulers include Round-Robin (RR). This scheduler handle all the tasks without priority in a cyclic manner. In short, RR is similar to the "First In First Out" (FIFO) scheduler, but with pre-defined time quantum maintained for preemption.

This research considers FJP schedulers.

Assuming a single core, FJP schedulers can be categorised according to their ability to pause the execution of the running job prior to its completion in order to execute another job with a higher priority. Hence, we distinguish between *preemptive*, *non-preemptive* and *limited preemptive* schedulers, with the following interpretation.

- *Preemptive schedulers*. These schedulers allow to pause the execution of the running job at the release of each job with a higher priority for this job to execute.
- *Non-preemptive schedulers*. These schedulers preclude pausing the execution of the running job, from its execution start time to its completion time, irrespective of whether there are pending jobs with a higher priority.
- *Limited preemptive schedulers*. These schedulers allow pausing the execution of the running job, but only at pre-defined scheduling points between its execution start time to its completion time, to execute other jobs.

This research considers preemptive schedulers.

When multiple cores are available on the target platform for the execution of a given task set, a new dimension, orthogonal to the previous one, adds to the problem: besides deciding *when* to schedule each job, it must also be decided *where* to execute this job. To deal with this new challenge, schedulers on multi-core platforms are traditionally classified as belonging to one of the following two categories: *global schedulers* or *partitioned schedulers* (Carpenter et al., 2004). These two categories differ by their ability to allow tasks to migrate among cores during their execution.

- *Global schedulers*. These schedulers do not perform a task-to-core mapping at design time. Instead, the scheduler manages a single ready queue and decides at runtime the core on which the selected ready job will execute. As such, different jobs belonging to the same task may happen to execute on different cores. Further, assuming such a scheduler, jobs are allowed to start their execution on a core and to complete on a different core from the one on which they have started. This allows us to classify global schedulers according to the granularity of the task migrations, as follows.
  - *Task-level migration*, where various jobs of a task are allowed to be assigned to different cores, but once a job is assigned to a core, migration of this job prior to its completion is forbidden.
  - *Job-level migration*, where various jobs of a task are allowed to be assigned to different cores, and migration of each job prior to its completion is also allowed.

  Assuming a global scheduling policy and a platform with m cores, the scheduler allocates the cores to the m highest priority jobs at every time instant in a *work conserving* manner. That is, the scheduler allows a core to idle only when there is no ready job pending for execution. As such, this scheduling paradigm allows for load-balancing among cores at runtime. However, it comes at the cost of job migrations. Although it is possible to make use of the scheduling algorithm techniques originally designed for single-core platforms as there is a single queue of ready jobs to be managed, specific multi-core algorithms have been proposed and have proven to achieve more efficient processor utilisation (Baker, 2010). Along this line, the global schedulers have witnessed the development of numerous and sometime sophisticated algorithms over the past 30 years. In this category, the Pfair (Baruah et al., 1996) family of schedulers has been devised to provide proportionate-fairness to the progress of each task with respect to its utilisation factor: PF (Baruah et al., 1996); PD (Baruah et al., 1995); and PD^2 (Srinivasan, 2003) are just few examples of such schedulers. All these schedulers are *optimal*, with the interpretation that if any other global scheduler is capable of successfully scheduling a given task set τ, then each of these schedulers is also capable to do so. However, the fairness property results in a high number of context switches and migrations among cores. This may drastically penalise the system in terms of runtime overhead costs. An alternative to Pfair schedulers includes U-EDF (Nelissen et al., 2012), which is an unfair but optimal global scheduler. Another limitation of most global schedulers is recognized in the literature as the *Dhall effect* (Dhall and Liu, 1978) as the jobs are scheduled in a greedy manner. To illustrate this claim, let us consider a multi-core platform, say π, consisting of $m > 1$ identical cores and let us consider either the global-RM or the global-EDF scheduler to manage the queue of jobs ready for execution. Also, let us consider a hard real-time system consisting of only $m+1$ tasks, say $\tau = \{\tau_1, \tau_2, \ldots, \tau_m, \tau_{m+1}\}$, where the first m tasks, say τ_i with $i \in \{1, \ldots, m\}$, have as parameters: $C_i = 2\varepsilon$ and $T_i = 1$ for any $0 < \varepsilon << 1/2$, and task τ_{m+1} has as parameters: $C_{m+1} = 1$ and $T_{m+1} = 1 + \varepsilon$. The total system utilisation is given by $U_\tau = m \cdot \frac{2\varepsilon}{1} + \frac{1}{1+\varepsilon}$ and we have

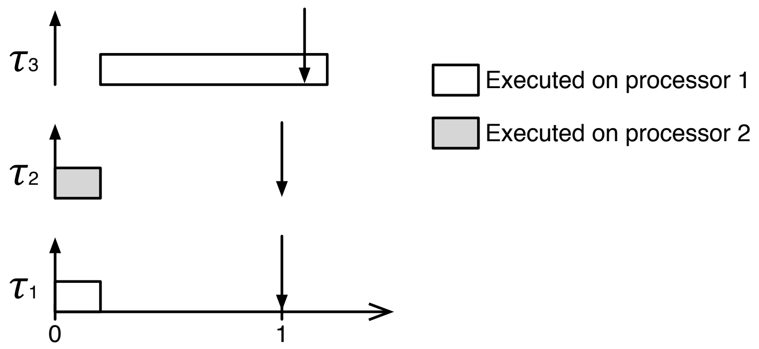


Figure 2.3: Dhall's effect. Job τ_3 misses the deadline because it cannot use the available capacity of the two processors when they overlap in time.

$U_\tau < m$. Note that $U_\tau \to 1$ when $\varepsilon \to 0$. Dhall and Liu (1978) have shown that even though this system may have a very low utilisation in comparison to the platform capacity, it is not schedulable on π by following both the global-RM and the global-EDF schedulers. Indeed, by following any of these schedulers at runtime, the m first tasks, i.e, $\tau_1, \ldots, \tau_m$, are selected for execution at time $t_0 = 0$ and are assigned to the cores in a one-to-one manner, as these tasks have the highest priorities. Then task τ_{m+1} is selected for execution at time $t_1 = 2\varepsilon$. The cores idle simultaneously at this time. While in this scenario, the first m tasks easily meet their deadlines, unfortunately the picture changes for task τ_{m+1}. As a matter of fact, this task will not have enough room to meet its deadline on any of the cores (see Figure 2.3 for the special case where $m = 2$). In this figure, the jobs of tasks τ_1 and τ_2 are assigned to cores π_1 and π_2 at time $t_0 = 0$, respectively, as they have a higher priority than task τ_3. At time $t_1 = 2\varepsilon$, the two cores idle simultaneously and τ_3 can start executing. However, τ_3 can execute for at most $((1+\varepsilon) - 2\varepsilon) = 1 - \varepsilon$ time units prior to its absolute deadline (at $d_3 = 1 + \varepsilon$) on any core. Since τ_3 cannot execute in parallel on the cores due to its sequential nature, there will be a deadline miss at time d_3 as $1 - (1 - \varepsilon) = \varepsilon$ time units will remain to be executed. These issues have slow down the investigations on global schedulers for almost two decades (Davis and Burns, 2011). In addition, experimental results by Bastoni et al. (2010) have shown that global scheduling approaches do not scale with an increase in the number of cores.

- *Partitioned schedulers*. These schedulers perform a task-to-core mapping at design time and do not allow any migration among cores at runtime. As such, every task is assigned to a core on which all of its jobs will execute.

  Assuming such a scheduler and a platform with m cores, the design time task-to-core mapping has been shown to be equivalent to a bin-packing problem, which in turn has been shown to be NP-hard (Garey and Johnson, 1979). Assuming a special mapping, a local ready job queue is managed on each core. Each local scheduler determines the job execution sequence based on the subset of tasks that has been assigned to this core. Despite the limitations of these schedulers, (e.g.: (1) They are not optimal in the sense that they may not be able to find a valid schedule for a task set for which there actually exists one (Baker, 2010); (2) They are unable to dynamically balance the workload between the cores as migrations are forbidden among cores at runtime; (3) They are not work-conserving as a core can idle while there are ready jobs pending for execution on other cores, etc.), they present the huge advantage of transforming a multi-core scheduling problem into a finite number of single core scheduling problems. This allows for example to take advantage of the precious scheduling theory and practice developed in this context over the years. Also, the Dhall effect is avoidable provided that a cautious task-to-core mapping is performed. These facts contribute in a non negligible manner to the attractiveness of partitioned schedulers to close the scheduling problem gap between uni-core and multi-core platforms (Davis and Burns, 2011). In addition, partitioned schedulers are the most used in the embedded systems industry as they provide very interesting features as reported below (Baker, 2010; Brandenburg, 2011).

- *Isolation of scheduling failures.* If a task misses a deadline, the resulting scheduling effects are contained and tied up only to the subset of tasks assigned to the same core.
- *No migration overheads.* As migrations are forbidden, there is no overheads related to preempting a job on one core and migrating its context on another core.
- *No global queue overheads.* As each core manages its own ready-queue, the overhead related to manipulating a single global queue, which might be considerable as it grows with the number of cores, is avoided.
- *Extensive knowledge of unicore scheduling.* For a given task-to-core mapping, the extensive knowledge, practice and results developed over the years for single core platforms can be reused to derive a system-level schedulability test provided that the tasks are sufficiently independent (Baker, 2010).

In between these two extremes, few alternatives have emerged and have proven to be viable for scheduling tasks onto multi-core platforms. *Semi-partitioned schedulers* (Kato et al., 2009; Dorin et al., 2010) and *cluster-based schedulers* (Calandrino et al., 2007; Baker and Baruah, 2008) are few examples. Semi-partitioned schedulers perform a task-to-core mapping at design time. All tasks that are successfully assigned in this phase are tied up to their corresponding cores and are not allowed to migrate at runtime. The remaining tasks, i.e., those that could not be successfully assigned to a core, are allowed to migrates in order to seek for a valid schedule. On the other hand, cluster-based schedulers partition tasks onto clusters of cores, and a global scheduler is applied inside each cluster. Note that the clusters may be of different sizes in terms of number of cores and may allow to align clusters with the underlying hardware topology, especially in order to take advantage of the cache sharing configuration between cores. This approach has shown to scale well on large heterogeneous multi-core systems (Bastoni et al., 2010).

This research considers partitioned schedulers.

2.4 Relevant works in the real-time scheduling theory

Following the discussion conducted in the previous sections we recall that this research made assumptions on three fronts, i.e., (f_1) the type of systems considered; (f_2) the type of target platforms; and finally (f_3) the adopted scheduling paradigm.

- ▷ Regarding (f_1): this research focuses on hard real-time systems which are modelled by using synchronous, sporadic and implicit deadline tasks (see Section 2.1).
- ▷ Regarding (f_2): the target platform is assumed to be agnostic about the memory access mechanism and composed of identical multi-cores with a NoC-based interconnect and non-coherent caches (see Section 2.2).
- ▷ Regarding (f_3): this research considers partitioned, preemptive and FJP schedulers (see Section 2.3).

In this context, preemptive scheduling algorithms have been developed, analysed and used in systems with real-time requirements since the 1960s. Liu and Layland (1973) developed two schedulers, referred to as Rate Monotonic (RM) and Earliest Deadline First (EDF), for hard real-time systems while assuming a single core platform. RM is an FTP scheduler which assigns a priority to each task according to its period: *The smaller the period of a task, the higher its priority*. Here, every job issued from a task inherits the priority of this task. On the other hand, EDF is an FJP scheduler which assigns a priority to each job according to its absolute deadline: *The closer the absolute deadline of a job, the higher its priority*. Assuming this latter scheduling policy and a single core platform, Liu and Layland proved that EDF is capable of successfully scheduling any hard real-time system τ, consisting of synchronous, sporadic and implicit deadline tasks, provided the following two conditions hold:

c_1: The total utilisation of τ does not exceed one (i.e., $U_\tau \leq 1$).

c_2: Tasks are independent (i.e., tasks do not share any other resource but the core).

Assuming that these two conditions hold, they proved that EDF is optimal among preemptive FJP schedulers. In the same vein, they showed that RM is optimal among preemptive FTP schedulers. As such, RM has become the *de facto* scheduler for hard real-time systems as it is simple to implement and it guarantees the schedulability of the jobs of each task independently from their absolute deadlines. Indeed, the higher the priority of a task, the higher the probability of its jobs to meeting their timing requirements. Liu and Layland showed that RM is guaranteed to successfully schedule τ only if $U_\tau \leq n \times \left(2^{\frac{1}{n}} - 1\right)$, where n denotes the number of tasks in τ. Note that $n \times \left(2^{\frac{1}{n}} - 1\right) \to ln(2) \simeq 0.69$ when $n \to +\infty$. This rather low bound has been revised and improved so far. For example, Bini et al. (2003) showed that τ is guaranteed to be successfully schedulable by following the RM policy if $\prod_{i=1}^{n}(U_i + 1) \leq 2$, where $U_i = C_i/T_i$ is the utilisation factor of task τ_i. This second test dominates the test proposed by Liu and Layland.

These results that guarantee the schedulability of a task set according to a given scheduler by using information on the utilisation of the system are referred to as *utilisation bound tests*. Such a test is formally defined as follows.

Definition 11 (Utilisation bound test)**.** Let τ be a task set with utilisation U_τ to be scheduled by following a scheduler A on a platform π. The *utilisation bound test* of A, denoted as U_A, is defined as the largest value of U_τ that guarantees the schedulability of τ by A upon π. Formally, τ is guaranteed to be schedulable by A if $U_\tau \leq U_A$ holds true.

A necessary condition for a sporadic implicit deadline task set τ to be schedulable upon a single core platform π by following any scheduler A is that the utilisation U_τ does not exceed the processor capacity, i.e. $U_\tau \leq 1$. On another front, EDF is an optimal scheduler for such a task set upon a single core platform. That is, EDF is capable of finding a valid schedule (i.e., a schedule in which all deadlines are met for all tasks) if one exists and a sufficient condition for this to hold true is $U_\tau \leq 1$ (see condition c_1). Consequently, the utilisation bound of EDF for a sporadic implicit deadline task set is $U_{\text{EDF}} = 1$. This bound provides an *exact* test for EDF since it cannot be further improved whereas the utilisation bounds derived for RM provide only a sufficient test. As a matter of fact, it is a trivial exercise to build a sporadic implicit deadline task set, say τ_0, such that $U_{RM} < U_{\tau_0} \leq 1$, but τ_0 is schedulable by following RM.

An alternative schedulability test for the EDF scheduler is based on the so-called *demand bound function* (dbf) (Baruah et al., 1990). The dbf is an abstraction of the computation requirements of tasks which has been observed to correlate very closely with schedulability property of the task set. This test is valid even for sporadic (*non*-implicit) deadline task sets on single core platforms.

Definition 12 (dbf (Baruah et al., 1990))**.** Let τ be a sporadic constrained deadline task set comprising of n tasks. The *dbf* for any task τ_i at any positive time instant t, denoted as $\text{dbf}(\tau_i,t)$, is defined as the maximum cumulative execution requirement of jobs of τ_i in any interval of length t. Formally, $\text{dbf}(\tau_i,t)$ is defined as follows.

$$\text{dbf}(\tau_i,t) \overset{\text{def}}{=} \max\left\{0, \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1\right\} \cdot C_i. \tag{2.1}$$

From Equation 2.1, note that $\text{dbf}(\tau_i,t)$ is a step-case function in t with first step occurring at time $t = D_i$ and subsequent steps separated by exactly Ti time units. the *dbf of* τ is defined as $\text{dbf}(\tau,t) \overset{\text{def}}{=} \sum_{i=1}^{n} \text{dbf}(\tau_i,t)$.

Definition 13 (dbf-based test)**.** Let τ be a sporadic constrained deadline task set comprising n tasks to be scheduled by following EDF on a single core platform π. The *dbf-based test* of τ upon π is defined as the test that verifies if $\text{dbf}(\tau,t)$ is always kept below the platform capacity at any positive time instant t. To this end, this operation is performed for the scenario where all tasks simultaneously release a job at time $t = 0$ (i.e., the *critical instant*), and all subsequent jobs for every task are released as soon as it is legally permitted to do so (i.e., the *worst-case scenario*).

Formally, τ is guaranteed to be schedulable by EDF if and only if $U_\tau \leq 1$ and

$$\mathrm{dbf}(\tau, t) \leq t, \quad \forall t \geq 0. \tag{2.2}$$

As the search-space of Equation 2.2 may be very large, Baruah et al. (1990) showed that the points in which the test has to be performed can be restricted to those deadlines within the hyperperiod H — $H \overset{\text{def}}{=} \mathrm{lcm}(T_1, \ldots, T_n)$, where T_i is the minimum inter-arrival time (period) between two consecutive jobs of task τ_i — not exceeding the value

$$L_{\max} \overset{\text{def}}{=} \max\{D_1, D_2, \ldots, D_n, L^*\}$$

In this expression, D_i is the relative deadline of task τ_i and

$$L^* \overset{\text{def}}{=} \frac{\sum_{i=1}^{n}(T_i - D_i) \cdot U_i}{1 - U_\tau}$$

Hence, condition 2.2 has to be tested $\forall t \in$ dlSet, where

$$\mathrm{dlSet} \overset{\text{def}}{=} \{d_{i,k} \mid d_{i,k} \leq \min(L_{\max}, H)\}$$

Zhang and Burns (2009) further improved the efficiency of this analysis by reducing the number of considered time instants through the so-called *Quick convergence Processor-demand Analysis* (QPA).

Another popular technique to address the schedulability analysis of a task set τ upon a given platform π by following a scheduler $\mathscr{S}$ is referred to as the *worst-case response time analysis*. This approach considers the time span between the release time and the completion time of each job of every individual task, say τ_i, when scheduled by following $\mathscr{S}$. Then, it determines the longest time among these values, i.e., the worst-case response time R_i of τ_i. Finally, R_i is compared against the relative deadline D_i of task τ_i to decide its schedulability according to $\mathscr{S}$. Spuri (1996) proposed a method that allows us to determine the worst-case response time of task τ_i, which occurs after a so-called *deadline-d busy period* — A deadline-d busy period is a time interval in which the processor continually executes a sequence of jobs until it eventually idles before the deadline d of a job of τ_i —. This busy-period starts when all tasks except τ_i are released synchronously (e.g., at time $t = 0$) and all subsequent jobs are released as soon as it is legally permitted to do so. Assuming this scenario, the job of τ_i that experiences the longest response time is the last job released before the end of the busy period. The author describes an algorithm that determines the longest deadline-d busy period for all tasks in a task set in a finite computation time, such that an upper bound on the response time of each task can be computed.

The above mentioned schedulers (i.e., RM and EDF) form the basis of a set of scheduling techniques and intuitions that have been developed for multi-core platforms over the passed 30 years. Although they are optimal upon single core platforms, none of them has shown to remain optimal while assuming a scheduling paradigm as the one considered in this work, see (f_3). There are implicit-deadline sporadic systems for which any of these schedulers would fail to find a valid schedule (i.e., one in which all the deadlines are met for all the jobs) while one actually exists. This statement holds true even if the total system utilisation is well below the total platform capacity. As such, the previously mentioned necessary and sufficient condition for an implicit-deadline sporadic task set to be schedulable by EDF as long as its utilisation does not exceed the platform capacity (i.e. $U_\tau \leq m$, being m the number of available cores) no longer holds for partitioned scheduling on multi-core platforms.

Andersson et al. (2001) proved that the utilisation bound for any partitioned approach on a platform with m cores is $U_{\text{OPTIMAL}} = (m+1)/2$. In other to illustrate this claim, the authors considered a task set τ of $m+1$ tasks with the following parameters for each task τ_i: $C_i = 1+\varepsilon$ where ε is an arbitrarily small positive real number; and $T_i = 2$. Then, they showed that this task set cannot be scheduled on a platform with $m \geq 1$ cores even when the utilisation of the task set τ is less than m, i.e. $U_\tau = (m+1) \times (1+\varepsilon)/2 < m$. This statement holds true because at least two tasks will necessarily be assigned to the same core (there are $m+1$ tasks and m cores). The core hosting these tasks has a utilisation greater than 1, and no partitioned approach is able to schedule such task set. Thus, an upper-bound on the task set utilisation which guarantees that all deadlines are met is $\lim_{\varepsilon \to 0} U_\tau = (m+1)/2$. Despite this rather low utilisation bound in general, it is known that there are special case of task sets with utilisation $U_\tau = m$ that are schedulable by using a partitioned algorithm, provided that a specific task-to-core mapping is performed.

Recall that in our setting, a task-to-core mapping is performed at design time and migrations among cores are forbidden at runtime.

This task-to-core mapping is the one of the most challenging cares of the big picture of our proposed approach as an optimal local scheduler can be used on each core, say for example EDF. As solving this mapping problem has been recognized to be NP-hard, we adopt a two-phase *allocation algorithm* that: (1) sorts the tasks according to a pre-defined criteria and (2) perform a task-to-core assignment in a sequential manner so that the subset of tasks assigned to each core is schedulable. During this process, the second phase is carried by a bin-packing heuristic (i.e., a *sub-optimal algorithm to find approximate solutions*), which can produce different results provided that different sorting criteria are used in the first phase. Hence, the efficiency of the task-to-core mapping algorithm in terms of capability to produce a valid partition of the task set (i.e., a partition such that every subset is schedulable on its associated core), when an optimal algorithm is capable to do so, depends on both the combination of the sorting/mapping methods and the selected scheduler on each core. The utilisation bounds have been determined as a good metric for this evaluation.

López et al. (2004) addressed the efficiency problem of well known bin-packing heuristics for the partitioning of sporadic implicit-deadline task sets upon multi-core platforms assuming a preemptive EDF scheduler running on each core. For the first phase, the authors considered that the tasks are sorted by *increasing (I)* or *decreasing (D)* order of their utilisations. Then for the second phase, three allocation heuristics are considered in their evaluation: namely (1) the *Worst Fit* (WF), (2) the *Best Fit* (BF) and (3) the *First Fit* (FF). The specifics of these heuristics are briefly recalled below.

1. *WF*: In this allocation strategy, each task is assigned to the core with the maximum remaining capacity.

2. *BF*: In this allocation strategy, each task is assigned to the core with the minimum remaining capacity where the task fits (i.e., the remaining capacity of the core exceed the task utilization).

3. *FF*: In this allocation strategy, each task is assigned to the first core that is capable of accommodating it (i.e., the remaining capacity of the core exceeds the task utilisation).

As the task-to-core mapping is performed according to the sorted list of task utilisations obtained in the first phase, the selected allocation algorithm is suffixed with the adopted sorting method (e.g., Worst Fit Decreasing — WFD — when the tasks are sorter in an decreasing order of utilizations). The utilisation bounds derived by the authors for all the considered heuristics and sorting variants reached the upper bound (i.e. $(m+1)/2$ when the utilisation of each task can reach 1) except for WF and WFI (for which the utilisation bounds reached 1 when the utilisation of each task reaches 1).

Baruah (2013) stated that utilisation bounds do not provide a good intuition on the efficiency of an allocation algorithm, specifically about *resource utilisation*, i.e., how the selected algorithm is capable of making use of the available computing capacity. As an alternative, the author proposed the *speedup factor* defined as follows.

Definition 14 (Speedup factor)**.** Let τ be a task set to be scheduled upon a multi-core platform π and partitioned among the cores according to an algorithm A. Provided that there exists a schedulable partition of τ generated by an optimal algorithm, say Opt, the *speedup factor* of A, denoted as $f_A \geq 1$, defines how much the speed of the platform π must be increased so that the partition by A becomes schedulable.

According to Definition 14, the smaller the speedup factor of a partitioning algorithm, the more efficient it is. To support this claim, the author showed that the partitioning algorithms that sort tasks according to decreasing utilisations (i.e., WFD, BFD and FFD) have a smaller speedup factor in comparison to other algorithms that shared the same utilisation bound. Note that this result matches previous experimental observations.

2.5 Summary

This chapter presented a basic background on real-time embedded systems and scheduling techniques. The functionality of a real-time application is commonly modeled by using a set of recurrent tasks, with timing constraints, that are meant to share the computing resources of the target platform. This paradigm is at the foundation of an entire and extensive body of knowledge that addresses the problem of scheduling tasks of the target platform, and the problem of formally certifying that the timing requirements of each task are all met so that a failure of the system is guaranteed never to occur. Multi-core architectures have introduced parallelism capabilities into real-time embedded computers. Consequently, scheduling real-time tasks has become an exercise that involves mapping the task executions temporally (i.e., at what time instant) and spatially (i.e., in which core) to the computing resource. This newly added dimension represents a dramatic increase to the complexity of the problem. Along with this challenge, lies the fact that tasks often share other resources on the target platform (e.g., data stored in memory and I/O devices). In these cases, when tasks must perform a sequence of operations in memory (e.g., an atomic operation) or an I/O operation in mutual exclusion, then they must be *synchronised* such that their concurrent operations do not invalidate data in memory and/or I/O devices. A common approach to address this issue consists of adopting a *synchronisation mechanism* that will inform the scheduler that a task is performing a special operation and requires an exceptional handling. This is the subject of Chapter 3.

Chapter 3

Background on synchronisation mechanisms

Most real-time systems are usually modelled by using a set of recurrent tasks as mentioned in the previous chapter. For such a system, all tasks are concurrent for the computing elements (cores) at runtime and each task is responsible for a specific aspect of the overall system functionality. Besides this fact, tasks also share data and/or I/O devices in order to carry out their function. However, the access to these shared resources may have to follow specific rules in order to ensure the integrity of the system. The validity of data in memory and the correct operation to name a few. Operations on these shared resources are typically carried out in mutual exclusion in order to avoid unexpected effects produced by concurrent operations. Therefore, *synchronisation mechanisms* are employed. The synchronisation of jobs that operate shared resources affects their respective response times as concurrent accesses must be serialised in such a way that each job is granted an exclusive access to the resource. Hence, for a given task set and a target platform, every synchronisation mechanism works in conjunction with a scheduler in order to guarantee all temporal constraints.

During the last four decades, the real-time community has specified an entire body of scheduling algorithms that ensures the timing requirements of a set of tasks with assigned priorities in a deterministic and convincing manner. Also, it has developed a set of resource sharing policies that limit the impact of the intrinsic resource behaviour on the schedulability of the tasks. Unfortunately, this suite of resource sharing mechanisms was mainly developed for uniprocessor platforms only. The current multi-core platforms have added a true parallelism flavour to the equation to be solved as multiple tasks can be executed in parallel. This makes the resource sharing mechanisms developed so far for real-time systems outdated. In order to circumvent this issue, a tremendous effort is put into developing multi-core adaptations of the previous mechanisms or reasoning about new multi-core specific mechanisms.

Regarding the first trend, *lock-based synchronisation* is a classical mechanism that provides mutual exclusion to a given system. However, it does not cope with an increase in the number of cores. We distinguish between two types of locks: the coarse-grained locks and the fined-grained locks. Coarse-grained locks have a negative impact on parallelism, whereas fined-grained locks have a negative impact on the system composability. As these are two key desired features when executing a set of real-time tasks upon a multi-core platform, alternatives need to be found. Credible candidates for this task are the so-called non-blocking objects. They suit parallel systems: on a shared non-blocking object, there is no need to explicitly acquire a lock. Unfortunately, they do not allow compound operations to be built on the same object or on a set of objects, as is the case for critical sections supported by locks. Other credible candidates are transactional memories. They allow us to fill this gap and the system designer to define *transactions* (i.e. *atomic sections*) in which the validity of all shared data involved depends on the success of the transaction, otherwise, the transaction is rolled back and its effect is discarded. Here, any conflicts between concurrent transactions are transparently solved by the underlying software transactional memory (STM) mechanism. Unlike in lock-based critical sections, each transaction does not explicitly request a lock and in contrast to non-blocking objects, a transaction generally allows for multiple operations on multiple shared data.

This chapter presents an overview of the solutions in the literature to maintain the consistency and validity of shared resources, i.e., the shared data. In addition, it discusses relevant works published on lock-based synchronisation, non-blocking data structures and STM.

3.1 Lock-based synchronisation

Locks have been the primary choice among the synchronisation mechanisms to provide exclusive access to shared resources and to avoid conflicting operations that can void the soundness of a given resource. Before going into the specifics, let us provide a set of definitions that will help us understand the basic concepts behind this mechanism.

Definition 15 (Critical section)**.** For a program to be executed on a given platform, a *critical section* is defined as a block of sequential instructions that cannot be executed concurrently with another block of sequential instructions that access the same set of (shared) resources.

Definition 16 (Conflict)**.** Let us consider a set of critical sections and shared resources. A *conflict* is defined as the situation where (1) multiple critical sections are allowed to concurrently access a shared resource and (2) at least one critical section modifies the state of the resource.

Figure 3.1 illustrates a practical case of conflict where `Function1` and `Function2` are assigned to core π_1 and core π_2, respectively, and are allowed to execute concurrently. In this example, `Function1` requires to read the values of data objects A and B and computes the next values of data objects A and C, whereas `Function2` modifies the value of A. At runtime, if the value of A is modified by `Function2` between the read and the later write operations on A by `Function1`, then the final value of A is invalid as the effects of `Function2` are lost. This situation represents a conflict on A, which has not been detected. In this case, the two concurrent functions are not

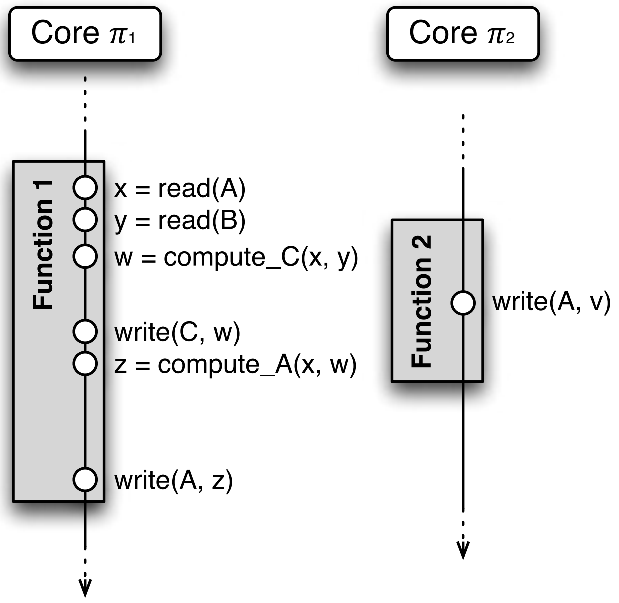


Figure 3.1: Race condition due to non-atomic execution of operations on multiple data objects.

serialised properly. Although the two functions are executed, the system progresses as if only `Function1` is executed. At this point, it is worth noticing that if `Function2` is not allowed to write to A between the read and write operations of `Function1`, then the problem is circumvented. To this end, a practical implementation consists of using a lock that controls the access to A. This solution ensures an exclusive access to the function that owns the lock and is viable as long as all the functions that are accessing a shared resource agree to operate the resource exclusively when they own the lock. As multiple shared resources can be managed by a single lock, it is important to formalise a qualitative indicator that will allow us to distinguish between the different types of accesses to one shared resource. Such an indicator is referred to as the locking granularity and is defined as follows.

Definition 17 (Locking granularity)**.** For a given set of shared resources, the *locking granularity* is defined as the number of shared resources covered by a single lock.

According to Definition 17, we distinguish between the *coarse-grained locking* and the *fine-grained locking*, with the following interpretation:

- *Coarse-grained locking.* Each lock controls the access to a large group of shared resources. This type of locking mechanism provides a simple mapping between the resources and their respective locks, which eases the programming of the critical sections. Moreover, if each lock controls all the resources accessed by the critical sections that request that lock, then nested critical sections (i.e. additional locks inside a critical section) can be eliminated. This is sufficient to avoid deadlock. However, this also increases the probability of *false conflicts*, in which two or more critical sections access disjoint resources that are covered by the same lock. Therefore, coarse-grained locking has a negative impact on concurrency, especially when jobs can execute in parallel.

- *Fine-grained locking.* Each lock controls the access to one or a few number shared resources. This type of locking mechanism improves concurrency as each lock controls only a limited number of resources, which reduces or even eliminates false conflicts. However, in contrast to coarse-grained locking, the mapping between resources and the locks is more extensive. In addition, critical sections that compound the access to multiple resources have to acquire multiple locks (nested critical sections), which requires additional measures to avoid deadlocks. This can complicate the programming of the critical sections.

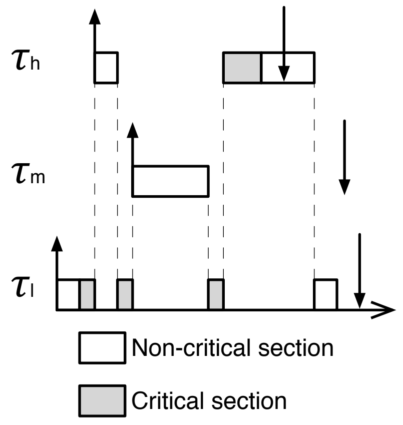
It is crystal clear that the execution of a critical section as mediated by the ownership of the locks has an impact on the scheduling of tasks. Let us illustrate this claim in the unicore context. To do so, let us consider a task set scheduled by following a priority-driven scheduling policy. Let us consider two jobs, say τ_h and τ_ℓ, such that τ_h is assigned a higher priority than τ_ℓ. Let us assume that τ_h preempts τ_ℓ while τ_ℓ was owning a lock ℓ. Later on, let us assume that τ_h requests lock ℓ. As ℓ is currently unavailable, τ_h must suspend and wait for τ_ℓ to complete the critical section and release ℓ before it can resume its execution. This represents a priority inversion as a lower priority task is executing while a task with a higher priority is ready and awaiting for execution. Such a situation allows us to introduce the concepts of *priority inversion* and *blocking* in a formal manner as follows.

Definition 18 (Priority inversion)**.** Let us consider a task set scheduled by following a priority-driven scheduler. A *priority inversion* is defined as a situation where a job is executing when there is at least one ready job, with a higher priority, pending for execution.

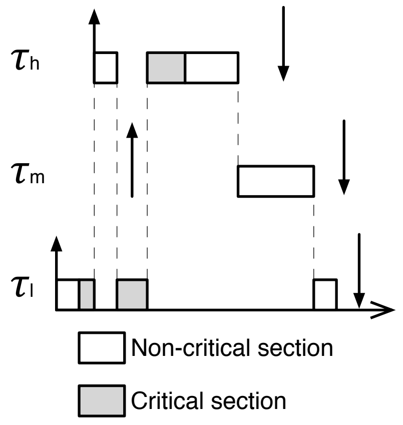
Definition 19 (Blocking time)**.** Let us consider a task set scheduled by following a priority-driven scheduler. The blocking time of a job is defined as the amount of time this job, which is pending for execution, awaits while another job with a lower priority is running.

Considering the previous example, note that τ_h is said to be *blocked* while τ_ℓ holds the lock ℓ. This represents a *priority inversion*, which is a violation of the underlying priority-driven scheduler. Consequently, the response time of τ_h is augmented by at least the blocking time. Revoking the ownership of a lock is a rather complex operation because all the operations carried out so far in the critical section have to be undone. Hence, the priority inversion is tolerated and accepted, as long as it is *bounded*.

Scheduling anomalies during a priority inversion time interval are the reason for potentially *unbounded* priority inversions. To illustrate this claim, let us consider the example depicted in Figure 3.2a where three jobs, τ_h, τ_m and τ_ℓ, are involved. We assume that these are assigned priorities high, medium and low, respectively. In this example, τ_ℓ is released first, then τ_h, and finally τ_m is released. We assume that τ_ℓ acquires a lock, which is shared with τ_h, and enters its critical section. Then, τ_h requests this lock and gets blocked, as it is owned by τ_ℓ. In this case τ_ℓ resumes the execution of its critical section. Upon the release of job τ_m, which does not execute



(a) Potentially unbounded priority inversion.



(b) Bounded priority inversion.

Figure 3.2: τ_h is assigned the highest priority and is blocked when it requests the lock that is owned by τ_ℓ. When priority inversion is not bounded, τ_m is able to complete before τ_h.

any critical section, τ_ℓ is preempted (see Figure 3.2a), and then the execution of τ_h is also delayed by the execution of τ_m. Unfortunately, as a consequence, this additional delay causes a deadline miss of τ_h. Note that in this special case, we considered a single task with a medium priority. The situation could clearly get worse if many jobs with a medium priority were released before τ_ℓ releases the lock. In order to get around this issue, which represents a real hurdle to a tight and sound analysis of any hard real-time, a solution is to adopt a policy that forbids jobs with intermediate priority to execute before a blocked higher priority job has completed execution (see Figure 3.2b). In this figure, τ_h meets its deadline because the blocking time is bounded to the critical section of τ_ℓ. Below we report on a number of relevant works in the literature that have been contributing to this end purpose.

Sha et al. (1990) proposed two protocols, namely the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP), in order to solve the problem of potentially unbounded priority inversions in the uniprocessor and FTP scheduler context. To this end, the priority of every job is raised to a level such that the job avoids any preemption by intermediate priority jobs while it is executing a critical section. PCP has a higher implementation complexity than PIP, but it allows avoiding deadlocks per se. Here, each shared resource is assigned a priority ceiling that corresponds to the highest priority of all jobs/tasks that can access that resource. Under this assumption, the lock is granted to the newly requesting job if its priority is greater than the priority ceilings of all the currently owned locks. Once the lock is granted, the job is guaranteed to acquire all the nested locks in the critical section.

Baker (1991) proposed the Stack Resource Policy (SRP) in order to address the priority inversion problem for uniprocessor and FJP schedulers. This protocol assigns a (constant) preemption level λ_i to every task τ_i according to an increasing order of the relative deadlines, i.e., $\lambda_h > \lambda_\ell$ iff $D_h < D_\ell$. For this protocol and likewise for PCP, each resource is assigned a priority ceiling that corresponds to the highest priority level of all jobs/tasks that can access that resource and the system maintains a ceiling that is the maximum ceiling of all currently owned locks. A released job is immediately scheduled if the following two claim hold true: (1) it holds the earliest deadline of all ready jobs and (2) its preemption level is higher than the current system ceiling. If this is not the case, the newly released job will suspend. In summary, SRP ensures that once the job is scheduled all the resources that it may request are available, so a deadlock can never occurs. Assuming a uniprocessor platform, both PCP and SRP guarantee that a job can be blocked at most by one concurrent job executing a critical section. However, the picture changes completely upon multiprocessors: all these protocols are less effective (Easwaran and Andersson, 2009b). In order to illustrate this claim, let us consider the following example. Let us assume the PCP protocol and a multi-core platform. Let us assume that a job, say τ_h, acquires lock ℓ_a. Now, by assuming that a lower-priority job τ_ℓ requests for some other lock, say $\ell_b \neq \ell_a$, it follows that τ_ℓ is forced to suspend, even if there are cores available and both τ_h and τ_ℓ do not compete for the same resource. Hence, a multi-core resource sharing protocol should not only bound the blocking times from lower priority jobs but it should also improve the number of jobs that can execute in parallel at any time instant.

Rajkumar et al. (1988) proposed the Multiprocessor PCP (M-PCP) for partitioned FTP schedulers. This protocol is an extension to multiprocessors of the PCP in order to synchronise *global shared resources*, i.e. resources that are shared between tasks assigned to different processors, while *local shared resources* (i.e. resources that are shared exclusively between tasks assigned to the same processor) are synchronised by using the classical uniprocessor PCP. In this approach, the access to a global resource is controlled by a global semaphore such that a task suspends until it acquires the lock. Once a task holds a global lock, its effective priority is raised to the highest local priority so that the critical section is executed without suffering any preemption. Unfortunately, this protocol is limited in that it does not allow for mixed nesting of local and global critical sections.

In the same vein, Gai et al. (2001) proposed the Multiprocessor SRP (M-SRP) for partitioned scheduling by adapting the uniprocessor SRP protocol. In this protocol, a task busy-waits in a FIFO queue until it acquires the requested lock. This represents a hurdle to the analysis of the system that needs to be addressed. In order to reduce the time wasted in the busy-waiting state, the critical sections must complete as soon as it is possible to do so, thus the priority of a job while it is executing a critical section should be boosted to a non-preemptible level. On another front, M-SRP allows for nesting local critical sections inside a global resource; however, it does not allow for nesting global critical sections because of the risk of deadlock. This precludes the possibility to perform critical sections accessing multiple global resources with fine grained locking.

Block et al. (2007) proposed the Flexible Multiprocessor Locking Protocol (FMLP) for both global and partitioned schedulers. This protocol distinguishes among them globally shared resources and classifies them as *long* or *short* according to the expected duration of the critical sections that access them. In this regard, the classification criteria is defined by the application designer. Each lock controls a group of resources that are exclusively short (referred to as *short group*) or exclusively long (referred to as *long group*). This is performed in such a way that every task is granted immediate access to all short or long resources requested in the critical section once it acquires the lock. This is referred to as *group locks*. Note that a short request can be contained within a long request, but the opposite is not allowed. A job blocked on a long lock enters a FIFO queue and suspends. When it eventually holds the lock, the job executes the long critical section preemptively, inheriting the maximum priority of any task blocked on this lock. In contrast, a job blocked on a short lock busy-waits non-preemptively in a FIFO queue, and continues executing the short critical section non-preemptively. As such, a job blocked on a short critical section has to wait at most for $(m-1)$ critical sections (being m the number of cores) while requesting the same group lock to complete. The strategy of grouping all short/long resources accessed by critical sections that share at least on resource under a single short/long lock eliminates deadlock but, unfortunately, it configures a potential coarse-grained locking mechanism.

Brandenburg and Anderson (2010) proposed the O(m) Locking Protocol (OMLP) for three schedulers: (1) the fully preemptive global EDF (G-EDF) scheduler, (2) the partitioned EDF (P-EDF) scheduler and (3) the partitioned fixed priority (P-FP) scheduler. This protocol operates in a similar way as FMLP, i.e., it joins the resources requested by intersecting critical sections under one group lock. However, OMLP assumes some practical simplifications as compared to FMLP. The distinction between long and short resources does not exist. This has two consequences: (1) nesting is not allowed, and (2) every job blocked on a lock can only enter a FIFO queue and suspend. This approach for partitioned schedulers allows for at most one job requesting a global resource per core in order to limit the contention on global resources (this means that there is at most m requests for a group lock, being m the number of cores). Hence, every requesting job may have to wait for at most $(m-1)$ other jobs (possibly with a lower priority) in other cores. Once a job is at the head of the queue and owns the lock, its priority is boosted until it releases the lock. Unfortunately here, as in FMLP, the use of group locks configures a potential coarse-grained locking mechanism.

Ward et al. (2012) proposed the Real-time Nested Locking Protocol (RNLP) that allows for fine-grained nested resource requests for global, semi-partitioned and partitioned fixed-job priority schedulers. This protocol allows for an arbitrary number, say $k > 0$, of jobs to concurrently request resources. In addition, deadlocks are avoided by imposing a total order in which locks can be requested such that resource l_a cannot be requested after resource l_b if $l_a < l_b$. Here, each job is assigned a timestamp when it requests the first resource. Then, each resource has a timestamp-ordered queue by which concurrent requests are serialised. Finally, it up to the system designer to select jobs busy-wait or suspend when waiting for a resource. A job eventually acquires a lock when (*i*) the job is at the head of the queue for the lock, or (*ii*) the previous resources in the defined total order of acquisition do not have jobs with earlier timestamps at the head of their respective queues. As such, although RNLP is a fine-grained mechanism, if the first locks in the total order are owned by jobs with earlier timestamps, then all the other locks will not be immediately available for jobs with later timestamps. Unfortunately, this holds true even if those locks are never requested by the previous jobs.

The same authors (Ward and Anderson, 2014) further extended RNLP to support multiple concurrent read-only accesses and mutually exclusive write access to shared resources. The modified request satisfaction mechanism alternates access to each object between read- and write-phases. To this end, each shared resource has a read-queue and a write-queue. Every task issuing a read request inserts an entry in the read-queues of each resource requested. A task issuing a write request inserts an entry in the write-queues of each resource requested. Additionally, it inserts an entry in the write-queue of resources that are currently read-shared (requested together) with the requested dataset by concurrent read accesses, to avoid inconsistent phases. In the beginning of read-phase, all the requests that are on the queue acquire the read-lock. Later read requests are enqueued and are not satisfied in this read-phase. Once all read-locks are released, the resource switches to a write-phase in which the request at the head of the write-queue is satisfied. Finally, the cycle repeats once the write-lock is released.

Although the presented synchronisation approaches offer practical solutions for multi-cores, they still incur the same limitations as uniprocessor lock-based solutions, i.e., a decreased concurrency with coarse-grained locking and no composability with fine-grained locking. An alternative to these lock-based synchronisation techniques are referred to as *non-blocking data structures*. Such a mechanism takes advantage of the natural characteristics of each data structure in order to implement operations that can be non-blocking.

3.2 Non-blocking data structures

Non-blocking data objects provide a safe parallel execution of operations on shared data structures without any explicit manipulation of locks by the programmer. Here, every operation is automatically controlled by a synchronisation mechanism that is part of the data object and is responsible for the maintenance of the object validity. As such, the parallelism is managed by the data object itself and the programmer does not need to explicitly request for a lock. This increases safety as all operations are guaranteed to be performed under concurrency scrutiny. Non-blocking objects present strong conceptual advantages (Tsigas and Zhang, 1999): (1) Priority inversion and deadlock are eliminated because accesses to the object proceed in parallel and concurrent accesses can always be resolved in favour of the most critical tasks; (2) The convoy effect is eliminated because no task will block-waiting for a task that is failing or prevented from executing; (3) The consistency of the object is more robust against unexpected task failures. These advantages are diminished by the complexity of the mechanisms that maintain the consistency of the shared object as this consistency depends on the progress guarantees that is provided by the non-blocking object. Three categories can be distinguished, namely *wait-free*, *lock-free* and *obstruction-free* (Herlihy and Shavit, 2008).

- **Wait-free objects.** Such an object ensures that every call finishes in a finite number of steps, independently from the pace other concurrent calls on the same object execute. This means that all tasks are ensured to progress. To achieve this goal, contending tasks have to collaborate in order to complete the on-going concurrent accesses. Although this is the strongest type of guarantee and the most appealing to multiprocessor systems, it expresses higher complexity in the implementation of mechanisms that are, probably, less efficient than others that offer less strict types of progression.

- **Lock-free objects.** Such an object relaxes the progress condition, so that the system is able to progress in general, although some calls may suffer starvation. Any modification to the state of the data object only take effect if no contention occurs, otherwise at least one call is guaranteed to complete, while the remaining calls fail and have to retry. Lock-free objects generally have less complex implementations and may be more efficient if the pattern of operations on the object does not induce too many fail-retry sequences.

- **Obstruction-free objects.** Such an object provides the most relaxed guarantee of progress, ensuring that one call will eventually progress if it executes isolated from other concurrent calls. This progress condition reflects that some operations may have to execute in mutual exclusion, but the access to the object is not performed by means of any explicit lock. Obstruction-freedom avoids deadlocks, but in case a group of concurrent calls mutually abort each others such that none makes progress, livelocks can occur.

Early non-blocking implementations of data structures took advantage of the characteristics of data structures themselves and their operations. Below we report on a number of relevant works in the literature on non-bloking synchronisation paradigm.

Lamport (1977) proposed a non-blocking solution for the multi-processor problem of multiple readers/single writer. Since then, several dedicated implementations of non-blocking data structures have been developed for systems with multiple processor and shared memory.

Herlihy (1993) and Anderson and Moir (1999) proposed universal methodologies to construct lock-free and wait-free objects from correct sequential code. Along the same line, Anderson and Ramamurthy (1996) described the implementation of a lock-free framework for uniprocessor systems, based on a *multi-word compare-and-swap* primitive (MWCAS). This implementation allows for the simultaneous atomic update to multiple data objects. This ability to perform atomic transactions is equivalent to the nested access to critical sections provided by lock-based synchronisation.

Comparative studies on synchronisation mechanisms for multi-core platforms have been conducted in the literature and the outcome indicates that non-blocking approaches represent a better candidate for real-time systems. However, they do not cover the sequential composition of operations over multiple data as a single, atomic operation, in a convenient manner. Therefore, a broader solution, referred to as *transactional memory*, is preferable as it provides the semantics to express an atomic/critical section.

This research will adopt transactional memory.

3.3 Transactional memory

We recall that the previous synchronisation approaches — i.e., lock-based and non-blocking data objects — provide means to maintain the consistency of shared resources. However, some issues are still pending. On the one hand, locks are not composable and this may represent an hurdle for the analysis of applications that can take advantage of the parallel capabilities of the underlying multi-core platform. On the other hand, non-blocking data objects do not provide a semantic that allows us to aggregate multiple operations on multiple shared objects into a single atomic block, i.e. an *atomic* or *critical section*. In order to illustrate this claim, let us consider once more the practical case depicted in Figure 3.1. We assume that variables A, B and C are non-blocking data objects and that concurrent read and write operations on these objects are guaranteed to be safe. In this example, `Function1` executes two correct operations on object A, intermediated by a correct operation on the same object by `Function2`. Although the three operations on A are correct, the final value of A is inconsistent. This is due to the fact that the effects of `Function2` are lost and thus the two concurrent functions were not properly serialised. This shows that there is a direct correlation between the consistency of the data and the given composition of these operations. The whole sequence of operations is referred to as a *transaction* and is formally defined as follows.

Definition 20 (Transaction)**.** A transaction is defined as a sequential block of instructions that is used to access or modify concurrent data objects, satisfying the serialisability and atomicity properties.

Each transation must be perceived in shared memory space by concurrent tasks as if it was performed atomically. As such that the intermediate states are invisible. This concept is referred to as *transactional memory*.

Software Transactional Memory (STM) exploits the same basic principles as transactions from databases or fault-tolerant systems, i.e., transactions can execute speculatively in parallel, but their effects should appear as if they executed atomically in sequence. However, the field of application of STM has its own specifics. Unlike databases and fault-tolerant systems, STM does not store consistent states in a persistent medium, but simply keeps data consistent in memory, such that STM transactions can be considered lightweight memory transactions (Harris and Fraser, 2003). In order to achieve this, the STM mechanism keeps track of accesses to transactional objects that are exclusively memory locations (words or structures). Since memory accesses to transactional objects become indirect accesses, the STM mechanism must be light enough to avoid performance issues. The concept of transaction is closely related to the lock-based critical section: a section of the sequential code of a task with atomicity requirements. As such, the multithreaded transaction concept of fault-tolerant systems does not apply to STM. According to the transactional memory access pattern, transactions can be divided into *read-only* and *update* transactions with the following interpretations.

Definition 21 (Read-only transaction)**.** A *read-only transaction* is defined as a transaction that does not write in any of the accessed transactional objects.

Definition 22 (Update transaction)**.** An *update transaction* is defined as a transaction that writes at least in one of the accessed transactional objects.

In general, a transaction is executed sequentially in isolation, irrespective of the other parallel transactions. Each transaction must complete its execution such that the transaction *commits* or *aborts*. Before completing, all accessed locations are checked for conflicting updates (see Definition 16) that may have occurred during the execution of the transaction. If no conflicts are detected then the accessed data is *valid* and updates are *committed*, thus becoming effective. Consequently, a transaction can execute multiple *attempts* until it eventually commits.

Definition 23 (Transaction attempt)**.** A *transaction attempt* is defined as one execution of a transactional section that results in a commit or an abort.

The transactional memory system must solve every conflict upon its detection such that transactions can progress. This is performed by applying a conflict solving policy, that is either by *helping* or by consulting a *contention manager*.

- **Helping.** Assuming wait-free and lock-free STMs, this policy helps the contender to complete upon the detection of a conflict between two transactions, such that both transaction can progress. However in lock-free STMs, a transaction is allowed to abort the contender when the cost of helping is high.
- **Contention manager.** Assuming obstruction-free STMs, this policy allows a transaction, which detects a conflict, to ask the STM contention manager[1] to solve the conflict. Specifically, the contention manager solves each conflict by following one of these two approaches: (*i*) by aborting one of the contenders such that the other can proceed or (*ii*) by giving some time to one of the contenders to have a chance to commit such that the conflict is naturally solved before aborting one of the contenders.

The combination of an obstruction-free STM with a contention manager has less complex implementations than lock-free and wait-free implementations. Moreover, the contention manager allows us to implement conflict solving policies that address specific application requirements.

This research considers the combination of an obstruction-free STM with a contention manager.

Conflicts can always be solved by selecting one transaction that commits and aborting the other contending transactions. Consequently, the performance of a STM depends on the frequency of occurrence of the conflicts and the transaction abort ratio. As such, STM behaves very well for systems that exhibit a predominance of read-only transactions, short-running transactions and a low ratio of context switches during the execution of a transaction (Maldonado et al., 2010).

[1]The contention manager is a module of the STM that solves conflicts by considering a pre-determined criteria, i.e. a set of rules defined by the system designer.

STM implementations may differ according to the *version management* or the *conflict detection* (Harris et al., 2010). Version management relates to the manner in which tentative writes are performed. These tentative writes can be categorised as *eager* or *lazy* with the following interpretation.

- **Eager version management.** This version management allows for the object to be modified immediately, but requires that it is unavailable to concurrent transactions until the updating transaction commits. If the updating transaction aborts, the value of the object is rolled-back to the previous version.

- **Lazy version management.** This version management defers the update of the object until the transaction commits, thus leaving the object available to other concurrent transactions. This implies that a transaction must work with an image of the object that will eventually replace the value of the transactional object. If the transaction aborts, then no roll-back is required.

Lazy version management requires more memory overhead, as each transaction needs to have its own set of copies of the accessed objects. However, eager version management requires book-keeping in order to allow rolling back the modifications in case the transaction aborts.

This research considers lazy version management.

Besides assuming a lazy version management, we adopt a *lazy conflict detection* policy.

This research considers lazy conflict detection.

This means that each conflict is detected only when the transaction tries to commit. This conflict detection mechanism allows transactions to execute in total isolation. In addition, lazy conflict detection can be more efficient with read-write conflicts. This is the case as long as read-only transactions commit before update transactions (Spear et al., 2006). We distinguish between STM implementations with visible reads and STM implementations with invisible reads.

For STM implementations with *invisible reads*, an update transaction that modifies a given object is not aware of concurrent transactions that are simultaneously reading that object. As such, update transactions commit regardless of the concurrent reads on their write sets. Furthermore, this transaction cannot inform the concurrent transactions that they are working with an outdated value. Hence, transactions with invisible reads have to check the validity of their read sets when they try to commit. In contrast, for STM with *visible reads*, every read operation is registered. As such, update transactions are able to detect write-read conflicts[2]. Visible reads require to register all operations. This represents memory overhead, but allows full control in detecting and solving conflicts. This is a very important feature when a deterministic contention manager is desired.

[2]A write-read conflict occurs when a transaction that modifies an object tries to commit before a concurrent transaction that reads the same object does.

In addition, transactions are aborted by updates in their read sets such that the verification of the validity of their read sets becomes redundant.

This work considers STM implementations with visible reads.

Last but not least, the granularity of conflict detection determines the possibility of false conflict detection. We can distinguish between *finer granularity* and *coarser granularity*. Finer granularity (on the word or cache-line level) implies less false conflicts detected and lower transaction abort ratio, but at the expense of higher memory overheads, whereas coarser granularity (at the object level) presents lower memory overheads but more false positive conflicts.

This research does not focus on any particular level of granularity of conflicts.

3.4 Relevant works on software transactional memory

Herlihy and Moss (1993) proposed the first transactional memory mechanism based on hardware. In this work, the authors described how adding extensions to the cache-coherence protocol of a multiprocessor architecture allows us to implement transactions on memory.

Shavit and Touitou (1995, 1997) adapted the concept of transactional memory and implemented all the synchronisation operations at the software level: this resulted in the *Software Transactional Memory* (STM). This approach supports static transactions, which requires that both the transactions and the memory usage are defined in advance. This limitation is overcomed by following STM implementations that on the one hand provide dynamical transactions, i.e., a transaction can decide the addresses to access based on the values read at runtime; and on the other hand provide dynamical memory usage, such as the Dynamic STM (DSTM) (Herlihy et al., 2003), the object-based STM (OSTM (Fraser, 2003), the Adaptive STM (ASTM) (Marathe et al., 2005), Transactional Locking 2 (TL2) (Dice et al., 2006), the multi-core runtime STM (McRT-STM) (Saha et al., 2006) or the SwissTM (Dragojević et al., 2009).

Some of these implementations have been tested against traditional synchronisation mechanisms. Cascaval et al. (2008) analysed the performance of a highly optimised STM and concluded that the current implementations suffer from overheads that do not make STM appealing in the situations where less than four parallel transactions are assumed. Dragojević et al. (2011) showed that STM is capable of outperforming the sequential coding paradigm, especially when the number of parallel threads increases although the overheads of the compiler instrumentation and transparent privatization are substantial.

Despite the large body of knowledge developed during the last decades on STM for distributed and parallel systems, only a few works dealt with dynamical memory usage in the context of real-time systems. Manson et al. (2005) propose a data access synchronisation mechanism based on transactions for real-time systems assuming a uniprocessor platform — the Preemptible Atomic

Regions — and provide a response time analysis that bounds the response time of any job executing atomic regions. In this analysis, an atomic region is guaranteed to be free from other tasks interference because the transaction is immediately aborted and its effects undone if any job that is executing a transaction is preempted by a higher-priority task. This policy implies that no concurrent transactions are allowed in the system, which is impractical for multiprocessor systems.

Fahmy et al. (2009) describe an algorithm that allows us to compute an upper-bound on the worst-case response time of every task in a multiprocessor system, assuming STM as the synchronisation mechanism and a Pfair scheduler. This analysis is limited to small atomic regions, assuming that every transaction will execute in at most two quanta. It is assumed that each task can have multiple atomic regions and concurrent transactions can interfere with each other. Also, conflicts are detected and solved during the commit phase.

Sarni et al. (2009) proposed a scheduling policy for concurrent transactions in soft real-time systems. The authors characterised each transaction by using scheduling parameters, e.g., a relative deadline which expresses the maximum amount of time desirable for the transaction to commit once it has started. Upon the start of every instance, it is assigned an absolute deadline that is determined by the sum of its release time and its relative deadline. Conflicting transactions are serialised based exclusively on their absolute deadlines. Unfortunately, this may have a negative impact on transactions with larger absolute deadlines. On another front, the authors adapted a practical STM to run on a real-time kernel and modified the contention manager to apply their proposed policy. The experimental results showed the benefits of using scheduling data to improve the number of transactions that meet their deadlines.

Schoeberl et al. (2010) proposed a hardware transactional memory for hard real-time tasks. In this contribution, the system model assumes that each task contains one single atomic region and conflicts are detected and solved during the commit phase. They provided a response time analysis, which demonstrates that every job of a task will meet its deadline as soon as two consecutive transactions are separated by the so-called *resolve time*, i.e. the worst-case time a transaction takes to successfully commit. The analysis provides a method to compute an upper-bound of the resolve time. Despite the guarantee that deadlines will be met, this work does not describe any method to solve the transaction conflicts, based on scheduling data.

Cotard (2013) developed a wait-free STM for hard real-time embedded systems on multi-core platforms, in which transactions help their contenders to commit. Based on the profiles of tasks in practical automotive applications, this work assumes that transactions are *homogeneous*, i.e. the data set of a transaction is exclusively formed either by the read set (*read-set transaction*) or by the write set (*write-set transaction*). Consequently, a task can have multiple transactions of different nature: e.g. an earlier read-set transaction to acquire data and a later write-set transaction to output the results of a computation. In this work, the system model does not consider a specific scheduling algorithm, but requires transactions to be executed non-preemptively, until they commit. In addition, write-set transactions that have intersecting write sets must be allocated to the same core and since transactions cannot be preempted, it is impossible to have a write-write conflict between transactions. The devised policy provides two results: (i) a write-set transaction will never abort because there are no write-write conflicts, and (ii) a read-set transaction will abort, at most, once because concurrent updates help beforehand read-set transactions that have previously aborted. Unfortunately, the restrictions assumed in this work do not allow *heterogeneous* transactions that combine read and write operations in one single atomic section, and that may be required in more complex applications.

El-Shambakey and Ravindran (2012b) provided the response time analysis of two contention managers that take the scheduling priorities of jobs as the decision criteria: the EDF contention manager (ECM) for tasks sets scheduled by following the global earliest deadline first (G-EDF) scheduler and the RMA contention manager (RCM) for tasks sets scheduled by following the global rate monotonic (G-RMA) scheduler. When a conflict is detected, the ECM selects the transaction with the earliest deadline and the RCM selects the transaction with the highest priority to commit. The system model assumes that each transaction accesses one single STM object, but each task can have multiple transactions. Note that transactions cannot be nested. On one front, the restriction on the number of accessed objects per task simplifies the response time analysis, it allows the authors to perform a direct comparison with the lock-free retry loops (Devi et al., 2006) and determines the conditions under which the STM approach provides lower retry costs than the lock-free approach. On another front, this same restriction also severely limits the atomic section where multiple operations on multiple data must be viewed has one atomic operation.

The same authors (El-Shambakey and Ravindran, 2012a) proposed the length-based contention manager (LCM) to be used in conjunction with the G-EDF or G-RMA scheduler. Here, a transaction that detects a conflict voluntarily aborts under two circumstances: (*i*) it has a lower priority than the concurrent transaction, or (*ii*) the concurrent transaction has executed for a predefined amount of its total execution time. This work assumes the same system model as in (El-Shambakey and Ravindran, 2012b), i.e. one object accessed per transaction and multiple non-nested transactions per task. Consequently, it presents the same limitations. To worsen the picture, the contention manager must keep track of the execution time of all jobs executing in all cores in order to decide which transaction will commit upon the detection of a transaction conflict. Unfortunately, this may lead to a non-negligible execution time overheads.

They proposed in (El-Shambakey and Ravindran, 2013a) the so-called *Priority contention manager with Negative values and First access* (PNF) in order to circumvent the aforementioned issues. Here, the system model allows transactions to access multiple objects. PNF partitions transactions in progress into two groups: (1) the m-set (the maximum size of the group is m, the number of cores) in which transactions execute non-preemptively and will commit immediately, and (2) the n-set (the maximum size of the group is n, the number of tasks) in which transactions execute with the lowest priority defined by the scheduler (i.e. -1) and will abort. Unfortunately, this approach relies heavily on a centralised contention manager and therefore may not scale with an increase in the number of cores.

They provide in (El-Shambakey and Ravindran, 2013b) a more dynamic and decentralised approach to circumvent this last limitation, while assuming global-based schedulers. Here, the contention manager does not require the data set of each transaction to be known beforehand, it is referred to as *First Bounded, Last Timestamp* (FBLT). Each transaction is assigned a maximum number of aborts and conflicts are solved by following the same principle as in LCM. When a transaction reaches its maximum number of aborts, FBLT registers the timestamp and rises the priority of the job to a non-preemptable level. Unlike PNF, conflicts between non-preemptable transactions are possible. FBLT selects the transaction with the earliest registered timestamp to commit. The main limitation of this approach is that the maximum number of aborts per transaction is assumed to be known beforehand. This may not be the case for complex real-world applications. Furthermore, although the adopted global schedulers provided interesting features such as better load-balancing and flexible solutions in terms of schedulability, they presented serious overheads (in terms of number of migrations and preemptions). These are not desirable in the context of hard real-time systems, where applications have stringent timing requirements.

The aforementioned works provide interesting perspectives on how to deal with STM in the context of real-time systems. However, it is clear that there are many pending issues, and further research is necessary in order to take full advantage of the future parallel architectures.

3.5 Summary

This chapter presented an overview of the relevant works published on lock-based synchronisation, non-blocking data structures and STM. Synchronisation mechanisms have been developed for uniprocessor platforms only, but the advent of multi-core platforms add a new dimension to the problem of data integrity since multiple tasks can be executed in parallel. Lock-based synchronisation mechanisms do not scale with an increase in the number of cores, as they have a negative impact on parallelism and/or the system composability. Non-blocking objects could have been a good alternative, but they do not allow compound operations to be built on the same object or on a set of objects, unfortunately. Because of these drawbacks, we adopted transactional memories as they allow us to define transactions to transparently solve conflicts by relying on the underlying software transactional memory (STM) mechanism. Chapter 4 introduces the system model used throughout this research work.

Chapter 4

System model

In this chapter we introduce all the parameters that will be used throughout this research work. To this end, we distinguish between three levels of abstraction, namely: the *task specifications*, the *platform and scheduler specifications*, and finally the *STM specifications*. We assume all timing characteristics to be non-negative integers, i.e. they are multiples of some elementary time interval.

4.1 Task specifications

We assume that the workload is carried by a set of n periodic tasks $\tau \overset{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$. Each task τ_i releases a potentially infinite number of jobs and is characterised by a worst-case execution time C_i, a relative deadline D_i, and a period T_i. These parameters are given with the following interpretation. The j^{th} job of task τ_i executing on processor π_k, referred to as $\tau_{i,j}^k$, is characterised by its release time $r_{i,j}$ such that $r_{i,j+1} \overset{\text{def}}{=} r_{i,j} + T_i, \quad \forall i \in \{1, \ldots, n\}, \forall j \geq 1$; and an absolute deadline $d_{i,j} \overset{\text{def}}{=} r_{i,j} + D_i$. We assume the system to be constrained-deadline, so $D_i \leq T_i, \forall i$. This condition imposes that each job must finish before the following job is released, so no consecutive jobs can overlap in time. We also assume that the system is synchronous, i.e., all tasks in τ release their first jobs at the same time instant, say $t = 0$. Formally, we assume that $r_{i,1} = 0, \forall i$. For such a system, interval $[0, P)$, where P is the hyper-period ($P = \text{lcm}(T_1, \ldots, T_n)$ – the least common multiple of the periods of all tasks), is a feasibility interval (Cucu-Grosjean and Goossens, 2011). This means that if all jobs meet their deadline in this finite time window, then we are guaranteed that every job will always meet its deadline.

Tasks may *not* be independent, meaning that they may concurrently access common data located in shared memory. Accesses to these shared data are performed in the context of a *STM transaction* (Shavit and Touitou, 1997) — The specification of each STM transaction is provided below. In the rest of this document, we will simply mention transaction to refer to a STM transaction. The outcome of a transaction must appear as if the operations that constitute it were performed atomically, isolated from the interference of concurrent jobs, i.e. as if performed in mutual exclusion. All reads and updates must be performed over a single state of the subset of STM objects that are accessed by the transaction: non intermediate concurrent updates on this subset can occur during the execution of the transaction, otherwise the results would be inconsistent. The mission of an STM system is to maintain the consistency of the share data, validating the outcome of transactions. If the outcome of a transaction results in an inconsistent state, then the transaction *aborts*; otherwise, the transaction *commits*.

We assume that each task τ_i performs *at most* one STM transaction, denoted as ω_i. Nevertheless, the results obtained throughout this work are extensible to tasks that execute multiple non-nested transactions with minor efforts. The transaction of τ_i is characterised by:

- C_{ω_i}: the maximum time required to execute the sequential code of ω_i once, without any external interference from other tasks or the system itself, and try to commit.
- **The data set** (DS_i): the collection of shared objects that are accessed by ω_i. This data set can be partitioned in two subsets: the *read set* (RS_i) and the *write set* (WS_i) where:
  - RS_i is the subset of objects that are accessed by ω_i solely for reading, and
  - WS_i is the subset of objects that are modified by ω_i during its execution.

  The size of the data set, read set and write set are denoted as $|DS_i|$, $|RS_i|$ and $|WS_i|$, respectively.

4.2 Platform and Scheduler specifications

We assume that all the jobs are executed on a multi-core platform $\pi \overset{\text{def}}{=} \{\pi_1, \ldots, \pi_m\}$ composed of m homogeneous cores, i.e., all cores have the same computing capabilities and are interchangeable. The task set τ is scheduled by following a *partitioned Earliest Deadline First* (P-EDF) scheduler, i.e., each task is statically assigned to a specific core at design time (and task migrations from one core to another at runtime are not allowed) and each core schedules its subset of tasks at runtime by following the classical EDF scheduler, where at each time instant the job with the earliest absolute deadline is selected for execution. Ties are broken in an arbitrary manner.

We define σ as a function that returns the core to which a task is assigned. Thus, if task τ_i is assigned to core π_k, then $\sigma(\tau_i) = \pi_k$.

4.3 STM specifications

We assume that a collection of p STM objects $O \overset{\text{def}}{=} \{o_1, \ldots, o_p\}$ are located at the globally shared memory and are accessible to all tasks executing a transaction, independently from the core on which they are executing. Multiple simultaneous transactions are supported and for each object there is a chronologically ordered list that records all transactions currently accessing the object. The number of transactions that have a specific object, say o_j, in their data sets is denoted as $|o_j|$.

Contention occurs when two or more transactions, executing in parallel, have intersecting data sets and at least one transaction modifies the value of a shared object. Hence, we can map

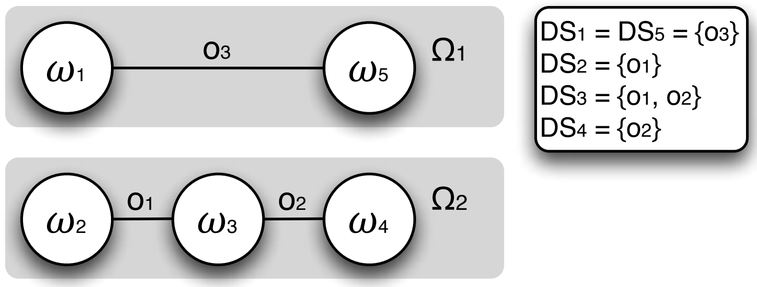


Figure 4.1: Transaction dependencies by object concurrency.

contentions by using a graph G in which *vertices* represent transactions and *edges* represent the shared objects between a pair of transactions.

Definition 24 (Contention group)**.** Given a contention graph G, a *contention group*, denoted as Ω_k (with $k \geq 1$), is defined as a set of connected vertices of G in which any two transactions are connected by a path.

Figure 4.1 illustrates a very simple example of contention groups in which a set of five transactions $\{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5\}$ are sharing a set of three STM objects $\{o_1, o_2, o_3\}$. In this example, the data sets of the transactions form two distinct contention groups: $\Omega_1 = \{\omega_1, \omega_5\}$ and $\Omega_2 = \{\omega_2, \omega_3, \omega_4\}$.

Each instance of a transaction has a life cycle that follows the states represented in Figure 4.2. When a transaction is released, it enters the active state in which the transaction code is executed. At the end of the code execution, the STM system checks whether the transaction is sound, in terms of data consistency and current data access conflicts. If the transaction is sound, then it commits and concludes, otherwise the transaction enters the failed state, in order to restart. During the active state, the data accessed by a transaction can become inconsistent when (1) another transaction, referred to as *contender* commits, and (2) the transaction enters the zombie mode, in which it continues to execute, but it will inevitably fail the final validation, transiting to the failed state. A transaction may be aborted multiple times until it successfully commits.

Definition 25 (Cores allocated to a contention group)**.** Given a contention group Ω_a, then Π_a is defined as the set of $m_a \leq m$ cores allocated to the transactions of Ω_a. Formally, $\Pi_a = \{\pi_k \mid \sigma(\omega_i) = \pi_k,\ \omega_i \in \Omega_a\}$.

Definition 26 (Direct contender of a transaction)**.** The *direct contender of a transaction* ω_i is defined as a transaction ω_j that shares at least one STM object with ω_i. Formally, ω_j is a direct contender of ω_i if $DS_i \cap DS_j \neq \emptyset$.

Definition 27 (Indirect contender of a transaction)**.** The *indirect contender of a transaction* ω_i is defined as a transaction ω_j that does not share any STM object with ω_i, but belongs to the same contention group as ω_i. Formally, ω_j is an indirect contender of ω_i if the following two conditions hold: (1) $DS_i \cap DS_j = \emptyset$ and (2) $\exists \Omega_a$ such that $\omega_i, \omega_j \in \Omega_a$.

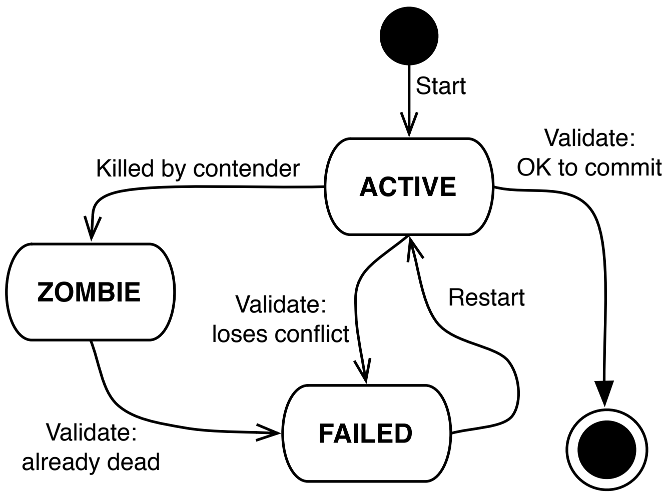


Figure 4.2: State diagram of a transaction.

Definition 28 (Independent transactions)**.** Two transactions ω_i and ω_j are said to be *independent* when they belong to different contention groups. Formally, ω_i and ω_j are independent if the following two conditions hold: (1) $\exists\Omega_a,\ \omega_i \in \Omega_a;\ \exists\Omega_b,\ \omega_j \in \Omega_b$ and (2) $\Omega_a \cap \Omega_b = \emptyset$.

Definition 29 (Transaction overhead of a job)**.** The *transaction overhead of job* $\tau_{i,j}$, denoted as $W_{i,j}$, is defined as the time wasted in executing aborted commit attempts of ω_i. Formally, $W_{i,j}$ is given by:

$$W_{i,j} \overset{\text{def}}{=} A_{i,j} \cdot C_{\omega_i} \tag{4.1}$$

In Equation 4.1, $A_{i,j}$ represents the number of failed attempts of ω_i before it commits.

Definition 30 (Execution time of a job executing a transaction)**.** Given task τ_i executing a transaction ω_i, the *execution time of the j^{th} job*, denoted as $C_{i,j}$, is defined as the sum of four terms: (1) the time ($C_{\text{a-}\omega_i}$) required to execute the code of τ_i *before* ω_i starts, (2) the time ($C_{\text{p-}\omega_i}$) required to execute the code of τ_i after ω_i has committed, (3) the execution time (C_{ω_i}) of a successful transaction of ω_i, and (4) the transaction overhead ($W_{i,j}$) of that job. Formally, $C_{i,j}$ is given by:

$$C_{i,j} \overset{\text{def}}{=} C_{\text{a-}\omega_i} + C_{\omega_i} + C_{\text{p-}\omega_i} + W_{i,j} \tag{4.2}$$

Definition 31 (Task utilisation)**.** The *utilisation of task* τ_i, denoted as U_i, is defined as the execution ratio of the jobs of τ_i within one hyper-period. Formally:

$$U_i = \frac{1}{P/T_i} \cdot \sum_{j=1}^{P/T_i} \frac{C_{i,j}}{T_i} \tag{4.3}$$

Definition 32 (System utilisation)**.** The *utilisation of system* τ, denoted as U_s, is defined as the sum of the utilisations of all tasks in τ. Formally:

$$U_S = \sum_{i=1}^{n} U_i \tag{4.4}$$

Chapter 5

FIFO-CRT: a predictable STM contention management

Memory hierarchy represents an important hurdle to the way applications share data upon multi-core platforms. To address this problem, the software transactional memory (STM) was proposed as an appealing solution as (i) it allows for transactions to execute in isolation and (ii) conflicts to be solved by applying a contention policy. This contention policy is responsible for selecting the transaction that will commit, while the contender(s) will most likely abort and repeat. Therefore, it is crucial to design control mechanisms for serialising concurrent transactions, while maintaining the consistency of shared data objects. The time overhead resulting from successive aborts of a transaction affects the worst-case execution time (WCET) of every task that executes a transaction. This implies that the timing behaviour of a task can only be predictable if the transaction overhead is bounded. In this chapter, we formalise a FIFO-based contention management algorithm, referred to as FIFO-CRT, that provides predictability and prevents transaction starvation. FIFO-CRT has proven its efficiency as a fairer way to limit the number of times transactions are aborted.

5.1 Requirements

The contention manager relies on criteria that express the expected behaviour of the system. Specifically, it must avoid live-lock situations, i.e. a situation in which a group of transactions indefinitely abort each other, without ever committing, and it must prevent transaction starvation, i.e. a situation in which a transaction is excessively aborted as compared with its contenders. An STM contention management policy oriented for real-time systems must meet the following three requirements.

▷ *Predictable.* When a transaction arrives, it must be guaranteed that its execution does not exceed a predefined time limit to commit. This requirement imposes an upper bound on the number of aborts that the transaction under analysis can undergo. To this end, read-only transactions must be visible.

▷ *Avoid starvation.* The ability to commit must be distributed in a fair manner among the set of contending transactions. As such, no task will have an excessive abort overhead.

▷ *Decentralised.* The algorithm that implements the contention management policy should be preferably decentralised and executed by each transaction at the moment it tries to commit, and not on a dedicated processor. This way, the overall system overhead will be drastically limited.

5.2 Classical contention management policies

Herlihy et al. (2003) were the first to introduce the concept of *contention manager* as a mechanism with the objective to avoid live-lock in an obstruction-free STM. Since then, the problem of managing accesses to transactional data received a lot of attention and became the main focus of many research work. More precisely, the main question to answer/address is: given a set of transactions competing for a set of shared data objects, how to select the transaction to commit and what actions (typically wait and/or abort) should the remaining transactions take in order to improve the overall throughput (i.e. the number of commits per time unit)? To this end purpose, various heuristics have been proposed – see the bullet list below for further details.

Although the contention managers proposed for general-purpose systems focus specially on throughput (i.e. systems without stringent timing requirements), they can provide some good insights on how to choose the most suitable decision criteria for the contention manager of a system with stringent timing requirements. Some of the most popular heuristics found in the literature were originally designed for STM with eager conflict detection and/or lazy conflict detection. Few examples of such heuristics are described below, assuming the scenario where a transaction, referred to as the *victim transaction*, has accessed shared data and another transaction, referred to as *attacking transaction*, detects a conflict. The remaining contention management policies available are not suitable as they do not meet the decentralized requirement (see Section 5.1). They present a high complexity and/or require a high amount of data exchange between cores.

- **Passive** (Dice et al., 2006). In this heuristic, the attacking transaction aborts when it detects a conflict at commit time. It was designed for lazy conflict detection.

- **Aggressive** (Herlihy et al., 2003). Here, the attacking transaction aborts immediately the victim transaction once a conflict is detected. It was originally designed for eager conflict detection but can be tuned and used for lazy.

- **Polite** (Herlihy et al., 2003). Here, the attacking transaction enters an exponential back-off cycle after it has detected a conflict in order to allow the victim transaction to commit. At the end of a predetermined number of wait cycles, the attacking transaction aborts the victim. It was designed for eager conflict detection.

- **Timestamp** (Scherer III and Scott, 2004). In this heuristic, each transaction is assigned a time stamp at the moment it starts. If the attacking transaction has the earliest time stamp, then the victim transaction is aborted. Otherwise, it waits for a given time before aborting the victim. Ties are broken in an arbitrary manner. It can be used for eager or lazy conflict detection.

- **Greedy** (Guerraoui et al., 2005b). The Greedy contention manager uses time stamps to assign priorities to transactions. In addition, each transaction has a boolean flag that indicates if the transaction is currently waiting for the outcome of a concurrent transaction, or not. The decision rule is not straightforward as in Timestamp. Here, when an attacking transaction detects a conflict, it aborts the victim if the victim has a later time stamp or is waiting for yet another transaction. Otherwise, if the victim transaction has an earlier time stamp and is not waiting for another transaction, then it is aborted only if it starts waiting for another transaction. This policy attempts to increase throughput of the number of commits. It was originally designed for eager conflict detection.

- **Karma** (Scherer III and Scott, 2005). In this heuristic, each transaction carries a value that represents the work it has carried out since it has started referred to as the *karma*. The karma of each transaction is calculated by using the current number of transactional objects accessed by the transaction since it has started and the number of aborts it has suffered. Note that the karma of each transaction may change over time. When an attacking transaction detects a conflict, it aborts the victim if it has an higher karma, otherwise it waits for a predetermined time and aborts itself, thus increasing its karma. It was originally designed for eager conflict detection.

From this sample of contention managers, Timestamp and Greedy are, in theory, free of live-lock and starvation. Indeed, for these two contention managers, each transaction is assigned a finite time stamp that will eventually become the earliest at a given instant. Once this is the case, it can commit. Guerraoui et al. (2005a) show that alternative contention managers may outperform those based on time stamps in practice. However, the winning candidates are all exposed to live-lock and starvation. To illustrate this claim, below are a few examples:

1. The Passive and Polite managers do not provide any guarantee on the time at which a transaction will commit.

2. The Aggressive manager may suffer from live-locks if a group of transactions mutually abort each other without ever committing.

3. The Karma manager may induce starvation on long transactions if short transactions accessing many transactional objects abort-and-repeat fast enough to achieve a higher karma.

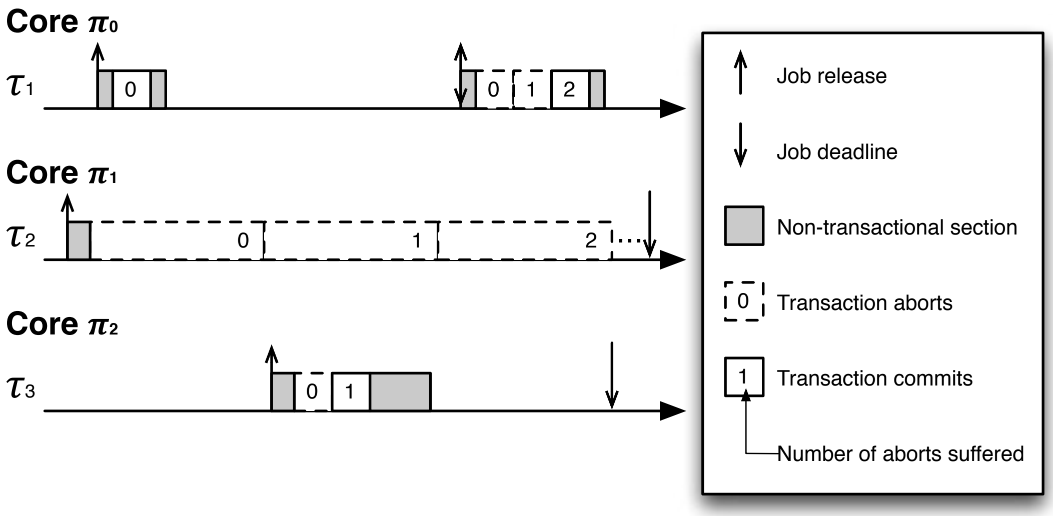


Figure 5.1: Contention management based on number of aborts.

5.3 Discussion of criteria

Regarding the requirements in Section 5.1, the contention manager is responsible for deciding, upon each conflict, which transaction to commit so that the application timing requirements are not violated. To this end, it must rely on criteria that monitor the progress rate of the transactions and/or the scheduling parameters of jobs that host a transaction. In this section, we carry out the discussion about four criteria out of which: (1) two are dynamic job parameters, namely the *number of aborts* of a transaction and the *current job laxity*; and (2) two fixed-job parameters, namely the *transaction release time* and the *job absolute deadline*. These criteria have been selected as they appear to be the most suitable ones to help us meeting all the timing requirements of the application.

Number of aborts of a transaction. This parameter can be used in a contention manager that opted for aborting the transaction with the smallest number of aborts. If two transactions with the same number of aborts are in conflict, then the first one trying to commit wins the conflict. This policy seems to provide fairness as the more a transaction suffered from aborts, the higher are its chances to commit. This is the case if the transactions have similar execution times. However, this policy may discriminate against long transactions as illustrated in the following example.

Consider three jobs belonging to tasks τ_1, τ_2, τ_3, and executing transactions ω_1, ω_2, ω_3, respectively. We assume that: (1) τ_1, τ_2, τ_3 execute on three different cores; (2) τ_2 is the first task to release its job, followed by τ_1 and then τ_3; (3) τ_1 releases a second job within the first period of τ_2; and finally (4) ω_2 is significantly longer than ω_1 and ω_3. Figure 5.1 illustrates a situation where all the three transactions try to access the same memory address, thus resulting in a transactional conflict. In this figure, ω_2 fails several times because of the short transactions (here, ω_1 and ω_3) that had been released later, but managed to commit before. This situation is due to the number of aborts (number inside the dashed rectangles) of these short transactions, which grows faster than that of ω_2.

Current job laxity. This parameter can be used in a contention manager that solves conflicts by committing the transaction belonging to the job with the smallest laxity. The laxity of a job at a given time instant, say t, after its release time is determined by the amount of time this job can be delayed, but still meet its deadline. Formally speaking, if we consider a job, say $\tau_{i,j}$, which still needs $c'_{i,j}(t) \leq C_i$ units of execution at time t to complete prior to its deadline, then its laxity at time t is defined as follows.

$$\ell_{i,j}(t) = (d_{i,j} - t) - c'_{i,j}(t). \tag{5.1}$$

Equation 5.1 shows that the laxity of every job in a non-increasing function of time. That is, the more the laxity of a job, the more this job can be delayed. Therefore, this parameter indicates the flexibility to schedule every job. The intuitive idea of using a contention manager based on this logic resides in the fact that by repeating the transaction belonging to the job with the smallest laxity, we are more inclined to miss its deadline. Nonetheless, this solution is not free from pitfalls as illustrated below.

1. *Runtime overhead.* The runtime overhead associated to the computation of each job laxity can be significant. Indeed, an accurate estimation of the laxity of a job requires monitoring the execution of this job from its release time up to its finishing time.

2. *Live-locks.* Every contention manager based on job laxity is unfortunately exposed to live-locks. Indeed, at every time instant, say t, if a transaction ω_i belonging to task τ_i aborts, then the execution time C_{ω_i} associated to this transaction adds naturally to the remaining execution time of τ_i as this transaction is forced to perform a new attempt to commit. Note that the more the number of aborts of ω_i, the higher the remaining execution of τ_i at time t. Now, let us construct a scenario resulting in a live-lock. To this end, we consider two tasks τ_a and τ_b executing transactions ω_a and ω_b respectively, and executing on two different cores in parallel (see Figure 5.2). We assume that these transactions access the same memory location. Also, we assume without any loss of generality that $C_{\omega_a} = C_{\omega_b} = C_\omega$. Furthermore, we assume that: (1) ω_a tries to commit first, say at time instant t_1, and the laxities of the two jobs are such that $\ell_a(t_1) > \ell_b(t_1)$ and $\ell_a(t_1) - \ell_b(t_1) < C_\omega$; and (2) ω_b tries to commit at time instant $t_2 > t_1$ such that $t_2 - t_1 < C_\omega$.

   • At time t_1, ω_a will abort (increasing its remaining execution time of τ_a by C_ω) as $\ell_a(t_1) > \ell_b(t_1)$. Then it will perform a new attempt at time $t_3 \overset{\text{def}}{=} t_1 + C_\omega$.

   • At time t_2, there is a conflict between ω_a and ω_b as $t_2 - t_1 < C_\omega$. At this time instant, $\ell_a(t_2) = \ell_a(t_1) - C_\omega$ and $\ell_b(t_2) = \ell_b(t_1)$ as task τ_b was busy executing in the time interval $[t_1, t_2]$. Consequently, $\ell_a(t_2) - \ell_b(t_2) = (\ell_a(t_1) - C_\omega) - \ell_b(t_1) = (\ell_a(t_1) - \ell_b(t_1)) - C_\omega < 0$ as $\ell_a(t_1) - \ell_b(t_1) < C_\omega$. This means that $\ell_a(t_2) < \ell_b(t_2)$, thus ω_b will abort (increasing its remaining execution time of τ_b by C_ω). It will perform a new attempt at time $t_4 \overset{\text{def}}{=} t_2 + C_\omega$.

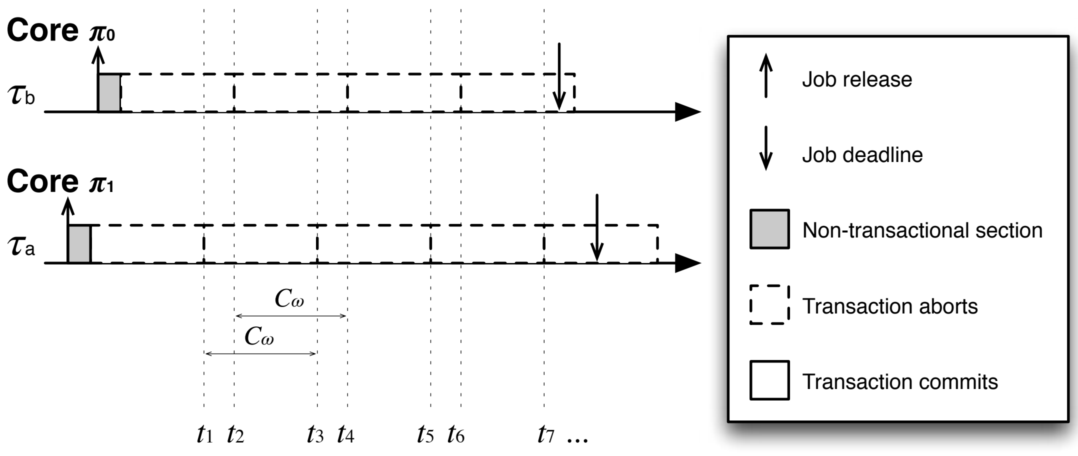


Figure 5.2: Live-lock due to laxity criteria.

• At time t_3, there is a conflict between ω_a and ω_b as $t_3 - t_2 = (t_1 + C_\omega) - t_2 = C_\omega - (t_2 - t_1) < C_\omega$ as $(t_2 - t_1) < C_\omega$. At this time instant, $\ell_a(t_3) = \ell_a(t_2)$ as task τ_a was busy executing in the time interval $[t_2, t_3]$ and $\ell_b(t_3) = \ell_b(t_2) - C_\omega$. Consequently, $\ell_a(t_3) - \ell_b(t_3) = \ell_a(t_2) - (\ell_b(t_2) - C_\omega) = (\ell_a(t_1) - C_\omega) - (\ell_b(t_1) - C_\omega) = \ell_a(t_1) - \ell_b(t_1) > 0$. This means that $\ell_a(t_3) > \ell_b(t_3)$, thus ω_a will abort (increasing the remaining execution time of τ_a by C_ω). It will perform a new attempt at time $t_5 \overset{\text{def}}{=} t_3 + C_\omega$.

• From time instant t_4 onwards, the previous reasoning can be carried out at any time instant $t_{k+1} \overset{\text{def}}{=} t_{k-1} + C_\omega$, with $k \geq 3$, thus leading to successive aborts of transactions ω_a and ω_b; and ultimately to a live-lock.

3. *Size of the laxity.* Given two concurrent transactions ω_a and ω_b belonging to tasks τ_a and τ_b, respectively. If we assume that $\ell_a(t)$ and $\ell_b(t)$ are the laxities of τ_a and τ_b at a given time instant $t \geq 0$, then choosing to abort the transaction with the largest laxity at time t does not guarantee that its deadline will be met. Indeed, if we assume without lost of generality that $\ell_a(t) > \ell_b(t)$, and we assume that there is an abort of ω at time instant t, then an upper bound on the number of aborts that ω_a can undergo and still meet its deadline is given by $\max\left(0, \left\lfloor \frac{\ell_a(t)}{C_{\omega_a}} \right\rfloor - 1\right)$. As a matter of fact, when deciding to abort a transaction, the cost of executing an additional attempt within the job laxity must be considered. Consequently, any contention manager that decides exclusively based on the value of the job laxity may select the less appropriate transaction to abort.

Transaction release time. The time instant at which a job τ_a starts executing its transaction ω_a is called the *transaction release time*. The contention manager that selects the transaction with the earliest release time for commit is referred to as the *Timestamp contention manager*. This criteria provides fairness as conflicting transactions are serialised by the order at which they are released, and predictability as once a transaction is released, the concurrent transactions already in progress

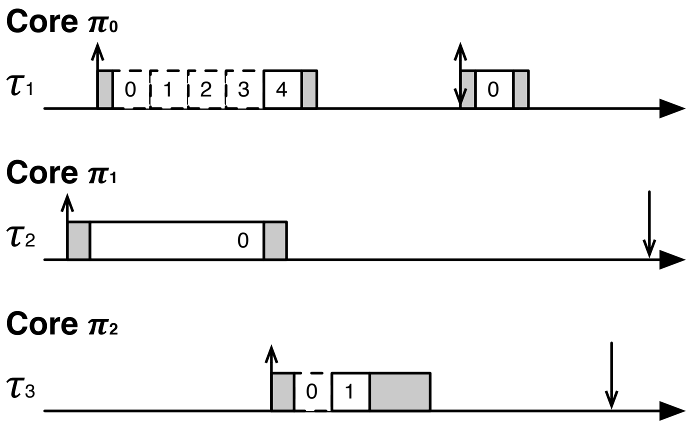


Figure 5.3: Contention management based on transaction release times.

will have to commit before, and no transaction arriving in the future will abort it. As illustrated in Figure 5.3, ω_2 executed by task τ_2 is the first transaction to commit, because it was the first to be released. Then, transaction ω_1 can only commit after ω_2 has committed.

Comparing against the scenario in Figure 5.1, where the number of aborts is the criteria for selecting the transaction to abort, we observe that the timestamp based manager is more convenient because all transactions can commit within a bounded amount of time, although this is done at the cost of a higher number of aborts for short transactions. Unfortunately, such a contention manager may induce deadlock in the subset of tasks assigned to the same core as a job executing a transaction can be blocked by another job with a lower-priority and executing a concurrent transaction with an earlier timestamp. To illustrate this claim, consider the following example.

Let τ_a and τ_b be two tasks assigned to the same core and executing transactions ω_a and ω_b, and contending for the same shared data object. We assume that τ_a has a lower priority than τ_b and preemption is allowed. We consider a scenario where a job of task τ_a is released first and is preempted by τ_b while it was executing its transaction ω_a. During the execution of task τ_b, a strict application of the "release time first" criteria would not allow transaction ω_b to commit, as its release time is later than that of ω_a. In this case, transaction ω_b will then abort indefinitely and this situation will ultimately result in a deadline miss for both τ_a and τ_b (see Figure 5.4). In order to circumvent this issue, the following solutions can be adopted: (i) allowing a transaction to commit before any other preempted transaction on the same core does, (ii) not allowing more than one transaction in progress per core.

Job absolute deadline. Consider tasks τ_a and τ_b assigned to two different cores. We assume the jobs of τ_a and τ_b execute transactions ω_a and ω_b, respectively. Upon detecting a conflict between these two transactions, a contention manager which relies on the "Job absolute deadline" criteria will select the transaction that belongs to the job with the earliest absolute deadline to commit. At first glance, this choice looks appealing, but this criteria does not indicate whether the laxity of the concurrent job (i.e., the job that has not been selected) allows for an additional abort of its transaction. Alternatively, if this job has a large absolute deadline and can bear a non negligible

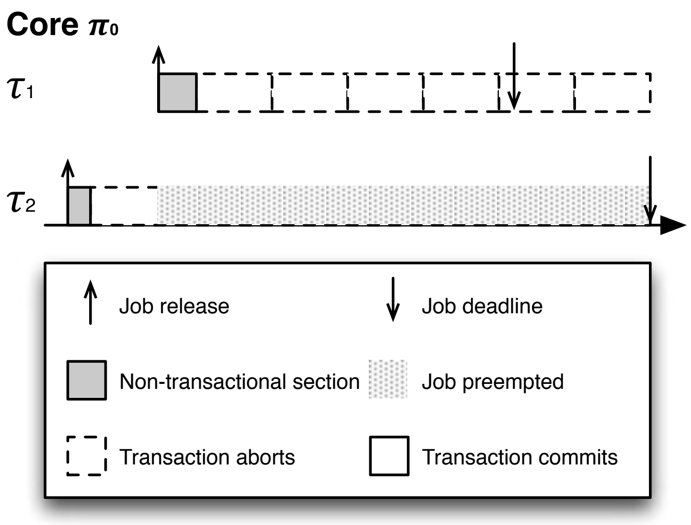


Figure 5.4: Contention between transactions on the same core resulting in deadlock.

number of aborts of its transaction, then this criteria may unfortunately induce excessive aborts. Figure 5.5 illustrates this effect where the same job releases as in figures 5.1 and 5.3 is considered.

This example shows that transaction ω_2 is aborted twice, in order to allow its contenders (here, transactions ω_1 and ω_3), released later but with earlier deadlines, to commit. In this example, it is noticing that the large absolute deadline of task τ_2 together with the large execution time associated to transaction ω_2 amplify the probability of conflict with upcoming concurrent transactions. Similar to the "transaction release time" criteria, the job absolute deadline criteria provides us with the following two desired features:

- for any set of conflicting transactions, the total order of the transactions is established beforehand and immutable, thus avoiding live-locks, and

- every transaction will possess the earliest absolute deadline at some point in time and eventually commit, thus enforcing liveness.

5.4 FIFO-CRT: a predictable contention manager for real-time systems

Considering the requirements in Section 5.1, a contention management for real-time systems must rely on criteria that always produce the same scheduling decisions irrespective of the time instant and/or the core on which it is executed. The static criteria discussed in Section 5.3 (i.e., the transaction release time and the job absolute deadline) serve this goal. Additionally, they ensure that every transaction will eventually commit, because at some time, it will be assigned the highest priority from the contention manager standpoint. From these two criteria, we opted to design our contention manager by using the transaction release time as it provides more fairness.

Algorithm 1: The FIFO-CRT contention manager algorithm for real-time systems.

```
Precondition : Current job of task $\tau_i$ finished executing transaction $\omega_i$.
Postcondition: Transaction $\omega_i$ commits if and only if it wins all conflicts.
if $\omega_i$ status is ACTIVE then
    forall $o_k \in DS_i$ do
        if $\omega_i$ status is ACTIVE then
            Acquire ownership of $o_k$;
            if $o_k \in WS_i$ then
                forall $\omega_j$ contending with $\omega_i$ on $o_k$ do
                    if $\omega_j$ status is ACTIVE then
                        if $\tau_j$ status is RUNNING then
                            if release$(\omega_i) >$ release$(\omega_j)$ then
                                Set $\omega_i$ status as FAILED;
                            else if release$(\omega_i) =$ release$(\omega_j) \wedge \sigma(\tau_i) > \sigma(\tau_j)$ then
                                Set $\omega_i$ status as FAILED;
                            end
                        end
                    end
                end
            end
        else
            Stop checking further objects;
        end
    end
    if $\omega_i$ status is ACTIVE then
        forall $o_k \in RS_i$ do
            Remove $\omega_i$ entry from list;
            Release $o_k$;
        end
        Commit updates;
        forall $o_k \in WS_i$ do
            Remove $\omega_i$ entry from list;
            forall $\omega_j$ accessing $o_k$ do
                Set $\omega_j$ status as ZOMBIE;
            end
            Release $o_k$;
        end
    else
        Release all currently owned objects;
        Abort and repeat $\omega_i$;
    end
else
    Abort and repeat $\omega_i$;
end
```

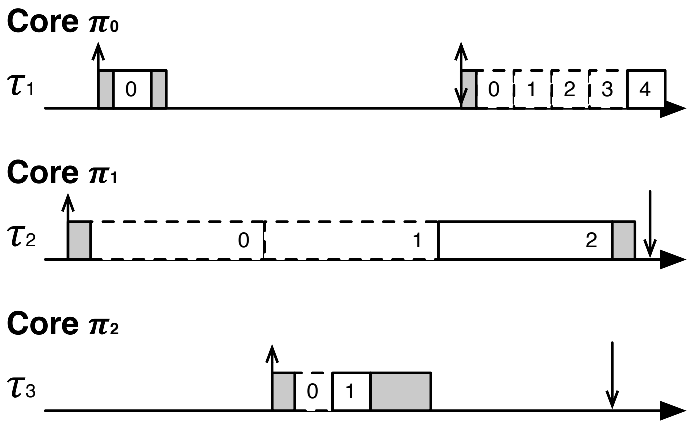


Figure 5.5: Contention management based on job absolute deadlines.

Algorithm 1 details the operations executed by every transaction when trying to commit. If the transaction is in the *active* state, then it will acquire the ownership of the objects in its data set (see lines 1 to 21). Note that every transaction may have to wait until an object is free before it acquires its ownership. For every object in its write subset, the transaction checks if it is the oldest active transaction accessing it amongst the currently running jobs (see line 9). This condition ensures that the transaction wins against all contenders on that object as they followed a FIFO-based approach. If the transaction fails on one object (see lines 10 and 12), then it immediately releases all its currently owned objects, aborts and repeats (see lines 36 and 37). Otherwise, if all conflicts are won, the transaction immediately releases the objects in the read subset (see lines 23 to 26), commits updates (see line 27), and mark the contenders in the write subset as *zombies*, before releasing the objects (see lines 28 to 34).

When checking for potential conflicts in lines 6 to 15, the target transaction just considers the subset of contending transactions that are *active* at that moment and the respective job is *running* (i.e., the job is executing on some core). A transaction in the *zombie* state can be overtaken by newer transactions without affecting its number of aborts since it is going to abort anyway. Thus, this algorithm increases the chances of commit for each transaction. In addition, active transactions that are not running (i.e., transactions belonging to a job that has been preempted) are ignored as a measure to avoid deadlocks.

Unlike locking solutions, the shared objects are owned just during the commit process, and not during the whole transaction section. This improves the parallelism. Note that the ownership process is controlled by the STM and should be transparent to the programmer. This feature improves the system composability. On another front, it is interesting to acquire ownership of the read subset of a transaction during its commit phase because it provides us with the following two key advantages:

- It allows to safely operate the data structures that keep track of transactional accesses (see line 24).

- It allows to prevent concurrent transactions from aborting. Specifically, consider a situation where two transactions, say ω_a and ω_b , are sharing an object, say O. Let us assume that transaction ω_a has the ownership on O for reading and transaction ω_b wants to update O. Then as ω_a owns O, transaction ω_b will be busy waiting for ω_a to commit instead of aborting and when ω_a commits, it will no longer be a contender to ω_b. In this case, ω_b can thus update O.

5.5 Summary

In this chapter we discussed the requirements for a contention manager designed for real-time systems. We elaborate on a selection of classical contention managers proposed for general purpose systems. Then, we conducted an indepth discussion on criteria that would help us meet all the timing requirements for an application with stringent timing requirements. Finally we proposed a contention management algorithm that is based on the transaction release time and which complies with all the requirements. Since predictability depends on the level of preemptions of every job while it executes a transaction, it follows that further improvements can only be achieved by managing the number of preemptions while transactions are in progress. This problem is addressed in Chapter 6.

Chapter 6

Scheduling tasks and transactions under FIFO-CRT

Chapter 5 presented the FIFO-CRT contention manager that provides predictability and fairness. However, scheduling decisions such as preempting a job while it is executing a transaction can have adverse effects on the intended behaviour of the contention manager. This chapter addresses this issue, proposing three extensions of the partitioned EDF (P-EDF) scheduling policy in order to further improve predictability. The first two approaches called Non-Preemptive Until Commit (NPUC) and Non-Preemptive During Attempt (NPDA) disable preemptions during the execution of the transactional sections. In contrast, the third approach allows preemptions. It builds on the Stack Resource Protocol (SRP) (Baker, 1991) and is called Stack Resource Protocol for Transactional Memory (SRPTM). This approach introduces the concept of *preemption levels* that the scheduler uses to control the number of preemptions of transactions in a wiser manner.

6.1 Impact of the scheduling policy on the contention manager

FIFO-CRT serialises transactions according to their release times, and commits these transactions by following this order: first arrived, first served. However, by applying this rule blindly, the scheduling decisions at runtime may lead to situations where the timing requirements of the application are violated, i.e. some jobs miss their absolute deadlines. To avoid these situations, it might be necessary to violate the "transaction release time first" rule in specific cases in order to meet all the timing requirements. To this end, we recall (see Section 5.3) that the following two solutions can be adopted: (1) allowing a transaction to commit before any other preempted transaction on the same core does, (2) not allowing more than one transaction in the active state per core.

▷ The first solution, where every transaction is allowed to commit before any other preempted transaction on the same core, has been implemented in the FIFO-CRT contention manager (see Section 5.4, Algorithm 1, line 8).

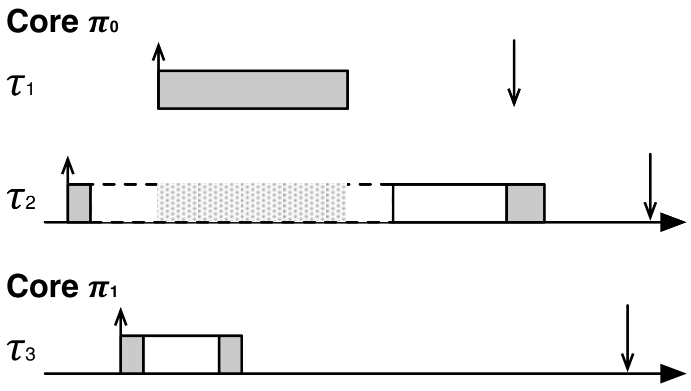


Figure 6.1: A preempted transaction with earlier release time being aborted.

Figure 6.1 illustrates a case in which this solution is applied. In this example, tasks τ_1 and τ_2 are assigned to core π_0 and task τ_3 is assigned to core π_1. Task τ_1 does not execute any transaction, whereas τ_2 and τ_3 execute transactions ω_2 and ω_3, respectively, that are contenders on the same memory object. Each core executes the EDF scheduler. Task τ_2 releases its first job before τ_3, and is preempted by τ_1 while it was executing its transaction. While τ_2 is preempted, transaction ω_3 tries to commit updates on the data set of ω_2. Although transaction ω_3 has a later release time, the contention manager decides that it should commit because τ_2 is not running. This decision invalidates the transactional data upon which ω_2 was working, so its state becomes zombie and it is doomed to abort.

Although this solution avoids deadlocks, it exposes transactions to starvation. Figure 6.2 illustrates this limitation. In this example, tasks τ_1 and τ_2 are assigned to core π_0 and task τ_3 is assigned to core π_1. Task τ_1 does not execute any transaction, whereas τ_2 and τ_3 execute transactions ω_2 (which is long) and ω_3 (which is short), respectively, that are contenders on the same memory object. Each core executes the EDF scheduler. Task τ_2 releases its first job before τ_3, and every time it is preempted by τ_1 while it was executing its transaction, τ_3 is released and executes its transaction ω_3. In this scenario, the instances of transaction ω_3 abort ω_2 and task τ_2 ends up missing its deadline. A way to get over this issue is to leverage on the scheduling decisions on the scheduler side, as no further improvement can be achieved by tuning the contention manager.

$\triangleright$ The second solution advocates for at most one transaction in the active state on each core. If this restriction applies, then we no longer require a transaction to abort a preempted one with an earlier release time, in order to avoid deadlocks. In practice, this restriction can be easily implemented in different ways on the scheduler side (see Sections 6.2 and 6.3 for further details).

6.2 Non-preemptive approaches

The issues of transaction starvation and associated unpredictability as mentioned in Section 6.1 can be solved by prohibiting preemptions during the execution of each transaction. If a transaction is guaranteed not be preempted, then the success of transaction will depend only on the contention

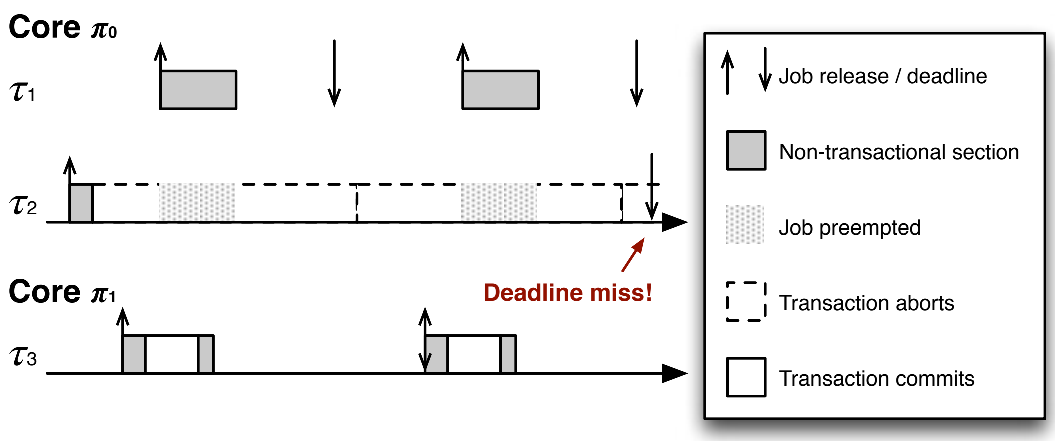


Figure 6.2: Long transaction is excessively aborted because of preemptions.

management policy. This solution is similar to priority boosting (Rajkumar, 1990): raising the priority of a job to the highest priority level during the execution of each critical section. The Flexible Multiprocessor Locking Protocol (FMLP) (Block et al., 2007) also follows a similar approach. It executes critical sections in a non-preemptive manner, by their order of arrival. In this manuscript, we consider two non-preemptive approaches to schedule transactions:

- *Non-preemptible until commit (NPUC)* in which every job is guaranteed to be scheduled from the moment the transaction is released until it successfully commits.
- *Non-preemptible during attempt (NPDA)* in which every job is non-preemptible during the execution of its transaction, but has preemption points between attempts. That is, every job executing a transaction can be preempted between an abort and the subsequent restart of its transaction.

6.2.1 Non preemptible until commit (NPUC)

Under NPUC, each transaction will take-over the core in which it is executing (see Algorithm 2), and will abort until all active direct contenders (transactions whose data accesses will conflict with the accesses of the transaction in consideration) that arrived earlier have committed and finished. Only after commit, the transaction releases the core (see Algorithm 3). The verification of the state of each job executing a transaction (running or preempted) by the contention manager (Algorithm 1: line 8) thus becomes redundant and should be omitted. In Figure 6.3, although task τ_2 has an earlier deadline, it is able to preempt τ_3 only after its transaction has committed.

NPUC is totally predictable, in the sense that the time required for the transaction to successfully commit depends solely on the transactions that are already executing when the transaction is released. Since direct contenders (transactions that have, at least, one conflict with the write set) will also wait for their own earlier direct contenders to finish, contention is propagated in chain. So, in the worst case, a transaction will have to wait for $(m-1)$ transactions to complete, assuming that every other core is already executing one transaction. However, this predictability comes

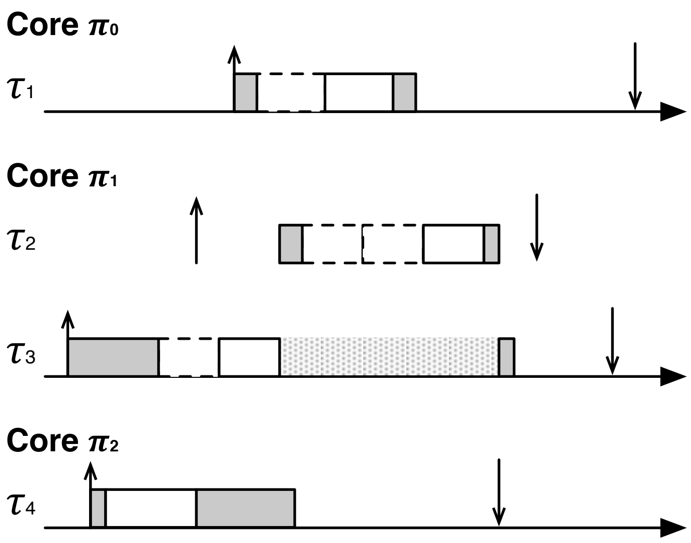


Figure 6.3: Transactions scheduled under NPUC.

with a cost: higher priority tasks will have their responsiveness degraded due to the blocking time associated to lower-priority tasks. To illustrate this phenomenon, Figure 6.4, displays three tasks τ_1, τ_2 and τ_3 executing transactions ω_1, ω_2 and ω_3, respectively. Transactions ω_1 and ω_2 are direct contenders on object o_1, and the same applies to transactions ω_2 and ω_3 on object o_2. In this setting, ω_1 and ω_3 are not contenders. In this example, ω_3 aborts on its first attempt, because ω_2 is executing. Then, ω_3 can commit only after ω_2 has committed, which in turn can commit only after ω_1 has committed.

Algorithm 2: STM actions taken when transaction ω_i starts under NPUC.

```
**Precondition :** Current job of task $\tau_i$ is starting transaction $\omega_i$
**Postcondition:** The scheduler disabled preemptions.
1 Disable preemptions;
2 Register $\omega_i$ in the STM;
```

Algorithm 3: STM actions taken when transaction ω_i tries to commit under NPUC.

```
**Precondition :** Current job of task $\tau_i$ finished executing transaction $\omega_i$
**Postcondition:** The scheduler enables transactions only if the transaction commits.
1 Validate $\omega_i$;
2 if $\omega_i$ aborts then
3     Repeat attempt;
4 else
5     Commit $\omega_i$;
6     Enable preemptions;
7 end
```

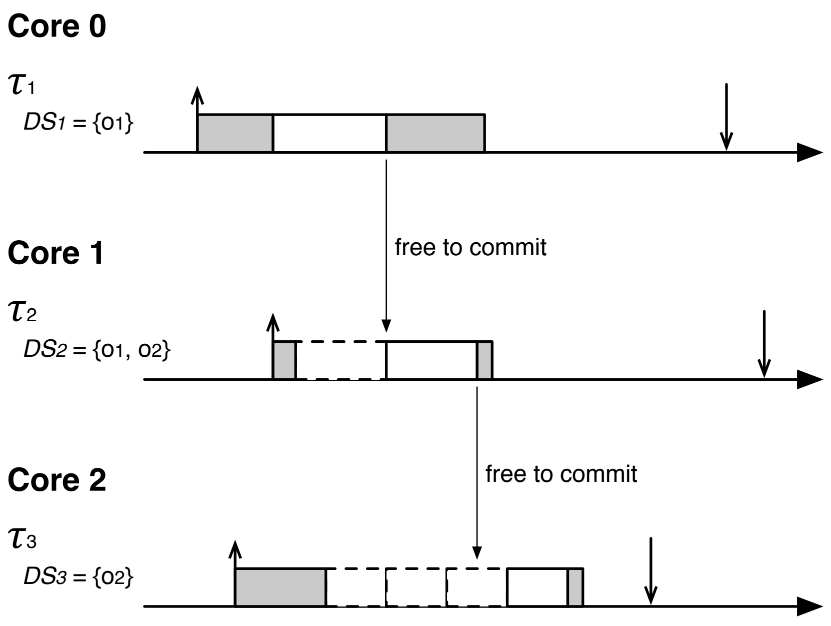


Figure 6.4: Contention cascades.

6.2.2 Non preemptible during attempt (NPDA)

Under NPDA, preemptions are limited during a transaction to preemption points inserted between consecutive attempts. That is, between a transaction abort and the time instant it tries a new attempt (see Algorithm 4). During this time period, the transaction is deemed as in the failed state. In other words, it does not compete with other transactions. This policy ensures that the success to commit of each attempt depends only on the running transactions at every time instant. In addition, it reduces the blocking incurred by low-priority tasks, as compared to NPUC, thus improving responsiveness of higher-priority tasks. Figure 6.5 illustrates the same scenario as in Figure 6.3, but assuming that tasks are scheduled by following the NPDA policy. Here, in contrast to NPUC, task τ_2 is able to preempt τ_3 immediately after its first abort.

In NPDA, since every job can be preempted between two consecutive attempts of its transaction, then each core can hold more than one transaction in progress at any time instant. This means that the number of contenders with earlier release time to a given transaction is not limited to the number of remaining cores $(m-1)$, as in NPUC. Consequently, although NPDA improves the responsiveness of tasks, it is less predictable than NPUC. In the depicted example (Figure 6.5), task τ_1 in core π_1 aborts its transaction twice because of the transactions executed by tasks τ_2 (first abort) and τ_3 (second abort) that execute both in core π_2.

6.3 Preemptive approach (SRPTM)

Although the non-preemptive approaches described in Section 6.2 enforce the intended behaviour of the contention manager, i.e. (1) all transactions are served relative to their release times, and

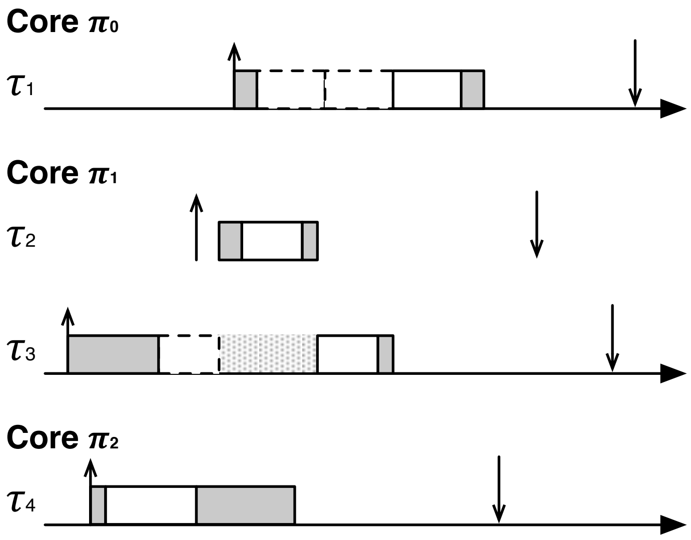


Figure 6.5: Transactions scheduled under NPDA.

(2) the number of aborts is bounded for each transaction, both approaches have their advantages and disadvantages.

- Non-preemptive Until Commit (NPUC) provides a fully predictable serialisation of transactions, derived from the fact that there can only be one transaction in progress per core. However, it has a negative impact on the responsiveness of urgent tasks.
- Non-preemptive During Attempt (NPDA) improves the responsiveness of the more urgent tasks. However, it is a non trivial exercise to determine a tight upper-bound on the transaction overheads. This is because the system allows multiple simultaneous transactions in progress in the same core.

The approach described in this section tackles the drawbacks of NPUC and NPDA as it already (1) provides a predicable serialisation of transactions and (2) maintains the responsiveness of jobs with shorter absolute deadlines. To this end goal, at most one transaction per core is allowed. In addition, preemption during the execution of every transaction is not disabled.

Before going into the technical details behind our proposed approach, let us summarise the main idea. This approach is based on the Stack Resource Protocol (SRP) (Baker, 1991). Each task τ_i is assigned a preemption level, denoted as λ_i, by following a non-increasing order of task relative deadlines, i.e. $\lambda_i < \lambda_j$ if and only if $D_i > D_j$. Two tasks with the same relative deadline are assigned the same preemption level. Preemption levels are assigned globally, meaning that tasks are ordered by relative deadline irrespectively of the core in which they execute.

In addition, each transaction is also assigned a preemption level that can be higher than the preemption level of the task that executes the transaction. The transaction preemption level expresses the highest preemption level of a task that can be affected by the outcome of the particular transaction. For example, assume two transactions that are direct contenders (see Chapter 4, Definition 26), the first transaction being executed in a job with a short relative deadline and the other

Algorithm 4: STM actions taken when transaction ω_i tries to commit under NPDA.

```
Precondition : Current job of task $\tau_i$ finished executing transaction $\omega_i$
Postcondition: The scheduler enables jobs with earlier deadlines to execute if transaction
               aborts.
1  Validate $\omega_i$;
2  if $\omega_i$ aborts then
3      Enable preemptions;
4      Schedule blocked jobs;
5      Disable preemptions;
6      Repeat attempt;
7  else
8      Commit $\omega_i$;
9      Enable preemptions;
10 end
```

transaction being executed in a job with longer relative deadline. The first transaction may have to wait for the second one to commit, and so the preemption level of the second transaction should reflect the preemption level of the first job.

Similar to SRP, a job can be preempted by another job (with earlier absolute deadline) while executing a transaction only if the latter job has a higher preemption level than the running transaction. This way, the scheduler ensures that preempting a transaction will not delay a potential contender with a shorter relative deadline.

Besides the interference due to direct contenders, each transaction can undergo an additional delay due to indirect contenders (see Chapter 4, Definition 27), when serialised in FIFO order, as seen in Figure 6.4. This example illustrates a transaction of task τ_3 waiting for a transaction of task τ_2 to commit, which in turn is waiting for a transaction of task τ_1 to commit. Transactions of tasks τ_1 and τ_3 are indirect contenders in this sequence, because although their data sets do not intersect, ω_3 is delayed by ω_1. Therefore, the preemption level set to a particular transaction must reflect the highest premption level of all contenders (direct and indirect) that could have to wait on its success, i.e. all transactions in the same contention group (see Chapter 4, Definition 24).

6.3.1 Assigning preemption levels to tasks and to transactions

Main idea. We initially assign sequential preemption levels to tasks by following a non-increasing order of their relative deadlines so that each task τ_i has a *task preemption level*, denoted as λ_i. Before going into further details, let us introduce a couple of key definitions.

Definition 33 (Ceiling of a STM object)**.** Given a set of transactions accessing (either reading or writing) an STM transactional object o_k, we define the ceiling of o_k, denoted as $ceil(o_k)$, as the maximum preemption level of all tasks executing a transaction accessing o_k. Formally, the ceiling

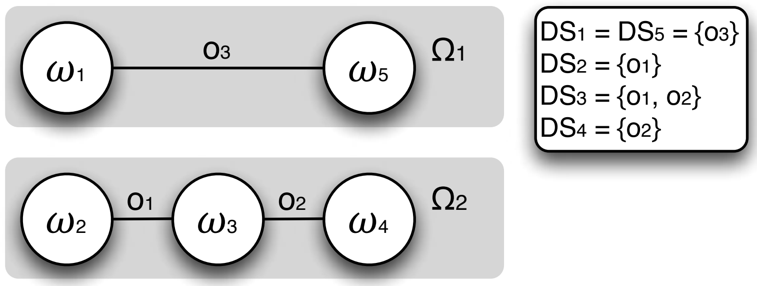


Figure 6.6: Transaction dependencies by object concurrency.

of o_k is given by

$$ceil(o_k) \stackrel{\text{def}}{=} \max_{i\in[1..n]} \{\lambda_i \mid (\tau_i \text{ executes } \omega_i) \text{ and } (\omega_i \text{ uses } o_k)\} \tag{6.1}$$

Definition 34 (Ceiling of a contention group)**.** Let Ω_k be a contention group (see Definition 24), we define the ceiling Ω_k, denoted as $ceil(\Omega_k)$, as the maximum preemption level of all tasks that belong to Ω_k. Formally, the ceiling of the contention group Ω_k is given by

$$ceil(\Omega_k) \stackrel{\text{def}}{=} \max_{i\in[1..n]} \{\lambda_i \mid (\tau_i \text{ executes } \omega_i) \text{ and } (\omega_i \in \Omega_k)\} \tag{6.2}$$

Definition 35 (Preemption level of a transaction)**.** Given a contention group Ω_k such that transaction ω_i belongs to Ω_k, we define the preemption level of transaction ω_i, denoted as λ_{ω_i}, as the ceiling of Ω_k. Formally, the preemption level of ω_i is given by:

$$\lambda_{\omega_i} \stackrel{\text{def}}{=} ceil(\Omega_k), \quad \omega_i \in \Omega_k \tag{6.3}$$

Taking into account the definitions 33 to 35, and for practical implementation purposes, we set the transaction preemption level of any task that does not execute any transaction to zero. Now, in order to get a better understanding of all the concepts defined above, we provide the example below.

Example. Consider a task set $\tau = \{\tau_1, \ldots, \tau_6\}$ in which tasks τ_1 to τ_5 execute transactions ω_1 to ω_5, respectively, and τ_6 does not execute any transaction. We assume that transactions ω_1 to ω_5 are sharing a set of three STM objects $O = \{o_1, o_2, o_3\}$, as illustrated in Figure 6.6, forming two distinct contention groups: $\Omega_1 = \{\omega_1, \omega_5\}$ and $\Omega_2 = \{\omega_2, \omega_3, \omega_4\}$. Table 6.1 provides the relative deadline (see Table 6.1: column 2) and the preemption level (see Table 6.1: column 3) for every task, as well as the dependencies between the transactions, where the STM objects accessed by each transaction are marked by a dot.

Table 6.2 illustrates the results of assigning preemption levels to transactions. The ceiling of each transactional object o_i $(i = 1, 2, 3)$ (bottom line of Table 6.2) is set to the maximum preemption level of all tasks that access it. Specifically, object o_1 is accessed by transaction ω_2 which is executed by task τ_2 with preemption level 2, and transaction ω_3 which is executed by task τ_3

Tasks	D_i	λ_i	STM objects		
			o_1	o_2	o_3
τ_1	50	6			●
τ_2	100	2	●		
τ_3	80	3	●	●	
τ_4	70	4		●	
τ_5	120	1			●
τ_6	60	5			

Table 6.1: Task parameters and transaction dependencies.

with preemption level 3. Therefore, $ceil(o_1) = \max(2,3) = 3$. In the same manner, it follows that $ceil(o_2) = \max(3,4) = 4$ and finally $ceil(o_3) = \max(6,1) = 6$. According to Definition 34, the preemption level of each transaction (see Table 6.2: column 7) is the highest preemption level found in the group to which the transaction belongs. Hence, $\lambda_{\omega_1} = \lambda_{\omega_5} = ceil(\Omega_1) = \max(\lambda_1, \lambda_5) = 6$ and $\lambda_{\omega_2} = \lambda_{\omega_3} = \lambda_{\omega_4} = ceil(\Omega_2) = \max(\lambda_2, \lambda_3, \lambda_4) = 4$. Note that $\lambda_{\omega_6} = 0$ as task τ_6 does not execute any transaction.

At this point, we have introduced all the materials required to specify in details our scheduling policy.

6.3.2 Scheduling policy

In this section, we specify our scheduling policy, which is based on the P-EDF scheduler i.e., each task is statically assigned to a specific core at design time (and task migrations from one core to another at runtime are not allowed) and each core schedules its subset of tasks at runtime by following the classical EDF scheduler. To this end, we introduce additional rules to tune the aforementioned general policy. These rules apply only when a job is executing a transaction.

Rule 1 (Core ceiling)**.** We associate to each core π_k a non-negative value Λ_k, referred to as the core ceiling, and given by the preemption level of the transaction in progress on that core. That is, if task τ_i starts executing its transaction ω_i on core π_k, then the core ceiling of π_k is set to the

Tasks	D_i	λ_i	TM objects			λ_{ω_i}
			o_1	o_2	o_3	
τ_1	50	6			●	6
τ_2	100	2	●			4
τ_3	80	3	●	●		4
τ_4	70	4		●		4
τ_5	120	1			●	6
τ_6	60	5				0
ceil(o_i)			3	4	6	

Table 6.2: Example of calculation of preemption levels.

preemption level of ω_i, i.e $\Lambda_k \leftarrow \lambda_{\omega_i}$. Λ_k reflects the urgency to commit the transaction in progress on core π_k. When no transaction is in progress on core π_k, then the core ceiling is set to zero, i.e. $\Lambda_k = 0$.

Rule 2 (Core ceiling reset)**.** Upon a successful commit of a transaction, the core ceiling is set back to zero (i.e. $\Lambda_k \leftarrow 0$).

Rule 3 (Unique transaction in progress per core)**.** At any time instant on each core π_k, we allow only one transaction to be in progress and all released jobs executing a transaction are blocked until the transaction in progress has committed. This rule holds true irrespective of the status of the job executing the transaction in progress (running or preempted).

Rule 4 (Priority inversion prevention)**.** Assume transaction ω_p belonging to job τ_p is in progress on core π_k. A newly released job, say τ_a, can preempt τ_p only if the following three conditions hold true: (C1) τ_a has a higher priority (i.e. $d_a < d_p$); (C2) τ_a is not executing any transaction (i.e. $\lambda_{\omega_a} = 0$) and (C3) the preemption level of τ_a is higher than the core ceiling (i.e. $\lambda_a > \Lambda_k$). In the case where job τ_a is executing a transaction (i.e. $\lambda_{\omega_a} > 0$), then the following three actions are taken: (A1) τ_a is blocked; (A2) the core ceiling is risen to the preemption level of τ_a and (A3) the core π_k resumes the execution of ω_p.

According to rules 1 to 4, the condition of FIFO-CRT where a transaction in a preempted job can be aborted by a contender released later (see Algorithm 1: line 7) is no longer necessary, and must be dropped. Rule 3 improves the predictability of the transactions in progress, since each transaction will have to wait, at most, for $(m-1)$ simultaneous transactions in progress before it can commit. In addition, it also reduces the preemption overhead because every job can be blocked at most once, at its release time. Algorithm 5 details the decision tree that the scheduler follows when a job τ_a is released and a job τ_r is currently running on core π_k.

6.4 Summary

In this chapter, we provided solutions to mitigate the potentially negative effects of enabling preemptions while executing transactions. To this end purpose, we proposed three extensions of the P-EDF scheduler, referred to as Non-Preemptive Until Commit (NPUC), Non-Preemptive During Attempt (NPDA) and Stack Resource Protocol for Transactional Memory (SRPTM) to tune the scheduling decisions when a transaction is in progress on a core. These new policies come up with interesting features such a substantial reduction of the number of preemptions. Preemptions are known to be a potential source of indeterminism on multi-core platforms executing a set of parallel tasks. In addition, these new policies allow us improving both the predictability and the fairness of the contention manager in comparison to the classical P-EDF. In the next chapter we provide the schedulability analysis of a task set scheduled by following each of these policies.

Algorithm 5: Scheduling decisions taken when a job τ_a is released and job τ_r is running.

```
Data: The transaction in progress, if any, belongs to a job, say $\tau_p$.
if $d_a < d_r$ then // Released job $\tau_a$ has an smaller absolute deadline.
    if $\Lambda_k > 0$ then // Core $\pi_k$ has a transaction in progress.
        if $\lambda_a > \Lambda_k$ then
            if $\lambda_{\omega_a} > 0$ then // $\tau_a$ has a transaction.
                $\Lambda_k \leftarrow \lambda_a$;
                insert $\tau_a$ in ready-queue;
                run $\tau_p$;
            else // $\tau_a$ is not executing any transaction.
                preempt $\tau_r$;
                run $\tau_a$;
            end
        else
            insert $\tau_a$ in ready-queue;
        end
    else // No transaction in progress exists.
        preempt $\tau_r$;
        run $\tau_a$;
    end
else // Running job has earlier absolute deadline.
    insert $\tau_a$ in ready-queue;
end
```

Chapter 7

Schedulability analysis of tasks under NPDA, NPUC and SRPTM

In this chapter, we report on the schedulability analysis assuming that tasks are scheduled by following NPDA, NPUC and SRPTM policies, respectively. Indeed, the schedulability analysis of a set of real-time tasks is usually assessed through simulation or by using closed-form mathematical expressions. We opted for the second alternative and derived a worst case response time (WCRT) based analysis. Such an approach correlates closely with the behaviour of the system at runtime, especially when tasks may not be independent and/or may execute transactions.

7.1 WCRT analysis for NPDA

NPDA does not restrict the number of transactions that can simultaneously be in progress on each core, thus increasing the complexity of determining a tight upper-bound on the number of times each transaction may be aborted. The worst-case scenario in this context occurs when each transaction has to wait for all of its direct contenders before it can commit. To support this claim, Figure 7.1 illustrates a contention group that involves three cores $\{\pi_0, \pi_1, \pi_2\}$ and six tasks $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$. Here, we assume that task τ_i executes transaction ω_i and that transactions $\omega_2, \omega_3, \omega_4$ are direct contenders of ω_1. Additionally, we assume that in the illustrated time segment[1], the chronological order in which transactions are released is $\omega_2 \prec \omega_4 \prec \omega_6 \prec \omega_3 \prec \omega_1$, and that τ_2 and τ_4 are preempted when we start observing the system. In this figure, thin vertical lines indicate the time instants at which a transaction is fated to abort, i.e. either (1) by being invalidated when a contender commits, or (2) by the contention manager decision at the commit time. The transaction that aborts and the one that causes the abort are displayed on the extremes of each thin line. Transaction ω_1 executes on core π_0, and may abort in favour of transactions that were released earlier on different cores. The sequence of events in this special case shows that ω_1

[1]Note that τ_5 is not released in the illustrated time segment.

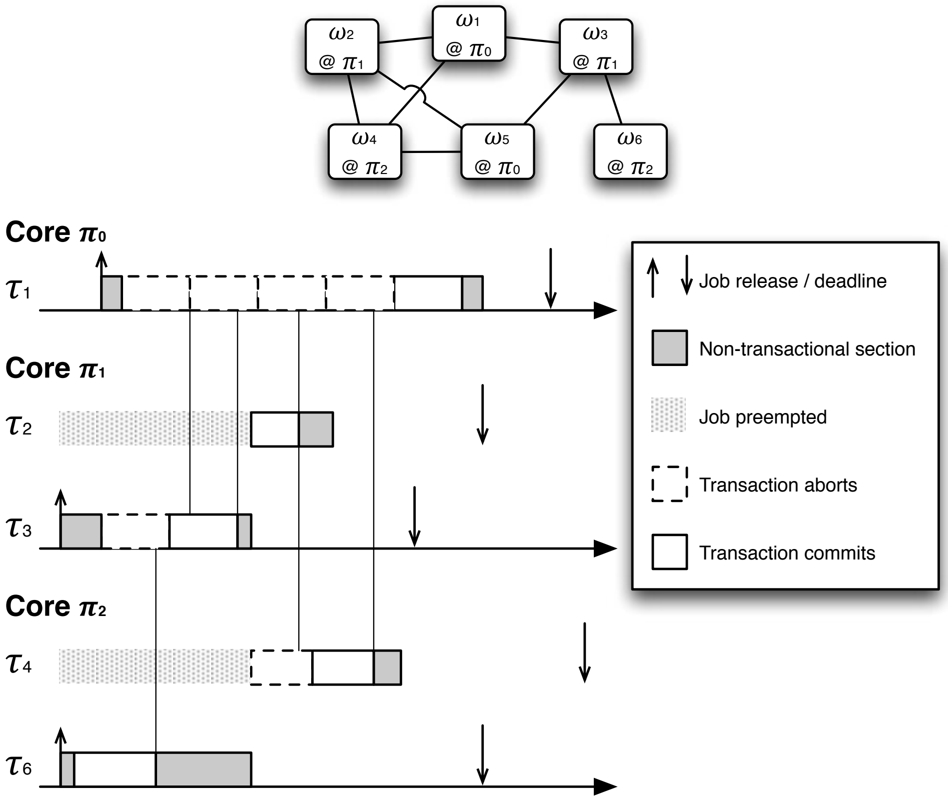


Figure 7.1: Preempted jobs executing ω_2 and ω_4 are rescheduled in times to abort transaction ω_1.

commits only when all its direct contenders (i.e. ω_2, ω_3 and ω_4) have committed. Note that this delay includes waiting for ω_6 (an indirect contender) to commit.

Although it is straightforward to identify the direct contenders of a transaction, the exercise becomes very challenging when it comes to compute the delay they impose on the commit of that transaction. As a matter of fact, when (1) the number of transactions (i.e., the vertices), (2) the number of dependencies (i.e., edges) and (3) the number of assigned cores in a contention group grow, the search-space where to find the sequence of transactions that leads to the longest commit delay also grows exponentially. Specifically, the computational complexity of completing this operation is in order of $\mathcal{O}(n^n \times m)$, where n is the total number of tasks and m is the total number of cores. Therefore, a tight feasibility analysis for NPDA is computationally intractable and not sound in the context of this thesis.

7.2 WCRT analysis for NPUC

NPUC schedules transactions in a non-preemptive manner until they commit (see Chapter 6, Section 6.2.1). This property ensures two essential predicates for the scheduler: (*i*) at most one transaction can be in progress on each core at any time instant, and (*ii*) the delay experienced by any transaction prior to its commit depends exclusively on its set of direct contenders, which in turn, depend on their own set of direct contenders. We can compute an upper-bound on the WCRT of a transaction by determining the sequence of transactions that will produce its longest delay. This longest delay occurs when the transaction under analysis is released at a time instant when the pending workload associated to its contenders is maximum. This maximum workload is reached when the remaining execution time for each contender corresponds to its worst case execution time. Therefore, we assume that the transaction under analysis is released at the same time instant as all its contenders.

Main idea of the analysis. We consider that task τ_i executes a transaction ω_i and is the task under analysis. To compute a sound and tight upper-bound on the WCRT of τ_i, we should compute the following four expressions:

1. the WCRT of the transaction ω_i;
2. the WCRT of the section of non-transactional code a-ω_{i} before the transaction starts executing;
3. the WCRT of the section of non-transactional code p-ω_{i} after the transaction has committed; and finally
4. the sum of all these three values[2].

7.2.1 WCRT of transaction ω_i

This section presents two methods for the computation of an upper bound on the commit delay of transaction ω_i. The first method provides a tight bound, but requires that all possible sequences of transactions that produce a delay on ω_i are known. The second method, in contrast to the previous one, provides a pessimistic upper bound, but it presents the advantage of being agnostic to this input. This second method has a linear-time complexity. The selection of one method over the other will depend on the user needs and the available computational resource and time.

$\triangleright$ *Method 1: Tight upper bound on the WCRT of transaction ω_i.* This method is based on the contention graph (see Chapter 4, Definition 24) to determine all the possible transaction sequences that may delay ω_i. From this, the sequence that produces the longest delay on ω_i is found by considering the *simple paths*[3] converging to the vertex ω_i with no repetition of cores. This is possible as the contention graph is finite. For each transaction sequence, an upper-bound on the WCRT of ω_i is computed by upper-bounding the times to commit of all the transactions released prior to ω_i. Figure 7.2 depicts an example in which three transactions — ω_6, ω_3 and ω_1 — are released almost simultaneously on cores π_2, π_1 and π_0, respectively. Assuming that the transactions are released in this order, the WCRT of ω_1 depends on that of ω_3, which in turn depends on the WCRT of ω_6. Note that for each sequence of transactions: ($P1$) *each transaction performs at most two attempts before success once it becomes eligible to commit*; ($P2$) *the very first transaction in the sequence may abort after ω_i is released, due to a contender executing on the same core as ω_i.* Figure 7.3

[2]As the WCRT of τ_i is upper-bounded by the sum of the response times of the non-transactional sections and the transactional sections, we consider only one transaction per task.

[3]A simple path is a sequence of connected vertices with no repetition.

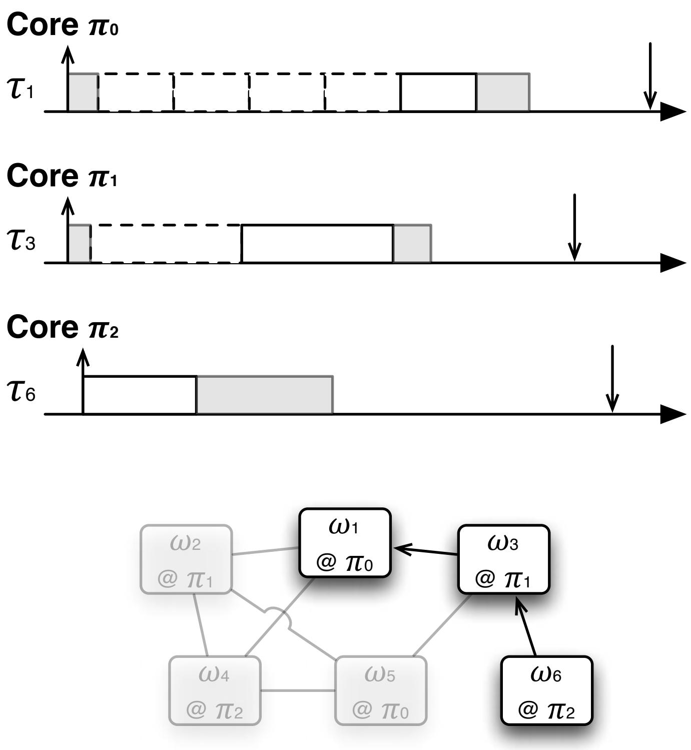


Figure 7.2: Sequence of transactions until ω_1 commits.

illustrates a case in which such a phenomenon occurs. Here, three tasks τ_1, τ_2 and τ_3 are assigned to cores π_0, π_1 and π_2, respectively, and are executing concurrent transactions ω_1, ω_2 and ω_3. Transaction ω_1 is released first and ω_2 and ω_3 are released at the same time. We assume that another transaction, say ω_x, not belonging to the sequence and executing on core π_2 invalidated the attempt of ω_1 just before ω_3 is released. In this scenario, ω_1 is fated to abort due to the commit of ω_x, thus delaying the commit time of ω_3.

Points ($P1$) and ($P2$) allow us to formulate an upper bound on the WCRT of transaction ω_i in a given sequence. To this end, we consider a sequence of $k > 0$ transactions and υ as the function that returns the transaction at each position in this sequence. We have $\upsilon(k) = \omega_i$.

Lemma 7.1. *The WCRT of ω_i in a given sequence, denoted as $R^{(k)}$, is computed recursively by using Equation 7.1.*

$$\begin{cases} R^{(1)} = 2 \cdot C_{\upsilon(1)} \\ R^{(q)} = \left(\left\lceil \frac{R^{(q-1)}}{C_{\upsilon(q)}} \right\rceil + 1 \right) \cdot C_{\upsilon(q)} & \text{if } 1 < q \leq k. \end{cases} \tag{7.1}$$

Proof. The proof follows directly from Point (1) and Point (2). In the worst-case scenario, the first transaction in the sequence takes at most two attempts to commit (see Point (2)). Then, transaction at position $q \geq 2$ has to wait for the transaction at position $q-1$ to commit plus at most one additional attempt to successfully commit (see Point (1)). □

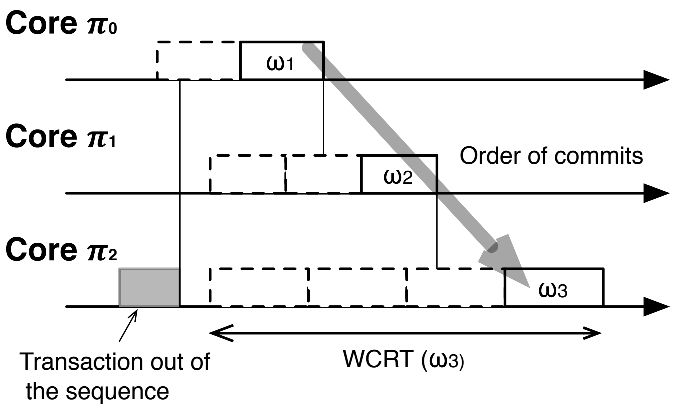


Figure 7.3: ω_1, the first transaction in the sequence aborts once before commits.

Now, let $\mathscr{S}_i$ denote the set of all simple paths that converge towards ω_i and let $\mathscr{S}_{i,k}$ denote the subset of $\mathscr{S}_i$, which consists only of simple paths of length k. We assume that ω_i belongs to contention group Ω_g.

Lemma 7.2. *An upper bound on the WCRT of ω_i is given by:*

$$R_{\omega_i} = \max_{k=1}^{m_g} \left\{ \max_{\mathscr{S}_{i,k}} \left(R^{(k)} \right) \right\} \tag{7.2}$$

Proof. The length of the simple path to transaction ω_i is at most m_g as ω_i belongs to Ω_g. For any sequence of transactions in Ω_g, an upper-bound on the WCRT of the transaction at position q is given by Equation 7.1. From these two predicates, it follows that the WCRT of ω_i cannot exceed the RHS of Equation 7.2. Now, as Ω_g is finite by assumption, this RHS is also an upper-bound on the WCRT of ω_i. □

Although this method provides a tight upper bound on the WCRT of transaction ω_i, it suffers from the fact that all the possible simple paths must be considered. This limitation becomes impractical with an increase of the number of transactions and cores in the system.

▷ *Method 2: Linear-time, but pessimistic, upper bound on the WCRT of transaction ω_i.* The main idea behind this method is to avoid the usage of all sequences of transactions leading to ω_i. To this end, we consider a single sequence of transactions in the same contention group as ω_i, and with the longest execution time on each core. In other words, even if the resulting selection of transactions does not form a possible sequence at runtime, it determines the longest possible delay that the commit of transaction ω_i can undergo. Formally, first we determine the longest execution time C_{Ω_g,π_ℓ} of all the transactions in Ω_g that are assigned to each core $\pi_\ell \in \Pi_g$ by using Equation 7.3.

$$C_{\Omega_g,\pi_\ell} = \max \left\{ C_{\omega_j} \mid \omega_j \in \Omega_g \wedge \sigma(\tau_j) = \pi_\ell \right\} \tag{7.3}$$

Then, from Point (1) and Point (2) in Method 1, we compute an upper-bound I_{Ω_g,π_k} on the delay that any transaction in Ω_g assigned to core π_k may suffer by using Equation 7.4.

$$I_{\Omega_g,\pi_k} \overset{\text{def}}{=} \sum_{\pi_\ell \in \Pi_g \setminus \pi_k} 2 \cdot C_{\Omega_g,\pi_\ell} \tag{7.4}$$

Note that I_{Ω_g,π_k} is common to all transactions in Ω_g assigned to π_k, so it is computed only once for each pair of core and contention group.

Lemma 7.3. *An upper-bound on the WCRT of transaction ω_i is given by Equation 7.5.*

$$R_{\omega_i} = I_{\Omega_g,\pi_k} + 2 \cdot C_{\omega_i} \tag{7.5}$$

Proof. The WCRT of ω_i is upper-bounded by the delay that ω_i suffers from all concurrent transactions with an earlier release time (see Equation 7.4), augmented by the time that ω_i takes to commit once it is possible to do so (see Point (1)). $\square$

This method is pessimistic as Equations 7.4 and 7.5 consider a sequence of transactions that may never occur in practice.

7.2.2 WCRT of task τ_i

We recall that the non-transactional sections of code of task τ_i (a$-\omega_i$ and p$-\omega_i$) are scheduled by following the fully preemptive P-EDF scheduler, whereas the transaction ω_i is scheduled with disabled preemption. Therefore, we need to compute the WCRT of a$-\omega_i$ and p$-\omega_i$ in order to derive the WCRT of task τ_i.

7.2.2.1 WCRT of the section of non-transactional code a$-\omega_i$

We compute an upper-bound on the WCRT of a$-\omega_i$ by tuning the technique described by Spuri (1996) for the WCRT of a task. In that contribution, a single core platform is assumed and tasks are scheduled by following the fully-preemptive EDF scheduler. The model of computation assumed in this manuscript requires the following three adaptations:

1. a computation of the WCET of a task executing a transaction,
2. an extension of the concept of "deadline-d busy period" in order to compute an upper bound on the WCRT of such a task,
3. an adaptation of the blocking term associated to the tasks with a lower priority than the one under analysis.

1st adaptation: WCET of a task executing a transaction. The technique presented by Spuri (1996) requires the WCET of all tasks to be known beforehand. To this end, we use an approximation of the actual WCET value of each task τ_i, executing a transaction, by including the overhead associated to the aborts of its transaction, which is given by the WCRT of ω_i. Hence, the WCET of task τ_i is approximated by Equation 7.6.

$$C_i \overset{\text{def}}{=} C_{\text{a}-\omega_i} + R_{\omega_i} + C_{\text{p}-\omega_i} \tag{7.6}$$

2nd adaptation: Extension of the deadline-d busy period. In order to determine an upper bound on the WCRT of a$-\omega_i$, the concept of "deadline-d busy period", as introduced by George et al. (1996), is relevant until the completion time of this section of non-transactional code. To this end purpose, the length of the deadline-d busy period is adapted from Spuri (1996) as shown in Equation 7.7.

$$\begin{cases} L^{(0)}_{\text{a}-\omega_i}(a) & = 0 \\ L^{(q+1)}_{\text{a}-\omega_i}(a) & = \sum\limits_{\tau_j \in \mathscr{D}_i} \min\left\{ \left\lceil \frac{L^{(q)}_{\text{a}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a+D_i-D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left\lfloor \frac{a}{T_i} \right\rfloor \cdot C_i + C_{\text{a}-\omega_i} \end{cases} \tag{7.7}$$

In Equation 7.7, variable a denotes the release time of the job under analysis, $\mathscr{D}_i \overset{\text{def}}{=} \{\tau_j \mid D_j < a + D_i; j \neq i; \sigma(\tau_i) = \sigma(\tau_j)\}$ and the deadline-d busy period, denoted as $L_{\text{a}-\omega_i}$, is computed by using a recursive algorithm, which stops as soon as $L^{(q+1)}_{\text{a}-\omega_i}(a) = L^{(q)}_{\text{a}-\omega_i}(a)$ for some integer $q \geq 0$ or when $L^{(q+1)}_{\text{a}-\omega_i}(a)$ exceeds $a + D_i$. In this latter case, the system is not schedulable. Note that the longest busy period, denoted as $L_{\text{a}-\omega_i}$, occurs when the last job of τ_i in the window of interest is released at instant a_m such that $a_m = \text{argmax}(L_{\text{a}-\omega_i}(a))$, see Spuri (1996) for details.

3rd adaptation: blocking term associated to lower priority tasks. If a job is released when the transaction belonging to a lower priority job is in progress, then the newly released job is blocked until the transaction in progress commits as preemption is disabled in this time window. Upon the commit, the newly released job can preempt any lower priority job as the classical EDF scheduling rules apply again. In this case, the WCRT of the section of non-transactional code a$-\omega_i$ must consider a possible blocking occurring at the released time of the job. Specifically, in the worst-case, this blocking time corresponds to the longest response time of a transaction executed by a job with a lower priority than τ_i, and assigned to the same core. This blocking is formalised in Equation 7.8 below.

$$B_i \overset{\text{def}}{=} \max\left\{R_{\omega_j} \mid D_i < D_j \wedge \sigma(\tau_i) = \sigma(\tau_j)\right\} \tag{7.8}$$

From Equation 7.8, it follows that an upper-bound on the WCRT of section of non-transactional code a$-\omega_i$, denoted as $R_{\text{a}-\omega_i}$, is given by Equation 7.9.

$$R_{\text{a}-\omega_i} \overset{\text{def}}{=} B_i + \max\{C_{\text{a}-\omega_i}, L_{\text{a}-\omega_i} - a_m\} \tag{7.9}$$

The intuition leading to quation 7.9 is related to the fact that an upper-bound on the WCRT of a$-\omega_i$ is defined by the longest delay incurred by the last job that completes in the busy-period starting at time instant a_m.

7.2.2.2 WCRT of the section of non-transactional code $\text{p}-\omega_i$

When a transaction is in progress, any job with a higher priority than the job executing the transaction in progress is blocked as preemption is disabled under the NPUC policy. Only upon commit, preemption is enabled again. As such, the section of non-transactional code $\text{p}-\omega_i$ may suffer interference from any concurrent job, say τ_j, with the following characteristics: (i) Job τ_j has an earlier deadline than τ_i and (ii) either this job is released while ω_i was in progress; or this job is released prior to the completion time of τ_i. Hence, an upper-bound on the WCRT of $\text{p}-\omega_i$ is obtained by maximizing the interference this section of non-transactional code may suffer. Specifically, this maximum is reached when ω_i is released at the earliest possible time instant of its executing job. This scenario allows us to accommodate the maximum number of concurrent jobs with an earlier deadline between the transaction release time and the deadline of the job. It is worth noticing that any other scenario where ω_i is released later would necessarily lead to a smaller number of pending jobs with a higher priority than τ_i, to be executed before its deadline. Therefore, the largest deadline that can be associated to transaction ω_i in order to avoid a deadline miss of τ_i is defined by Equation 7.10.

$$D_{\omega_i} \overset{\text{def}}{=} D_i - C_{\text{a}-\omega_i} \tag{7.10}$$

An upper-bound on the WCRT of $\text{p}-\omega_i$ can be computed in an iterative manner by using a fixed-point algorithm (see Equation 7.11) wherein $\mathscr{D}_{\omega_i} \overset{\text{def}}{=} \{\tau_j \mid D_j < D_{\omega_i}; \sigma(\tau_i) = \sigma(\tau_j)\}$.

$$\begin{cases} R^{(0)}_{\text{p}-\omega_i} & = C_{\text{p}-\omega_i} \\ R^{(q+1)}_{\text{p}-\omega_i} & = \sum\limits_{\tau_j \in \mathscr{D}_{\omega_i}} \min\left\{ \left\lceil \frac{R^{(q)}_{\text{p}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j + C_{\text{p}-\omega_i} \end{cases} \tag{7.11}$$

This fixed point algorithm stops as soon as $R^{(q+1)}_{\text{p}-\omega_i} = R^{(q)}_{\text{p}-\omega_i}$ for some integer $q \geq 0$ or when $R^{(q+1)}_{\text{p}-\omega_i}(a)$ exceeds D_{ω_i}.

7.2.2.3 WCRT of task τ_i

At this stage, we have everything required to derive an upper-bound R_i on the WCRT of task τ_i (see the following theorem).

Theorem 1. *The WCRT of task τ_i is obtained by combining the WCRTs of the three sections that compose τ_i, as defined in Equation 7.12*

$$R_i = R_{\text{a}-\omega_i} + R_{\omega_i} + R_{\text{p}-\omega_i} \tag{7.12}$$

Proof. This theorem follows directly from Equation 7.9 (which bounds the execution of $\text{a}-\omega_i$); Equation 7.2 or Equation 7.5 (which bounds the execution of ω_i); and finally Equation 7.11 (which bounds the execution of $\text{p}-\omega_i$). □

Note: An upper-bound on the WCRT of a task τ_i that does not execute a transaction is a special case of the previous theorem where $R_{\omega_i} = R_{\text{p}-\omega_i} = 0$ and $C_{\text{a}-\omega_i} = C_i$.

7.3 WCRT analysis for SRPTM

In NPDA and NPUC, preemptions are disabled during the execution of the transaction in order to avoid a situation where a transaction can be aborted by a later released transaction. The approach presented in this section (SRPTM) achieves the same goal, but does not disable preemption during the progress of a transaction. This feature allows us to improve both the schedulability as more systems that originally were not schedulable would become schedulable, and on the other hand the responsiveness of the system as every task not executing a transaction will be scheduled as soon as it is possible to do so. To this end, we apply the four rules stated in Chapter 6, Section 6.3.2, and replicated below, to tune the P-EDF policy.

Rule 1 (Core ceiling)**.** We associate to each core π_k a non-negative value Λ_k, referred to as the core ceiling, and given by the preemption level of the transaction in progress on that core, i.e $\Lambda_k \leftarrow \lambda_{\omega_i}$. When no transaction is in progress on core π_k, then the core ceiling is set to zero, i.e. $\Lambda_k = 0$.

Rule 2 (Core ceiling reset)**.** Upon a successful commit of a transaction, the core ceiling is set back to zero (i.e. $\Lambda_k \leftarrow 0$).

Rule 3 (Unique transaction in progress per core)**.** At any time instant on each core π_k, we allow only one transaction to be in progress and all arriving jobs executing a transaction are blocked until the transaction in progress has committed.

Rule 4 (Priority inversion prevention)**.** Assume transaction ω_p belonging to job τ_p is in progress on core π_k. A newly released job, say τ_a, can preempt τ_p only if the following three conditions hold true: (C1) τ_a has a higher priority (i.e. $d_a < d_p$); (C2) τ_a is not executing any transaction (i.e. $\lambda_{\omega_a} = 0$) and (C3) the preemption level of τ_a is higher than the core ceiling (i.e. $\lambda_a > \Lambda_k$). In the case where job τ_a is executing a transaction (i.e. $\lambda_{\omega_a} > 0$), then the following three actions are taken: (A1) τ_a is blocked; (A2) the core ceiling is risen to the preemption level of τ_a and (A3) the core π_k resumes the execution of ω_p.

These rules do not eliminate the interference but, they reduce it during the execution of a transaction. In a similar manner as for NPUC paradigm, deriving an upper-bound on the WCRT of a task under SRPTM consists of computing the WCRT of the individual parts of the task taken separately.

7.3.1 WCRT of transaction ω_i

SRPTM shares two fundamental features with NPUC: (1) every transaction suffers the delay associated to its direct contenders with an earlier release time before it can commit; and (2) no more than one transaction can be in progress on each core. From these two features, it remains true that every transaction requires at most two attempts to commit, once it is legally possible to do so. We denote an upper-bound on the WCRT of these last two attempts by $R^*_{\omega_i}$ for transaction ω_i.

Since SRPTM is based on the P-EDF scheduler, the computation of $R^*_{\omega_i}$ depends on the so-called *intra-core* interference only, i.e., the interference associated to the higher priority jobs not executing transactions and assigned to the same core as τ_i. The preemption level of such a task, say τ_j, is thus greater than the core ceiling (i.e., $\lambda_j > \Lambda_k \geq \lambda_{\omega_i}$).

Lemma 7.4. *An upper-bound $R^*_{\omega_i}$ on the WCRT of the last two attempts of ω_i is computed recursively by using Equation 7.13 where $\mathscr{D}^*_{\omega_i} \overset{\text{def}}{=} \{\tau_j \mid D_j < D_{\omega_i};\ \sigma(\tau_i) = \sigma(\tau_j);\ \lambda_j > \lambda_{\omega_i};\ \lambda_{\omega_j} = 0\}$.*

$$\begin{cases} R^{*(0)}_{\omega_i} & = 2 \cdot C_{\omega_i} \\ R^{*(q+1)}_{\omega_i} & = \sum\limits_{\tau_j \in \mathscr{D}^*_{\omega_i}} \min\left\{ \left\lceil \frac{R^{*(q)}_{\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j + 2 \cdot C_{\omega_i} \end{cases} \tag{7.13}$$

The calculation of $R^*_{\omega_i}$ stops as soon as the result converges, i.e., $R^{*(q+1)}_{\omega_i} = R^{*(q)}_{\omega_i}$ for some positive integer q or when $R^*_{\omega_i}$ exceeds D_{ω_i}.

Proof. $R^*_{\omega_i}$ is determined by maximising the possible intra-core interference. As such, it is given by the execution time taken by the last two attempts of the transaction, augmented by the execution time required by the concurrent jobs that are able to preempt τ_i during the time window corresponding to these two attempts. □

Let us assume that τ_i is assigned to core π_k and ω_i belongs to contention group Ω_g. Then, we have everything necessary to compute a tight upper bound on the WCRT of ω_i. For each core π_ℓ in Π_g, but π_k, the transaction that presents the longest response time to execute the last two attempts is selected and sum-up to produce the worst-case delay on ω_i. As such, the *inter-core interference*, denoted as I_{Ω_g,π_k}, can be upper-bounded and formalized as in Equation 7.14.

$$I_{\Omega_g,\pi_k} = \sum_{\pi_\ell \in \Pi_g \setminus \pi_k} \max\left\{ R^*_{\omega_j} \mid \omega_j \in \Omega_g \wedge \sigma(\tau_j) = \pi_\ell \right\} \tag{7.14}$$

Once an upper-bound on the intra-core interference and an upper-bound on the inter-core interference are computed, an upper-bound on the WCRT of ω_i can be determined by combining these two expressions as follows.

$$R_{\omega_i} \overset{\text{def}}{=} I_{\Omega_g,\pi_k} + R^*_{\omega_i} \tag{7.15}$$

7.3.2 WCRT of task τ_i

The blocking and interference factors when following the SRPTM policy for a set of tasks differ depending on whether the tasks execute transactions or not. We recall that every newly released job executing a transaction automatically blocked, irrespective of its priority, when a transaction is already in progress on the target core. Thus, we compute an upper-bound on WCRT of τ_i in two distinct approaches.

7.3.2.1 WCRT of τ_i executing a transaction

We consider the three sections of τ_i separately, i.e. the section of non-transactional code $\text{a}-\omega_i$ before the transaction is released; the section of non-transactional code corresponding to the transaction itself ω_i; and finally the section of non-transactional code $\text{p}-\omega_i$ after the transaction has committed.

Blocking term. SRPTM does not allow more than one transaction in progress per core. When a transaction is in progress, the newly released job belonging to task τ_i is *directly blocked* until the transaction commits. Upon the commit, preemption is possible again and ready jobs are scheduled by following a classical EDF scheduler. This implies that no job can incur an indirect blocking. Any job executing a transaction can be directly blocked at most once. Hence, the maximum blocking time, denoted as DB_i, is defined by longest response time of a transaction from all the transactions assigned to the same core. This is derived from the subset of tasks with a lower preemption level, as formalised in Equation 7.16.

$$\text{DB}_i \overset{\text{def}}{=} \max\left\{R_{\omega_j} \mid \lambda_j < \lambda_i \wedge \sigma(\tau_i) = \sigma(\tau_j)\right\} \tag{7.16}$$

WCRT of the section of non-transactional code $\text{a}-\omega_i$**.** Under P-EDF, any job can suffer interference only from other jobs released on the same core, as no migration among cores is permitted at runtime. An upper-bound on the WCRT of $\text{a}-\omega_i$ can once again be determined by adapting the technique presented by Spuri (1996). For the purpose of this analysis, the WCET of τ_i is approximated by using Equation 7.17.

$$C_i = C_{\text{a}-\omega_i} + R_{\omega_i} + C_{\text{p}-\omega_i} \tag{7.17}$$

In the same manner as for NPUC (see Section 7.2.2), the extension of the deadline-d busy period is determined by Equation 7.18.

$$\begin{cases} L^{(0)}_{\text{a}-\omega_i}(a) & = 0 \\ L^{(q+1)}_{\text{a}-\omega_i}(a) & = \sum\limits_{\tau_j \in \mathscr{D}_i} \min\left\{\left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a+D_i-D_j}{T_j} \right\rfloor\right\} \cdot C_j + \left\lfloor \frac{a}{T_i} \right\rfloor \cdot C_i + C_{\text{a}-\omega_i} \end{cases} \tag{7.18}$$

The computation of $L_{\text{a}-\omega_i}(a)$ is performed by using a fixed-point algorithm that converges if $L^{(q+1)}_{\text{a}-\omega_i}(a) = L^{(q)}_{\text{a}-\omega_i}(a)$ for some non-negative integer q. Otherwise, if $L^{(q+1)}_{\text{a}-\omega_i}(a) > a + D_i$ for a given q, then the system is "not schedulable". In the former case, the longest deadline-d busy period $L_{\text{a}-\omega_i}$ occurs when the last job of τ_i is released at the time instant a_m such that $a_m = \text{argmax}(L_{\text{a}-\omega_i}(a))$, see Spuri (1996) for further details.

Finally, an upper-bound on the WCRT of $\text{a}-\omega_i$ is given by an upper-bound on the WCRT of this non-transactional section, augmented by the delay produced by the busy period. The possible blocking time that a job can suffer when it is released is formalized as in Equation 7.19.

$$R_{\text{a}-\omega_i} = \text{DB}_i + \max\left\{C_{\text{a}-\omega_i}, L_{\text{a}-\omega_i} - a_m\right\} \tag{7.19}$$

WCRT of the section of non-transactional code $\mathrm{p}-\omega_i$**.** This section of the task τ_i can be preempted by: (1) the jobs executing transactions that were released while the transaction was in progress, and (2) the jobs with a higher priority that are released in its window of execution. In order to maximise the interference, we consider the same time window as defined in Equation 7.10 for the jobs executing a transaction. The *"critical time instant"* at which this section of the task can start executing occurs when the blocking time and the interference experienced are at their maximum. The length of the interval which starts at a critical instant, denoted as $D_{\mathrm{p}-\omega_i}$, is formally given as in Equation 7.20.

$$D_{\mathrm{p}-\omega_i} \overset{\text{def}}{=} D_i - (R_{\mathrm{a}-\omega_i} + R_{\omega_i}) \tag{7.20}$$

This interval defines the longest possible response time of $\mathrm{p}-\omega_i$ while assuming the lowest slack of τ_i upon the commit of ω_i. Thus, an upper bound on the WCRT of $\mathrm{p}-\omega_i$ can be defined by the execution demand of this section augmented by the execution demands of the concurrent jobs, with an earlier deadline, that are released inside both mentioned time windows as formalized in Equation 7.21, where:
$\mathscr{D}^1_{\mathrm{p}-\omega_i} \overset{\text{def}}{=} \{\tau_j \mid D_j < D_{\omega_i}; \sigma(\tau_i) = \sigma(\tau_j); \lambda_{\omega_j} \neq 0\}$ and $\mathscr{D}^2_{\mathrm{p}-\omega_i} \overset{\text{def}}{=} \{\tau_j \mid D_j < D_{\mathrm{p}-\omega_i}; \sigma(\tau_i) = \sigma(\tau_j)\}$.

$$\begin{cases} R^{(0)}_{\mathrm{p}-\omega_i} = C_{\mathrm{p}-\omega_i} \\ \\ R^{(q+1)}_{\mathrm{p}-\omega_i} = \sum\limits_{\tau_j \in \mathscr{D}^1_{\mathrm{p}-\omega_i}} \min\left\{ \left\lceil \frac{R^{(q)}_{\mathrm{p}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j \\ \qquad + \sum\limits_{\tau_j \in \mathscr{D}^2_{\mathrm{p}-\omega_i}} \min\left\{ \left\lceil \frac{R^{(q)}_{\mathrm{p}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\mathrm{p}-\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j + C_{\mathrm{p}-\omega_i} \end{cases} \tag{7.21}$$

The iterative computation stops when the result converges to a value such that $R^{(q+1)}_{\mathrm{p}-\omega_i} = R^{(q)}_{\mathrm{p}-\omega_i}$, otherwise if for a given q, we have $R^{(q+1)}_{\mathrm{p}-\omega_i} > D_i$, then the system is "not schedulable".

Theorem 2 (WCRT of a task τ_i executing a transaction)**.** *An upper-bound on the WCRT of task τ_i is obtained by combining the direct blocking and the WCRTs of the three sections that compose τ_i, as defined in Equation 7.22.*

$$R_i = R_{\mathrm{a}-\omega_i} + R_{\omega_i} + R_{\mathrm{p}-\omega_i} \tag{7.22}$$

Proof. This theorem follows directly from Equation 7.19, Equation 7.15 and Equation 7.21. □

7.3.2.2 WCRT of τ_i not executing a transaction

In addition to the classical interference that every task can suffer from the jobs executing on the same core, each task not executing a transaction can experience either direct or indirect blocking.

Direct blocking. It occurs when the job with the earliest deadline is released, but has a preemption level which is not greater than the current core ceiling. We recall that once the transaction commits, the classical EDF scheduler applies. At this moment, the job with the earliest absolute deadline is selected for execution.

Indirect blocking. It occurs when the job with the earliest deadline is released while a transaction is in progress and the job is able to execute because its preemption level is greater than the core ceiling at that specific time instant. In the meantime, if another job executing a transaction is released and fulfills the deadline and preemption level requirements to be scheduled, then the core ceiling is raised to the transaction preemption level of this job and this operation forces the first transaction that was already in progress to complete its execution. As such, this process helps the transaction in progress to commit as soon as it is legally possible to do so.

Blocking term. The longest direct blocking term that task τ_i can experience, denoted as DB_i, is given by an upper-bound on the WCRT of all the transactions which: (1) have a higher preemption level than λ_i, (2) belong to the subset of tasks with a lower preemption level than λ_i, and finally (3) are assigned to the same core as τ_i. This is formally expressed in Equation 7.23.

$$\mathrm{DB}_i = \max\left\{R_{\omega_j} \mid \lambda_{\omega_j} > \lambda_i \wedge \lambda_j < \lambda_i \wedge \sigma(\tau_j) = \sigma(\tau_i)\right\} \tag{7.23}$$

In contrast, an upper-bound on the longest indirect blocking term is given by an upper-bound on the WCRT of all the transactions with a lower preemption level than τ_i. Note that these transactions are assigned to the same core as τ_i. If a job of a task, say τ_g, executing a transaction is able to preempt τ_i, then task τ_i will be indirectly blocked as formally expressed in Equation 7.24.

$$\mathrm{IB}_i = \max\{R_{\omega_j} \mid \lambda_{\omega_j} < \lambda_i \wedge \sigma(\tau_j) = \sigma(\tau_i) \wedge \exists \tau_g : \lambda_g > \lambda_i \wedge \lambda_{\omega_g} > 0\} \tag{7.24}$$

The intuition behind Equation 7.24 is related to the fact that when a job of τ_i preempts the job with the current transaction in progress, τ_i may then be preempted by another job τ_g also executing a transaction, but with an earlier deadline and a preemption level greater than the core ceiling.

Under SRPTM, if a job of τ_i (not executing a transaction) is released while a transaction is in progress, then τ_i may be blocked at most once. Consequently, direct and indirect blocking terms are mutually exclusive for any job not executing a transaction. In this case, the longest blocking time is given by the maximum value between direct and indirect blocking, as formally expressed in Equation 7.25.

$$B_i = \max\{\mathrm{DB}_i, \mathrm{IB}_i\} \tag{7.25}$$

WCRT of task τ_i not executing a transaction. Assuming a task not executing a transaction, its jobs are scheduled by following the EDF scheduler. As such, until each released job completes its execution, it may suffer interference from any released job with an earlier absolute deadline. The computation of an upper-bound on the WCRT of task τ_i in this case can thus be achieved by the method described by Spuri (1996) without any adaptations. The iterative equation is replicated in Equation 7.26, where $\mathscr{D}_i$ is defined as in Equation 7.7.

$$\begin{cases} L_i^{(0)}(a) = 0 \\ L_i^{(q+1)}(a) = \sum\limits_{\tau_j \in \mathscr{D}_i} \min\left\{ \left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left\lfloor 1 + \frac{a}{T_i} \right\rfloor \cdot C_i \end{cases} \tag{7.26}$$

The iterative computation stops when the result converges to a value such that $L_i^{(q+1)}(a) = L_i^{(q)}(a)$, otherwise if the deadline is exceeded, the system is not schedulable. Assuming that a job of τ_i is released at time instant $a_m = \text{argmax}(L_i(a))$, the length of the deadline-d busy period that produces the longest delay is denoted as L_i, and the upper-bound on the WCRT of τ_i if no blocking could occur is given by $L_i - a_m$. This result can then be combined with the longest blocking time B_i (see Equation 7.25) to determine an upper-bound on the overall WCRT of a task not executing a transaction.

Theorem 3. *An upper-bound on the overall WCRT of a task τ_i not executing a transaction is formalized as in Equation 7.27.*

$$R_i = B_i + \max\{C_i, L_i - a_m\} \tag{7.27}$$

Proof. This theorem follows directly from the combination of Equation 7.25 and Equation 7.26. □

7.4 Summary

In this chapter, we addressed the response time analysis of hard real-time tasks, which share STM data under partitioned scheduling strategies. A framework wherein an upper-bound on the worst-case response time of each task has been discussed while assuming three scheduling approaches – namely NPDA, NPUC and SRPTM. Although NPDA was a promising approach for soft real-time tasks upon multi-core platforms, we showed that its associated analysis is intractable in the context of hard real-time tasks. Assuming NPUC, we provided a tight analysis and a linear-time analysis, which reduces the computational complexity of the proposed analysis of the former. Finally, we showed that SRPTM improves the responsiveness of the tasks as compared to the other two approaches, and allows for a computationally tractable analysis. In Chapter 8 we report on the evaluation results of these synchronisation policies, both from a qualitative and quantitative viewpoints.

Chapter 8

Evaluation

We proposed the FIFO-CRT contention manager for obstruction-free STM, in order (*i*) to solve the conflicts based on the release times of transactions (see Chapter 5); and (*ii*) to improve the predictability of multicore systems when it is used together with fully-partitioned scheduling approaches such as NPDA, NPUC and SRPTM (see Chapter 6). This chapter presents the evaluation of FIFO-CRT when it is associated to each scheduling policy.

We recall that solving conflicts based on the release times of the transactions (as performed in FIFO-CRT) without any scheduling support can produce undesirable anomalies (see Chapter 6, Section 6.1). A transaction which aborts another transaction with an earlier release time is an example of such an anomaly. To circumvent this problem, we devised NPDA, NPUC and SRPTM. This chapter evaluates the gain of each of these proposed techniques over FIFO-CRT when it is associated to P-EDF. The evaluation is performed both quantitatively and qualitatively. Section 8.1 presents the simulation setup and the quantitative evaluation. Furthermore, as STM is the best synchronisation scheme candidate for scenarios in which the contention is low and every transaction presents both a short execution time and a low probability of being preempted (Maldonado et al., 2010), we also perform a comparison against a lock-based sychronisation mechanism that suits this type of scenarios. On the one hand, synchronisation mechanisms that suspend to wait for locks (such as FMLP for long critical sections and OMLP) adapt easily to long critical sections as the host jobs would yield temporarily the core to lower priority jobs not to waste the processor capacity. This statement holds true as long as the waiting time is long enough to compensate for the context switch overheads. Indeed, the waiting time increases in a monotonic manner with the size of the critical sections and the size of the contention (expressed as the number of critical sections simultaneously competing for a lock). On the other hand, synchronisation that busy-wait non-preemptively for locks (such as FMLP for short critical sections) are the best candidates for applications in which the waiting times are generally short and the contention is low. Here, the context switching overheads may worsen the system performance (Brandenburg et al., 2008). STM also provides the best performance for these scenarios. Therefore, we choose to compare the simulation results against FMLP[1]. To this end, the task sets are first profiled so that the length of the atomic sections is not restricted. This results in an increase in the contention size, which in turn equally stressed both the STM and the locking approaches. Then, the length of the atomic sections is restricted to an arbitrary upper bound relative the execution time of the tasks. This results in a decrease in the contention size. Section 8.2 presents a qualitative evaluation of STM (i.e., the proposed contention manager and the associated scheduling approaches) against the locking mechanisms on various aspects in which the synchronisation policy can condition the performance of the system. This comparison is conducted while assuming a multicore architecture without cache coherence and a switched network interconnect between the cores.

8.1 Quantitative evaluation

This section reports on the quantitative results of the simulations conducted as a proof of concept of the contention manager FIFO-CRT proposed in Chapter 5 and the scheduling approaches proposed in Chapter 6.

8.1.1 Simulation set up

We developed a simulation environment to test the proposed contention management algorithm under four different scheduling policies – (1) pure P-EDF, (2) P-EDF with NPUC, (3) P-EDF with NPDA and finally (4) P-EDF with SRPTM – on multicore platforms containing from 2 up to 64 cores. Additionally, we implemented the FMLP rules over P-EDF, in order to compare our approaches with a state-of-the-art lock-based synchronisation mechanism.

We conducted our simulations by defining 12 task set profiles that share some common characteristics. The task sets are synchronous (i.e., all the tasks release a job at the time instant, say the first job at $t = 0$) and have implicit deadlines (i.e., for a task set with n tasks, $D_i = T_i,\ \forall i \in \{1,2,\ldots,n\}$). Each task set targets a 75% usage of the system capacity, when the execution time overhead related to atomic sections (i.e., aborts and retries under STM, and busy-waiting under lock-based mechanisms) is neglected. The individual utilisation of each task τ_i (neglecting possible atomic section overheads) is randomly selected from the interval $U_i \in (0, 0.3]$, with a uniform distribution. As suggested by Nelis et al. (2013), the period of each task τ_i is determined as the product of three non-negative integer values, i.e., $T_i = a_i \times b_i \times c_i$, where a_i is randomly chosen from $\{2,4,8,16\}$, b_i from $\{3,6,9,12\}$ and c_i from $\{5,10,15\}$, in order to keep the hyperperiod (i.e., the least common multiple of all task periods) reasonably small. The WCET of task τ_i is given by $C_i = \max(U_i \times T_i, 1)$. If the task has an atomic section, then the minimum WCET is 2, as we require one time unit to flag the beginning of the atomic section and another time unit to flag the end. We randomly select approximately 75% of the tasks to have one atomic section, and approximately half of the atomic sections modify some element in its data sets. The execution

[1]The FMLP definition states that resources are classified as long or short by the application designer (Block et al., 2007).

time of each atomic section is such that $C_{\omega_i} \in \{2,\ldots,C_i\}$ with a uniform distribution. The size of an atomic section data set ranges from 1 to 3, with a discrete uniform distribution.

For each profile, we calculated the number of transactional objects/resources required so that the expected value of tasks accessing each object is $E[|o|] = 3$. Considering that the average number of tasks accessing an object is given by

$$\overline{|o|} = \frac{\sum_i |DS_i|}{p} \tag{8.1}$$

we can calculate the number of objects given by the expected sum of data set sizes divided by the expected number of accesses per object:

$$p = \frac{U_s \times P(\tau_i \text{ has } \omega_i)}{E[U_i]} \times \frac{E[|DS|]}{E[|o|]} \tag{8.2}$$

We observed that purely randomly generated transaction data sets ended up creating a single global contention group (or a single global group lock, as known in FMLP). In order to circumvent this effect, we artificially created contention groups by organising the objects into g subsets as follows: the first $(g-1)$ groups are of size $|\Omega| = \lfloor p/g \rfloor$ each, and the last group is of size $|\Omega_{last}| = p - \lfloor p/g \rfloor \times (g-1)$. Note that the last group may be larger than the previous ones. Each transaction joins randomly a contention group, and its data set is confined to the objects in this group. As a consequence of this, a transaction will not compete with transactions in other contention groups. We did not balanced the allocation of transactions to the groups, so some groups may contain more tasks than others. We defined two groups of profiles.

1. *First group of profiles.* This group of profiles allows for the observation of the effects of the system size (number of cores and tasks) on the scheduling of tasks, for equivalent levels of contention. In this group, we vary the number of cores in the range $m \in \{2,4,8,16,32,64\}$. The number of shared objects varies proportionally with the number of cores, in the range $p \in \{5,10,20,40,80,160\}$, so the expected number of tasks accessing each object remains constant, equal to 3. Additionally, we vary the number of contention groups proportionally with the number of cores, in the range $g \in \{1,2,4,8,16,32\}$, so that contention groups maintain the same size (in terms of number of objects) and expected number of tasks.
2. *Second group of profiles.* This group of profiles allows for the observation of the effects of the contention granularity, for equivalent size of system (cores, tasks and shared objects). In this group, we keep the number of cores and the number of shared objects constant: $m = 64$ and $p = 160$, respectively. However, the number of contention groups varies in the range $g \in \{1,2,4,8,16,32,64\}$.

8.1.2 Simulation results

For each profile we generated 50 synchronous task sets in a random manner. Each task set was tested for a time period equivalent to its hyperperiod, which is sufficient to verify the feasibility of the schedule (i.e., to check whether no deadlines will ever be missed). Every released job was executed until completion, even if it missed its deadline. Deadlines were detected at the end of the execution of a job, so deadline miss propagation was allowed under heavy utilisation. This also means that under heavy utilisation, a backlog of workload could be passed to the following hyperperiod, so the number of unreleased or unfinished jobs were also accounted for as deadline misses.

During each simulation, we recorded the number of deadline misses (including jobs that were expected to complete inside the hyperperiod and did not) for each task. We also recorded the time overhead of the atomic sections (due to aborts or busy waits) performed inside completed jobs, and the total system capacity used during the simulation time by the task set.

8.1.2.1 Varying system size

In this experiment, we focused on comparing how the different scheduling approaches are capable of finding a valid schedule for the task sets in the first group of profiles. Figure 8.1 illustrates the feasibility rates for all the approaches as a function of the number of cores and the number of groups. The trend in this figure shows that P-EDF provides the highest success ratio in comparison to the approaches that disable preemptions during the atomic sections. When scrutinizing each individual task set, we observed that there was a considerable amount of deadline misses from some tasks because their periods were smaller than or closer to the execution time of the transactions of other tasks assigned to the same core. For task sets in which some task parameters have these characteristics, non-preemptive approaches were not capable of finding a valid schedule, while P-EDF could. SRPTM could provide a valid schedule for a partition of these task sets with no transactions. This explains why it dominates the non-preemptive approaches. As the system size increases, so does the probability of occurring longer sequences of serialisation of transactions. Thus, P-EDF feasibility ratio also drops to zero when transactions may have to wait longer than the available time to commit.

Analysis of the behaviour of the same task sets with smaller atomic sections. We assigned a 95% probability to each atomic section with an execution time longer than 20 time units to having its execution time reduced to 20 time units, and a 5% probability to each atomic section to keeping its execution time unchanged. We chose 20 because the smallest period in each task set is $2 \times 3 \times 5 = 30$ time units, and the maximum execution time for such a task is $0.3 \times 30 = 9$ time units. Hence, there is a slack of 21 time units dedicated to waiting for a concurrent job to execute one transaction attempt of 20 time units. As expected, the feasibility ratios increased (see Figure 8.2). We observed that approaches that do not allow preemption points before the atomic section completes (e.g., NPUC and FMLP) have more difficulties to finding valid schedules. Also, we observed that SRPTM and NPDA have similar feasibility ratios and behaved better than P-EDF for systems with few tasks.

Analysis of the overall number of deadline misses. We assumed that there exists a valid schedule for a given task set and we focused on evaluating how far are the different scheduling approaches from this valid schedule. Figure 8.3 illustrates the total amount of deadlines misses (including not released or not completed jobs by the hyperperiod) as a function of the number of

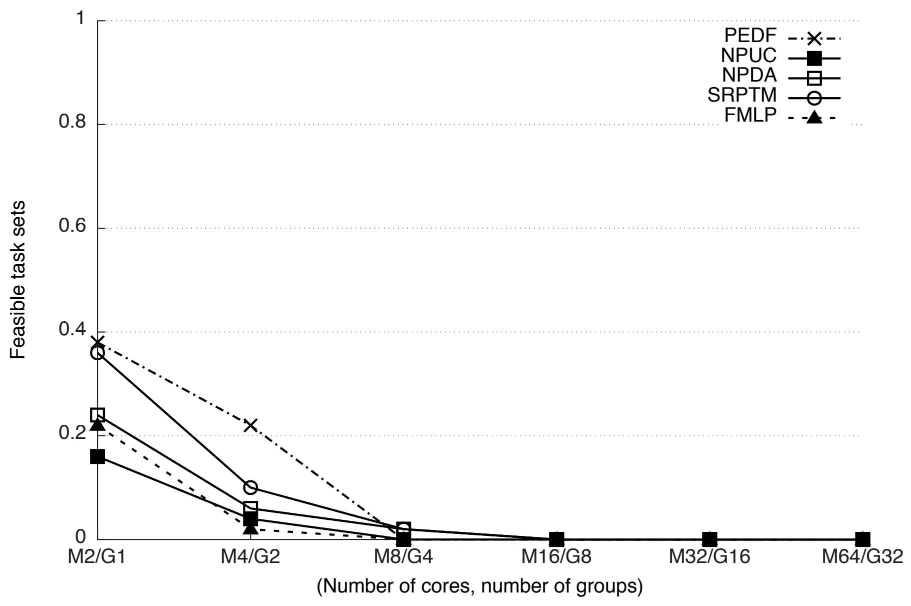


Figure 8.1: Feasibility rates: varying number of cores.

cores and the number of groups. We assumed 50 task sets with the same number of cores and the same number of contention groups. Figure 8.3a illustrates the total amount of deadline misses after 50 simulations. This figure shows that FMLP provides the worst results as we count over one million deadline misses, whereas SRPTM appears to provide the best results with less than 363.000 deadline misses. Our explanation of these trends is that a job under P-EDF can preempt any other job, irrespective of whether it executes a transaction or not, whereas this is not the case under SRPTM. In this case, a job executing a transaction cannot preempt another job executing a transaction.

Analysis of the total number of aborts. The maximum number of aborts that a transaction may suffer stresses the effort to meet the deadline of its host job. This holds true for jobs that are executing an atomic section as well as for jobs that are waiting for the completion of an atomic section to proceed with their execution. Jobs in the ready queue of a core and jobs with contending transactions in other cores are examples of such jobs. Figure 8.4 illustrates the total number of deadline misses for each task set after 50 simulations. From this figure, it follows that NPUC and NPDA present in average a low maximum number of aborts. These trends can be explained as follows. Under NPUC, transactions are not exposed to the delays caused by the preempted contenders in other cores, so it tends to present a low number of aborts. In constrast, these delays have a negative impact under SRPTM, which increase the number of aborts. This is especially true for the waiting transactions in the worst case scenario. Under P-EDF, the high number of aborts can be explained by the possibility of aborting older preempted transactions in order to avoid eventual deadlocks.

Analysis of the average performance of the atomic sections. For all the released jobs that completed their execution during our simulation time (i.e., in one hyperperiod), we measured the

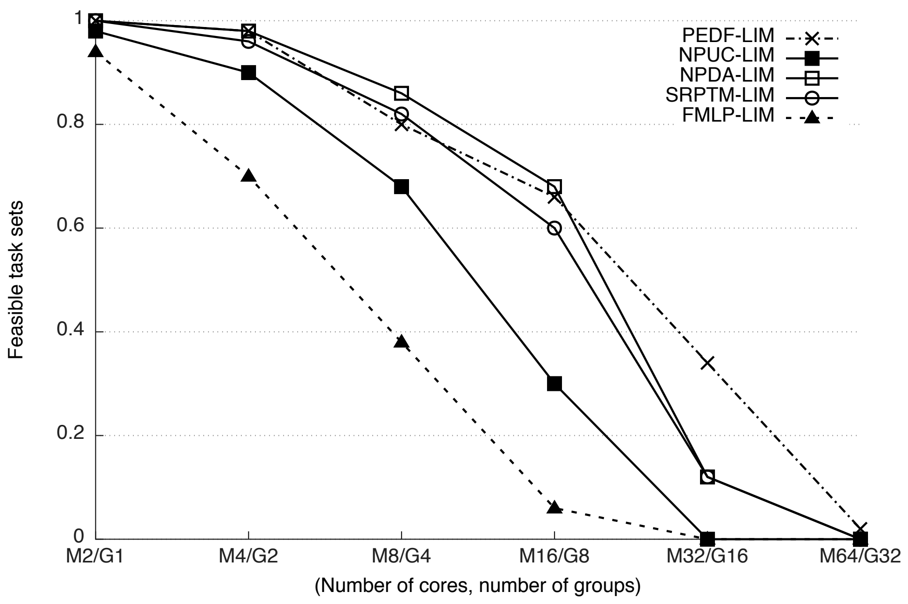


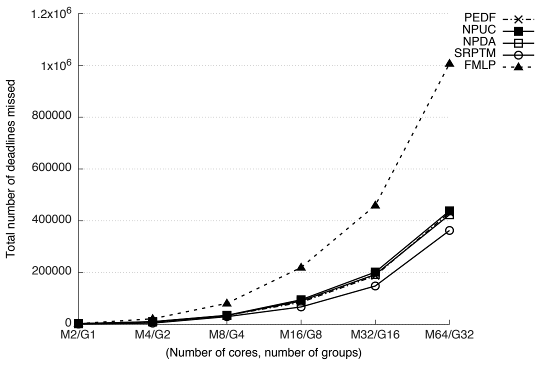
Figure 8.2: Feasibility rates: varying number of cores with smaller atomic sections.

execution time overhead related to the aborts under STM, and to busy waiting under FMLP. Figure 8.5 illustrates the extra time it takes in average to execute one job with an atomic section. More precisely, Figure 8.5a illustrates the results for long atomic sections and Figure 8.5b illustrates the results for limited length atomic sections. It follows that SRPTM presents low overheads in average in comparison to the other approaches. This trend is explained by the fact that under SRPTM, a preempted transaction on a core yield back the core to other tasks while this is not the case under other approaches: a preempted transaction is waiting for its contenders on other cores to commit before it can proceed. This reduces the number of aborts and improves the amount of workload performed per core. Unlike under P-EDF, the transaction is not killed by newer contenders when the job is preempted. The effect of this protection is visible in the gap that separates the SRPTM curve to the P-EDF curve for the attempt of the last transaction.

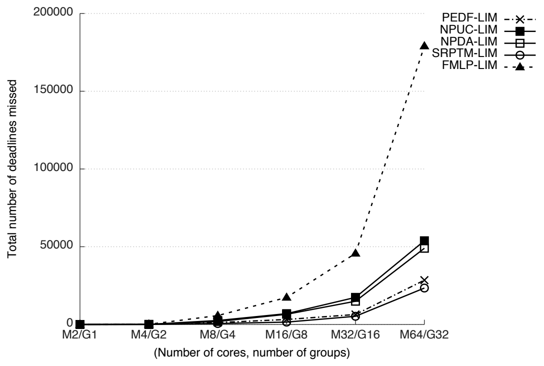
Non-preemptive approaches provide better results for systems with few tasks. This phenomenon can be explained by the fact these approaches tend to force transactions to commit as soon as it is legally possible to do so. When the systems grow, the possibility of longer sequences of transactions to serialise also grows. This is reflected in an increase of the number of aborts before commit. FMLP presents higher overheads because it does not profit from intra-group contention granularity, as this is the case for the transactions (non-intersecting data sets inside the same contention group).

8.1.2.2 Varying granularity of contention

In the second bunch of simulations, we focus on the evaluation of the contention groups granularity. To this end, we considered and tested various scheduling approaches on systems with the same size (i.e., the same number of cores, tasks, atomic sections and shared objects). We noticed

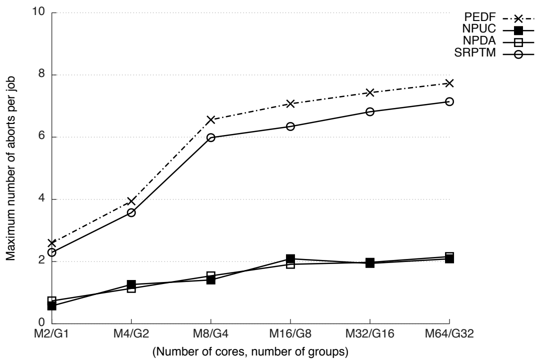


(a) Results for task sets with atomic sections with unrestricted size.

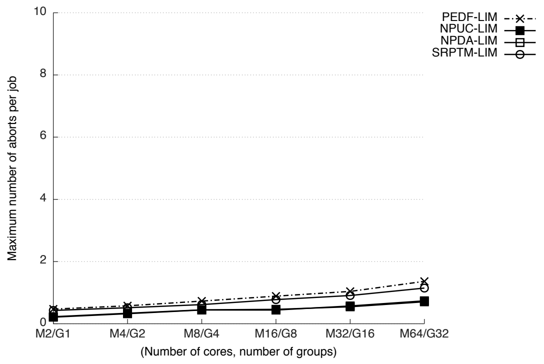


(b) Results for task sets with atomic sections with size limited to 20 time units.

Figure 8.3: Total number of deadline misses (50 simulations).

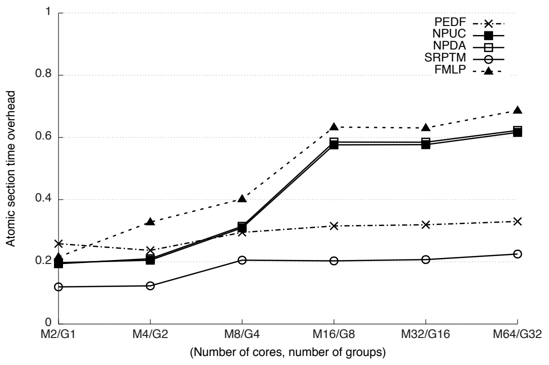


(a) Results for task sets with atomic sections with unrestricted size.

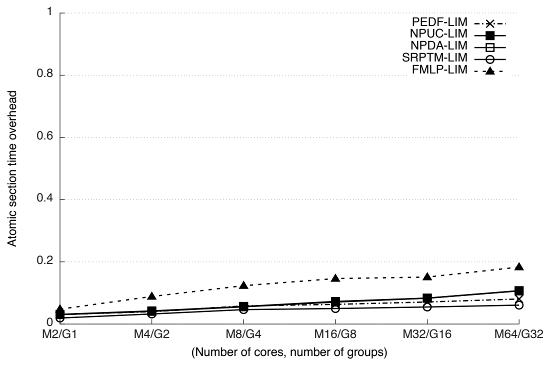


(b) Results for task sets with atomic sections with size limited to 20 time units.

Figure 8.4: Maximum number of aborts per job (average).



(a) Results for task sets with atomic sections with unrestricted size.



(b) Results for task sets with atomic sections with size limited to 20 time units.

Figure 8.5: Time overhead per atomic section (average).

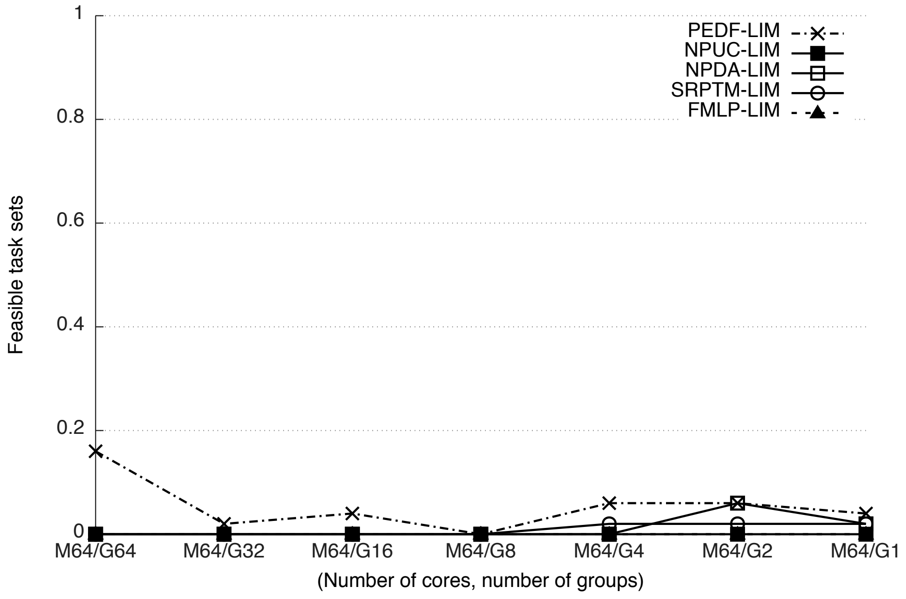


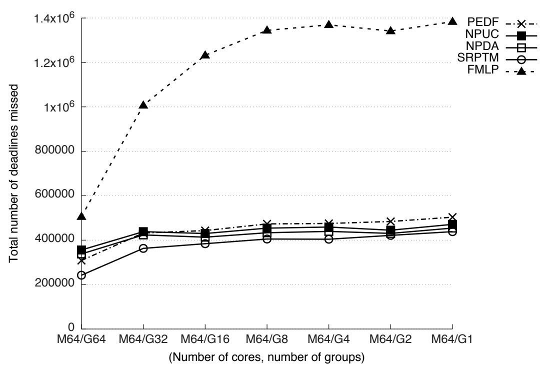
Figure 8.6: Feasibility rates: varying number of cores with smaller atomic sections.

that we could not find any valid schedule for any number of groups, when assuming a platform with 64 cores and allowing long transactions. For the same task sets but with limited transactions in terms of execution times, we could find valid schedules under P-EDF, NPDA and SRPTM, as illustrated in Figure 8.6. Again, it can be noticed that fully preemptive P-EDF yields the highest ability to schedule task sets.

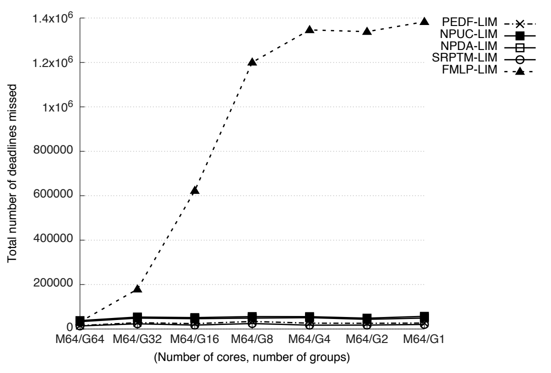
Analysis of the number of deadline misses. For a given task set, we assume that there exists a valid schedule, (i.e., one for which all the deadlines are met for all the jobs) and we evaluate how far the task set is from this valid schedule by using the different approaches. We recall that unreleased and/or unfinished jobs within one hyperperiod increase the number of deadline misses. Figure 8.7 illustrates the total number of deadline misses after 50 simulations for each profile. Figure 8.7a illustrates this number for long transactions and Figure 8.7b illustrates this number for the same task sets, but with limited transactions in terms of execution time. In both cases and for all profiles, SRPTM always yields the lowest numbers. It is also worth noticing that STM-based mechanisms present a slight increase in terms of deadline misses when the number of contention groups becomes smaller.

Analysis of the total number of aborts. Figure 8.8 illustrates the average of the maximum number of aborts per job. In this figure, where we can observe that the non-preemptive approaches suffer lesser aborts than both P-EDF and SRPTM in the worst case. The difference is more pronounced for long transactions (see Figure 8.8a). This trend is reduced when the size of the transactions is limited (see Figure 8.8b).

Although the non-preemptive approaches tend to present a better behavior when assuming the worst case scenario, SRPTM presents fewer overheads in the long run, as illustrated in Figure 8.9. In this figure, FMLP is heavily penalised when the granularity of contention is coarse and in

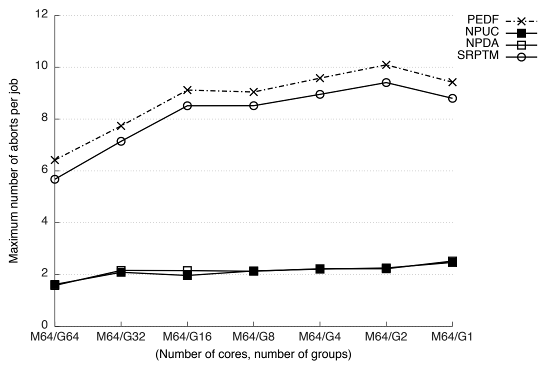


(a) Results for task sets with atomic sections with unrestricted size.

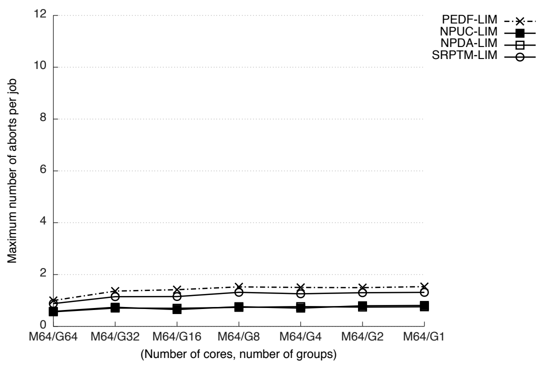


(b) Results for task sets with atomic sections with size limited to 20 time units.

Figure 8.7: Total deadlines missed (50 simulations).



(a) Results for task sets with atomic sections with unrestricted size.



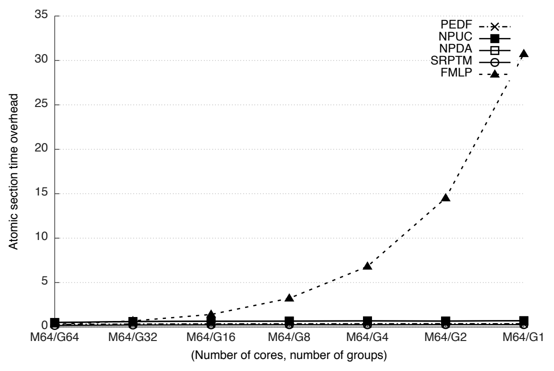
(b) Results for task sets with atomic sections with size limited to 20 time units.

Figure 8.8: Maximum number of aborts per job (average).

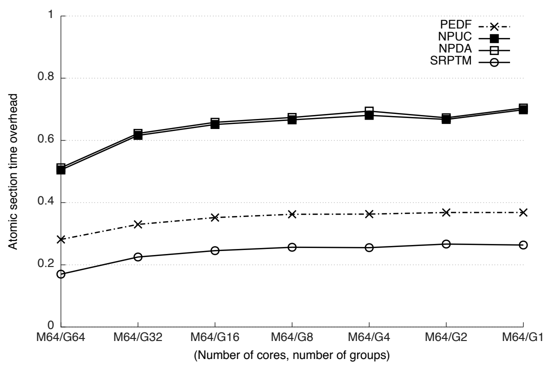
comparison to STM-based approaches, this penalty increases drastically as the number of groups decreases (see Figure 8.9a). Except for FMLP, both NPUC and NPDA use a high execution time overhead to execute their jobs executing transactions (see Figure 8.9b).

8.2 Qualitative evaluation

Section 8.1 covered the simulation of task sets with atomic sections under different combinations of partitioned real-time scheduling algorithms (namely, P-EDF, NPDA, NPUC, SRPTM) and resource sharing protocols (namely, STM, FMLP). The results from the simulations provided good quantitative comparisons on the performances of the proposed approaches. This section instead provides *a qualitative comparison* with respect to a set of characteristics (deadlock and livelock, composability, transparency, access to multiple data, priority inversion, convoying, impact on the platform and platform dependency) between the STM-based approaches devised in this work, i.e., the contention managers together with the associated scheduling policy and the lock-based approaches such as the Flexible Multiprocessor Locking Protocol (FMLP) and the more recent pro-



(a) Results for STM and lock-based approaches.



(b) Detail of the results for the STM approaches (excluding FMLP).

Figure 8.9: Execution time overhead per atomic section (average).

tocols such as the O(m) Multiprocessor Locking Protocol (OMLP) (Brandenburg and Anderson, 2013) and the Real-time Nested Locking Protocol (RNLP) (Ward and Anderson, 2012) associated to partitioned scheduling. Before going into more details, let us recall the specifics of these protocols.

- *FMLP*. In this protocol, the mutual exclusive access to a group of resources is managed by a group lock. A job that is waiting to acquire a group lock busy-waits non-preemptively (on short critical section) or suspends (on long critical section). Critical sections are executed non-preemptively. Only a long critical section can nest a short critical section; other types of nesting are not allowed.
- *OMLP*. This protocol follows the same principle as FMLP. However, a job that is waiting for a group lock always suspends, and resumes when the lock is granted. Also, nesting is not allowed.
- *RNLP*. This protocol is fine-grained and allows nesting critical sections, but with the constraint that locks must be acquired in a predetermined order. The chronological order by which the critical sections start define the order in which the locks are acquired.

8.2.1 Deadlock

Let us consider a set of jobs that are sharing a set of resources. A *deadlock* is a situation in which two jobs are each waiting for the other to release a resource, in order to proceed, and thus neither ever does. If more than two jobs are involved in this process, then a deadlock can be defined as a situation in which these jobs are waiting for the resources in a circular chain.

Lock-based synchronisation. This mechanism avoids deadlocks either by (*i*) not allowing nested critical sections so as to decrease the granularity of the contention, as this is the case in OMLP or (*ii*) imposing an explicit order to the programmer in which resources will be acquired, as this is the case in RNLP. In both cases, these measures must be designed and implemented off-line.

Software Transactional Memory. This mechanism avoids deadlocks either by helping or by aborting transactions upon the detection of conflicting requests for the resources. Hence, resources can be requested in any order and the conflicts are managed at runtime.

Concluding remarks. Following the previous discission, it follows that STM is able to avoid deadlocks in a more flexible and adaptive manner than lock-based mechanisms.

8.2.2 Livelock

A *livelock* is similar to a deadlock, except that the states of the jobs involved in the livelock constantly change with regard to one another, none progressing. In the context of atomic sections, a livelock corresponds to a situation in which concurrent atomic sections get mutually aborted upon a conflict. So, although the initial intention of the livelock is to allow the competitor(s) to progress, none of the concurrent atomic sections is able to progress in the end.

Lock-based synchronisation. This mechanism does not yield the ownership of resources upon each conflict, so livelocks do not occur.

Software Transactional Memory. Among the various STM strategies, Obstruction-free STM contention managers must deal with the possibility of livelock[2]. FIFO-CRT avoids livelock by applying a selection criteria based on static parameters when solving conflicts. The release time of each transaction is an example of such a parameter. Ties broken by using the core ids. This way, FIFO-CRT always determines the same serialisation sequence for any sample of transactions. This approach provides the same result independently from the time instant at which the sequence is computed.

Concluding remarks. Livelock is not an issue for any lock-based synchronisation mechanism, but it might be one for STM-based approaches. However, a properly designed contention management criteria allows us to avoid or, at least, bound livelocks.

8.2.3 Access to multiple objects per atomic section

In general, an atomic section allows operations on multiple shared objects that, from the system perspective viewpoint, can be considered as a large atomic operation on multiple objects. However, the way atomicity feature is achieved depends on the design of the synchronisation mechanism.

Lock-based synchronisation. This mechanism allows a critical section to get exclusive access to multiple shared objects either by (*i*) acquiring a lock that controls the whole data set of the

[2]For example, the Passive contention manager (Dice et al., 2006) may abort indefinitely two conflicting transactions in alternate sequence, so that none of the two ever commits. If such a conflicting situation occurs, then an additional rule such as exponential back-off must be employed to break this cycle.

critical section (non-nestable locks); or (*ii*) successively acquiring the multiple locks that control parts of the data set (nestable locks).

▷ *Non-nestable locks.* A non-nestable lock has to control the exclusive access to a set of shared objects, such that a critical section that acquires the ownership of that lock is able to access in mutual exclusion any of the controlled objects. In the worst case, a unique system global lock controls the access to all shared objects. In FMLP and OMLP, an off-line analysis of the data sets of all critical sections allows us to improve the granularity of contention by determining the unions of the intersecting data sets. This way, each of these subsets is assigned to a group lock. The impact on the parallelism of the critical sections depends on the number of objects controlled by each lock: the higher the number of objects per lock, the worse the impact is.

▷ *Nestable locks.* When locks are nestable, each lock controls very few objects or just one object. In this case, a critical section must acquire multiple locks in a sequence (nesting critical sections) in order to ensure exclusive access to its data set, such as in RNLP. Here, deadlocks are avoided by establishing a strict order in which locks will be acquired.

Software Transactional Memory. With few exceptions in literature, transactions are generally conceived to support accesses to multiple objects, without following a particular order. Consequently, the concept of nested transactions is not necessary for a transaction to access multiple shared objects. Nevertheless, some approaches allow nested transactions. In such cases, a inner transaction must be clear to commit so the outer transaction is able to proceed; however the updates of all inner transactions will be pending until the outermost transaction commits, and will be discarded if the outermost transaction aborts.

Concluding remarks. STM allows access to multiple shared objects in a transaction without specific rules, and conflicts are managed at runtime. Oppositely, lock-based mechanisms require off-line analysis in order to improve the granularity of the contention.

8.2.4 Composability

A common engineering principle in system design is to divide the complexity of a large system into smaller and simpler subsystems where each subsystem addresses a specific aspect of the desired functionality. Composability is a system design principle in which such subsystems are expected to be as much independent as possible. This way, an independent design, an implementation and testing of each subsystem, as well as a reliable composition of the system by assembling the subsystems is made possible and proved to be correct. The details of the implementation of each subsystem are hidden behind the so-called subsystem or component port/interface, thus making the component interchangeable and reusable. In theory, a highly composable designed system improves implementation, maintenance and future functionality extension.

Lock-based synchronisation. Lock-based synchronisation presents some relevant obstacles to achieve a high degree of composability. Below we discuss single global locks, non-nestable group locks and nestable locks.
▷ *Single global locks.* The simplest approach to lock-based synchronisation consists of having a unique global lock, such that the development of any system component must operate exclusively one lock. However, this solution voids any possibility of parallelism between non-contending atomic sections.

▷ *Non-nestable group locks.* The composition of synchronisation solutions based on the concept of non-nestable group locks, such as FMLP and OMLP, is a non-trivial exercise. To illustrate this claim, let us assume that the update of a system component requires the modification of a critical section, say A, such that, besides the previous objects controlled by the respective group lock, it becomes necessary to access another object, say B, that belongs to another group. It follows that B must join the initially accessed group (i.e., the group of A). The solution is to join the two groups under the same group lock. However, doing so requires that all components that access any of these two groups must also be updated to cope with the changes, otherwise the system will no longer be correct.

▷ *Nestable locks.* Fine-grained locking mechanisms, such as RNLP pose non trivial exercises to designers. These mechanisms avoid deadlocks by imposing a total order that rules the allowed sequences of object acquisitions in a (nested) critical section. Any component must be designed and implemented by taking into consideration the established total order. The same applies when a component is updated, or when a new component is developed to add functionality or replace a working component. However, testing a component individually does not reveal that the sequence is not respected, and the problem will reveal itself only when the whole system is assembled.

Software Transactional Memory. STM does not impose any order on the accessibility of the shared objects, and manages conflicts at runtime, with just-in-time contention granularity. Thus, STM is able to cope with updated and added components that have been proved correct. This is performed without requiring any modifications on concurrent modules and system control structures to maintain the system correctness.

Concluding remarks. From the previous discussion, lock-based mechanisms that are designed to allow parallelism are not composable, because any modification in a component or the adding of any new component may affect the correctness of the system. In contrast, STM allows that components can be modified or added without affecting the functional correctness of the system, as STM can adapt to these modifications.

8.2.5 Transparency

Given a model of tasks which are sharing a resource, *transparency* as it is addressed in this work refers to the degree of detail on the synchronisation mechanism that is required from the designer in order to be able to implement a correct and efficient multicore system.

Lock-based synchronisation. In order to take full advantage of the parallel feature offered by the multicore architecture, this mechanism must be designed based on the characteristics of the critical sections that will access the shared resources. In addition, the implementation of system components must meet the rules associated with the locking mechanism, namely the mapping of locks to the resources and the rules in sequential lock acquisition (if allowed). As the locks can be either nestable or non-nestable, we discuss each case in details below.
▷ *Nestable locks.* As already mentioned previously, lock-based approaches that allow nested critical sections (e.g., RNLP) usually impose a total order on the acquirement of the locks in order to avoid deadlocks. As the implementation of each system component require such a total order to be determined, it is the responsibility of the programmer to explicitly guarantee that locks are acquired in the right order when he is writing the source code.

▷ *Non-nestable locks.* We recall that lock-based approaches that do not allow nested critical sections are not exposed to deadlocks. In general, these approaches present coarser-grained contentions and practical implementations of this kind (e.g., FMLP and OMLP) use the notion of group locks in order to improve the granularity of the contentions. However, the one-to-one mapping between the locks and the resources must be known beforehand (i.e., at design time) from the programmer viewpoint, and the programmer must explicitly operate the lock that matches each resource when it accesses the critical section. Furthermore, it is also necessary to analyse the data sets of all critical sections of the system at design time in order to define the groups of resources as they are controlled by the group locks.

Software Transactional Memory. In contrast to lock-based approaches, STM saves the programmer from the burden of knowing the fine inner details of the synchronisation mechanism. Specifically, STM comes along with three features: (*i*) it allows objects to be accessed in any order inside a transaction; then (*ii*) it allows contentions to be detected with just-in-time granularity, and finally (*iii*) it allows contentions to be managed by the STM system seamlessly.

Concluding remarks. The design of a lock-based mechanism for a multicore architecture can be improved in terms of parallelism only by providing an analysis of the critical sections data sets at design time. In addition, the programmer must follow a predefined set rules defined by the designer of the locking mechanism. In contrast, STM does not require any knowledge of the transaction data sets a priori. Furthermore, the programmer is not required to follow any specific set of rules when he is implementing a transaction.

8.2.6 Priority inversion

Let us assume a set of tasks sharing a resource. Then, the *priority inversion* is a problematic situation in the schedule of this resource in which a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks. More specifically, it is a problematic situation which violates the priority model which infers that high priority tasks can only be prevented from executing by higher priority tasks. In this case, a high priority task is briefly prevented from executing by a low priority task which will quickly complete its use of the resource that is shared with the high priority task.

Lock-based synchronisation. For this mechanism, a priority inversion usually occurs when a low priority task, say τ_j, holds the lock to a resource (critical section), say R, and a high priority task, say τ_i, is constrained to wait for the lock on R to be released by τ_j before τ_i can proceed with its execution and enter R. Removing the ownership of the lock on R from τ_j in order to hand it to τ_i would require to rolling-back all operations that have already been performed in R. This is not desirable for the associated time and memory overheads. With this being said, lock-based synchronisation policies are designed not to avoid priority inversions, but rather to bound priority inversions to the minimal possible extent. As such, any high priority task τ_i can be blocked by a low priority task τ_j at most a predefined number of times. As the design-space for such a policy depends on both the class of the assumed scheduling approach (e.g., global or partitioned) and the sequence in which the requests for locks are serialised, it appears to be very large. We assume partitioned scheduling approaches in this work. Therefore, the following discussion about non-nestable and nestable locks will be restricted to their concrete implementations when they are associated to partitioned scheduling approaches.

▷ *Non-nestable locks.* Lock-based synchronisation policies such as FMLP and OMLP use a FIFO queue to sequence the access to each (group) lock. These two approaches allow for at most one task waiting for or owning a lock on each core. Consequently, the maximum length of a lock queue is m, where m is the number of cores. Assuming such a lock queue, the last task in the queue must wait for $(m-1)$ concurrent critical sections to complete before it can acquire the lock. In OMLP, a task which is waiting in a lock queue suspends and allows the corresponding core to execute other concurrent tasks whereas the story is different in FMLP: a task suspends if it is waiting for a long resource, and busy-waits with its priority boosted (non-preemptively) if it is waiting for a short resource. In either of these two approaches, the critical sections are executed with the task priority being boosted, i.e., all tasks with a higher priority are prevented from executing. The blocking may be short as in OMLP or long as in FMLP. The maximum blocking due to a low priority task which is requesting a short resource in FMLP includes the delay time for lock acquisition augmented by the execution time of the critical section. In contrast, OMLP allows another type of blocking, referred to as *transitive blocking*, when a task is not able to request a resource because a low priority task with another request pending is already executing on the same core. In the worst case, a high priority task is blocked for a delay corresponding to the lock acquisition augmented by the execution time of the critical section.

▷ *Nestable locks.* Lock-based synchronisation policies such as RNLP restrict the simultaneous requests of every resource to an arbitrary number of k tasks, and avoid deadlocks by imposing a total order on the accessibility to the resources on these tasks. In this case, each lock has its own timestamp ordered list of requests, but a task at the head of the list cannot acquire the lock until all tasks with associated requests with an earlier timestamp did with respect to the predefined partial order. Hence, each task may have to wait for $(k-1)$ concurrent critical sections to complete upon the issuance of a request. In its design, RNLP does not specify how the waiting mechanism is implemented (either busy-waiting or suspending). As such, the maximum blocking of a high priority task can range from the actual execution time up to the full response time of a critical section.

Software Transactional Memory. Let us consider the STM approaches devised in this work, in which the FIFO-CRT contention manager is used together with four different scheduling policies, namely P-EDF, NPDA, NPUC and SRPTM. Scheduling tasks by following P-EDF allows us to avoid priority inversion because any high priority job is able to preempt a running low priority job while it is executing a transaction. FIFO-CRT has been designed such that for a subset of tasks assigned to the same core, then it holds true that: (*i*) deadlocks are *always* avoided; and (*ii*) a transaction belonging to a high priority task *always* commit before the one belonging to a low priority task does. When the transactions are executed non-preemptively as this is the case in NPDA and NPUC, the priority inversion is unavoidable. More precisely, NPDA limits a priority inversion to the execution time of a transaction attempt in order to avoid aborts due to preemptions. As the delays related to the earlier concurrent parallel transactions add to the execution time of the attempt that commits, it follows that the blocking times are longer in NPUC. This type of blocking is similar in FMLP and RNLP when jobs busy-wait for a lock. However, NPUC can take advantage of the just-in-time contention granularity that STM provides to reach a higher parallelism than the lock-based approaches.

The number of blockings experienced in NPUC can also be reached in SRPTM, but the number of jobs that suffer from these blockings may be reduced as the transactions do not run non-preemptively. Typically, jobs not executing transactions and with a high preemption level may not suffer any priority inversion in SRPTM, as this could be the case in NPUC. This improvement in the average response times of the tasks is not possible for lock-based approaches as critical sections run with a boosted priority.

Concluding remarks. For a given multicore architecture, the priority inversion highly depends on the characteristics of the synchronisation mechanism that is used in conjunction with the assumed scheduling policy. STM is capable of eliminating priority inversions (as this is case when FIFO-CRT is considered together with P-EDF) or is capable of bounding its effect to the extension of a transaction attempt (as this is the case in NPDA), or to the whole length of each transaction until it commits (as this is the case in NPUC and SRPTM). Lock-based approaches do not eliminate priority inversions and have not been designed to do so. However, they are capable of bounding each priority inversion from the execution time of a critical section (as this is the case in OMLP and RNLP with suspension waiting) or from the execution time of the critical section augmented by the time to acquire the lock (as this is the case in FMLP and RNLP with busy-waiting).

8.2.7 Convoy effect

Let us consider a set of tasks to be scheduled on a resource by following a specific scheduling algorithm. Then, the *convoy effect* as introduced by (Bershad, 1993) is a non desirable situation in which the scheduling algorithm allows long-running tasks to dominate the resource, i.e., many tasks get stuck behind the execution of a single task on the target resource.

Lock-based synchronisation. Assuming such a mechanism for the management of the atomic sections, the convoy effect can be avoided by introducing special scheduling rules which will be specific to every atomic section. For example, jobs synchronised by FMLP and OMLP execute critical sections according to their boosted priorities, whereas RNLP requires the critical sections to be executed according to a priority inheritance scheme.

Software Transactional Memory. Assuming such a mechanism for the management of the atomic sections, the convoy can also be avoided as any transaction can be aborted in order to allow concurrent transactions to commit. In this context, we can conduct the following discussion on the different types of STM implementations.
▷ *Wait-free STM.* These implementations are the most complex since a fair access to memory is usually not guaranteed. They use sized buffers to bound the number of object access by any operation. However, a transaction that is not progressing can be helped by a concurrent transaction executing on another core. This way, no transaction will ever be indefinitely blocked.

▷ *Obstruction-free STM.* These implementations guarantee a transaction to make progress when all other transactions are suspended. However they require that the contention manager must take specific actions to overtake a transaction that is not progressing and thus they are not efficient. For example, FIFO-CRT allows a transaction trying to commit to abort a conflicting transaction with an earlier release time that is not executing. This rule applies when FIFO-CRT is used together with P-EDF and NPDA scheduling policies. However, when jobs are scheduled by following NPUC, transactions are executed non-preemptively (as this is the case in the previously mentioned lock-based mechanisms). Consequently, FIFO-CRT does not need to apply this rule as the convoy effect is solved at the scheduling level. On another front, SRPTM allows each job to be preempted while executing a transaction as long as it will not have a "critical impact" on concurrent transactions which are executing on other cores. Therefore, the convoy effect is not an issue for urgent jobs.

Concluding remarks. The convoy effect is addressed by lock-based and STM-based approaches that limit preemptions during the execution of atomic sections or by allowing the contention manager to overtake preempted competitors to proceed with their execution. The convoy effects caused by unsound jobs (either by software or hardware failure) are not considered in the practical lock-based and STM-based approaches as discussed in this section. Hence, measures to deal with these special issues must be implemented by using orthogonal solutions. The use of watchdogs that monitor the soundness of the jobs and reconfigure the system in the presence of an anomaly is just an example of such a solution.

8.2.8 Impact of the synchronisation mechanism on a multi-core architecture

It follows from everything that we have been discussing so far that the implementation of the developed synchronisation mechanisms have various impacts on the underlying multicore architecture. Nevertheless, the benefits gained from these implementations cover the inherent overheads associated with their operations. From a practical viewpoint, the overheads can be associated to the usage of the following three components at runtime: (1) the processing elements (i.e., the cores); (2) the platform interconnect; and finally (3) the memory. The overheads associated to each of these key components must weight in the system designer decision of choosing the most appropriate mechanism for the system. This choice must be performed with respect to the application requirements and the available platform features. In this work, we assume a multicore platform with the following characteristics:

- The platform does not implement any cache-coherency algorithm[3].
- The cores are interconnected by using a switched Network-on-Chip (NoC).
- The memory is accessible via a shared bus.

These characteristics are already available in the large majority of the commercialised processors in the embedded systems domain. The Kalray MPPA platform (Kalray2015) is an example of such a processor.

Lock-based synchronisation. The impact of such a mechanism on the target platform comes mainly from (*i*) the operations that access off-chip resources and (*ii*) the manner in which every core is used during the time period between the moment a lock is requested and the moment it is acquired. We recall that the three locking protocols discussed so far wait for a requested lock by busy-waiting (this is the case for FMLP and optionally RNLP) or by suspending the requesting job (this is the case for OMLP and optionally RNLP).
▷ *Busy-waiting.* Assuming this policy, it follows that acquiring a lock involves testing atomically the current value of the lock. Once this operation is performed, the value of the lock must be coherent across all cores that are assigned a job competing for this lock. In the absence of a cache-coherency protocol, this may require an operation on the main memory. If the lock is not immediately acquired, a pointer is inserted in a wait-queue to this job and the job is busy-waiting, i.e. it idles in a non-preemptive manner up until the lock is granted to it. During this interlude time period, the processing capacity is wasted as concurrent ready jobs are not capable of preceeding with their execution. However, as the job does not execute on the core, nor does it perform any operation, then the power consumption of the core to which it has been assigned is kept low and

[3]Given a platform, cache coherency is the protocol which ensures that changes in the values of shared data are propagated throughout the system in a timely fashion.

the shared platform resources are not requested. Finally, upon the lock release, a signal is sent to the job at the head of the wait-queue for it to acquire the lock. This operation requires a message between two cores.

▹ *Suspending.* As for the busy-waiting policy, acquiring a lock also involves testing atomically the current value of the lock. This operation may also require an operation on the main memory. If the lock is not immediately acquired, a pointer is inserted in a wait-queue to the job but in contrast to the behavior got for the busy-waiting policy, the job suspends this time. This allows the scheduler to select another ready job for execution. Once the lock is granted to the suspended job, the scheduler preempt the running job irrespective of its priority in order to resume the execution of the suspended job. It follows that this approach presents the advantage of improving the core usage in an effective manner; but the disadvantage of requiring at least two context switches. Unfortunately, this contributes in turn to the overhead associated with this policy. Finally, similar to the busy-waiting policy, when the lock is released, a signal must be sent to the job at the head of the wait-queue in order for it to acquired the lock. This operation also requires a message between two cores.

Software Transactional Memory. Before discussing the specifics of STM-based mechanisms, it is worth recalling that the assumed target platform does not implement any cache-coherency algorithm. In this context and assuming an STM-based mechanism, we note that transactions can be executed in an optimistic manner, i.e., they can operate on local copies of shared objects. This feature matches well with an absence of cache-coherence as data validity is maintained by the STM itself. To this end, local updates become globally available only after a commit and transactions with invalid data abort and refresh the local copies for the following attempt. This procedure may require readings from the main memory. Below, we discuss the relationship between STM and the various types of locks.

▹ *Hidden locks.* Although from the programmer standpoint STM is free from locks, their actual implementation may use locks to guarantee mutually exclusive accesses to control and data structures. In this context, conflict detections can be of two types: *eager* and *lazy*. Eager conflict detection is active and detects a conflict early on, while lazy conflict detection is passive and detects a conflict when tasks are about to commit their results. Even though performance comparisons show that eager conflict detection heuristics are better in benchmarks where tasks share several objects among themselves, we adopted lazy policies in this work as they are easier to implement and faster in benchmarks where tasks share a small number of objects and, most importantly, they do not mask read transactions (Spear et al., 2006) which is very relevant for real-time systems. This allows us to have full control on the resolution of conflicts and, therefore, the serialisation of transactions. With this choice, we benefit from the fact that locks are owned only for the time required for a transaction to commit (i.e., to detect conflicts and to commit updates), and not for the whole critical section, as is the case in locking mechanisms.

▹ *Querying job status on other cores.* We recall that FIFO-CRT avoids deadlocks by allowing any contender that is trying to commit to abort a preempted transaction. This rule is necessary only while following either P-EDF or NPDA. Indeed, these approaches allow many transactions in progress at the same time on the same core. To implement this rule, the scheduler must be inquired when each job executing a conflicting transaction is running. This operation may incur significant runtime overheads. As a matter of fact, upon each detected conflict the scheduler must interrupt the running job and check the status of the job executing the transaction on each core.

▷ *Execution time overhead due to non-preemptively executed aborts.* On one front, we recall that approaches such as NPDA and NPUC execute transactions non-preemptively. On another front, we recall that SRPTM grants the capability to preempt only to jobs that are more urgent than the transaction in progress. It follows that the result of each attempt can be discarded if the transaction aborts, thus augmenting the execution time overhead. Under NPUC, a transaction can dominate the core upon which it is executing until it commits. In a benchmark with lock-based suspension-waiting, the worst case execution time overhead can be far superior, whereas it is at most comparable when comparing with lock-based busy-waiting. The reason for this behaviour is related to the fact that STM can profit from a finer granularity in conflict detection than in group locks. FIFO-CRT allows a transaction to commit before concurrent transactions with earlier release times do when the concurrent transactions are zombies[4]. Although the worst case execution overhead may be higher, a transaction may commit earlier than expected in some time windows. This is not possible with lock-based mechanisms.

Concluding remarks. Lock-based synchronisation impacts the underlying platform in terms of lock operations that require communication between cores whereas STM matches architectures without cache-coherency, but may have a significant impact on the execution time overhead due to the number of aborts.

8.2.9 Platform dependency

The platform characteristics can affect the performance of the transactions as they may favor specific synchronisation mechanisms over others. In this section, we discuss the *platform dependency* as the degree at which the synchronisation mechanism is integrated into the physical hardware and environment runtime that supports the system. The more the synchronisation mechanism is integrated inside the core components (e.g., in the scheduler, the kernel) the more dependent it is to the platform. In this case, it is less portable to other platforms.

Lock-based synchronisation. Locks are implemented in the kernel space. Therefore, they are intrinsically bounded to the operating system in which they are implemented.

Software Transactional Memory. In general, STM are implementated in user space libraries, which make them highly portable. Now, as our main goal is to meet the timing requirements of

[4]This is possible as at the time the transaction commits, this operation does not augment their abort counts.

real-time embedded systems, it follows that the approaches devised in this work combine both STM-based techniques and scheduling policies. Below we discuss the various combinations.
▷ *STM and P-EDF.* This combination does not require any modification of the kernel. However it requires that the scheduler is capable of indicating the state of a job that has a transaction in progress.

▷ *STM and NPUC/NPDA.* We recall that NPUC and NPDA are based on the P-EDF scheduler and do not require any modification of the scheduler itself. Also, as this is already the case for STM and P-EDF, NPDA requires that the scheduler is capable of indicating the state of a job that has a transaction in progress. However, both approaches (i.e., NPUC and NPDA) require that the system is capable of enabling/disabling preemptions. To this end, this operation is performed through requests or by enabling/disabling interrupts if necessary.

▷ *STM and SRPTM.* The SRPTM policy is implemented in the kernel space. Indeed, even though its design is based on the P-EDF scheduler, it requires the modification of the scheduler, especially when a transaction is in progress in a core in order to integrate the necessary structures for the following key features: the core ceilings, the premption levels of the tasks and finally, the premption levels of the transactions.

Concluding remarks. Locks requires to add functionalities in the kernel space whereas STM-based approaches are usually implemented in user space. As real-time applications may require that the STM-based approaches make use of system calls to address timing requirements, complex approaches such as SRPTM require the implementation of scheduling rules inside the kernel.

8.3 Summary

In this chapter we discussed the evaluation of the FIFO-CRT contention manager when it is implemented together with the following four scheduling approaches: P-EDF, NPDA, NPUC and SPRTM, in both quantitative and qualitative terms.

The quantitative evaluation is based on the results of the simulations conducted on randomly generated synthetic task sets. The simulation set up is given in Section 8.1.1 and Table 8.1 highlights the results for the group of tasks with 64 cores and 32 contention groups in common. These partial results provide the trend of the characteristics observed in the experiments. That is:

1. The scheduling approaches that execute atomic sections in a non-preemptive manner have more difficulties to meet the deadlines. This is due to the fact that it is more difficult to schedule tasks with shorter relative deadlines in this case.

2. The scheduling approaches that allow preemptions during the execution of a transaction present a higher average number of aborts per job. P-EDF and SRPTM illustrate this claim. P-EDF allows to abort a transaction that is preempted and SRPTM allows a transaction to suffer longer delays waiting for concurrent transactions that may have been preempted.

3. SRPTM incurs a smaller overhead per transaction in average as: (*i*) a transaction is not aborted upon preemption; (*ii*) when a transaction is preempted, the time period spent in this state may be less than the time associated to the corresponding number of aborted attempts.

The qualitative evaluation discusses various aspects of the different mechanisms (see Section 8.2). Table 8.2 summarises the qualitative comparison of STM-based approaches against lock-based approaches for real-time embedded systems, when assuming a multi-core platform with a NoC interconnect and without cache-coherence.

In Chapter 9 we complete this in-depth analysis with the practical implementation of the FIFO-CRT contention manager together with the three scheduling policies on a two 12-core AMD Opteron Processor 6168 based computer.

	P-EDF	NPDA	NPUC	SRPTM	FMLP
Total deadline misses	432 176	423 635	438 299	362 877	1 006 744
	28 501	49 013	53 829	23 435	179 048
Maximum aborts per job	7.73	2.15	2.09	7.14	–
(average)	1.36	0.71	0.74	1.15	–
Average execution time	33%	62%	62%	23%	69%
overhead per job	8%	11%	11%	6%	18%

Table 8.1: Quantitative results for the group of 50 task sets with 64 cores and 32 contention groups in common. For each category, the bottom line corresponds to results of simulations with smaller atomic sections.

	Lock-based mechanisms	STM-based mechanisms
Deadlocks	Group locks: No Nestable locks: responsibility of the programmer	No
Livelocks	No	No
Access to multiple objects in an atomic section	Group locks: Yes (coarse grained) Nestable locks: Yes (fine grained)	Yes
Composability	No	Yes
Transparency	No	Yes
Priority inversion	Yes (bounded)	P-EDF: No NPDA, NPUC, SRPTM: Yes (bounded)
Convoying	Priority boosting/inheritance: No	P-EDF, NPDA, NPUC: No SRPTM: Bounded
Impact on the architecture	Busy-waiting: Intercore communication, execution time overhead Suspending: Intercore communication, context switching Locks owned during all critical section (impact on parallelism)	Optimistic execution of transactions matches architectures without cache-coherence P-EDF, NPDA: requests of jobs status to schedulers on other cores Execution time overhead due to aborts Lazy conflict detection: locks owned only during commit
Platform dependency	Locks are implemented inside the kernel (high dependency)	STM (with contention manager) is generally a user space library (low dependency) NPDA and NPUC: require system calls to enable and disable preemptions (low dependency) SRPTM: requires to rewrite the scheduler inside the kernel (high dependency)

Table 8.2: Summary of the qualitative evaluation of STM (opposed to locking) for real-time embedded systems.

Chapter 9

Implementation

The experiments performed in a simulation environment reported in Chapter 8 had as main goal the comparison of our STM-based approaches, i.e. the FIFO-CRT contention management policy combined with the Non-Preemptive Until Commit (NPUC), SRP for Transactional Memory (SRPTM) and Non-Preemptive During Attempt (NPDA) scheduling strategies with the state-of-the-art lock-based synchronisation mechanism Flexible Multiprocessor Locking Protocol (FMLP). The set of simulations revealed that our STM-based approaches were more scalable than FMLP.

However, these simulations considered only overheads related to the extra execution time required when a transaction aborts. We implemented a minimalistic STM system with the FIFO-CRT contention manager policy, and the NPUC, SRPTM and NPDA schedulers for the Linux operating system, so as to evaluate the performance of our approaches from a practical viewpoint. The experiments were conducted on a testbed that allows us to measure and compare both the performance and overheads that each approach presented for the executed task sets that were executed.

9.1 Experimental setup

This section describes the development of a computing system that provides an STM system in which concurrent accesses are ordered by the FIFO-CRT contention manager proposed in Chapter 5, and the three scheduling approaches (NPUC, SRPTM and NPDA) proposed in Chapter 6.

9.1.1 Platform specification

The target hardware platform used to conduct the experiments is a computer with 24 cores provided by two 12-core AMD Opteron Processor 6168. Each processor occupies one socket on the motherboard and contains three levels of cache: 12 x 64 KB instruction and 12 x 64 KB data Level 1 cache, 12 x 512 KB Level 2 cache, and 2 x 6 MB Level 3 cache. Each core is 64-bits and clocked at 1.90 GHz. Both processors share access to 4 GB of main memory.

The adopted Operating System is the Linux kernel 3.10.15, modified by using the Real-time TAsk-Splitting scheduling algorithms (ReTAS) framework. ReTAS was developed by Sousa et al. (2011) to support the development and testing of real-time scheduling algorithms.

9.1.2 Task set generation

We defined a number of task set profiles, and assume transaction and context switch overheads to be negligible. Each task set profile was characterised by the 2-tuple $\{m, U_S/m\}$, where m is the number of cores of the system and U_S/m is the ratio of the system capacity that is demanded. Specifically, we consider the following values for these two parameters:

- $m \in \{2, 4, 8, 16\}$ cores
- $U_S/m \in \{0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75\}$

For each task set, the maximum task utilisation U_i^{max} is randomly selected from an arbitrary set of values with equal probabilities, such that $U_i^{max} \in \{0.25, 0.50, 0.75\}$. In each task set profile, each task τ_i is categorised as LIGHT, MEDIUM and HEAVY, according to its individual utilisation, with the following interpretation:

Definition 36 (Light task)**.** Let us consider a system with m cores and a demand of U_S/m of its total capacity. Task τ_i with utilisation $U_i = C_i/T_i$ is categorised as a *light* task if $U_i \in \left[0.05,\ 0.20 \cdot \frac{U_s}{m}\right)$.

Definition 37 (Medium task)**.** Task τ_i with utilisation U_i is categorised as a *medium* task if $U_i \in \left[0.20 \cdot \frac{U_s}{m},\ \min\{0.50 \cdot \frac{U_s}{m}, U_i^{max}\}\right)$.

Definition 38 (Heavy task)**.** Task τ_i with utilisation U_i is categorised as an *heavy* task if $U_i \in \left[\min\{0.50 \cdot \frac{U_s}{m}, U_i^{max}\},\ U_i^{max}\right]$.

For every task set, task τ_i is generated as *LIGHT*, *MEDIUM* or *HEAVY* with probabilities $P(Light) = 0.50$, $P(Medium) = 0.35$ and $P(Heavy) = 0.15$, respectively. The utilisation of τ_i is randomly selected from an interval (with a uniform distribution) within the limits of the respective type of task.

The period of each task τ_i is determined as the product of three non-negative integer values, i.e. $T_i = a_i \times b_i \times c_i$, where a_i is randomly chosen from $\{2, 4, 8, 16\}$, b_i from $\{3, 6, 9, 12\}$ and c_i from $\{5, 10, 15\}$, in order to keep the hyperperiod (i.e. the least common multiple of all task periods) reasonably small, as suggested by Nelis et al. (2013). The WCET of task τ_i is given by $C_i = U_i \times T_i$. We assume all timing values to be in milliseconds.

We consider synchronous task sets (i.e. all the tasks release a job at the same time instant, say the first job at $t = 0$), with implicit deadlines (i.e. for a task set with n tasks, $D_i = T_i,\ \forall i \in \{1, 2, \ldots, n\}$).

Each task set is characterised according to the proportion of tasks executing a transaction, and the proportion of transactions that update a portion of its dataset. Before we discuss the actual details of our implementation, let us introduce the following definitions.

Definition 39 (Transaction ratio)**.** Let τ_{trans} be the subset of τ in which every task executes at least a transaction. The *transaction ratio* is defined as $P_t \overset{\text{def}}{=} \frac{n_{\text{trans}}}{n}$, where $\#\tau_{\text{trans}} = n_{\text{trans}}$.

Definition 40 (Update transaction ratio)**.** Let τ_{update} be the subset of τ_{trans} in which every task executes an update transaction. The *update transaction ratio* is defined as $P_u \overset{\text{def}}{=} \frac{n_{\text{update}}}{n_{\text{trans}}}$, where $\#\tau_{\text{update}} = n_{\text{update}}$.

For each task set, the transaction ratio P_t is randomly selected from the set of values with equal probabilities, such that $P_t \in \{0.40, 0.50, 0.60, 0.70, 0.80\}$. A number of tasks is randomly selected that the ratio of tasks executing a transaction is within a 0.05 error from the selected P_t. In the same vein, the ratio of update transactions among all transactions, denoted as P_u, is randomly selected from $\{0.25, 0.50, 0.75\}$. A number of tasks executing a transaction is randomly selected such that the ratio of tasks executing an update transaction is within a 0.05 error from the selected P_u.

The execution time of the transaction ω_i executed by τ_i, C_{ω_i}, is computed as a fraction of the task execution time C_i, given by a normal distribution with mean $\mu = 0.20$ and standard deviation $\sigma = 0.10$. The time at which ω_i starts inside task τ_i is random. The STM environment is defined based on the system characteristics. The number of transactional objects p is computed as $p = 2.5n$, and the number of contention groups g is defined as $g = \lfloor n/2 \rfloor$. The transactional objects are evenly distributed between the contention groups, such that the size of each group $|\Omega| = \frac{p}{g} = 5$. Each transaction ω_i is randomly assigned to a contention group. The size of ω_i dataset is randomly selected from $\{1, \dots, |\Omega|\}$. The objects in the dataset of ω_i are randomly selected from the assigned contention group, such that ω_i does not compete with transactions in other contention groups. If ω_i is an update transaction, each object in its dataset is randomly assigned a read or write operation. We assume that at least one object must be accessed by a write operation.

9.1.3 STM specification

We developed a minimalistic STM system by using the C language, as a Linux user space service. This STM stores the current version of the shared transactional objects and manages the current accesses to every transactional object. The STM is initialised with a fixed number of transactional objects. Each transactional object is identified by a unique index number and the STM keeps a pointer to the current object value, its current (sequential) version number, and a chronologically ordered list of pointers to the descriptors of the transactions that are currently accessing the object. The chronologically ordered list allows us to assign priorities to the transactions in the commit sequence. This list is updated everytime a transaction requests access to the transactional object, and when a transaction commits. Each transactional object has a mutex that synchronises the operations that involve modifying the current value of the object and insert or remove a transaction descriptor from the ordered list. The time required by the operations to own this mutex is expected to be a small fraction of the execution time of a complete transaction/atomic section. Figure 9.1 illustrates a simplified view of this architecture.

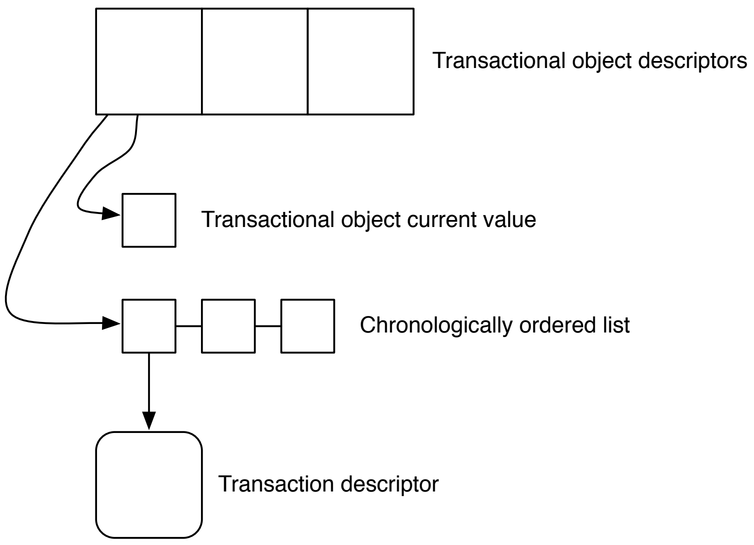

Transactional object descriptors
Transactional object current value
Chronologically ordered list
Transaction descriptor

Figure 9.1: STM transactional object descriptors.

Each transaction descriptor has a pointer to the local copy of the transactional object and a list of pointers to every transactional object descriptor. This list allows us to check the status of the transactional objects at the time the transaction is committing.

The STM system provides the tasks with the following calls:

`stm_start_transaction`. This function is called by the task when it reaches the beginning of the transactional section, and sets the environment for the transaction. This function signals the operating system that a transaction was released.

`stm_read_object`. This function opens a transactional object for read-only access by the calling transaction. The transaction is inserted in the transactional object list of accesses, and the transactional object is added to the transaction current dataset.

`stm_write_object`. This function opens a transactional object for read-write access by the calling transaction. The transaction is inserted in the transactional object list of accesses, and the transactional object is added to the transaction current dataset.

`stm_commit`. This function is called by the task at the end of the transactional section. The STM verifies if the transaction is allowed to commit (and apply the required STM state updates if this is the case) or it must abort. This function signals the operating system if the transaction committed or aborted.

9.1.4 Schedulers specification

The scheduling approaches – NPUC, NPDA and SRPTM – were implemented as schedulers into the ReTAS framework, in the Linux kernel 3.10.15. Every scheduler implemented two system calls that allow the STM to inform the operating system on the status of the current transaction:

`retas_set_active_transaction`. This system call is invoked at the beginning of a transaction and informs the scheduler that a transaction is currently active on core where the task has been assigned.

`retas_clear_active_transaction`. This system call is invoked at the end of a transaction and informs the scheduler on the status of the current active transaction (committed or aborted).

The three schedulers are based on the original partitioned-EDF scheduler that is part of the ReTAS framework, adding the respective extensions to the base policy. Specifically, each core executes its assigned tasks by following the EDF scheduler and the extensions are activated only when a transaction is released on the core and deactivated when the transaction commits. Both system calls `retas_set_active_transaction` and `retas_clear_active_transaction` have the effect of temporarily modify the scheduler behaviour, as described below.

NPUC. The call to `retas_set_active_transaction` executes the current transaction on the core in a non-preemptive manner. This is given with the interpretation that any arriving job, irrespective of its priority, is enqueued and not scheduled until the transaction has committed. The call to `retas_clear_active_transaction` is ignored by the scheduler when the transaction of the executing job aborts, and it switches back the scheduler to EDF once the transaction has committed.

NPDA. As for NPUC, the call to `retas_set_active_transaction` executes the transaction non-preemptively. However, the call to `retas_clear_active_transaction` allows jobs with a higher priority to be executed when the transaction aborts, and it switches back the scheduler to EDF when the transaction commits.

SPRTM. The call to `retas_set_active_transaction` sets the core ceiling to the transaction preemption level with the interpretation that only arriving jobs with a higher preemption level than the current core ceiling and with no transaction can be scheduled. The call to `retas_clear_active_transaction` is ignored by the scheduler when the transaction aborts, and it switches back the scheduler to EDF when the transaction commits.

9.2 Results

This section reports on the quantitative results of the experiments conducted in a practical computing system with real-time requirements assuming synthetic task sets. Each task set was executed for two hyper-periods for each of the three scheduling approaches – NPUC, SRPTM and NPDA – and was executed in the context of a Linux process. Each task was executed as a Linux thread. The master thread loads the task set specifications and sequentially creates the task threads, setting their individual core affinity, scheduling and STM parameters. It also sets a common first arrival time instant so that all tasks are released synchronously.

During the experiments, we monitored the following metrics:

- the time between the arrival of two consecutive jobs for the same task,
- the execution time of each job,
- the response time of each job,
- the number of preemptions of each job,
- the response time of each transaction from its release time until its commit time, and
- the number of aborts experienced by each transaction.

Before we conducted the experiments, we observed that the execution time in isolation of each job on the platform presents minor fluctuations, increasing with the number of tasks on each core. The fluctuation could be significant relative to the actual WCET, for tasks with small execution times. However, to take these fluctuations into account, we tuned the system such that the effective execution time of a task would be around 80% of the task WCET, on average. The remaining 20% margin were established to accommodate the eventual system overheads, so as to prevent the actual execution time of a job to exceed its predefined task WCET.

The period of each task was set exactly as the predefined value. The experimental results revealed that the arrival of two consecutive jobs of the same task was consistently close to the predefined period across all the task set profiles, with the average $\frac{T_{experimental}}{T} = 1.003$ and the maximum standard deviation of this ratio observed was 0.004.

9.2.1 STM performance

The influence of the scheduling strategy on the performance of the STM system was observed by measuring the number of aborts that were experienced by the transactions during the experiments. This metric is meaningful in this regard as the maximum number of times an instance of a transaction is aborted has a direct impact on the response time of the job that executes it. Figure 9.2 to Figure 9.5 present the maximum number of aborts observed for a single job of a task, normalised by the number of tasks that executed a transaction in each task set profile. The charts do not permit to determine a scheduling strategy that consistently provides the lowest maximum number of aborts per job. However, in some spurious cases, SRPTM presents results that are worst than the other two scheduling strategies (NPUC and NPDA). The same conclusion applies to NPDA as compared to NPUC and SRPTM.

On a different perspective, the total number of aborts per task during the experiments reveals the total amount of execution time overhead that is due to the STM synchronisation mechanism. Figure 9.6 to Figure 9.9 illustrate the observed total number of aborts per task, normalised by the number of tasks that executed a transaction in each task set profile. Again, the charts do

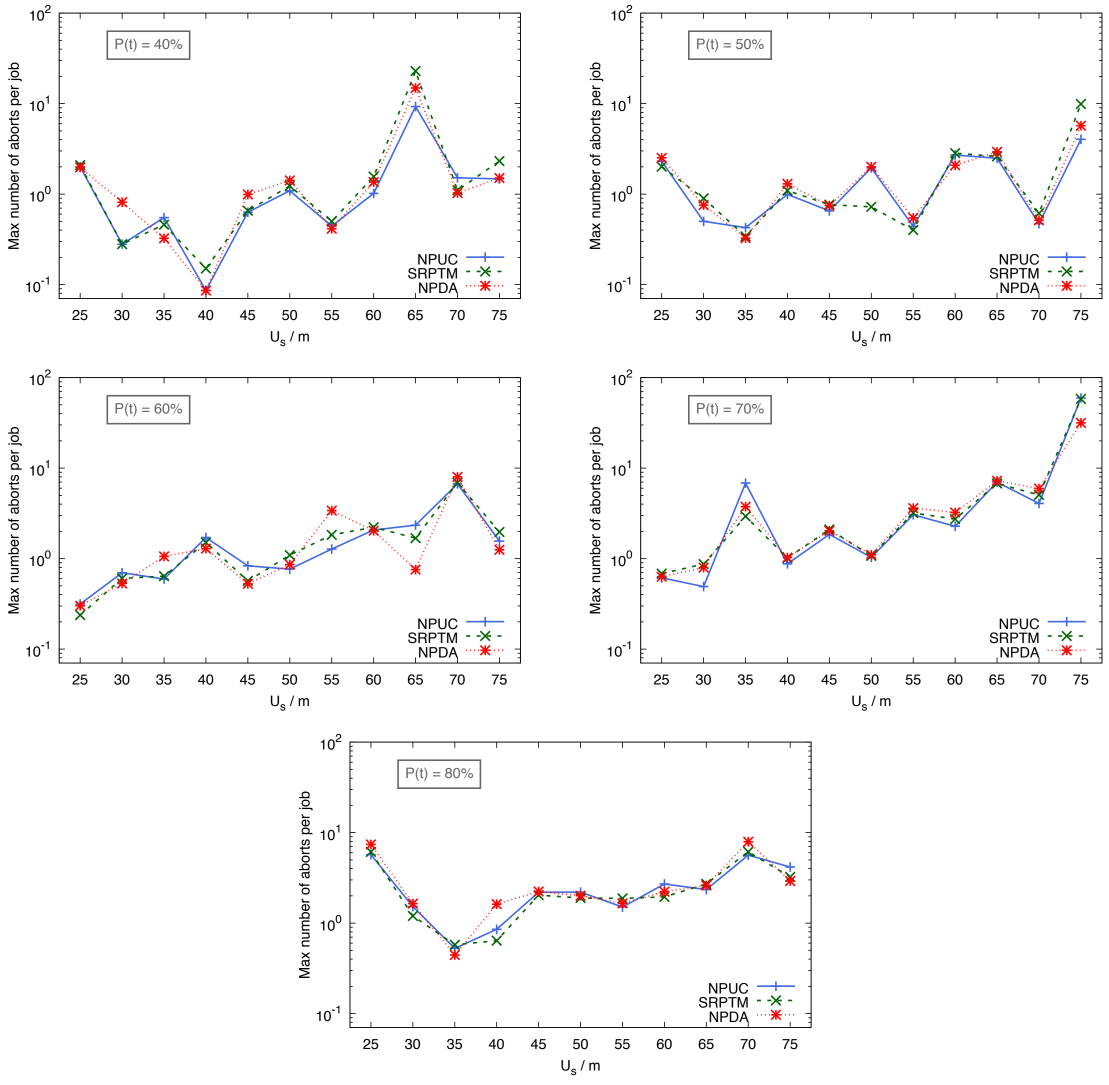


Figure 9.2: Maximum number of aborts of a transaction per job ($m = 2$).

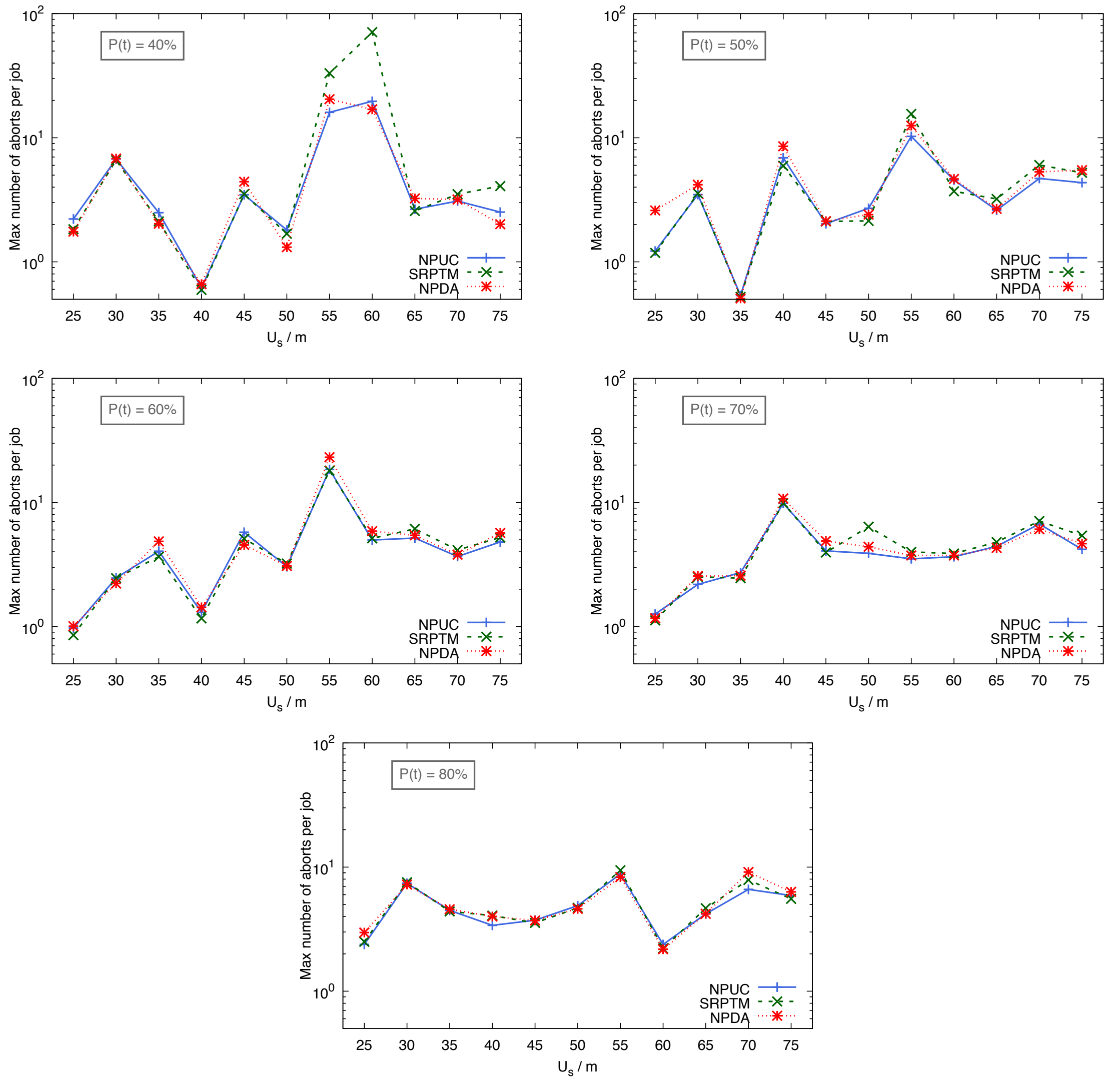


Figure 9.3: Maximum number of aborts of a transaction per job ($m = 4$).

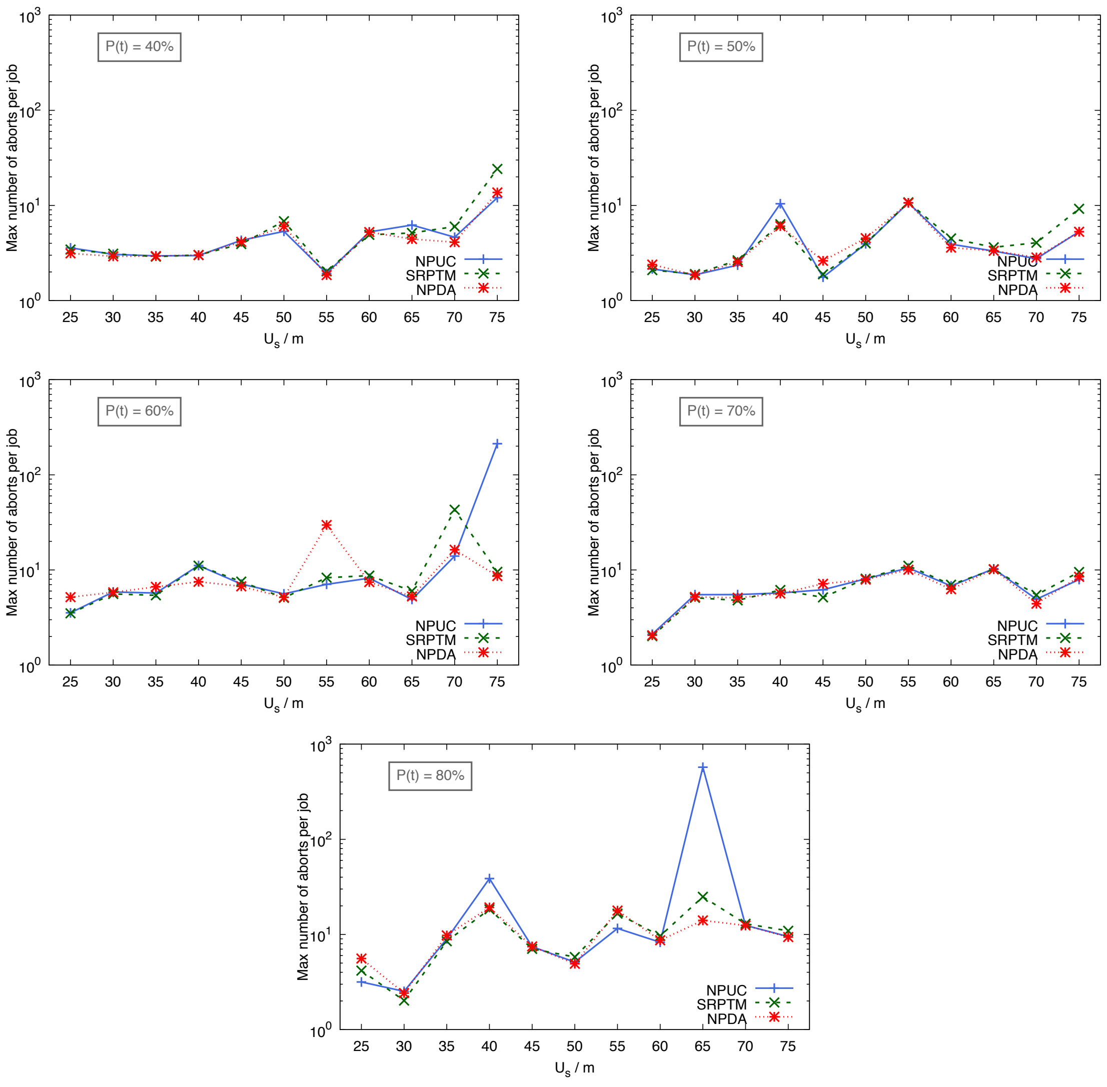


Figure 9.4: Maximum number of aborts of a transaction per job ($m = 8$).

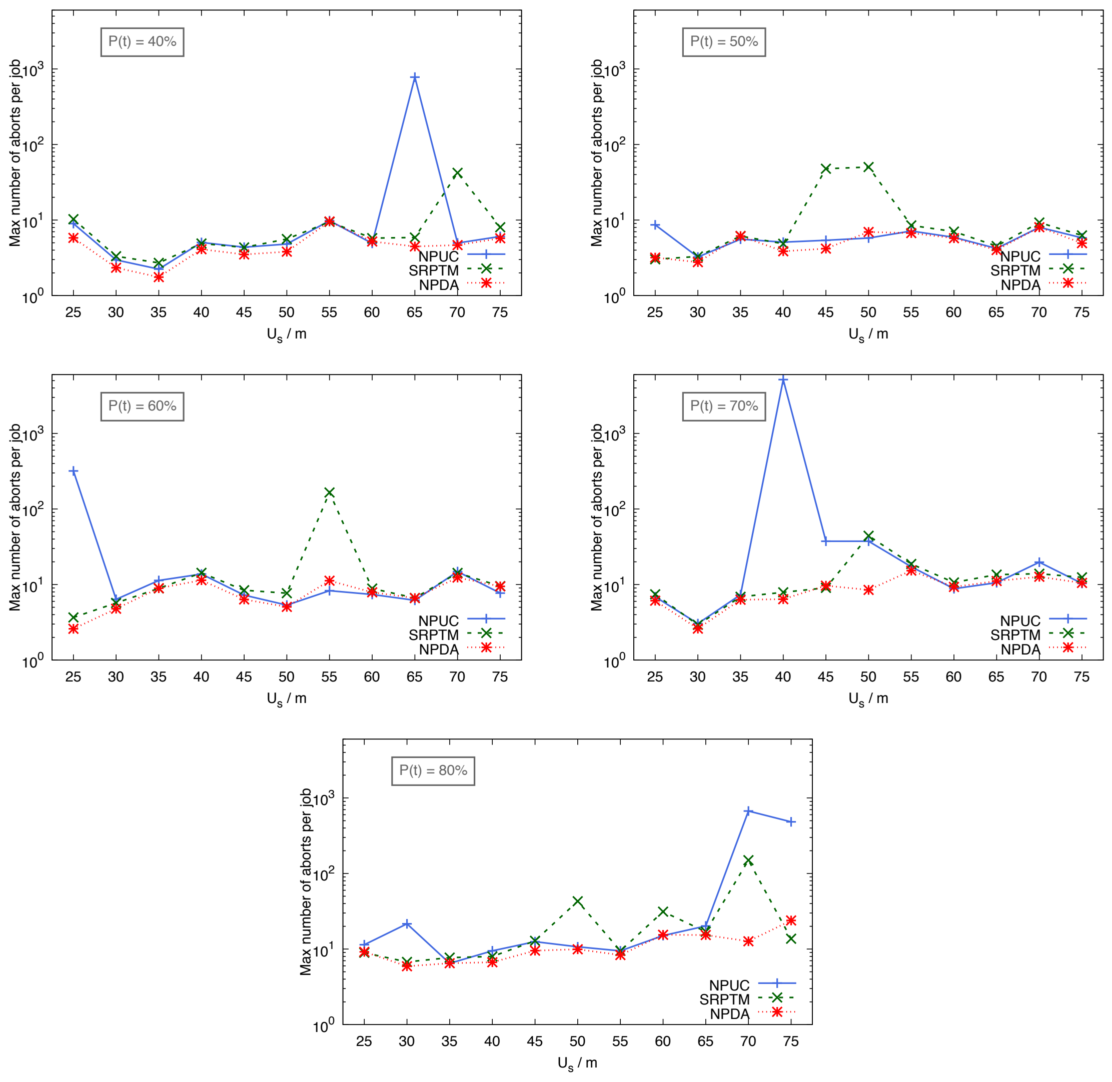


Figure 9.5: Maximum number of aborts of a transaction per job ($m = 16$).

not show a scheduling strategy that consistently provides the lowest transaction execution time overhead. However, for systems with a higher number of cores, NPUC presents worst results in general. This is due to short transactions that are non-preemptible and abort a significant number of times while waiting for concurrent transactions to commit on other cores. This effect sums up on multiple instances of such transaction, becoming visible in the charts.

In summary, the results indicate that the task set properties have a major influence on the number of aborts, rather than the scheduling strategy. However, NPUC is more prone to more aborts on short transactions that concur with longer transactions than SRPTM and NPDA.

9.2.2 System performance

This section reports on the influence of the proposed scheduling strategies on the overall task set performance. Specifically, we discuss the maximum number of preemptions a job suffers; the average execution time overhead that each task incurred; the workload; and, finally, the responsiveness of each task.

Maximum number of preemptions per job. Preemptive scheduling strategies are characterised by the ability of the scheduler to suspend the execution of a running job so as to schedule another job with a higher priority. This technique incurs in practice execution time overheads due to context switching operations. For each job, we measured the number of preemptions it suffered, and recorded the maximum observed value for each task. Then, for each task set profile, we determined the average value for all the tasks.

The results allow us to observe a common trend for all the three strategies: the maximum number of preemptions for a single job grows as the ideal system demand increases. This result is intuitive and expected as when the number of tasks increase, more interference is expected.

The results for the three scheduling strategies are very similar in the majority of task set profiles. NPUC presents the largest maximum number of preemptions per job for $m = 2$ cores, but the lowest for higher number of cores. This indicates that the longer the non-preemptive period, the larger the number of ready jobs enqueued to be scheduled in a sequential manner when the scheduler switches back to preemptive again. This leads to a potential reduction of the number of preemptions on those specific jobs.

Average execution time overhead per task. Although context switching due to preemptions is a practical source of overhead, diverse system events such as other scheduler operations (not related to preemption) and STM transaction aborts can add to the total execution time of a task. The average overhead of each task τ_i was inferred from the ratio between the total processor time demanded by the observed j_i jobs of τ_i, and the theoretical demand that τ_i would require during the observed experiment, i.e. $j_i \times C_i$. Finally, for each task set profile we determined the average execution time overhead for all tasks. We must recall that the execution time per task in practice was set to be approximately 80% of the predefined task WCET to accommodate eventual system fluctuations.

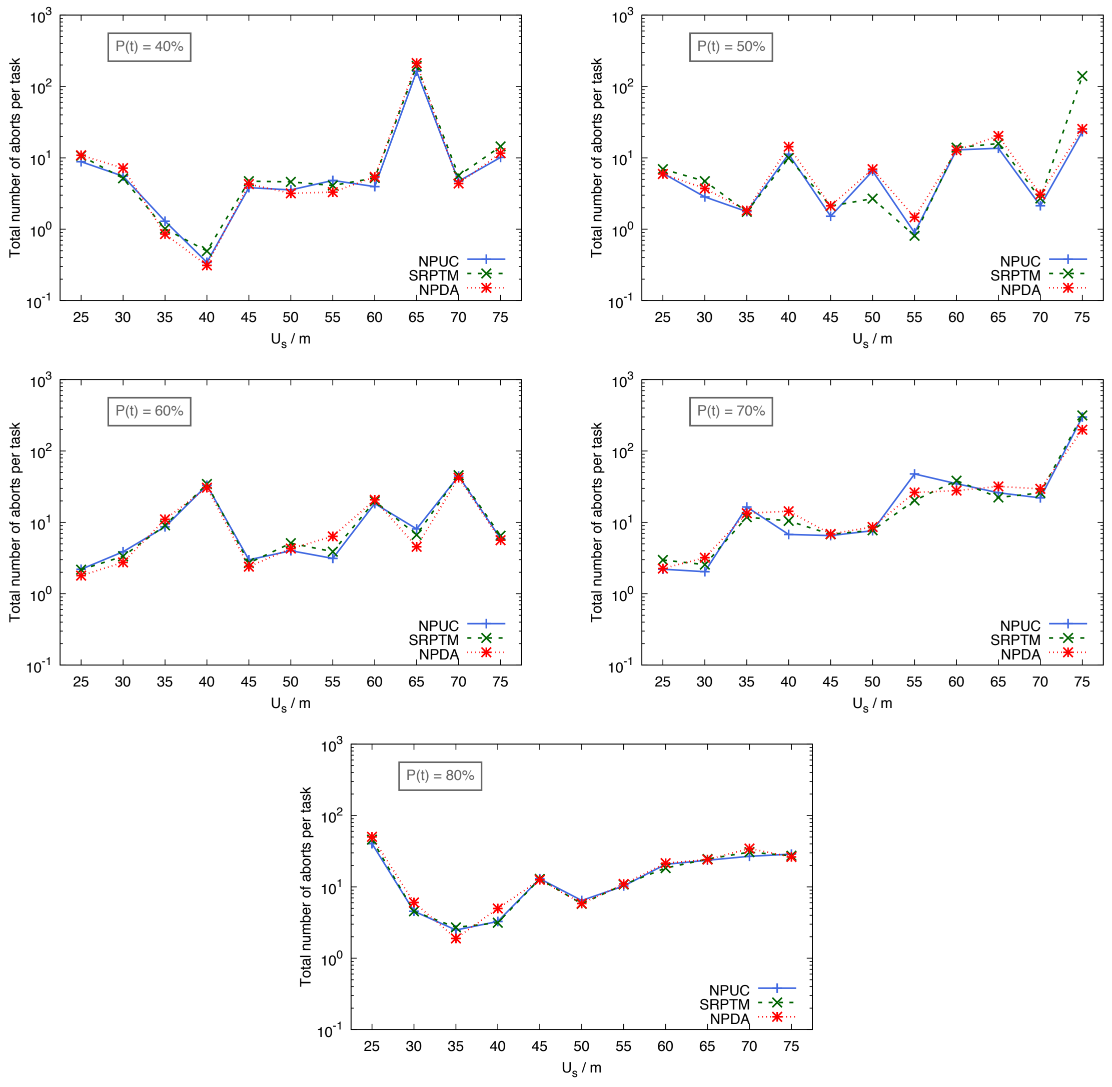


Figure 9.6: Average number of aborts per task ($m = 2$).

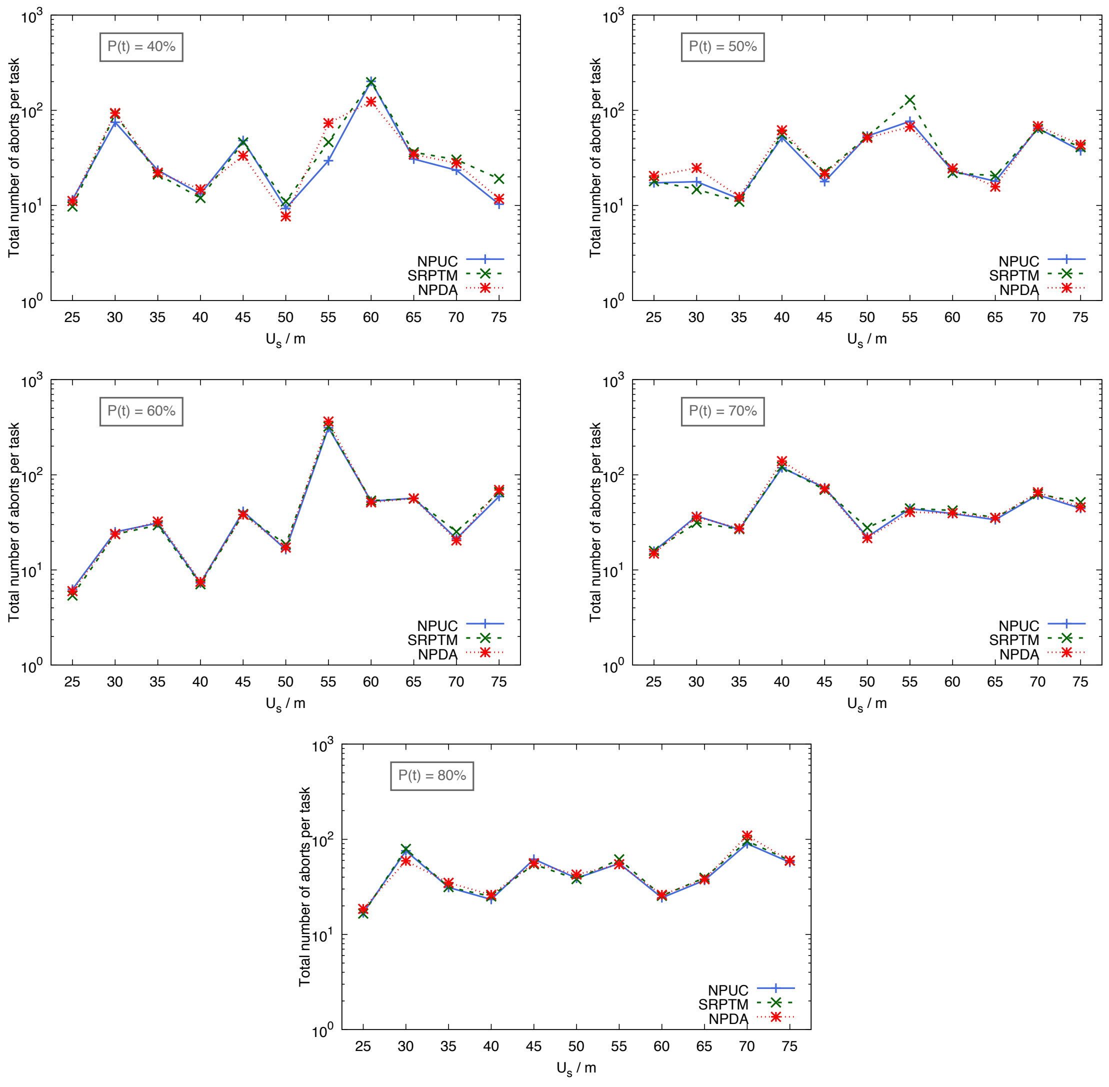


Figure 9.7: Average number of aborts per task ($m = 4$).

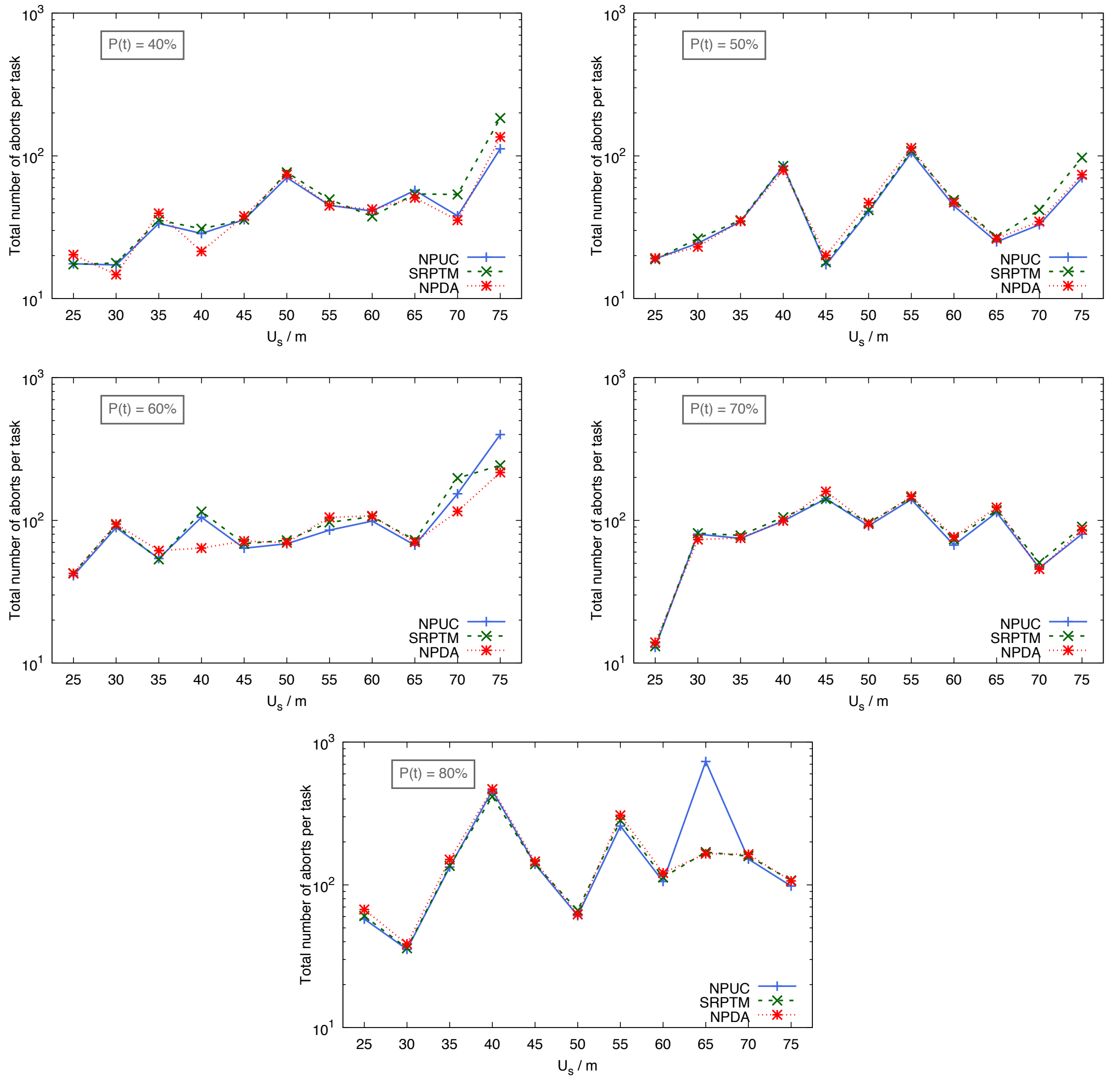


Figure 9.8: Average number of aborts per task ($m = 8$).

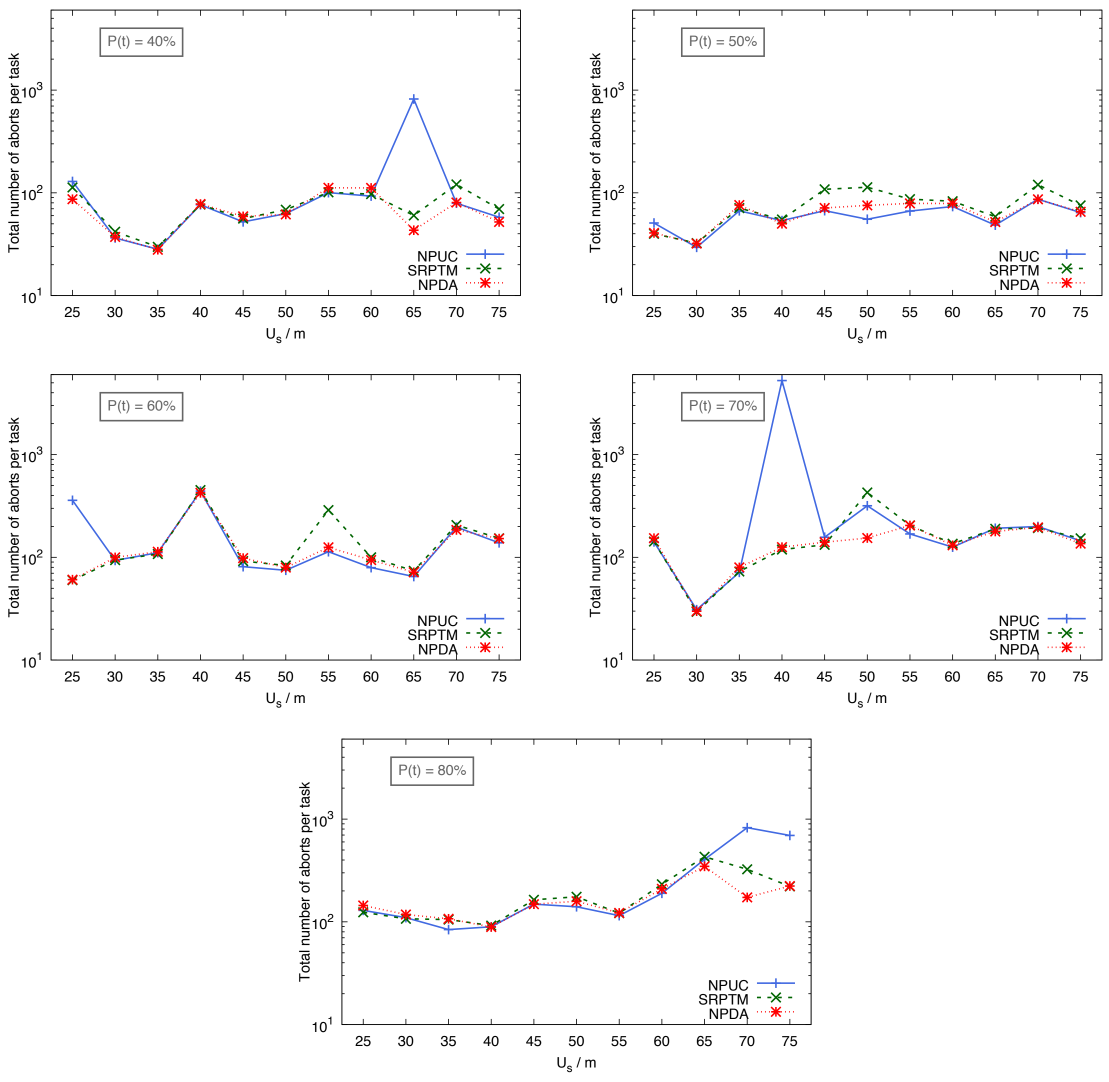


Figure 9.9: Average number of aborts per task ($m = 16$).

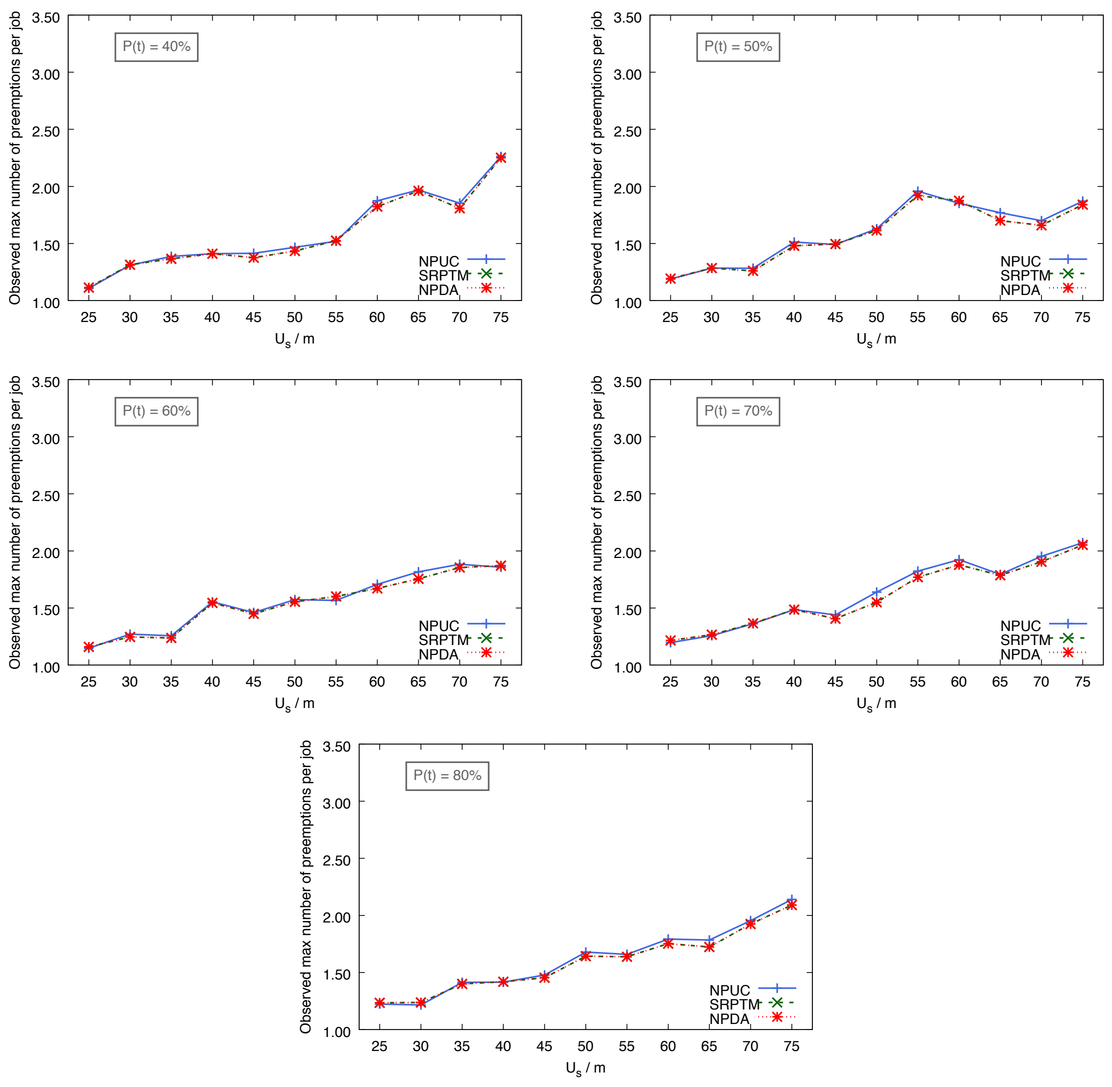


Figure 9.10: Maximum number of preemptions per job ($m = 2$).

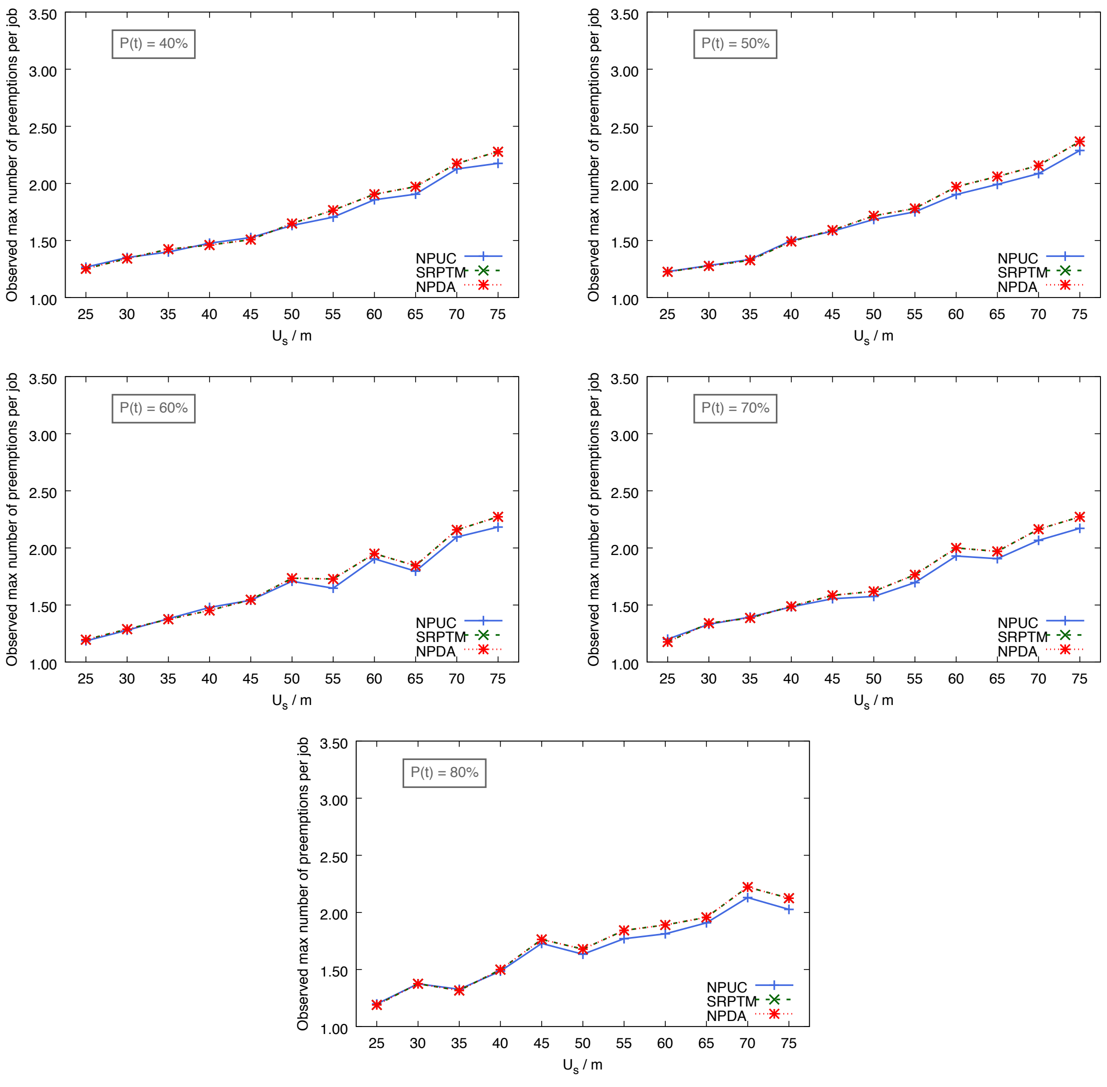


Figure 9.11: Maximum number of preemptions per job ($m = 4$).

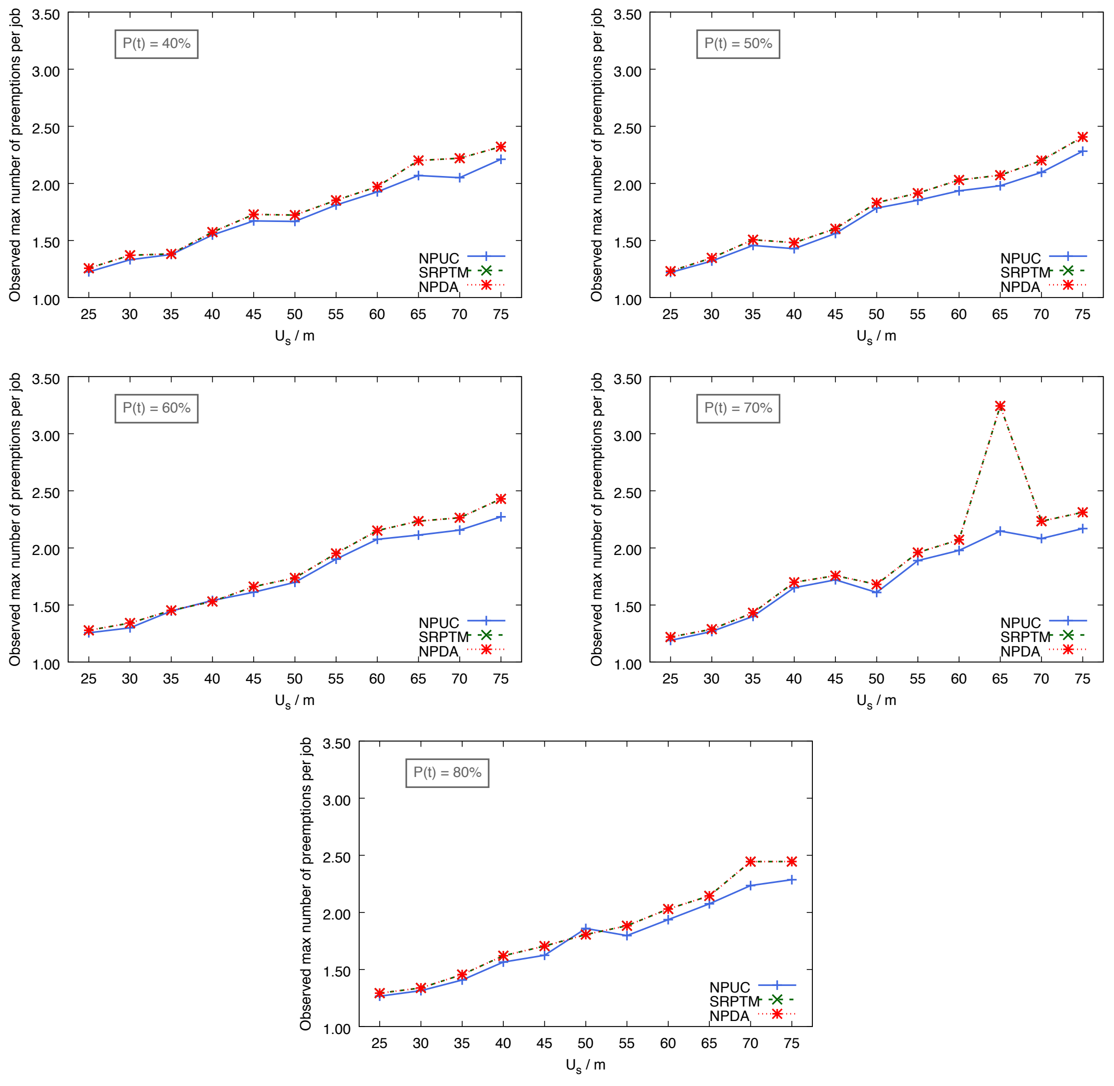


Figure 9.12: Maximum number of preemptions per job ($m = 8$).

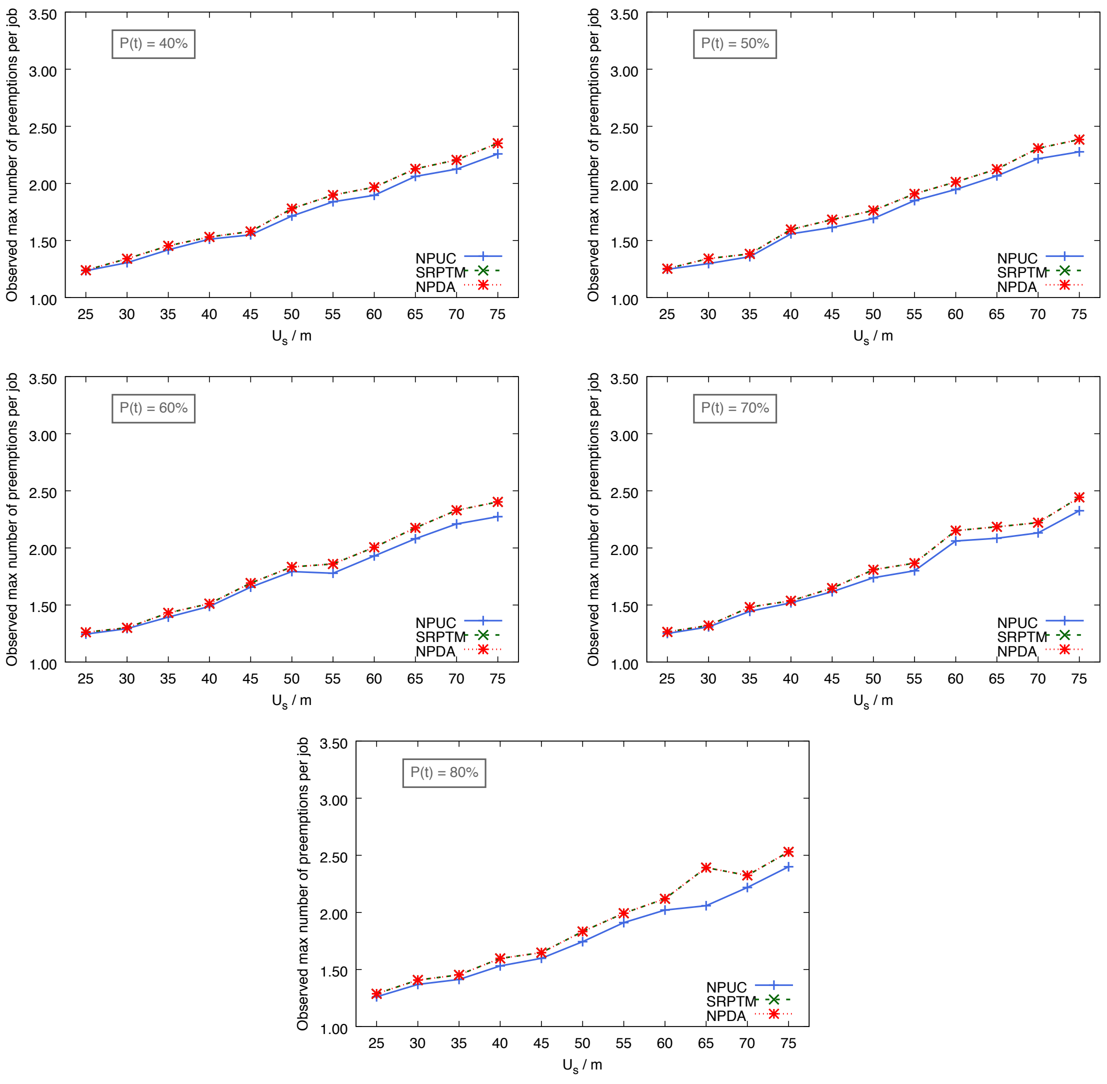


Figure 9.13: Maximum number of preemptions per job ($m = 16$).

The results show that, for 2 and 4 cores, NPDA presents the highest overheads for the majority of task set profiles, while SRPTM presents the lowest overheads for the majority of task set profiles. However, NPUC predominantly shows the lowest overheads when the number of cores increases.

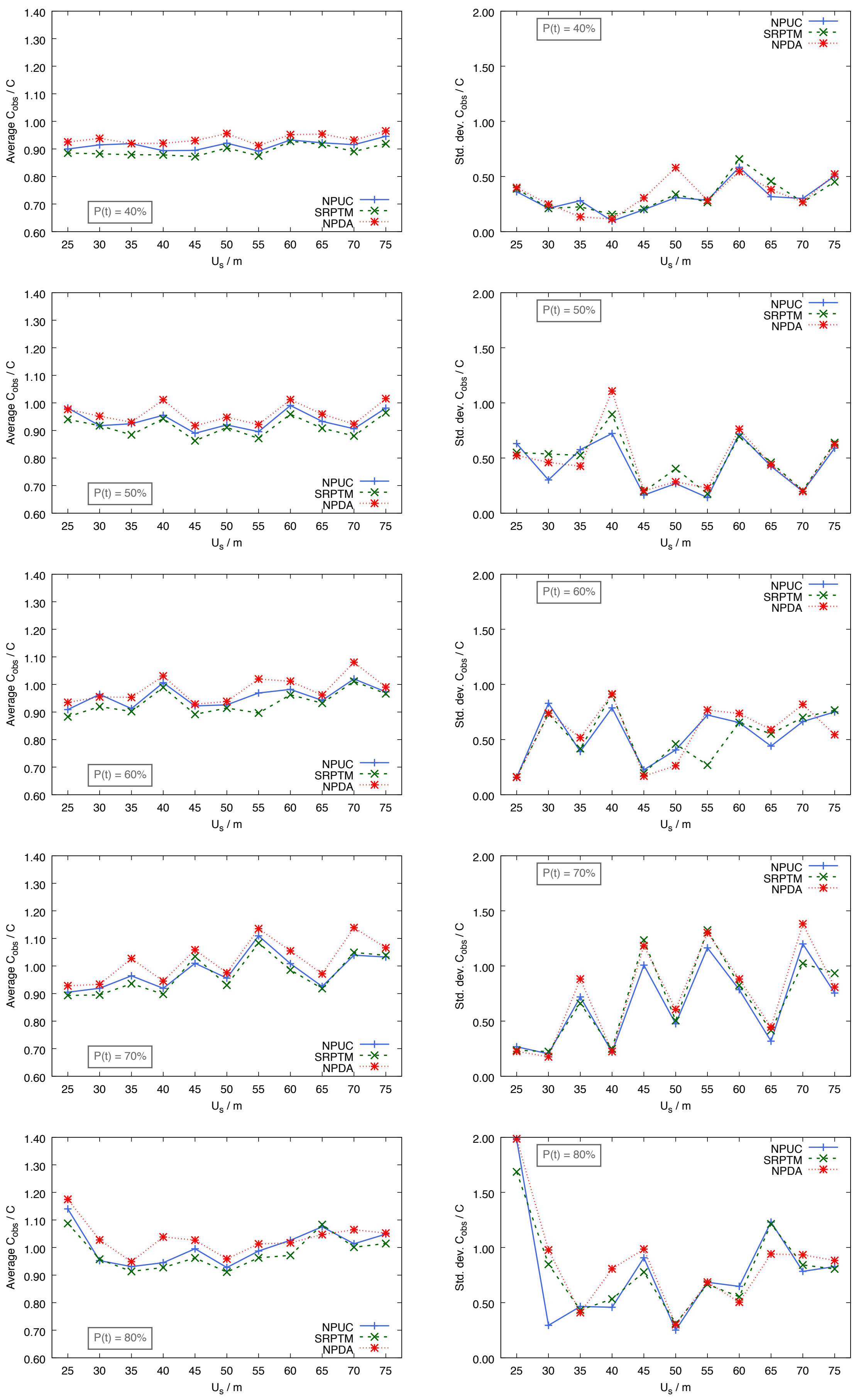


Figure 9.14: Execution time overheads ($m = 2$).

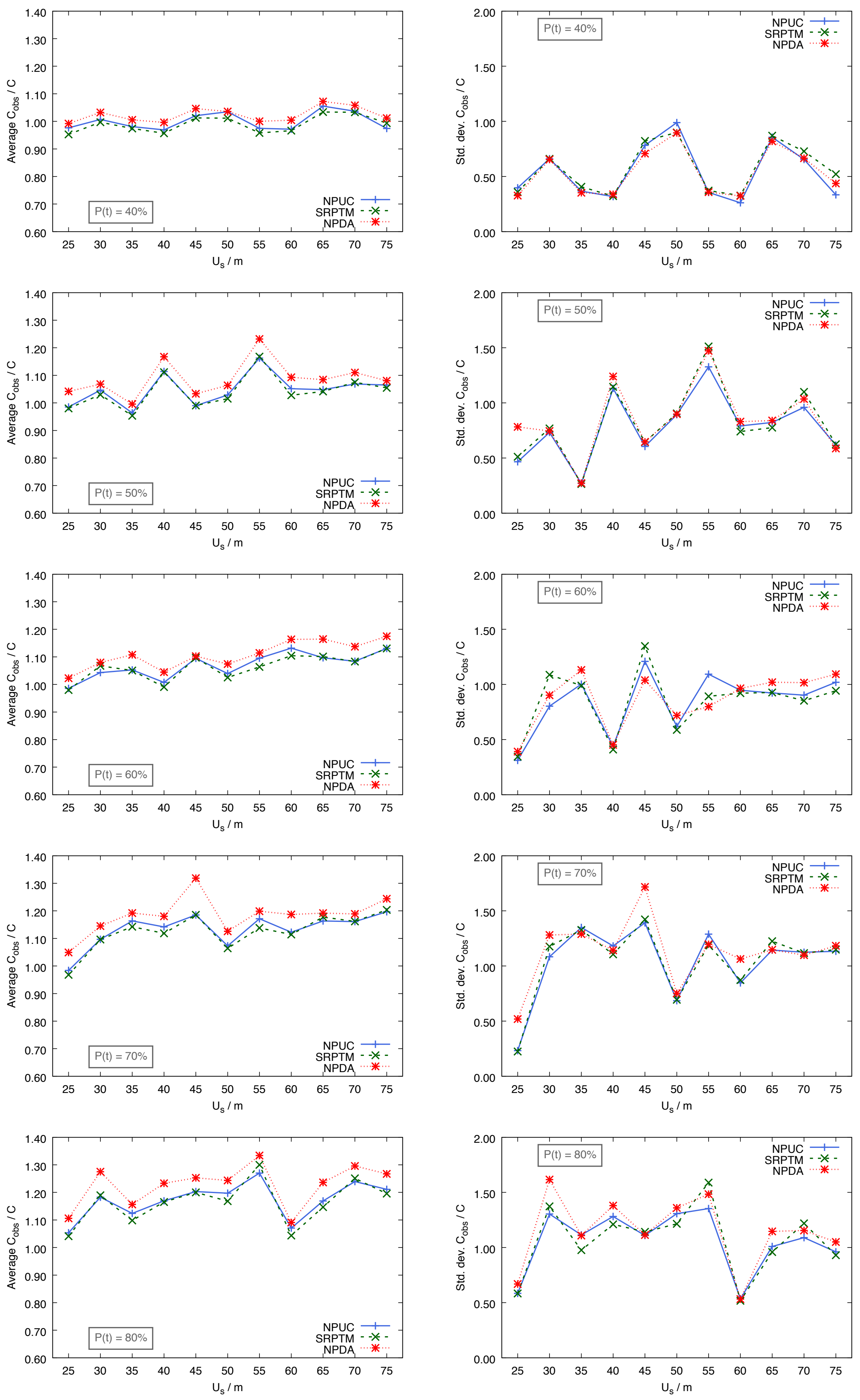


Figure 9.15: Execution time overheads ($m = 4$).

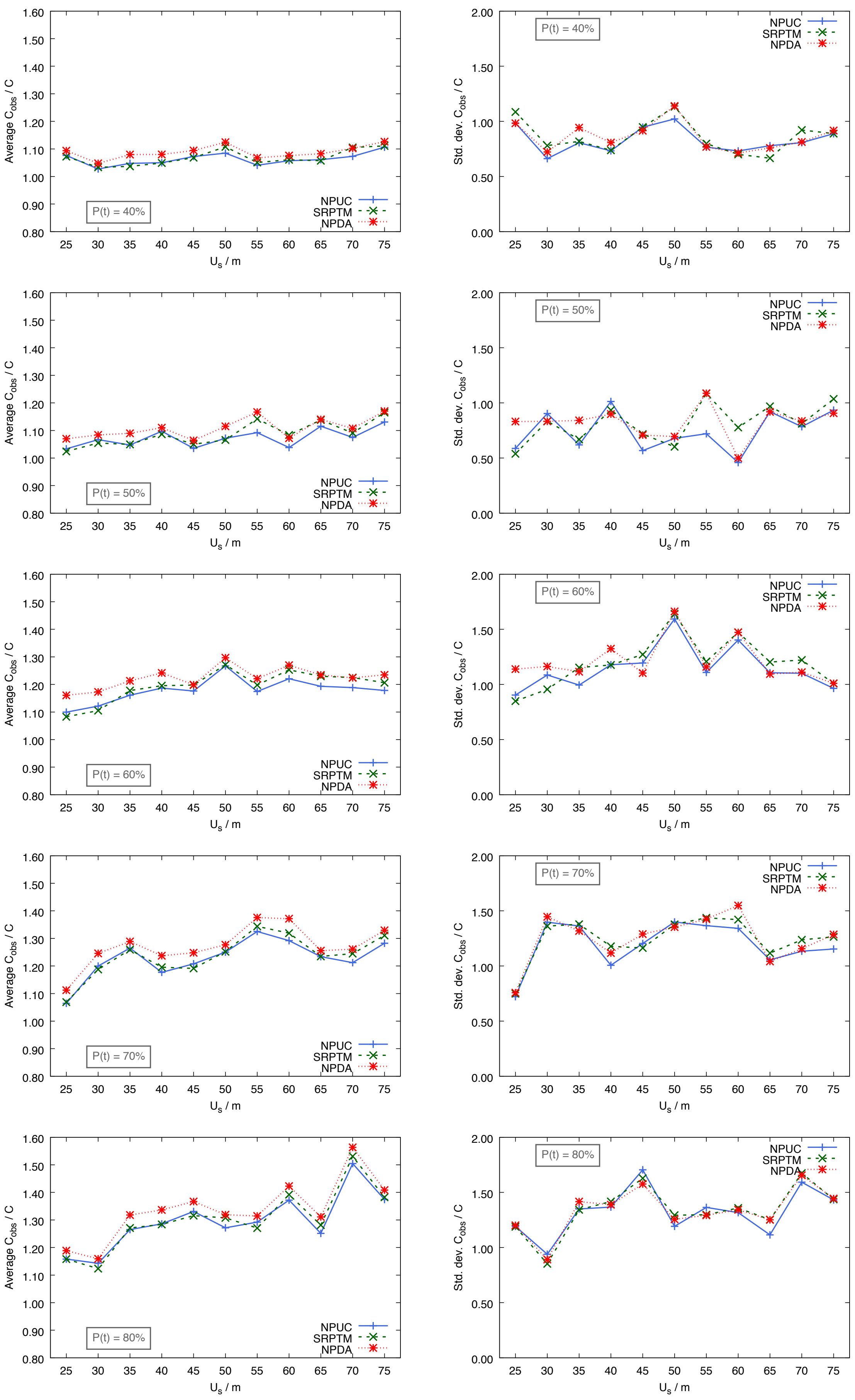


Figure 9.16: Execution time overheads ($m = 8$).

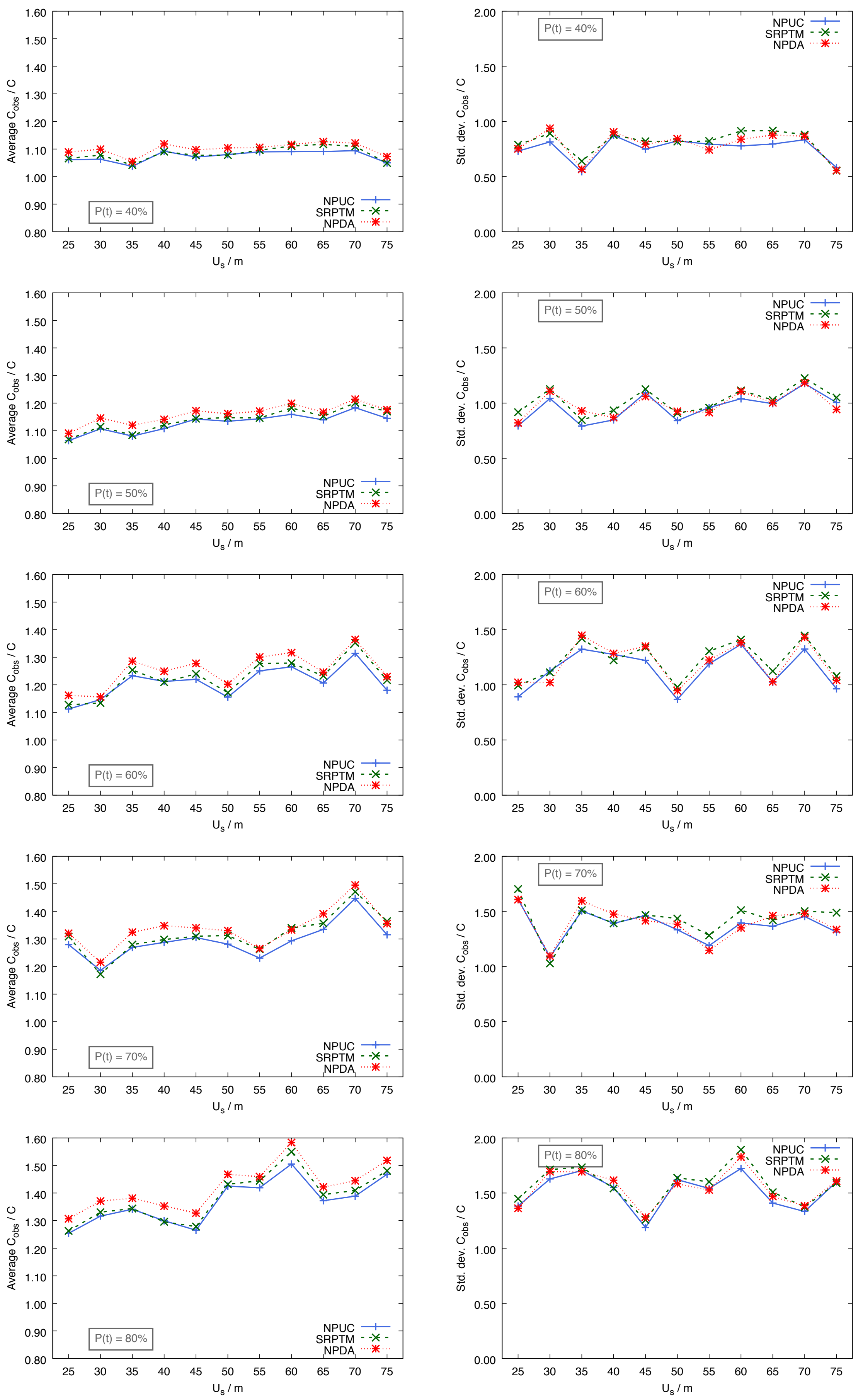


Figure 9.17: Execution time overheads ($m = 16$).

Workload. At a broader scale, we observed how the different scheduling strategies would affect the demand of system at runtime. For each task set, we determined the workload as the ratio between the actual processor demand and the processor capacity. Finally, for each profile, we computed the average workload as the sum of the workloads of the task sets in the profile over the number of tasks sets.

NPDA consistently presents the largest workload in all observed profiles, as compared to NPUC and SRPTM. This result indicates that NPDA requests more system capacity than NPUC and SRPTM to carry out the same amount of work. This behaviour can be explained by the amount of interference among jobs with a higher priority than the executing one when the transaction aborts, before it eventually commits.

Responsiveness of each task. We measured the responsiveness of each task as the ratio between its observed worst-case response time over its period. This metric reveals how the scheduling strategies handle the timing requirements of the task set, such that the tasks can meet their deadlines. The smaller the ratio, the more responsive is the scheduler.

Although the three schedulers present close ratios for all the task set profiles, NPDA consistently revealed the highest ratios. On the other end, SRPTM consistently presents the lowest ratios, providing the best responsiveness in the worst case for the considered task sets. This result indicates that SRPTM is effectively a promising candidate towards the responsiveness of tasks. It takes into account the urgency of ready jobs that may be blocked when a transaction is in progress.

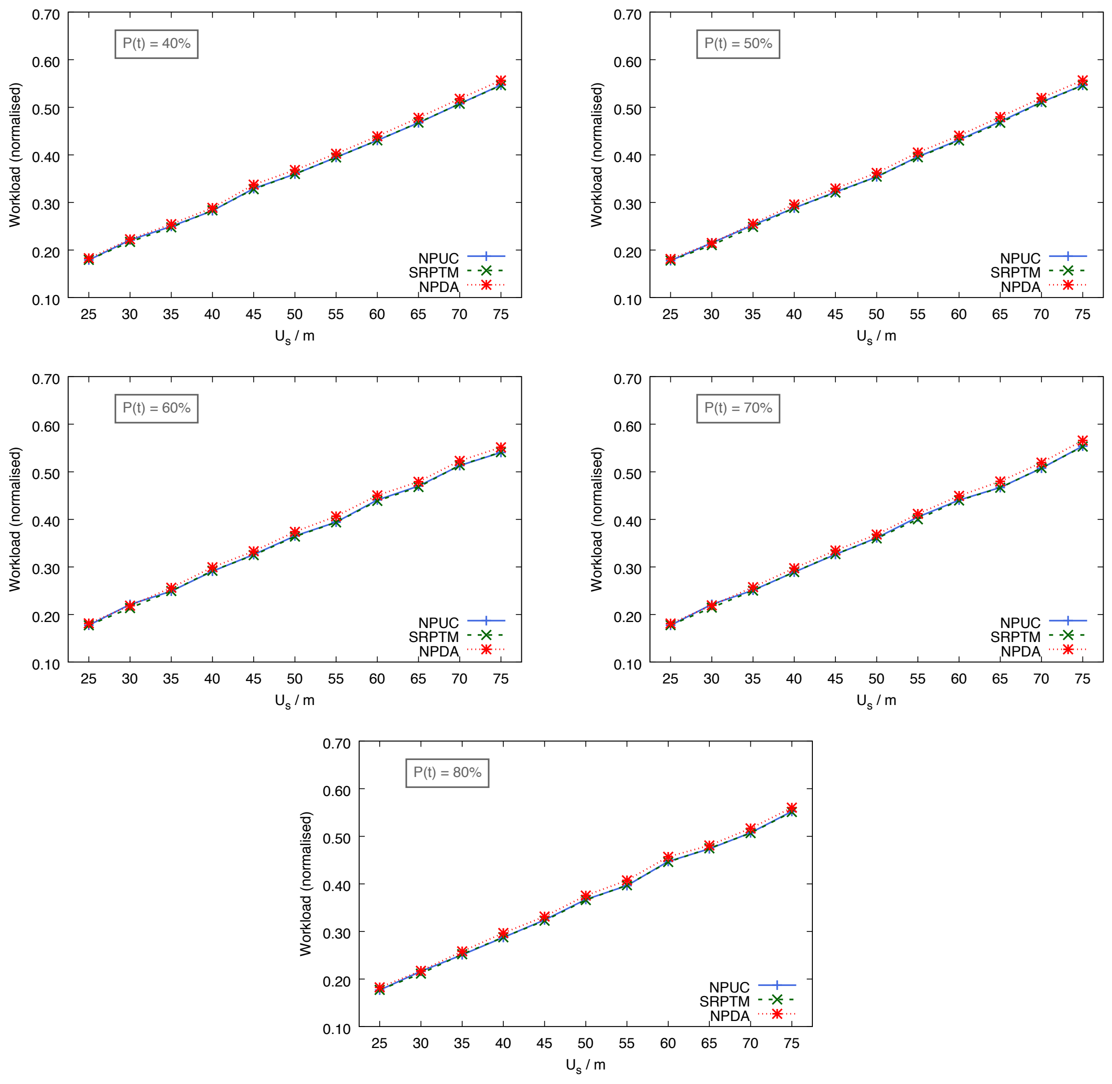


Figure 9.18: Workload (normalised by the number of tasks) ($m = 2$).

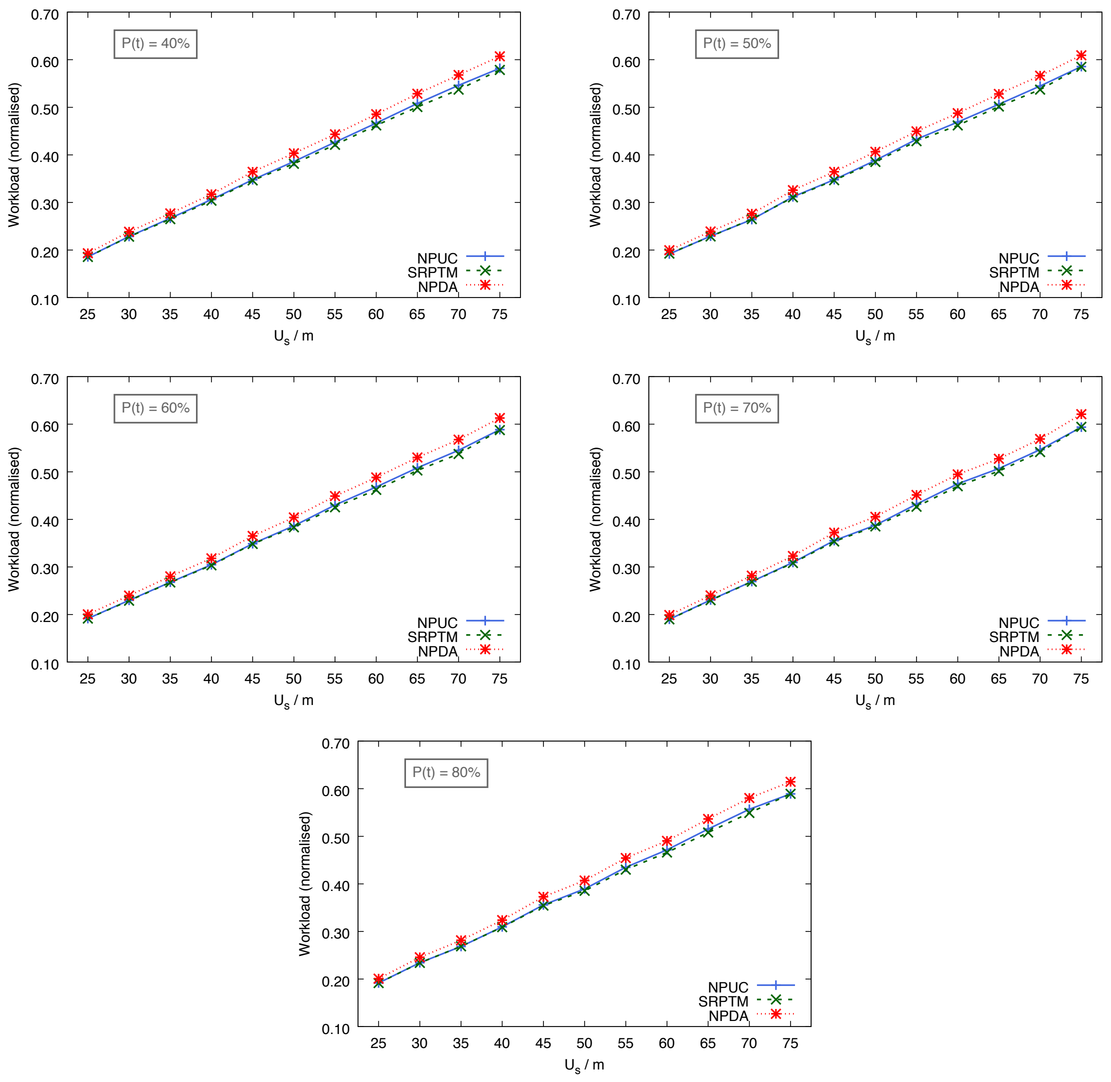


Figure 9.19: Workload (normalised by the number of tasks) ($m = 4$).

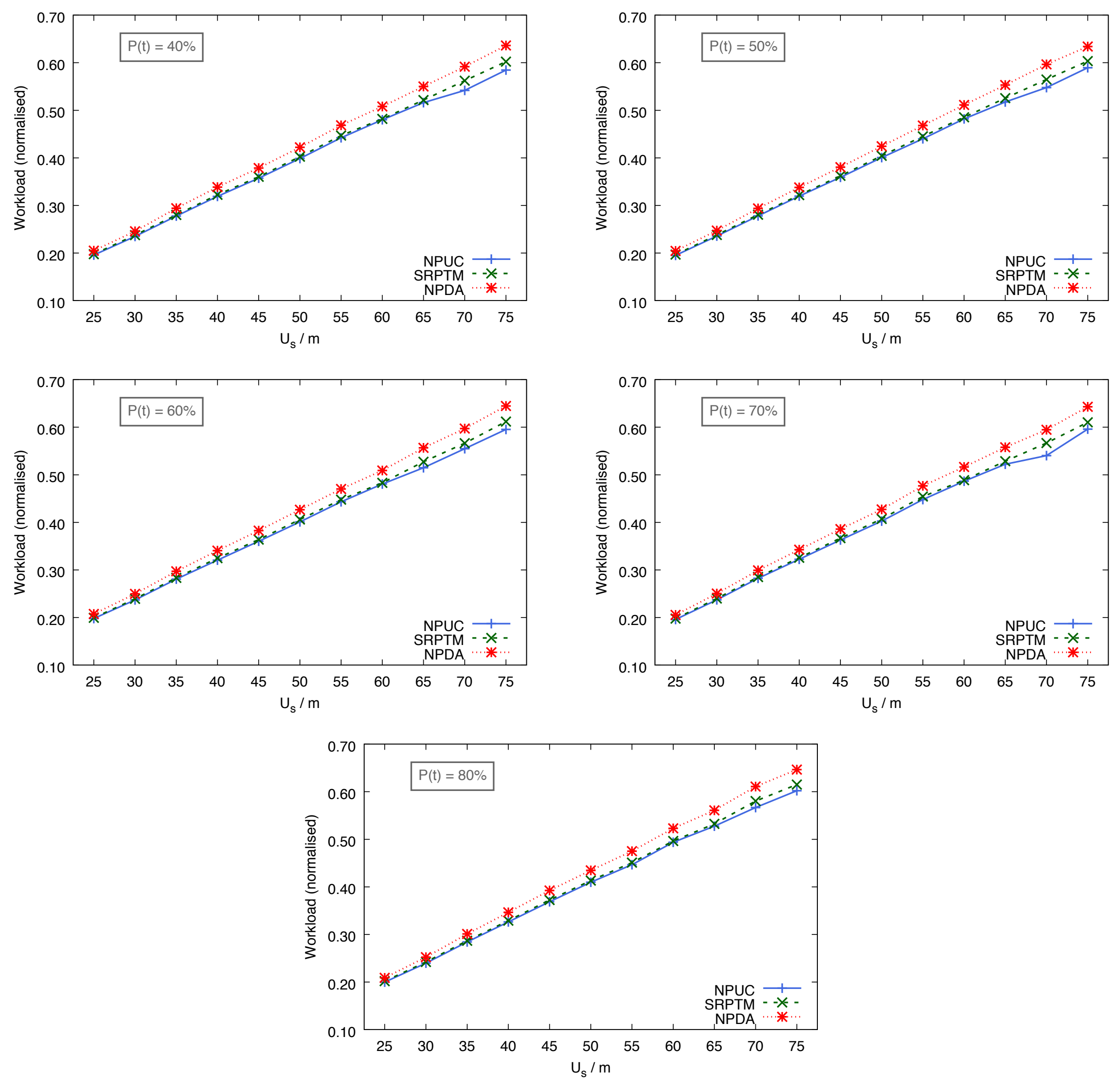


Figure 9.20: Workload (normalised by the number of tasks) ($m = 8$).

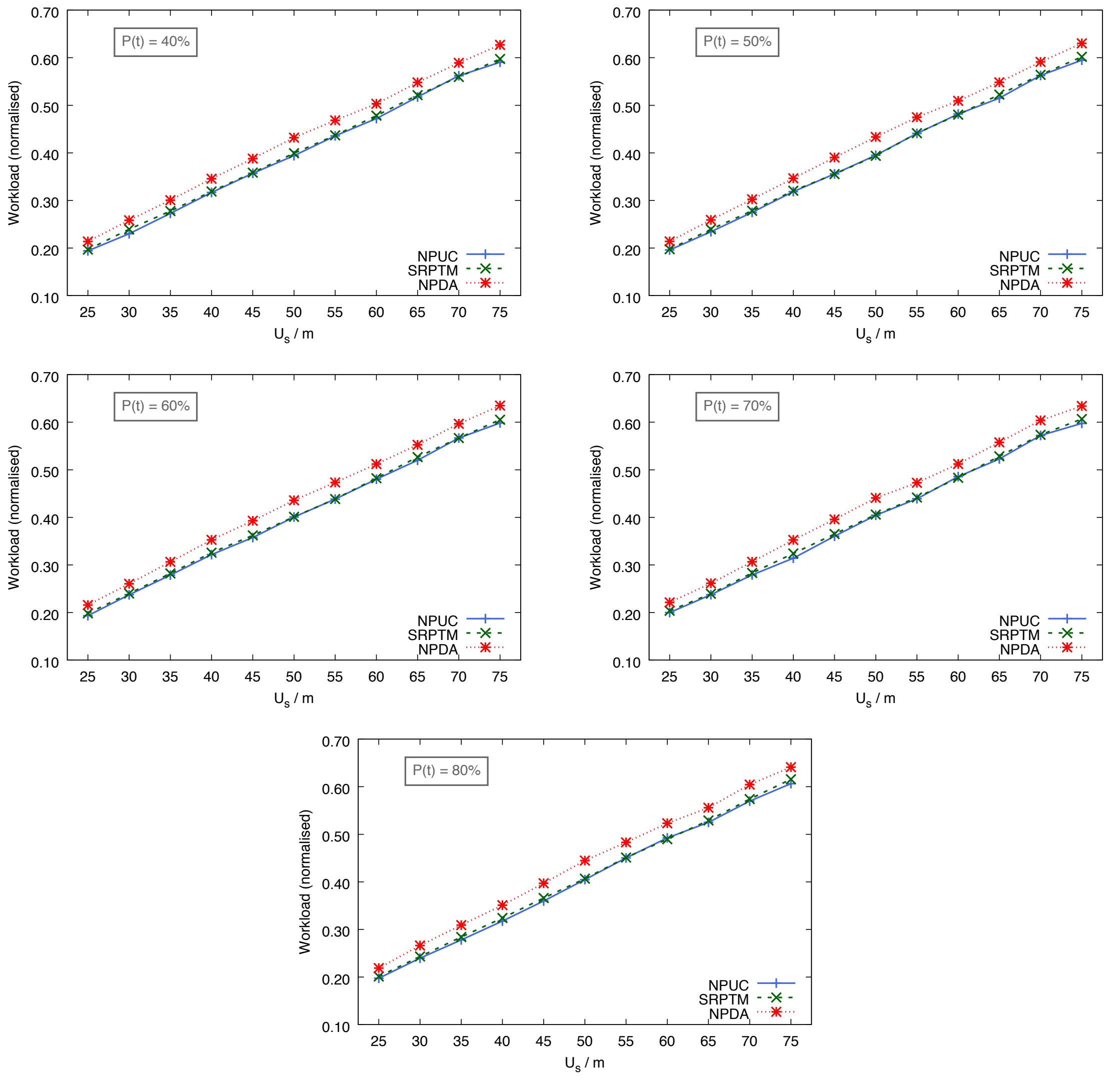


Figure 9.21: Workload (normalised by the number of tasks) ($m = 16$).

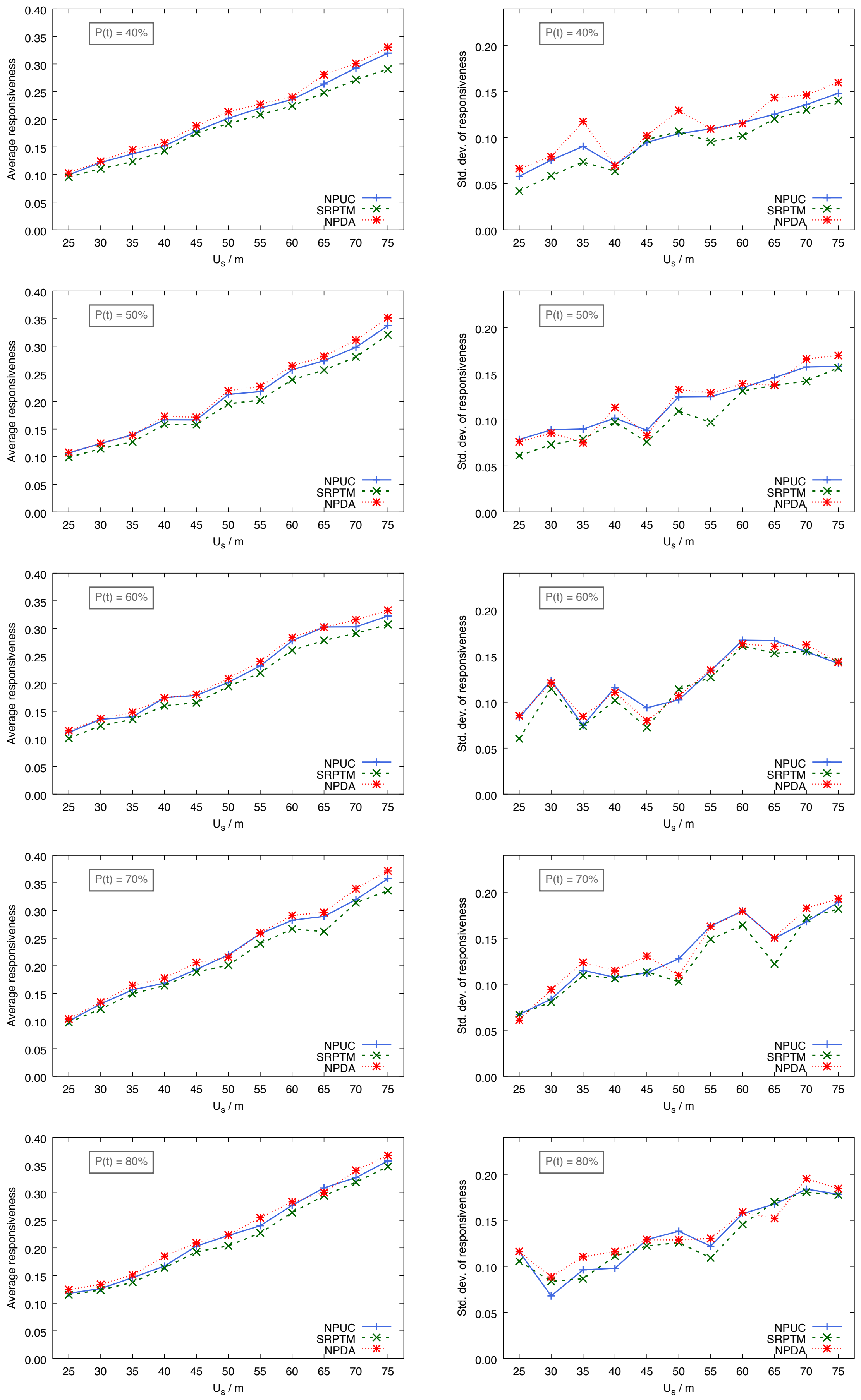


Figure 9.22: Responsiveness $\frac{RT}{T}$ $(m = 2)$.

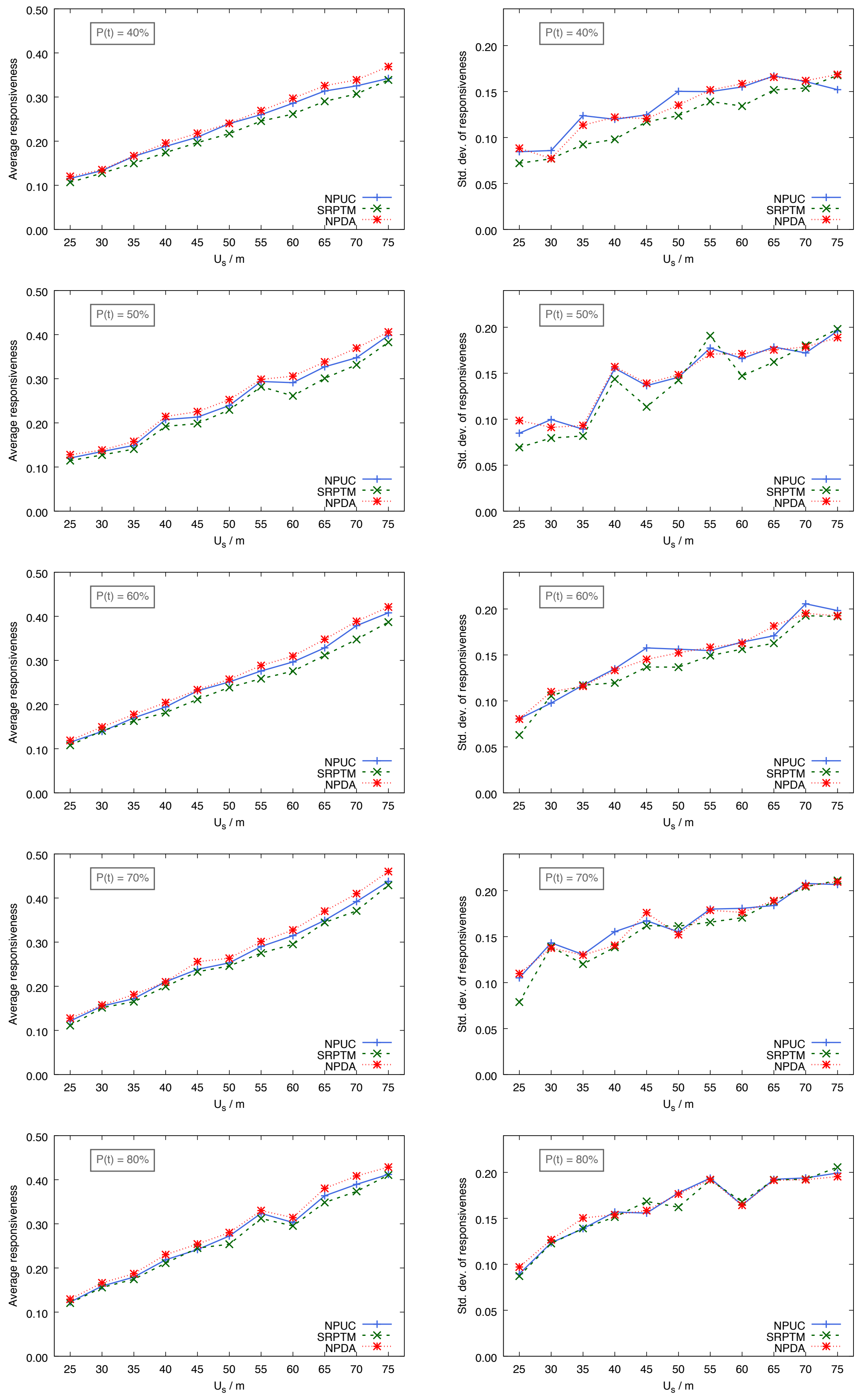


Figure 9.23: Responsiveness $\frac{RT}{T}$ $(m = 4)$.

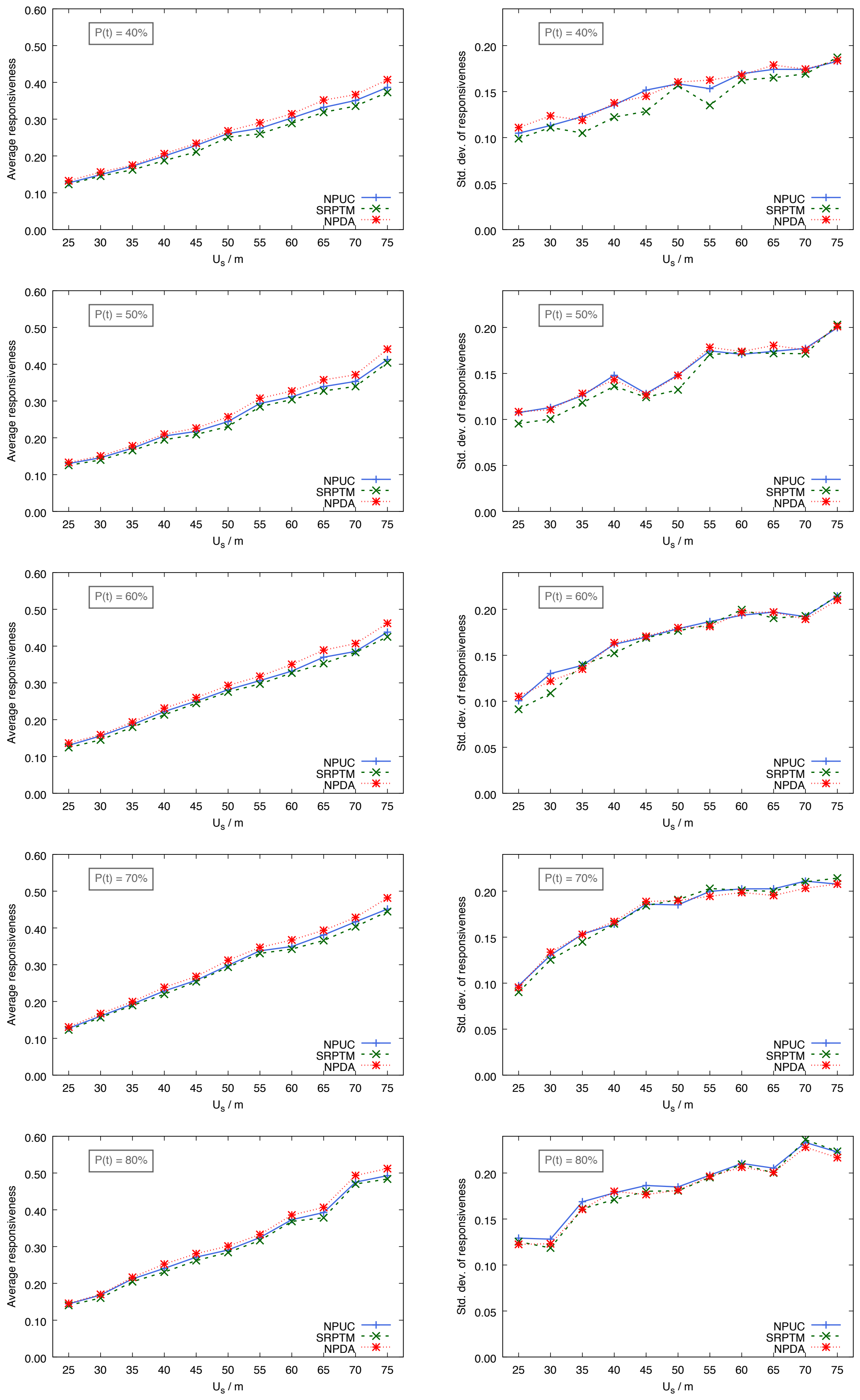


Figure 9.24: Responsiveness $\frac{RT}{T}$ $(m = 8)$.

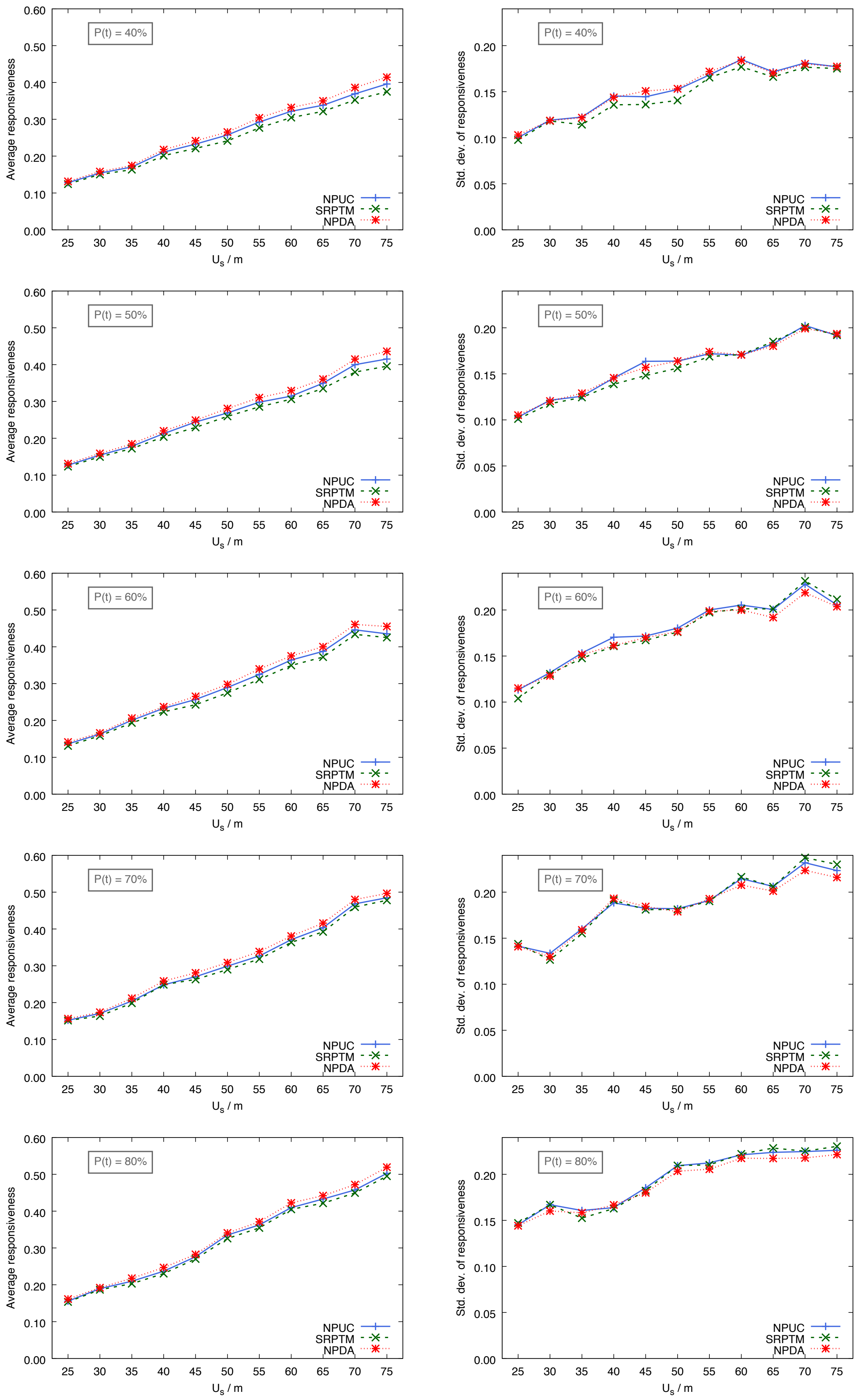


Figure 9.25: Responsiveness $\frac{RT}{T}$ $(m = 16)$.

9.2.3 Response Time Analysis (RTA) accuracy

In Chapter 7 we provided two analytical methods to determine upper bounds on the worst-case response time for transactions scheduled by using NPUC and SRPTM. Section 7.1 discussed the impracticality of a worst-case response time analysis for transactions scheduled by NPDA. The WCRT analyses for transactions scheduled by usign NPUC and SRPTM introduce non-negligible pessimism.

Accuracy of the RTA at the transaction level. In this paragraph, we compared the analytical worst case response time for each transaction against the observed worst case response time as follows: $\frac{WCRT^{\omega}_{analysis} - WCRT^{\omega}_{observed}}{WCRT^{\omega}_{observed}}$. Figures 9.26 to 9.29 present this comparison. The observed results illustrate the pessimism in the analysis of NPUC and SRPTM, as expected. On average, for the majority of task set profiles, the analytical worst response times are 20 to 60 times higher than the observed worst response times. SRPTM adds slightly more pessimism than NPUC, which can be explained as follows.

- the analysis for SRPTM yields results that are not smaller than those provided by NPUC, and
- the observed WCRT for transactions scheduled by SRPTM are smaller as compared to the WCRT for transactions scheduled by NPUC (see Section 9.2.2).

Accuracy of the RTA at the task level. In contrast to the analysis conducted in the previous paragraph targeting just the transactions, now we compare the analytical worst-case response time for each task against the corresponding observed value. We recall that the WCRT analyses for each task for the NPUC and SRPTM policies are based on the approach proposed by Spuri (1996). This analysis requires the processor utilisation to be less than or equal to one, i.e. $\sum C_i/T_i \leq 1$, to converge. In the described analyses, we assume that each job is scheduled as long as the transaction is in progress, then the WCET for any task τ_i is approximated by the execution time of the non-transactional parts of the task, summed with the analytical WCRT of the transaction. The pessimism that the transaction WCRT analyses add, inflates the task WCET. This additional pessimism results, in some cases, to a computed processor utilisation that exceeds the convergence requirement limit. In such cases, the WCRT analysis is not possible for the task subset allocated to the core. Figures 9.30 to 9.33 present the ratio of tasks that were analysed for each task set profile.

We observe that the ratio of tasks analysed decreases when the ideal system utilisation[1] increases. This trend was expected as the slack of the system to accommodate the pessimism in the analysis decreases. We also observe that the ratio slightly decreases with an increase in the number of tasks executing a transaction (i.e. P_t). Informally speaking, this phenomenon can be explained by the fact that the higher the number of transactions allocated to a core, the higher the pessimism induced in the analysis.

[1]The ideal system utilisation is the system utilisation assuming that no transaction ever aborts.

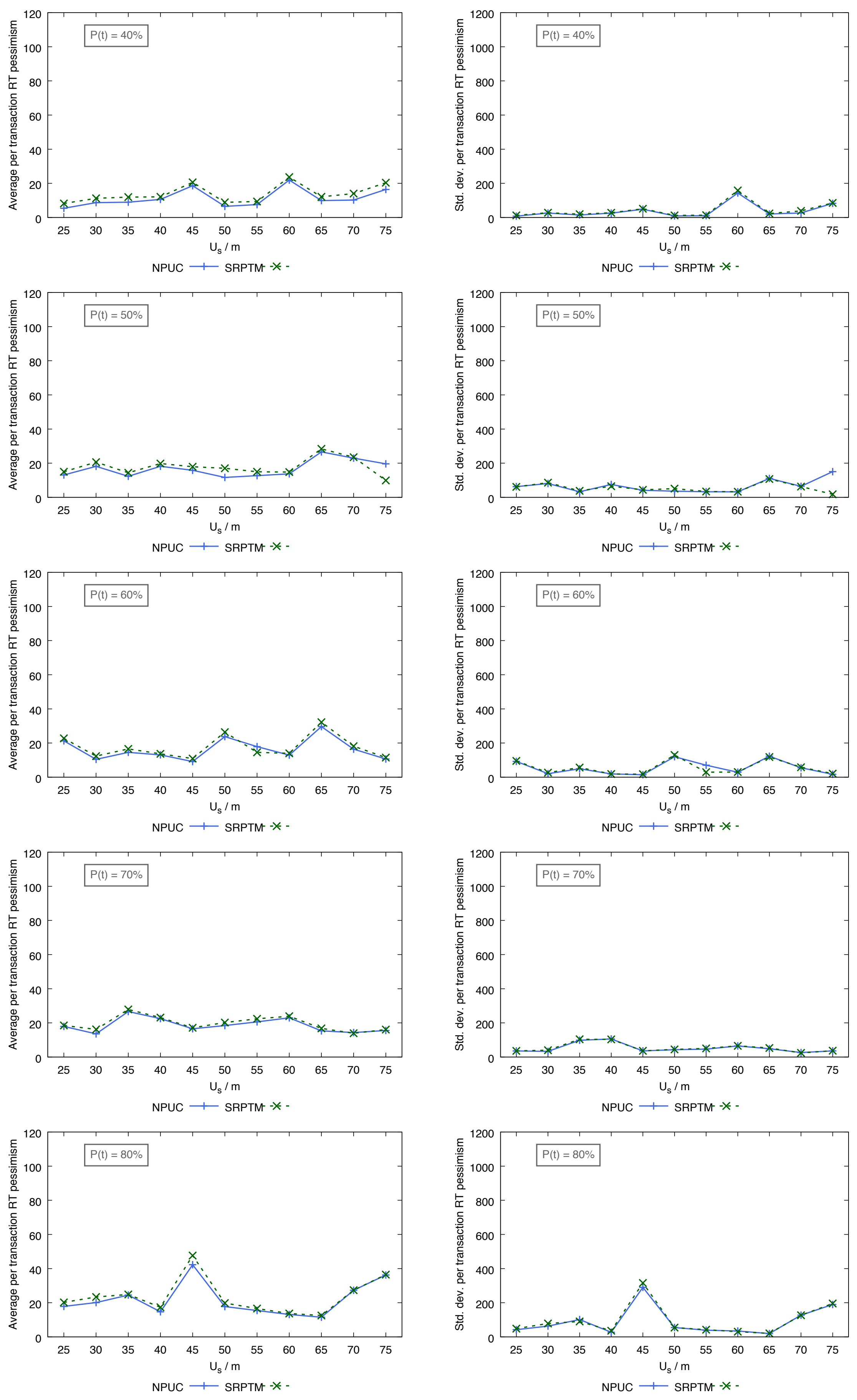


Figure 9.26: Observed pessimism for transaction RT analysis ($m = 2$).

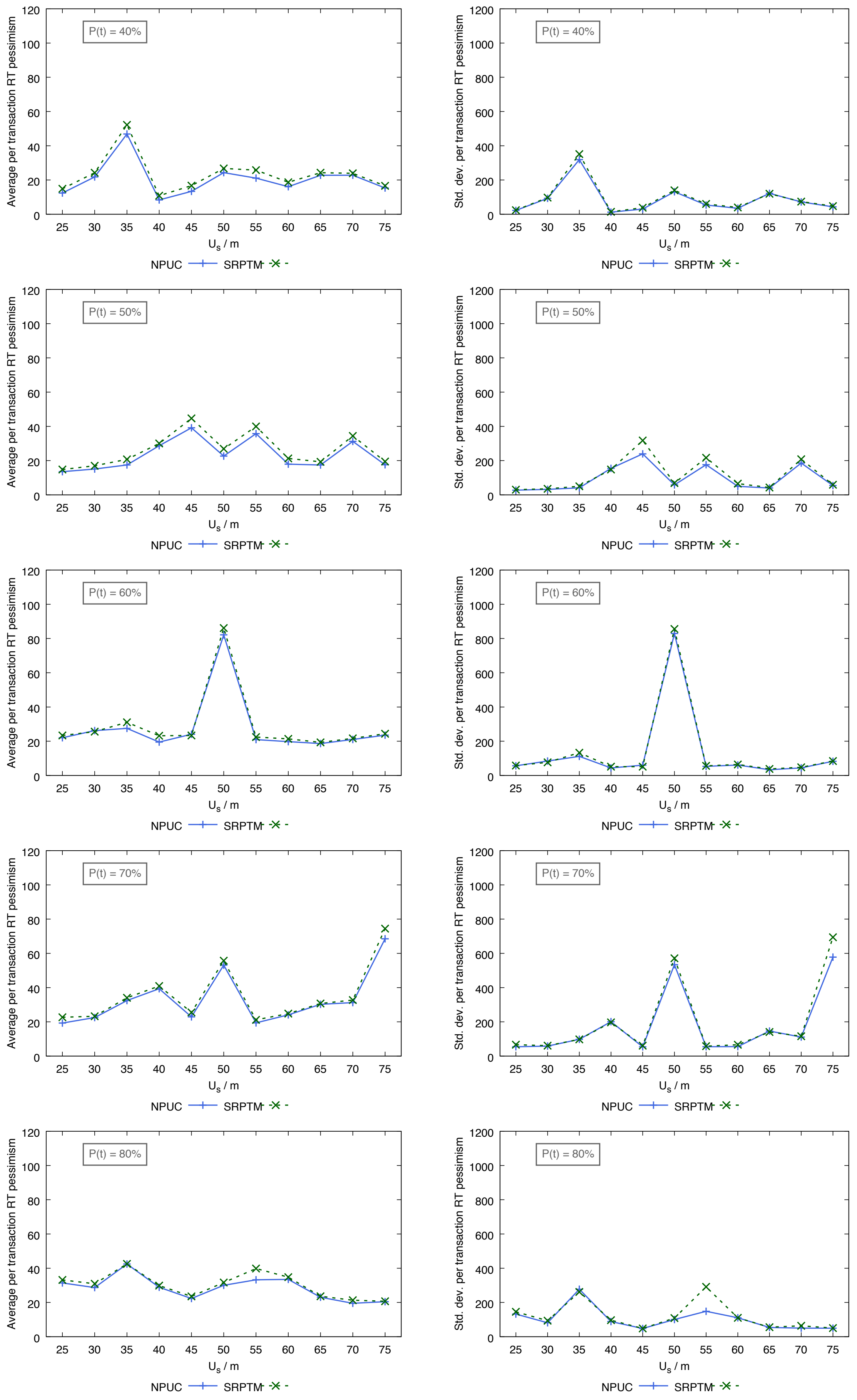


Figure 9.27: Observed pessimism for transaction RT analysis ($m = 4$).

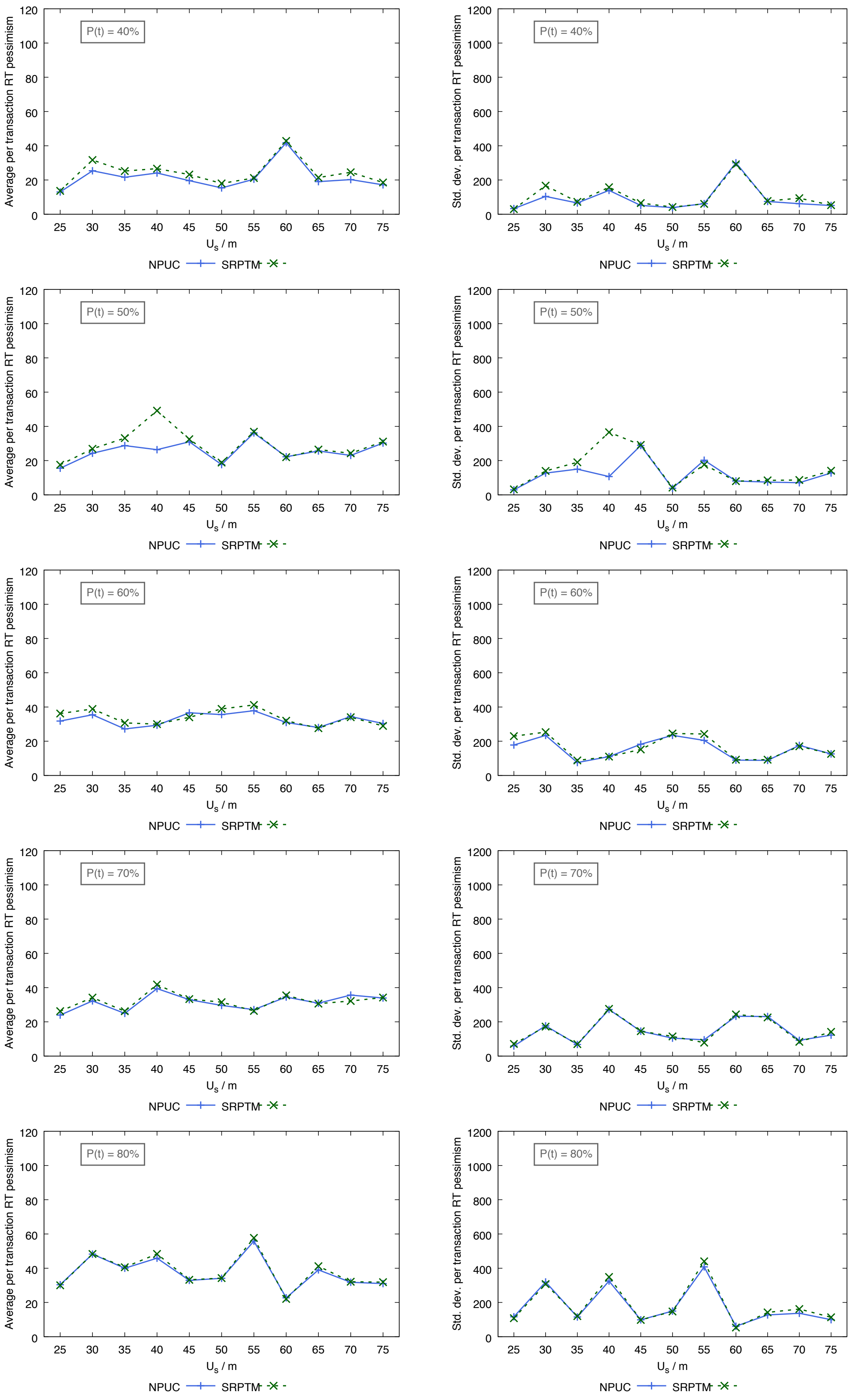


Figure 9.28: Observed pessimism for transaction RT analysis ($m = 8$).

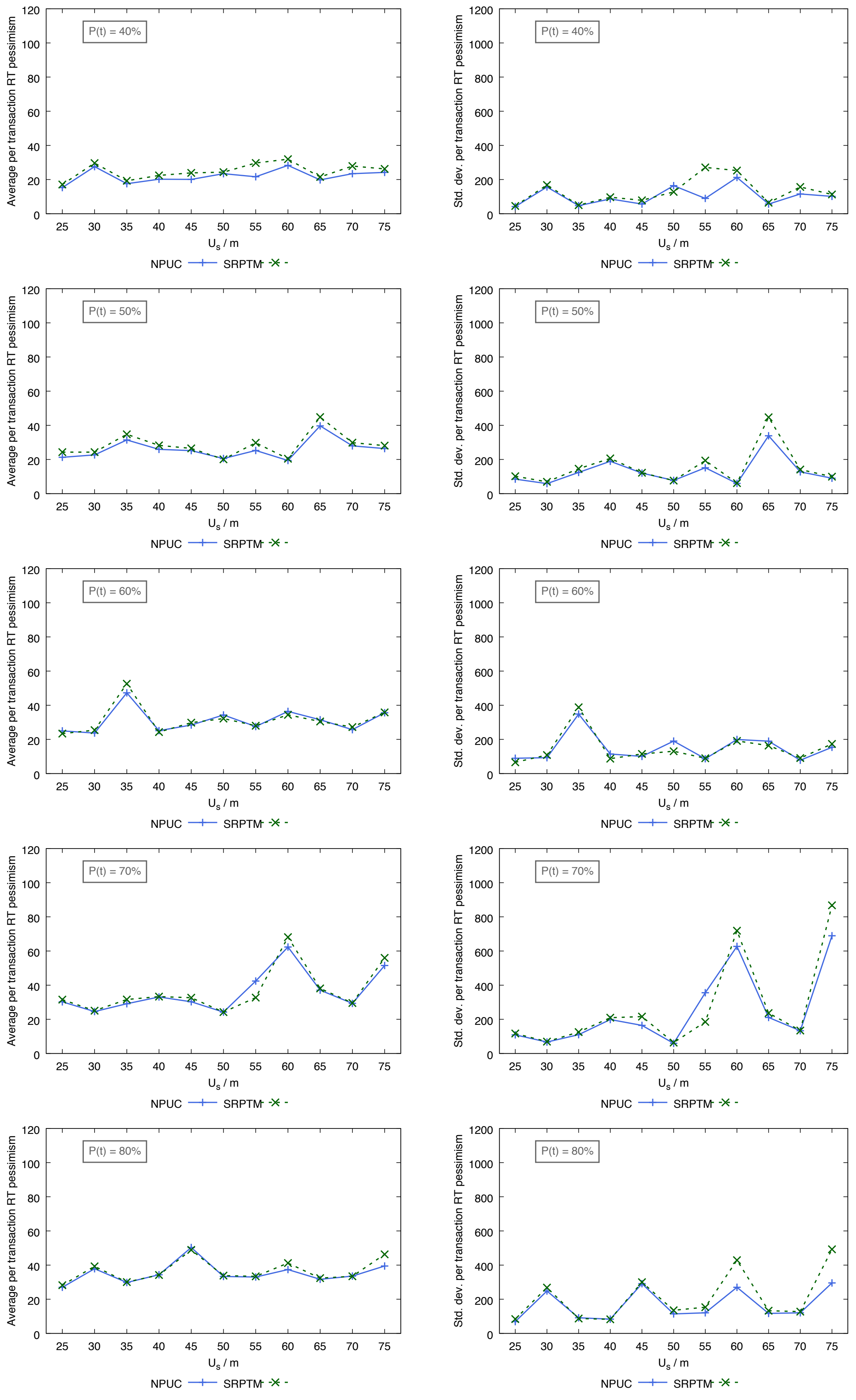


Figure 9.29: Observed pessimism for transaction RT analysis ($m = 16$).

The most noticeable trend in the figures is the decrease of the ratio of tasks analysed as the number of cores increases. In this case, the length of the sequences of transactions that may conflict becomes longer. Such long sequences result in higher upper bounds on the transaction WCRT.

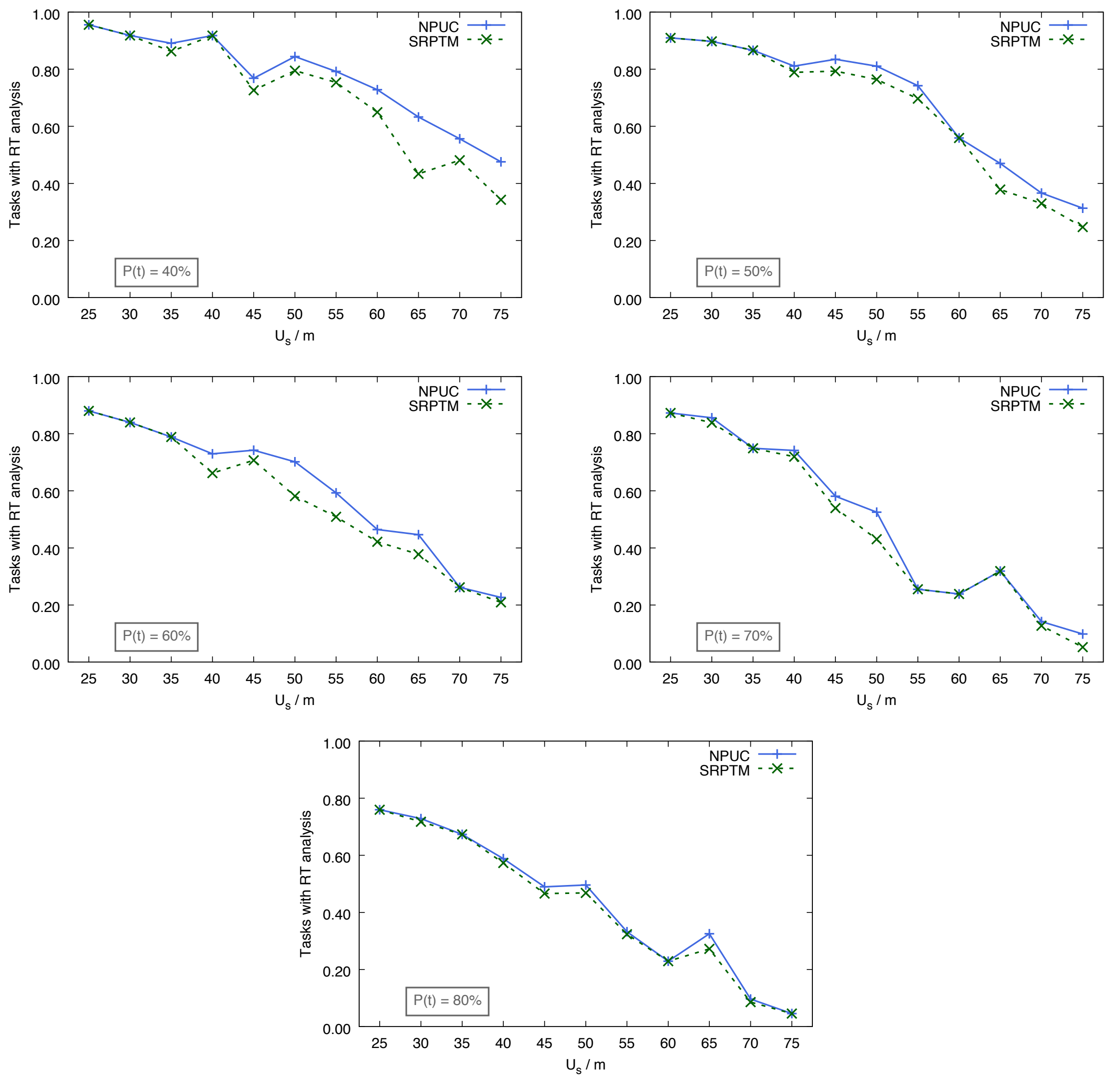


Figure 9.30: Ratio of tasks that were RT analysed ($m = 2$).

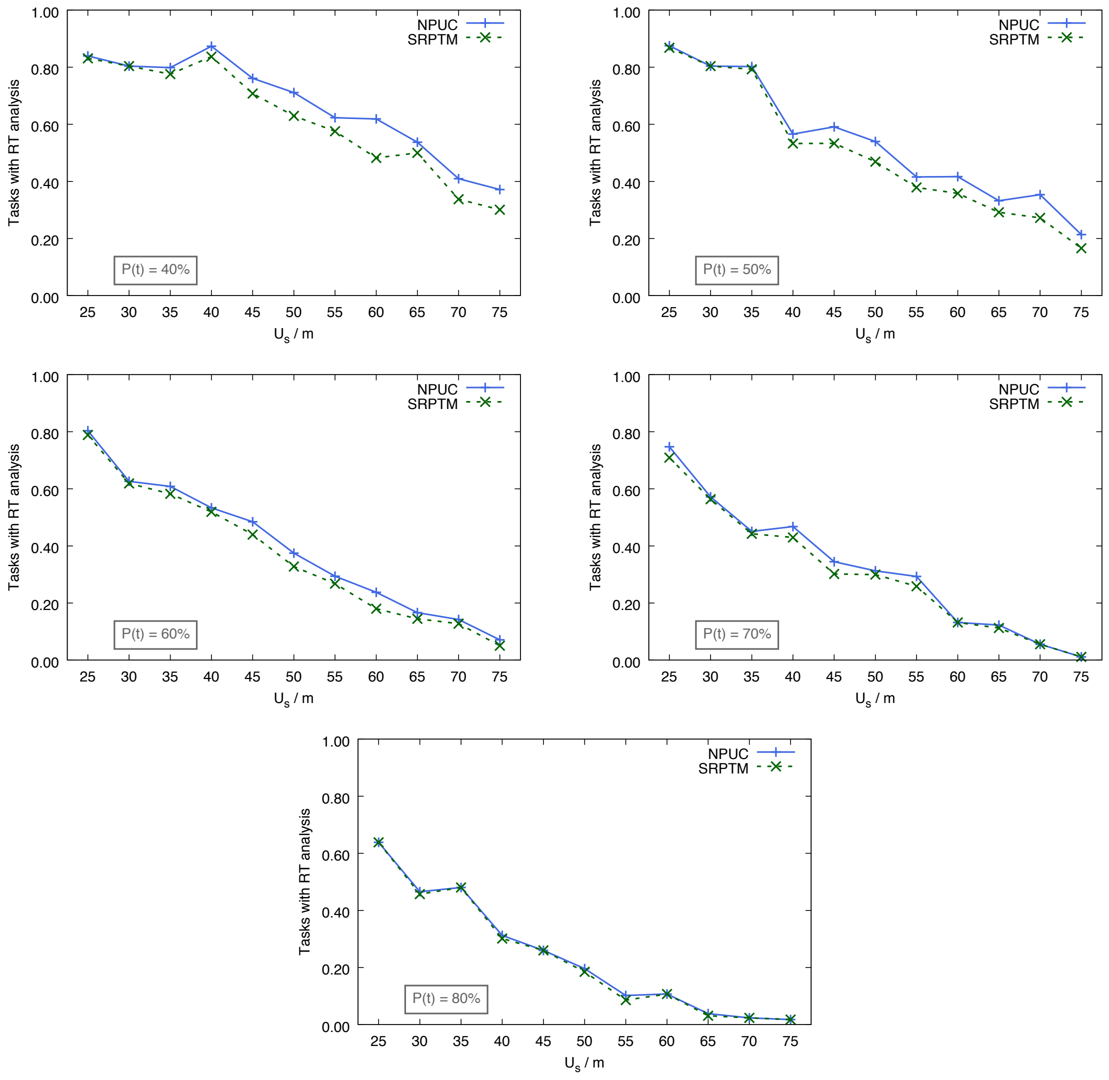


Figure 9.31: Ratio of tasks that were RT analysed ($m = 4$).

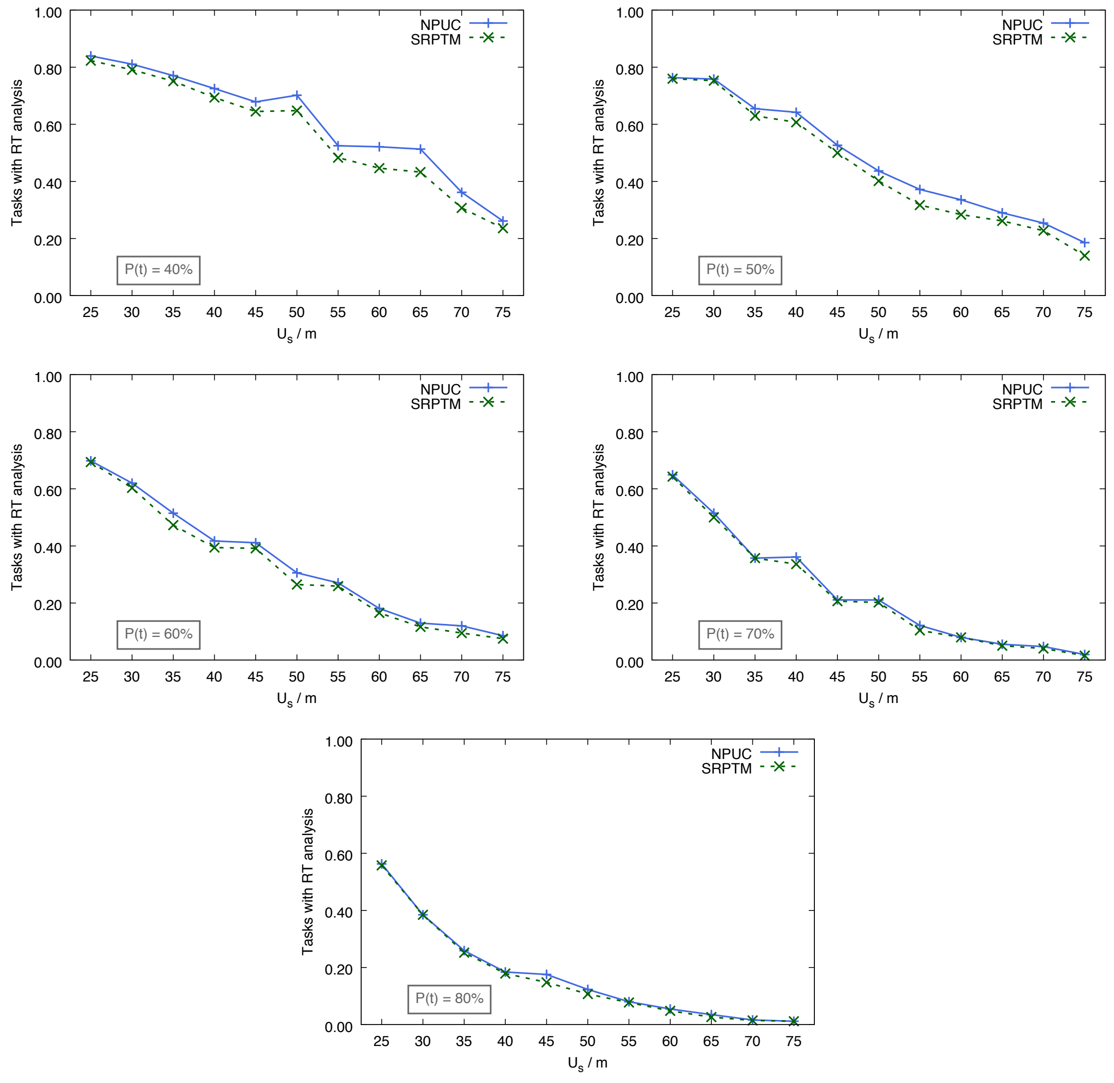


Figure 9.32: Ratio of tasks that were RT analysed ($m = 8$).

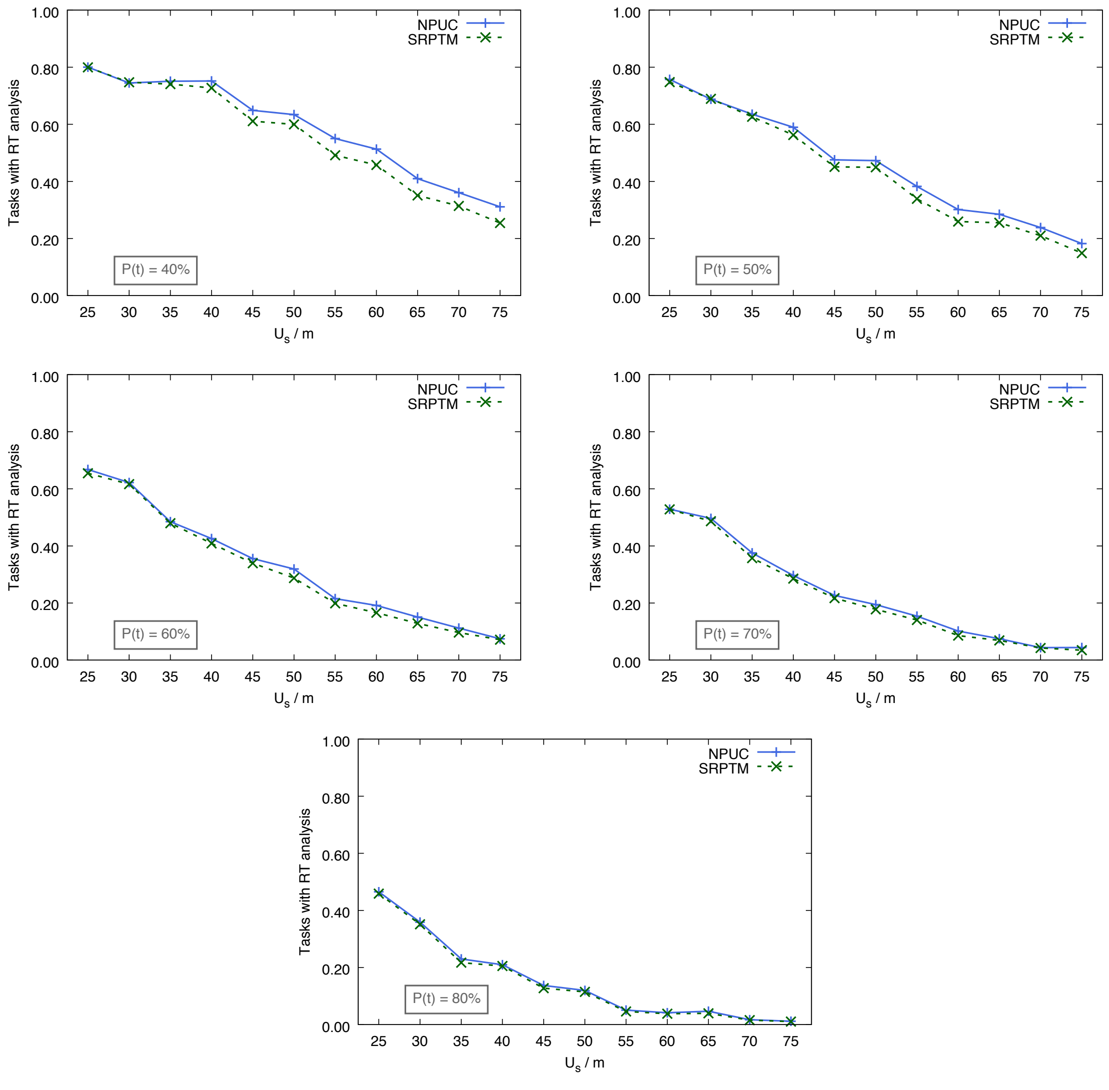


Figure 9.33: Ratio of tasks that were RT analysed ($m = 16$).

For the subset of tasks for which the WCRT analysis converged, we compared the upper bounds on the WCRT computed analytically with the observed WCRT for every task, the ratio $\frac{WCRT_{analysis} - WCRT_{observed}}{WCRT_{observed}}$. We must stress that the obtained results are not equally representative across the task set profiles, because of the significant different sample sizes between task set profiles with low system utilisation and number of cores and task set profiles with higher system utilisation and number of cores.

Figures 9.34 to 9.37 illustrate the results. We observe that the analytical results at the task level are closer to the observed values, as compared to the results obtained at the transaction level. This is due to the fact that only task sets that met the processor utilisation convergence requirement were analysed, i.e. task sets for which the pessimism induced by the computation of the transaction WCRT was not prohibitive. In almost all task set profiles, SRPTM presents the higher analytical-to-observed ratio. Again, this is an expected result as the analytical method for SRPTM adds slight more pessimism than the NPUC analysis. Furthermore, we noticed that SRPTM presents a better responsiveness than NPUC, thus increasing even more the distance between the observed and analytical WCRT values.

9.3 Summary

In this chapter we presented the implementation details of an experimental testbed based on a Linux kernel running on a 24-core platform. The NPUC, SRPTM and NPDA schedulers have been implemented and added to the kernel. A simple STM with the FIFO-CRT contention manager was developed as a user-space synchronisation mechanism. The experimental results showed that the STM performance depends more on the characteristics of the particular task set than on the scheduling algorithm. Regarding the system performance, while the results were not much distinct, SRPTM is the scheduling algorithm that presents less overheads and better responsiveness for the large majority of task set profiles. The experiments illustrate the pessimism introduced by the transaction WCRT analyses. The most influencing factor of the analysis in terms of performance is the number of cores as a larger number of transactions can conflict in this case. This pessimism undermines the task WCRT analysis since the WCET associated to each transaction is actually assumed to be the analytical WCRT of that transaction. This assumption can make the iterative task WCRT analysis diverge. Consequently, we conclude that the tasks-to-core mapping plays an important role on the outcome of the analysis. If every contention group is mapped to a minimal number of cores, the transaction and task WCRT analyses will be tighter, in the sense that less pessimism will be added.

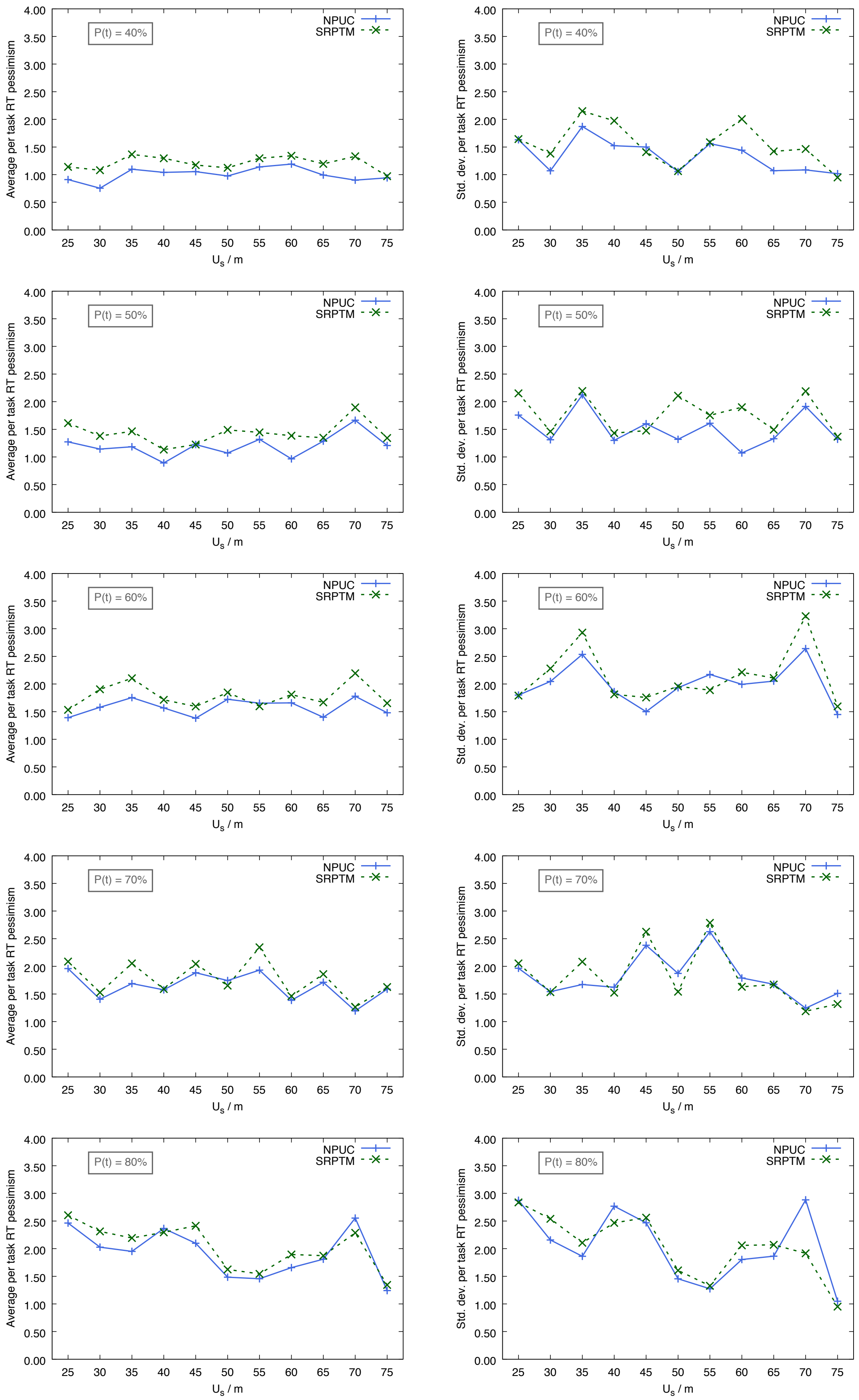


Figure 9.34: Observed pessimism for task RT analysis ($m = 2$).

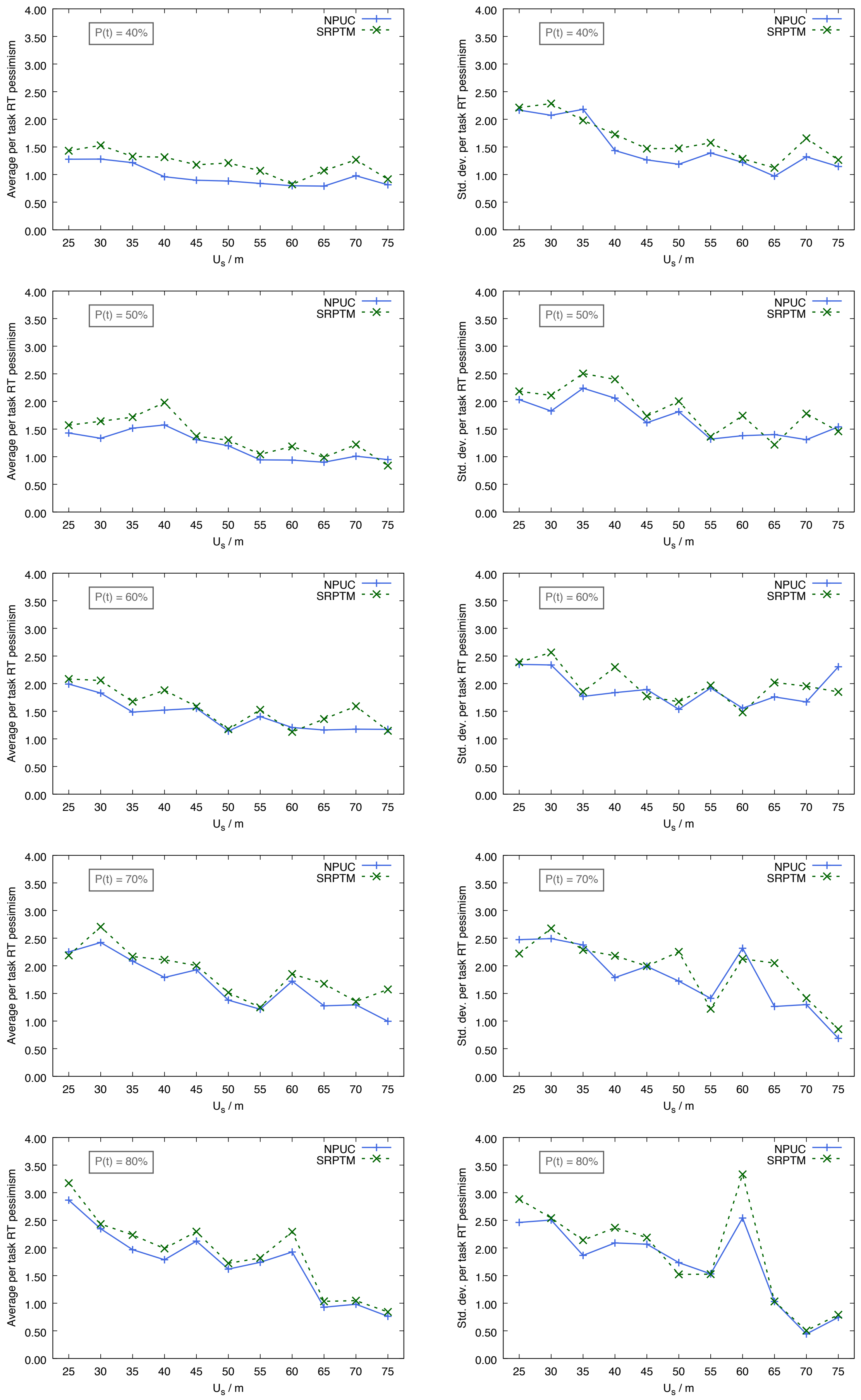


Figure 9.35: Observed pessimism for task RT analysis ($m = 4$).

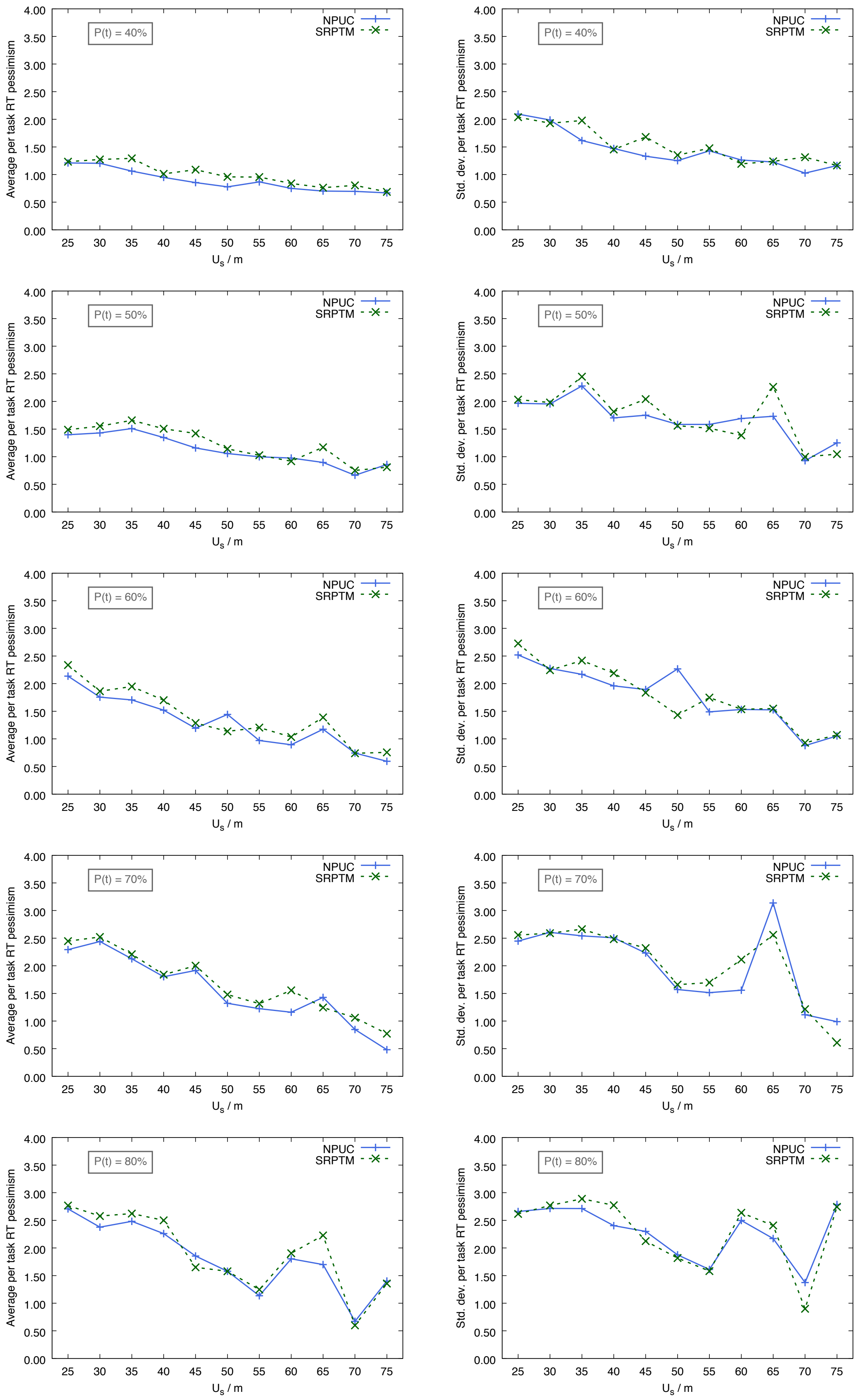


Figure 9.36: Observed pessimism for task RT analysis ($m = 8$).

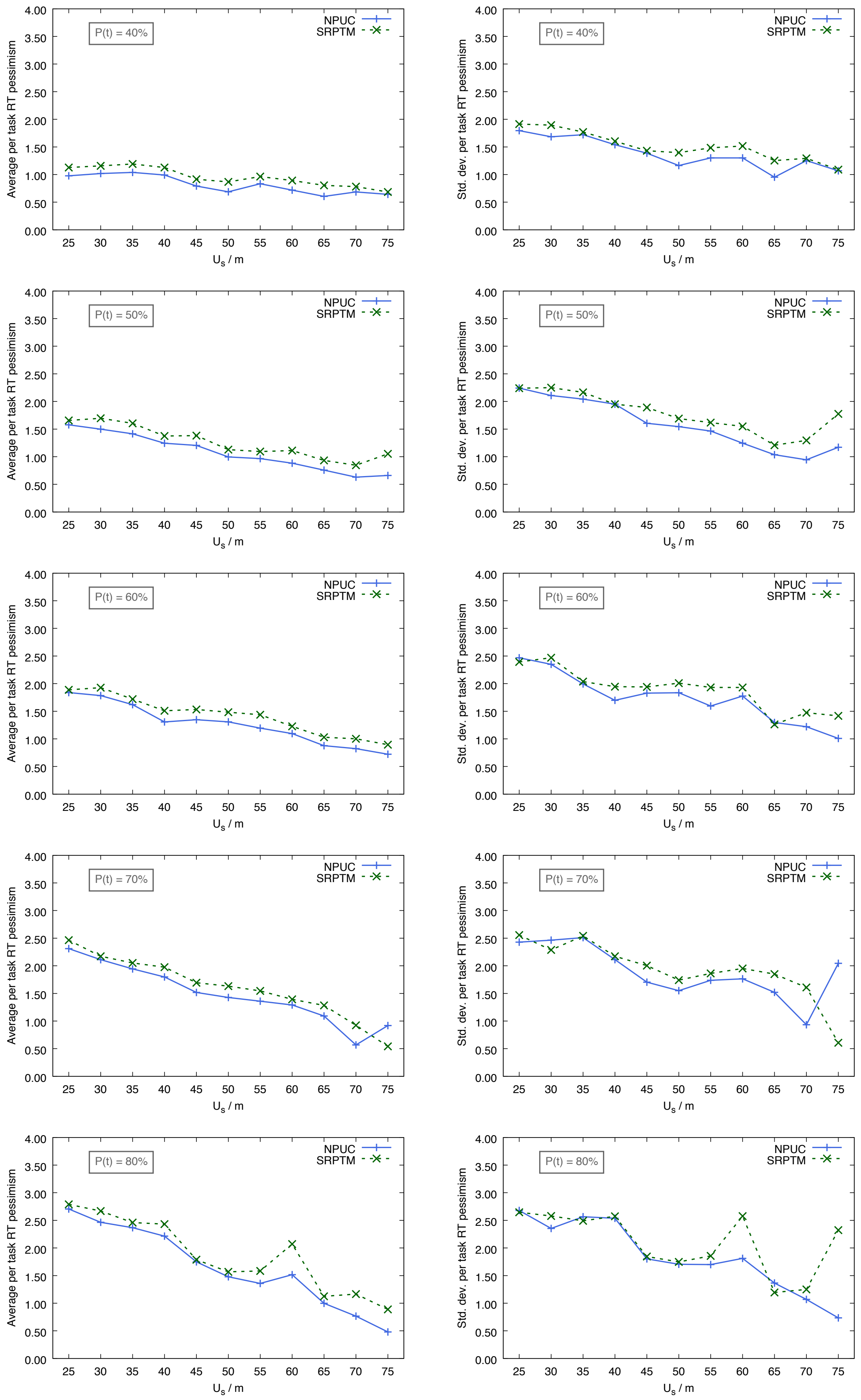


Figure 9.37: Observed pessimism for task RT analysis ($m = 16$).

Chapter 10

Conclusions

The current trend in increasing the computing capacity of a chip by adding more interconnected processing cores on the same integrated circuit has been an effective approach to deal with the physical limitations (thermal and capacitive) that single-processor chips faced. But the efficient use of the available computing capacity is heavily influenced by both the ability to design the application code as a set of components that can execute in parallel, and by the way those components are scheduled on an inherently parallel platform. This issue exacerbates even more when components are not independent and compete to access shared resources such as memory and external peripherals. While task scheduling and synchronisation techniques are matured and well established theories for single-core real-time systems, unfortunately the scale of the problem grows by several orders of magnitude when it comes to multi-core based systems as the same theory designed for single-core is no longer applicable. Indeed, in addition to the temporal dimension of *when* to schedule each task, multi-core add an extra spatial dimension of *where* to schedule this task. Furthermore, the synchronisation between these tasks should also be taken into account as the shared resources can be accessed simultaneously by concurrent tasks. From these facts, it follows that the combination of real-time scheduling and synchronisation techniques is an open research field on multi-core based platforms.

Traditional lock-based synchronisation mechanisms present serious issues when applied to concurrent applications upon multi-core platforms: coarse-grained locking does not allow to take advantage of parallel processing, while fine-grained locking negatively impacts the system composability. Software Transactional Memory offers an optimistic concurrency paradigm, in which transactions (critical sections) execute in parallel with private views of shared memory data. An underlying synchronisation mechanism determines whether the execution of each transaction is valid and commits, otherwise the transaction is invalidated, which makes it abort and deemed to repeat again. Some STM implementations apply a contention management policy to solve memory access conflicts, and several policies have been proposed along the time presenting different performances for different types of workloads. However, some of the proposed policies are known to favour some types of transactions over the others according to some specific characteristic (*e.g.* execution time, priority, number of current aborts). This approach suits well many parallel computing applications, but it does not comply with the predictability required for real-time systems. Indeed, while the burden of dealing with concurrency details is seamless to the programmer, the probability of executing multiple times the same transaction makes it challenging to provide tight upper bounds on the response time of each task. We analysed the applicability of STM as a synchronisation mechanism for real-time systems. To this end purpose, we developed a contention management algorithm that upper bounds the number of aborts for every transaction. We developed three partitioned scheduling policies – Non-Preemptive Until Commit (NPUC), Non-Preemptive During Attempt (NPDA) and SRP for Transactional Memory (SRPTM) – that tune the P-EDF scheduler when a transaction is in progress and provide us the ability to assess the response time of each transaction in a finite time window. These algorithms have been implemented on a simulation testbed as well as on a two 12-core AMD Opteron Processor 6168 based computer hosting a PREEMPT-RT-patched Linux kernel version. The performance have been compared from both qualitative and quantitative viewpoints against the Flexible Multiprocessor Locking Protocol (FMLP). FMLP is a lock-based synchronisation policy for multiprocessor systems. In this final chapter, we review and discuss the results and contributions presented in this dissertation. We also provide some directions for future work.

10.1 Summary of results

10.1.1 Fair and predictable contention management algorithm

This work proposed transaction serialisation by chronological order of transactions release times, referred to as First In First Out Contention manager for Real-Time (FIFO-CRT). Although this policy is agnostic to the underlying task scheduling policy, it provides a fair opportunity to commit to every transaction. As a matter of fact, a transaction aborts only when it is conflicting with another transaction released earlier and currently is *active* (i.e. not in the *zombie* of *failed* states). As the set of concurrent transactions that prevent the transaction to commit is finite, this set will progressively become empty as the all the concurrent transactions will commit, and then the transaction of interest will also eventually commit.

In a fully preemptive system where a transaction can be suspended and a conflicting transaction is released on the same processor, this policy is unfortunately exposed to deadlocks[1]. To address this issue, our contention management algorithm allows each transaction to abort a preempted contender that is suspended on the same core. While this rule avoids deadlock, it also undermines the main FIFO-CRT rule, i.e. *transactions are serialised by their release time instants*, and the number of aborts suffered by a transaction becomes unbounded.

The deadlock-avoidance rule becomes unnecessary when at runtime at most one active transaction is allowed to be in progress on each core. Therefore, we modified the behaviour of the

[1]A very similar situation occurs when a critical section requests a lock owned by a preempted critical section with the same processor affinity.

original scheduler (P-EDF) associated to FIFO-CRT so as to prevent multiple simultaneous transactions in progress on each core. In turn, this modification resulted in a limited number of aborts for each transaction, thus improving the responsiveness of each task.

10.1.2 Modifications performed on the original scheduler

This work proposed three fully-partitioned scheduling algorithms, all based on the classical P-EDF scheduler. Each algorithm modifies the behaviour of P-EDF when a transaction is in progress by adopting a set of rules that allow us to fully serialise the transactions, improve the predictability of the system and avoid deadlocks.

NPUC. This approach schedules each transaction without any preemption until they commit, thus ensuring that at most one transaction is in progress on each core at runtime.

NPDA. This approach also schedules each transaction in a non-preemptive manner when it is in progress. However, preemption points are inserted between every abort and the subsequent restart. In this time window, ready jobs with a higher priority can be scheduled. As a consequence, NPDA allows for multiple transactions to be simultaneously in progress on each core. In this case, preempted transactions are in the failed state and temporarily not exposed to additional aborts.

SRPTM. This approach extends the SRP mechanism introduced by (Baker, 1991). Tasks and transactions are assigned preemption levels. Only tasks with a higher priority and not executing any transaction are allowed to preempt the transaction in progress on each core.

In a simulation environment, these algorithms proved more responsive and scalable than a practical multiprocessor lock-based synchronisation mechanism that allows a granularity down to contention group level. The simulation results indicated that STM is able to dynamically set the level of contention granularity, thus allowing a better exploitation of the parallelism provided by multi-core architectures over the lock-based mechanism. Between the three proposed scheduling strategies, the simulation revealed that NPUC was able to feasibly schedule the smallest number of task sets, due to its lower responsiveness to schedule jobs with short deadlines when a transaction is in progress. SRPTM presented the worst case number of aborts for a single job, but presented the lowest average time overhead, meaning that in average, it provides a more efficient use of the computation resources.

We implemented the three scheduling approaches together with FIFO-CRT on a two 12-core AMD Opteron Processor 6168 based computer hosting a PREEMPT-RT-patched Linux kernel version by generating a large group of synthetic task sets. The results showed that SRPTM was predominantly the most responsive scheduler, while NPDA was the least responsive. In average, SRPTM presented the shortest worst-case response times for each task. In terms of practical time overheads (i.e. considering the time used for system operations), the resulting data showed that preemptions have a non-negligible impact: while SRPTM showed the smallest time overhead for 2 and 4 cores, NPUC became dominant for 8 and 16 cores, despite its reduced responsiveness.

10.1.3 Response time analysis

We developed a worst case response time analysis for transactions scheduled by NPUC and by SRPTM. We showed that an exact analysis is computationally intensive. Therefore, in order to be agnostic to the platform characteristics and task-to-core mapping, we assume that all other cores are executing the longest transaction in the same contention group assigned to it. We did not carried out an analysis for NPDA as it would be too pessimistic. Indeed, in order to collect sound and convincing results for each transaction under this scheduler, all transactions in the same contention group would need to be considered. We compared the observed WCRT of each transaction obtained from the experiments, against the corresponding WCRT resulting from the analysis. In all cases, the measured WCRT for both NPUC and SRPTM schedulers fell within the computed analytical bounds.

The transaction WCRT analysis set the basis for the complete task WCRT analysis. The task WCRT analysis was adapted from the framework proposed by Spuri (1996). To account for the time overhead incurred by the attempts to commit of each transaction, we incorporate the previous transaction WCRT into the worst case execution time of the task, thus adding some pessimism in our approach. The added pessimism becomes a particular issue when the considered tasks WCET are inflated to an extent such that the WCRT analysis no longer converges and no results can be extracted. The experimental results indicate that this issue exacerbates with an increase of the number of cores assigned to each contention group.

10.2 Future directions

We demonstrated the realism in constructing a synchronisation mechanism for real-time systems based on the Software Transactional Memory paradigm. Such a mechanism allows for the programmer to abstract from the low-level synchronisation details and focus only on the functional aspects of the code. However, this feature raises non-trivial issues in the assessment of the schedulability of the task set. A substantial number of these issues have been addressed in this manuscript, but a few more remain open and require an in-depth investigation. A few examples go below.

Task-to-core mapping. The mapping of tasks-to-cores has a significant impact either on the practical system performance and on the feasibility analysis. In this work we assume a given random mapping. Among the possible mappings, the next natural step along this line of research would be to derive heuristics to determine the mapping or the finite set of mappings that dominate the others in terms of performance as well as the responsiveness of the system. A few metrics to guide us in deriving such heuristics are the number of conflicts among the transactions and the number of cores assigned to each contention group. Regarding the former metric, it would substantially improve the schedulability of the system, whereas the latter metric would help in reducing the response time of each transaction. That is, conflicts will occur only between a smaller number of transactions, and will be solved faster at runtime. In addition, the pessimism added to the WCRT computation of each transaction is smaller.

Probabilistic analysis. An alternative to deal with the pessimism that impairs the analysis presented in this manuscript is to adopt a probabilistic approach for the actual execution time of each transaction and, subsequently, the execution time of each task. In this work, we assume the worst-case scenario in the computation of these parameters, which may never occur in practice. In order to circumvent this issue, a probabilistic task analysis might reduce the pessimism associated with the transaction execution time overheads. This assumes that the execution time for each task is not the same for all of its jobs. As a matter of fact, the number of times a transaction aborts within a job might be associated a probability. Although this increases a bit more the complexity of the problem, the results would be more precise in comparison to those derived from a deterministic approach as ours.

Mixed criticality. The advent of more and more sophisticated multi-core platforms and the increase in the complexity of the application make it commonplace to execute application functionalities with different levels of importance or criticality on the same architecture. Such applications are referred to as mixed criticality systems. A predictable STM model for real-time systems could be extended to cope with the requirements of such systems. Our model assumes that all tasks have the same criticality level, despite the diverse priorities of jobs. On another front, our STM contention management policy is based on a single FIFO serialisation process. A more sophisticated one can be tailored to match the mixed-criticality model.

References

Adapteva. *Epiphany Multicore IP*, August 2012. URL http://www.adapteva.com/products/epiphany-ip/epiphany-architecture-ip/. Cited on pages 5 and 6.

Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. *Draft specification of transactional language constructs for C++ (v1.1)*. C++ Transactional Memory Specification Drafting Group, February 2012.

James H. Anderson and Philip Holman. Efficient pure-buffer algorithms for real-time systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 57–64, Cheju Island, South Korea, December 2000. Cited on page 3.

James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, December 1999. ISSN 10459219. Cited on page 37.

James H. Anderson and Srikanth Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 94–105, Los Alamitos, USA, December 1996. Cited on page 37.

James H. Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*, pages 215–234. Kluwer Academic Publishers, Norwell, USA, May 1997. Cited on page 3.

James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-Time Scheduling on Multicore Platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 179–190, San Jose, USA, 2006. Cited on page 2.

Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193–202, London, UK, Dec 2001. Cited on page 26.

Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991. Cited on pages 3, 33, 61, 66, and 161.

Theodore P. Baker. What to Make of Multicore Processors for Reliable Real-Time Systems? In *Proceedings of the 15th International Conference on Reliable Software Technologies (Ada-Europe)*, pages 1–18, Valencia, Spain, 2010. Cited on pages 20 and 22.

Theodore P. Baker and Sanjoy K. Baruah. Schedulability Analysis of Multiprocessor Sporadic Task Systems. In Insup Lee, Joseph Y-T. Leung, and Sang Son, editors, *Handbook of Real-time* *and Embedded Systems*, pages 3.1–3.15. CRC Press, 2008. ISBN 1-58488-678-1. Cited on page 22.

António Barros and Luís Miguel Pinho. Software transactional memory as a building block for parallel embedded real-time systems. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 251–255, Oulu, Finland, August 2011a. Cited on page 180.

António Barros and Luís Miguel Pinho. Revisiting Transactions in Ada. In *Proceedings of the 15th International Real-Time Ada Workshop (IRTAW)*, Fuente Dé, Spain, 2011b.

Sanjoy Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013. ISSN 0922-6443. Cited on page 27.

Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, Lake Buena Vista, USA, Dec 1990. Cited on pages 24 and 25.

Sanjoy K. Baruah, J.E. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, pages 280–288, Santa Barbara, USA, April 1995. Cited on page 20.

Sanjoy K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996. ISSN 0178-4617. Cited on page 20.

Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*, pages 14–24, San Diego, USA, November 2010. IEEE. Cited on pages 21 and 22.

L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of theConference & Exhibition Design, Automation Test in Europe (DATE)*, pages 983–987, Dresden, Germany, March 2012.

Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, pages 264–273, Pittsburgh, USA, May 1993. Cited on pages 3 and 103.

E. Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, Jul 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1214341. Cited on page 23.

Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56, Daegu, South Korea, August 2007. Cited on pages 3, 34, 63, and 88.

Björn B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina, 2011. Cited on page 22.

Björn B. Brandenburg and James H. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 49–60, San Diego, USA, Nov 2010. Cited on page 35.

Björn B. Brandenburg and James H. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, June 2013. ISSN 0929-5585. Cited on page 96.

Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 342–353, Saint Louis, USA, April 2008. Cited on pages 3 and 87.

Peter Bright. *IBM's new transactional memory: make-or-break time for multithreaded revolution*. Arstechnica, August 2011. URL http://arstechnica.com/gadgets/2011/08/ibms-new-transactional-memory-make-or-break-time-for-multithreaded-revolution/.

Alan Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3): 116–128, May 1991. ISSN 0268-6961. Cited on page 1.

Alan Burns and Andy J. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, Cambridge, UK, 2007. ISBN 978-0-521-86697-2.

Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, Essex, UK, 4th edition, April 2009. ISBN 978-0-321-41745-9. Cited on page 2.

João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, December 2006. ISSN 01676423. Cited on pages 181 and 182.

John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–258, Pisa, Italy, July 2007. IEEE. Cited on page 22.

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H. Anderson, and Sanjoy K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, Laurie Kelly, and James H. Anderson, editors, *Handbook of Scheduling Algorithms, Models, and Performance Analysis*, chapter 30, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, USA, April 2004. ISBN 978-1-58488-397-5. Cited on page 19.

Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: why is it only a research toy? *Queue*, 6(5):40–46, September 2008. ISSN 15427730. Cited on page 41.

Byn Choi, R. Komuravelli, Hyojin Sung, R. Smolinski, N. Honarmand, S.V. Adve, V.S. Adve, N.P. Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–166, Galveston Island, USA, October 2011. Cited on page 5.

Sylvain Cotard. *Contribuition á la robustesse des systémes temps réel embarqués multicoeur automobile*. PhD thesis, Université de Nantes, 2013. Cited on pages 3 and 42.

Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture*, 57(5): 561–569, 2011. Cited on page 45.

Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computer Surveys*, 43(4):35:1–35:44, October 2011. Cited on pages 21 and 22.

UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson. Efficient Synchronization under Global EDF Scheduling on Multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, Dresden, Germany, July 2006. Cited on page 43.

Sudarshan K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, January 1978. ISSN 0030-364X. Cited on pages 20 and 21.

Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Distributed Computing: 20th International Symposium (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, Stockholm, Sweden, September 2006. Cited on pages 41, 50, 97, and 180.

Edsger Wybe Dijkstra. Cooperating Sequential Processes, EWD123. Technical report, Eindhoven University of Technology, Eindhoven, Netherlands, 1965. Cited on page 3.

François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. In *Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS)*, Toulouse, France, Nov 2010. Cited on page 22.

Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 155–165, Dublin, Ireland, 2009. ACM Press. Cited on page 41.

Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011. ISSN 00010782. Cited on pages 6 and 41.

Arvind Easwaran and Björn Andersson. Scheduling Sporadic Tasks on Multiprocessors with Mutual Exclusion Constraints. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)*, pages 50–57, Vienna, Austria, September 2009a. Cited on page 3.

Arvind Easwaran and Björn Andersson. Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386, Washington, USA, December 2009b. Cited on pages 3 and 34.

Mohammed El-Shambakey and Binoy Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 437–446, San Francisco, USA, June 2012a. ACM. Cited on page 43.

Mohammed El-Shambakey and Binoy Ravindran. STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1602–1609, Riva del Garda, Italy, March 2012b. ACM. Cited on page 43.

Mohammed El-Shambakey and Binoy Ravindran. On real-time STM concurrency control for embedded software with improved schedulability. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 47–52, Yokohama, Japan, January 2013a. Cited on page 43.

Mohammed El-Shambakey and Binoy Ravindran. FBLT: A Real-Time Contention Manager with Improved Schedulability. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1325–1330, Grenoble, France, March 2013b. EDA Consortium. Cited on pages 3 and 43.

Robert Ennals. Efficient Software Transactional Memory. Technical report, Intel Research Cambridge, Cambridge, UK, 2005. URL http://berkeley.intel-research.net/rennals/pubs/051RobEnnals.pdf. Cited on page 4.

Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. On Bounding Response Times under Software Transactional Memory in Distributed Multiprocessor Real-Time Systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 688–693, Nice, France, April 2009. Cited on page 42.

Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, September 2003. Cited on pages 4 and 41.

Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 73–83, London, UK, December 2001. Cited on pages 3 and 34.

Michael R. Garey and David S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman and Co., San Francisco, 1979. ISBN 0716710455. Cited on page 21.

N.H. Gehani. Concurrent c: real-time programming and fault tolerance. *Software Engineering Journal*, 6(3):83–92, May 1991. ISSN 0268-6961.

Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical report, INRIA, 1996. Cited on page 79.

Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In Pierre Fraigniaud, editor, *Distributed Computing: 19th International Conference (DISC)*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323. Springer Berlin Heidelberg, Cracow, Poland, September 2005a. ISBN 978-3-540-29163-3. Cited on page 51.

Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 258–264, Las Vegas, USA, 2005b. ACM. ISBN 1581139942. URL http://portal.acm.org/citation.cfm?doid=1073814.1073863. Cited on page 51.

J. Hansen, John P. Lehoczky, and Ragunathan Rajkumar. Optimization of quality of service in dynamic systems. In *Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, San Francisco, USA, April 2001. Cited on page 2.

Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, November 2003. ISSN 03621340. Cited on page 38.

Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, December 2010. ISSN 1935-3235. Cited on page 40.

Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993. ISSN 01640925. Cited on page 37.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th anual International Symposium on Computer Architecture (ISCA)*, pages 289–300, San Diego, USA, May 1993. Cited on pages 4 and 41.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, Burlington, MA, USA, 2008. Cited on pages 4 and 36.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC)*, pages 92–101, Boston, USA, 2003. Cited on pages 4, 5, 41, and 50.

C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. ISSN 00010782. Cited on page 3.

Intel. *The SCC Platform Overview*. Santa Clara, CA, USA, May 2010. URL http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-platform-overview-paper.pdf. Cited on pages 5 and 6.

Intel. *Intel Many Integrated Core Architecture – Advanced*, 2012. URL http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. Cited on pages 5 and 6.

Kalray. *MPPA256 – Product Brief*, 2015. URL http://www.kalrayinc.com/download/4640/. Cited on pages 5, 6, and 16.

Shinpei Kato, Nobuyuki Yamasaki, and Yutaka Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 249–258, Dublin, Ireland, 2009. IEEE. Cited on page 22.

Jörg Kienzle, Ricardo Jiménez-Peris, Alexander Romanovsky, and Marta Patiño-Martínez. Transaction Support for Ada. In *Proceedings of the 6th International Conference on Reliable Software Technologies (Ada-Europe)*, volume 2043, pages 290–304, Leuven, Belgium, 2001. Cited on page 175.

M Klein, T Ralya, B Pollak, R Obenza, and M Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, 1993. Cited on page 1.

Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977. ISSN 00010782. Cited on page 37.

C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. Cited on pages 18, 19, and 23.

J.M. López, J.L. Díaz, and D.F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004. ISSN 0922-6443. Cited on page 26.

Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 79–90, Bangalore, India, January 2010. Cited on pages 6, 39, and 87.

Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible Atomic Regions for Real-Time Java. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 62–71, Miami, USA, December 2005. Cited on page 41.

Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In Pierre Fraigniaud, editor, *Distributed Computing: 19th International Conference (DISC)*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368. Springer Berlin Heidelberg, Cracow, Poland, 2005. Cited on page 41.

Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012. ISSN 0001-0782.

Aloysius K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983. Cited on pages 12 and 19.

Vincent Nelis, Patrick Meumeu Yomsi, and Joël Goossens. Feasibility intervals for homogeneous multicores, asynchronous periodic tasks, and FJP schedulers. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS)*, pages 277–286, Sophia Antipolis, France, 2013. ACM. Cited on pages 88 and 112.

G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 13–23, July 2012. doi: 10.1109/ECRTS.2012.36. Cited on page 20.

Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Sergio Arévalo. Integrating groups and transactions: A fault-tolerant extension of Ada. In *Proceedings of the International Conference on Reliable Software Technologies (Ada-Europe)*, pages 78–89, Uppsala, Sweden, June 1998. Cited on page 175.

Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Sergio Arévalo. Implementing Transactions using Ada Exceptions : Which Features are Missing? *Ada Letters*, XXI(3):64–75, September 2001. Cited on pages 175 and 176.

Dmitri Perelman and Idit Keidar. SMV: Selective Multi-Versioning STM. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Paris, France, April 2010. Cited on pages 180, 181, and 182.

Dmitri Perelman, Rui Fan, and Idit Keidar. On Maintaining Multiple Versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 16–25, Zurich, Switzerland, July 2010. Cited on pages 181 and 182.

Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990. Cited on page 63.

Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS)*, pages 259–269, Huntsville, USA, December 1988. Cited on page 34.

James Reinders. *Transactional Synchronization in Haswell*. Intel, July 2012. URL http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/.

Torvald Riegel and Pascal Felber. Snapshot Isolation for Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, Ottawa, Canada, June 2006. Cited on page 182.

Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 284–298, Stockholm, Sweden, September 2006. Cited on pages 180, 181, and 183.

Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 47–56, Bangalore, India, January 2010. Cited on page 6.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. {McRT-STM}: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 187–197, New York, USA, 2006. ACM Press. Cited on page 41.

Toufik Sarni, Audrey Queudet, and Patrick Valduriez. Real-Time Support for Software Transactional Memory. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 477–485, Beijing, China, August 2009. Cited on pages 3, 42, and 180.

William N. Scherer III and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, July 2004. Cited on page 51.

William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 240–248, Las Vegas, USA, 2005. Cited on page 51.

Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-Time Transactional Memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 326–333, Sierre, Switzerland, March 2010. Cited on page 42.

Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990. ISSN 00189340. Cited on pages 3 and 33.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC)*, pages 204–213, Ottawa, Canada, August 1995. Cited on pages 4 and 41.

Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2): 99–116, February 1997. ISSN 0178-2770. Cited on pages 41 and 45.

Paulo B. Sousa, Konstantinos Bletsas, Eduardo Tovar, and Björn Andersson. On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems. In *Proceedings of the 13th Real-Time Linux Workshop (RTLWS)*, pages 207–218, Prague, Czech Republic, 2011. Cited on page 112.

Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 179–193, Stockholm, Sweden, September 2006. Cited on pages 40 and 105.

Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. Research Report RR-2772, INRIA - Institut National de Recherche en Informatique et en Automatique, Paris, France, 1996. Cited on pages 25, 78, 79, 83, 85, 144, and 162.

Anand Srinivasan. *Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2003. Cited on page 20.

John A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988. ISSN 0018-9162. Cited on pages 1 and 11.

John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems. *Real-Time Systems*, 2(4):247–254, November 1990. Cited on page 14.

Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005. ISSN 15427730. Cited on pages 2 and 3.

Tilera. *Tilera TILEPro64 processor (product brief)*, 2012. URL http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf. Cited on pages 5 and 6.

Tilera. *TILE-Gx72 Product Brief*, 2015. URL http://www.tilera.com/files/drim__TILE-Gx8072_PB041-04_WEB_7683.pdf. Cited on pages 5 and 16.

Philippas Tsigas and Yi Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 247–254, Hong Kong, China, December 1999. Cited on pages 3 and 36.

Bryan C. Ward and James H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 223–232, Pisa, Italy, July 2012. IEEE. Cited on page 96.

Bryan C. Ward and James H. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 177–186, Phoenix, USA, May 2014. Cited on page 35.

Bryan C. Ward, Glenn A. Elliott, and James H. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 280–289, Seoul, South Korea, August 2012. IEEE. Cited on page 35.

Andy J. Wellings, Alan Burns, Osmar Marchi dos Santos, and Benjamin M. Brosgol. Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 115–123, Santorini Island, Greece, May 2007.

Jeannette Marie Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. The Avalon/C++ programming language (version 0). Technical Report CMU-CS-88-209, Carnegie Mellon University, Computer Science Department, December 1988.

Fengxiang Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250–1258, Sept 2009. Cited on page 25.

Appendix A

Ada language support for transactions

This appendix specifies how to enrich the syntax of the Ada programming language so that it provides support to the execution of transactions on a multi-core platform by following an STM based approach.

A.1 State of the art

The Ada programming language was originally developed for writing concurrent software dedicated to embedded systems. Previous work on transactions was oriented for fault tolerant systems, as a mean to maintain the state sound despite failures. The solutions found in literature on this topic have several similarities with memory transactions: once a transaction starts executing, it must commit or abort. In this way, they are serialised as atomic sections that are meant to maintain data consistency. This work can be useful for the definition of new language constructs, and for the design of a STM service in real-time systems.

Support for transactions in Ada was already subject of research in the field of fault-tolerant systems. The concept of a transaction grouping a set of operations that appear to be executed atomically (with respect to other concurrent transactions) if succeed, or having no effect whatsoever on the state of the system if failed, is quite appealing as a concurrent control mechanism for fault-tolerant and/or distributed systems. The ability to abort a transaction due to data access conflict or to an unexpected error, rolling back any state modifications, permits to preserve automatically the system in a safe and consistent state. Safe consistent states can be stored in a durable medium, so they might be available even after a crash.

The loose parallelism provided by the isolation property of the transactional paradigm suits systems based on multiple processors (either clustered or distributed) and the inherent backward recoverability mechanisms suits fault-tolerant concerns.

Two paradigmatic implementations of transaction support in Ada are the Transactional Drago (Patiño-Martínez et al., 1998, 2001) and the OPTIMA framework (Kienzle et al., 2001).

Listing A.1: Transaction block (Transactional Drago).

```
transaction
  declare
    -- data declared here is subject to concurrency control
  begin
    -- sequence of statements
    -- can include tasks that work on behalf of transaction
    -- can include nested transactions
  exception
    -- handle possible exceptions here...
end transaction;
```

Both proposals share many common attributes, aiming to support competitive concurrency (between transactions) and cooperative concurrency (inside a transaction), and provide the essential interface to start, commit and abort transactions. Transactions can be multithreaded, i.e. multiple tasks can work on behalf of a single transaction. Both implementations support nested transactions and exception handling.

Despite the similarities, both implementations take different approaches.

Transactional Drago is an extension to the Ada language, so it can only be used with compilers that include this extension. Transactions are defined using the *transactional block*, an extension that resembles an Ada block statement, but identified with the keyword `transaction` (Patiño-Martínez et al., 2001). The transactional block creates a new scope in which data, tasks and nested transactions can be defined. Data declared inside a transactional block are volatile and subject to concurrency control. Tasks inside a transactional block work cooperatively on behalf of the transaction and their results will dictate the outcome of the transaction.

The transactional block provides a clean syntax, defining clearly the limits of the transaction. A transaction is aborted by rising an exception, as there is no explicit abort statement. The sample code in Listing A.1 illustrates a transactional block.

The OPTIMA framework is provided as a library, thus the language is not modified. In this framework, a transaction is created with the command `Begin_Transacion` and ends with either `Commit_Transaction` or `Abort_Transaction`, depending on the result of the computation. The OPTIMA framework supports open multithreaded transactions, so the additional command `Join_Transaction` allows one task to join and cooperate in an ongoing transaction. A task becomes linked to the transaction setting the appropriate parameters in the `Ada.Task_Attributes` record, which allows the transaction support to determine in which transaction context the task is working on. The code sample in listing A.2 illustrates a task that starts a transaction.

This example shows how this framework uses the exceptions mechanism to define handlers for foreseen exceptional cases, thus allowing forward recovery in such cases. However, unexpected exceptions will abort the transaction, like in Transactional Drago.

Listing A.2: Transaction (OPTIMA framework).

```
begin
  Begin_Transaction;
  -- perform work
  Commit_Transaction;
exception
  when ...
    -- handle recoverable exceptions here...
    Commit_Transaction;
  when others =>
    Abort_Transaction;
    raise;
end;
```

A.2 Ada language support for transactions

In this section, we elaborate on the specific instructions we propose to enrich the Ada language. Essential support to STM can be implemented in a stand-alone library, without introducing any modifications in the Ada programming language itself. This approach facilitates the portability of the STM service, without any need to modify compilers and debuggers. However, the burden rely on the programmer as he will be responsible for delimiting the boundaries of the transactional sections of code of the progrm.

The code in listing A.3 illustrates how a very simple transaction should be written.

This example shows how the initialisation of the transaction and the retry-until-commit loop have to be explicitly written.

This STM perspective requires two key classes of objects: the *transactional object* and the *transaction identifier*.

The transactional object encapsulates a data structure with the transactional functionality. For instance, a write operation will not effectively modify the value of the object if the STM applies lazy version management.

The transaction identifier is a structure that stores the data required by the contention manager to apply the conflict solving policy chosen for the system.

A.2.1 Transactional object

A transactional object is a type of class that wraps a classical data structure with the transactional functionality. The interface provided is similar to the non-transactional version, but adds the operations required to maintain the consistency of the object, according to the implementation details of the STM.

Thus, for every access, the identification of the transaction is required, either to locate the transaction holding the object (case of eager version management) or track all transactions referring the object (case of lazy version management). In each case, the object must locate one or all accessing transactions, respectively, so under contention, transactions' attributes are used

Listing A.3: TM transaction syntax for Ada.

```
-- we need a transaction identifier structure
My_Transaction : Transaction;

-- start an update transaction
My_Transaction.Start(Update);

loop
  -- read a value from a transactional object
  x := trans_object_1.Get(My_Transaction);

  -- write a value to a transactional object
  trans_object_2.Set(My_Transaction, y);

  -- try to commit transaction
  exit when My_Transaction.Commit;

exception
  -- handle possible exceptions here...
end loop;
```

to determine which transaction is allowed to proceed, according to the contention management policy.

Reading accesses can also be tailored for read-only transactions, if a multi-versioned STM is used. Transparently, the transactional object can return the latest version to an update transaction, or a consistent previous version to a read-only transaction.

Listing A.4 proposes an interface for a transactional object.

A.2.2 Transaction identifier

The transaction identifier provides the transactional services to a task, identifying uniquely a transaction. The essential interface of this class should provide the `Start`, `Commit` and `Abort` operations, and keep track of the accessed objects, as seen in listing A.5.

The `Start` procedure initialises the transaction environment. Starting an already active transaction is not allowed, and an exception should be raised.

The `Abort` procedure erases any possible effects of the transaction, but the transaction remains active and is allowed to undertake further execution attempts. Aborting an inactive transaction is not allowed, and an exception should be raised.

The `Terminate` procedure cancels the transaction, leaving the transaction inactive. Terminating an inactive transaction is not allowed, and an exception should be raised.

The last operation provided by this interface is the `Commit` function that validates accessed data and resolves possible conflicts. If the transaction is allowed to commit its updates, then this function will return the `True` value. Committing an inactive transaction is not allowed, and an exception should be raised.

Listing A.4: Transactional object.

```
-- Transactional object
package Transactional_Objects is
  type Transactional_Object is tagged private;
  -- examples of transactional class methods
  procedure Set(T_Object: Transactional_Object;
                Transaction_ID : Transaction;
                Value : Object_Type);
  function Get(T_Object: Transactional_Object;
               Transaction_ID : Transaction)
           return Object_Type;
private
  type Transactional_Object is tagged
  record
    Current_Value : Object_Type;
    Accesses_Set : <list_of_references_to_transaction_identifiers>
    -- some other relevant fields...
  end record;
end Transactional_Objects;
```

Listing A.5: Transaction identifier.

```
type Transaction_Type is (Read_Only, Update);

-- Transaction identifier
package Transactions is
  type Transaction is tagged private;
  procedure Start(T : Transaction;
                  TRX_Type : Transaction_Type);
  procedure Abort(T : Transaction);
  procedure Terminate(T : Transaction);
  function Commit(T : Transaction)
           return Boolean;

private
  type Transaction is tagged
  record
    Data_Set : <list_of_references_to_accessed_transactional_objects>;
end record;
end Transactions;
```

Listing A.6: Extension to base transaction identifier.

```
type Transaction_Status is (Active,
                            Preempted,
                            Zombie,
                            Validating,
                            Committed);

-- Transaction identifier
package Transactions_Foo_STM is
  type Transaction_Foo_STM is new Transaction with private;
  overriding
  function Commit(T : Transaction_Foo_STM) return Boolean;

private
  type Transaction_Foo_STM is new Transaction with
  record
    -- implementation specific elements
    -- some examples below
    Type_of_Accesses : Transaction_Type;
    Time_Started : STM_Time;
    Time_Current_Begin : STM_Time;
    Status : Transaction_Status;
    -- some other relevant fields...
  end record;
end Transactions_Foo_STM;
```

This class also stores the references to the transactional objects that were accessed in the context of the transaction. This data is required when trying to commit, to validate read and update locations.

Specific STM implementations will, most likely, require modified operation functionality and additional attributes. For example, some STM algorithms require to know the instant the transaction arrived, the current status of the transaction (Barros and Pinho, 2011a), transaction deadlines (Sarni et al., 2009), or the instant the current attempt of the transaction began (Dice et al., 2006; Riegel et al., 2006; Perelman and Keidar, 2010).

These attributes can be included in extensions of this class, deriving a new class for each implementation, as the example in listing A.6 illustrates.

Appendix B

Bounded-memory multi-version STM for real-time systems

Previous work on STM has consistently shown that the amount of contention has a relevant impact on the behaviour of STM. While contention may never be eliminated, measures that reduce contention may improve the performance of the STM. Multi-versioned STM was proposed as a way to reduce contention, improving directly the abort ratio of read-only transactions and, in consequence, improve the overall system performance (Cachopo and Rito-Silva, 2006). In this approach, multiple versions of shared objects are temporarily kept so each read-only transaction executes with a recent and consistent snapshot of its read set without conflicting with other concurrent transactions. However, published works on multi-versioned STM either propose a fixed number of versions for each object (Riegel et al., 2006) or a variable number of versions that are garbage collected (Cachopo and Rito-Silva, 2006; Perelman et al., 2010; Perelman and Keidar, 2010). In all cases, since there is no knowledge on the timing characteristics of tasks, the number of outdated versions is either set as a fixed parameter that reduces the probability of aborts, or eliminates this probability keeping a strict number of versions necessary for the ongoing transactions at the expense of using a garbage collector mechanism, which is known for not being suitable to real-time systems. This appendix demonstrates how the timing characteristics of real-time systems can be used to determine the maximum number of versions for each data object.

A conflict occurs when two or more transactions access concurrently a common data object and at least one of the transactions updates this object. Considering the example in Figure B.1, where shared objects X and Y are initially in version v, respectively denoted as X_v and Y_v. Transaction ω_1 first reads object X_v. Then transaction ω_2 is released and modifies both objects, updating them to versions X_{v+1} and Y_{v+1}, and finishes its execution. Finally, transaction ω_1 reads object Y_{v+1}, which is no longer consistent with X_v. This concurrent execution is not serialisable because ω_1 will not behave exactly the same way as if it was executed sequentially before or after ω_2: ω_1 reads X_v as if ω_2 has not executed yet, and reads $Yv+1$ as if ω_2 has finished executing. In this situation, and

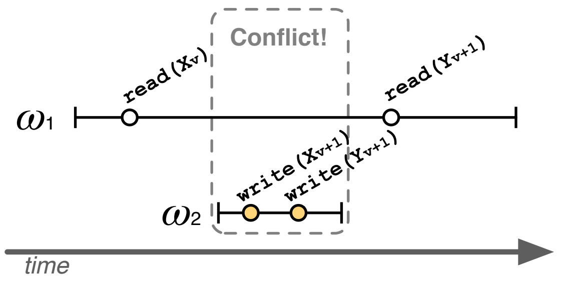


Figure B.1: Conflict between a read-only (ω_1) and an update (ω_2) transactions.

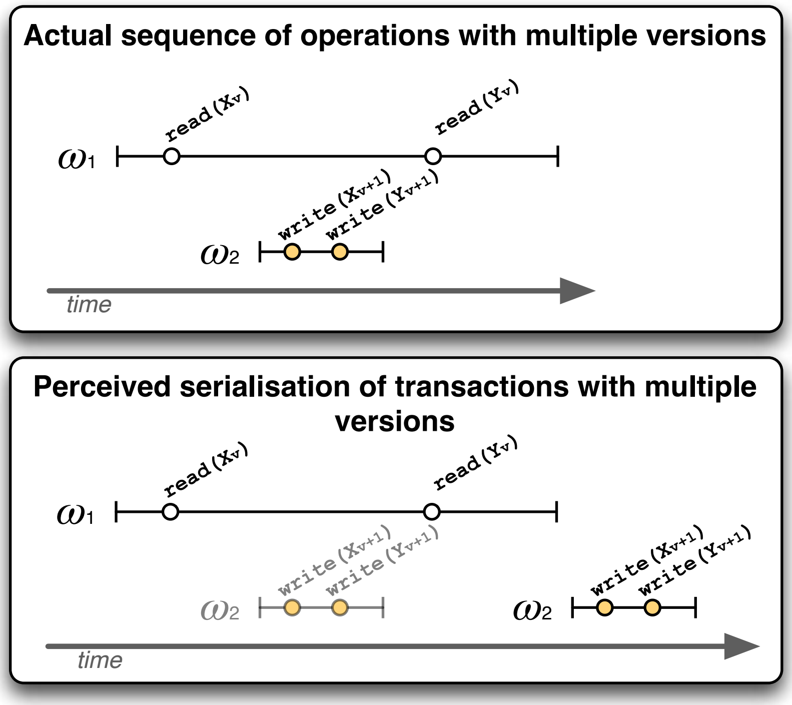


Figure B.2: Conflict between a read-only (ω_1) and an update (ω_2) transactions.

depending on contention management policy applied, one of the transactions must abort so the contender can commit.

Multi-versioned STM allows read-only transactions to work on recent consistent snapshots of their read sets without ever conflicting with other concurrent transactions (Cachopo and Rito-Silva, 2006) and, therefore, execute in a wait-free manner and commit at first try. Considering the same example in Figure B.1, both transactions can now commit, as long as ω_1 has access to the version of Y_v, previous to the update performed by ω_2. In this case, although ω_1 commits after ω_2, it can be serialisable as if it has executed before ω_2, as represented in figure B.2.

Multi-versioned STM reduces the amount of contention on object accesses at the expense of higher memory utilization, since previous versions of each object must be temporarily stored. The amount of memory overhead that optimises the system throughput is a subject of current research in the field of parallel systems (Perelman et al., 2010; Perelman and Keidar, 2010; Riegel and

Felber, 2006; Riegel et al., 2006), essentially relying on fixed number of versions or in garbage collecting techniques that will statistically reduce or eliminate the abort ratio of read-only transactions.

In real-time systems, it is known beforehand the timing characteristics of the task set which, in conjunction with the knowledge of the data set of each transaction, gives the means to determine the exact number of versions required for any given object. Knowing the exact number of versions each object must store, permits to design a STM with minimum memory overhead and predictable version management (excluding garbage collecting mechanisms), and guaranteeing read-only transactions will complete with deterministic execution time while avoiding collisions with update transactions.

To determine the number of versions required for any object it is necessary to:

1. bound the number of updates each object is subject to in a given interval, and

2. determine the time each object must store a individual version.

The maximum number of updates of an object in a given interval can be calculated considering the timing properties of the tasks that host update transactions that include the object in their write set. For an arbitrary time interval ΔT, the maximum number of updates of object o_k, denoted as $updates_k$, is given by the number of job releases of tasks that host transactions that include the o_k in its write set:

$$updates_k = \sum_i a_i \cdot \left\lceil \frac{\Delta T}{T_i} \right\rceil \tag{B.1}$$

in which the binary variable a_i is given by

$$a_i = \begin{cases} 1 & \text{if } o_k \in WS_i, \\ 0 & \text{otherwise.} \end{cases} \tag{B.2}$$

The time each version of object o_k must be stored, denoted as $store_k$, is given by the maximum time a read-only transaction including the o_k in its read set can execute. Since there are no restrictions on where the transaction is located inside the task, we must assume the object may be read any time during the period of the task. Thus, a version of object o_k must be stored for

$$store_k = \max\{T_i : o_k \in RS_i \wedge \omega_i \text{ is read only}\}. \tag{B.3}$$

Combining the two results from Equations (B.1) and (B.3), we can determine the number of versions required for O_k, denoted as $versions_k$, is

$$versions_k = \sum_i a_i \cdot \left\lceil \frac{store_k}{T_i} \right\rceil \tag{B.4}$$

in which a_i is given by

$$a_i = \begin{cases} 1 & \text{if } o_k \in WS_i, \\ 0 & \text{otherwise.} \end{cases} \tag{B.5}$$

Therefore, multi-versioned STM can be implemented efficiently in real-time systems with predetermined memory overhead, and assuring all read-only transactions will execute in a wait-free manner.