Hardware and software analyses for precise and efficient timing analysis

Claire Maiza

Hardware and software analyses for precise and efficient timing analysis

Claire Maiza – Université Grenoble Alpes
Defense on June 9, 2023.

Jury :
Reviewer: Christopher Gill, Prof. Washington University in Saint Louis
Reviewer: Sandrine Blazy, Prof. Université de Rennes
Reviewer: Marc Pouzet, Prof. Ecole Normale Supérieure
Jean-Luc Scharbarg, Prof. Université Toulouse
Laure Gonnord, Prof. Grenoble INP-UGA
Laurence Pierre, Prof. UGA

June 9, 2023

Acknowledgment

Along the nice adventure of scientific research, I always collaborated and learnt from each collaboration. I would like to thank Christine Rochange and Reinhard Wilhelm, for their help in starting in this domain and community. Among my collaborators, some shared long adventures with me and I enjoyed particularly what we did together: Sebastian Altmeyer, Jan Reineke, Rob Davis, Matthieu Moy, Pascal Raymond, Nicolas Halbwachs, Benoit Dinechin, Pascal Richard, Joël Goossens. Most research could not exist without the work of students. I would like to thanks the three doctors who lived the adventure of the PhD with us: Hamza Rihani, Valentin Touzeau and Matheus Schuh. As well as the doctors who collaborated with me along their PhD adventure: Julien Henry, Amaury Graillat, Guillaume Phavorin. Master and bachelor students are important to explore new ideas or better explore possibilities. I do not list them here, because I would not like to forget one but I would like to let them know that I enjoyed working with them and thank them for the weeks of collaboration. Along this habilitation report, all my co-authors are listed: many thanks to each one of you for the work we did together. I would like to thank also all colleagues who shared a discussion, an idea, a smile, happiness in the daily life at work. Particularly, many thanks to Verimag people.

Research projects are good opportunities to create particularly productive collaborations. I would like to thank all collaborators in W-SEPT, CAPACITES, and CAOTIC projects. Special thanks to Lionel Rieg.

In France, as a "maitre de conférence", the research activity depends on the efficiency of the teaching activity: I would like to thank all my friends and colleagues of Ensimag for their good work and for the nice daily life we can share.

Finally, many thanks to my family and friends, actors of my life for their happiness, their help when it was necessary and for all what we share. Special thanks to Chaouki and Lina.

Abstract

In hard real-time systems, encountered for example in the transportation, energy or medical domains, programs must not only provide a functionally correct result but also guarantee timing constraints. Our researches focus on timing models and analyses which aim to get precise bounds on execution time and delays. Generally, these models focus on a single hardware or software aspect. On the hardware side, our models and analyses focus on cache memories, memory bus arbitration and memory access ways in multi-core and many-core platforms. As part of software analysis, we took benefit from the infeasible path elimination during analysis: the idea is that the computation of the worst-case execution time should ignore paths that are not semantically possible. Additionally, we have shown that a better integration of software and hardware analyses leads to more precise bounds. In the context of multi-core timing systems, we have shown that timing analysis can benefit both from a better knowledge of software and hardware, and from the integration of timing analysis with the last step of implementation. We show how and where our work leads to changing the point of view on timing analysis in order to improve it. Finally, we expose the current state of the art and which gaps must still be filled to obtain a complete and efficient timing analysis.

Résumé

Les travaux présentés se situent dans le contexte des systèmes temps-réel critiques où les programmes ne doivent pas seulement être corrects mais aussi respecter des contraintes temporelles sous-peine de dommages importants. Parmi les domaines concernés sont ceux de l'énergie, des transports, le médical et l'espace. Nous nous intéressons particulièrement aux modèles et analyses qui permettent d'obtenir une borne garantie des temps d'exécution et délais. Pour réaliser ces analyses temporelles, des modèles du logiciel et du matériel sont nécessaires. Ces modèles et les analyses qui les utilisent se concentrent souvent sur un seul aspect du matériel ou du logiciel. Nos travaux ont permis d'obtenir des analyses plus précises etou efficaces de la mémoire cache, des arbitres de bus mémoire, et des structures d'accès mémoire dans les plateformes multi-cœur et pluri-cœur (groupement de multi-cœur). Du côté du logiciel, nos travaux ont exploité l'élimination de chemins d'exécution infaisables lors de l'analyse pour éviter que le temps d'exécution pire-cas ne corresponde à un chemin d'exécution qui n'est pas possible sémantiquement. Une autre partie de nos travaux a permis d'obtenir des bornes plus précises du temps d'exécution en travaillant sur des analyses conjointes du matériel et du logiciel. Pour les plateformes multi-cœurs, nos travaux ont montré qu'un gain de précision était possible au travers d'une meilleure prise en compte des structures logicielles et matérielles mais aussi d'une intégration des analyses temporelles avec les dernières phases d'implémentation (orchestration de code). Dans ce rapport, nous montrons en quoi nos travaux ont permis de changer le point de vue sur l'analyse temporelle et de la faire évoluer. Enfin, ce rapport termine par un exposé de la situation actuelle et des avancées nécessaires pour obtenir des analyses temporelles complètes et efficaces.

Contents

Chapter 1

Introduction

Among the embedded systems, some are subject to timing requirements where the program execution must not only fulfil a correct functional execution, but also a bounded duration (execution time). The timing constraints may come from different sources, among them: some hardware behavior (e.g. minimum inter-arrival delay of censored data) or safety timing requirements (e.g. the minimum delay to deploy an airbag). Most of the embedded systems are subject to timing requirements that do not impact human safety; e.g. video quality of service.

Our work is in the context of *hard real-time systems*, where any requirements must be fulfilled. Most of the industrial users or collaborators are in the transportation domain (automotive, avionics) or medical device domain. In such domains the functional and non-functional constraints must be guaranteed.

For instance, the timing constraints may consider a maximum execution time delay for a program or an application, an inter-arrival delay on data calculation to be sent, or a deadline to be respected.

To better understand the context, we need to give more details on what are the characteristics of the hardware and software that we consider. We consider *software at different levels* and getting different characteristics:

- *high-level design:* such as SCADE-Lustre codes, The specific properties that we consider are: the data-flow model where any application may be described by a precedence graph –each node is a program and each edge represents a communication and a constraint on the execution of the target node after the execution of the source node–, some specific properties due to the synchronous paradigm or the specificity of Lustre/SCADE;

- *C level:* at this level some semantic properties may be useful for timing analysis;

- *binary level:* that is the code that will be executed and must be considered by the timing analysis, for most of our binary analysis we do not directly analyze the code but an extracted model containing the necessary information. For that aim we use a timing analysis tool: generally the OTAWA academic tool [2].

The *hardware elements and platforms* we consider are:

- *cache memory*: this small memory is placed close to the execution core to get faster access to data and/or instructions. As it is small, it can only contain a subset of the memory and needs to often load new content from the distant memory.

- *memory bus*: a shared memory may be accessed by each processing element (or core). A memory bus is used to access this memory and needs to be arbitrated to select which core may access the shared memory among the cores requesting accesses: this is the role of a bus arbiter.

- *multi-core or cluster*: it refers to a set of cores sharing a local memory. Each core may have an instruction and/or data cache memory. A memory bus and its arbitration is used to access the shared memory.

- *Banked memory*: this multi-core memory can be partitioned and each partition may get a limited access by a core or a set of cores.

- *Many-core:* it is composed of a set of clusters (multi-core) that are communicating through a Network-On-Chip (NoC) or a specific memory bus for inter-cluster memory transfer– e.g. Advanced eXtensible Interface. The many-core may contain a global large memory, but we do not focus on this kind of memory and consider only the set of cluster memories.

The execution time of a program or application may vary largely. The main reasons for the variation of exection time are:

- the use of a hardware platform with a behavior that depends on the context (e.g. cache memory where the access delay is short in the case that the accessed memory content is already in the cache, and very long in the case where the content needs to be loaded from main memory);
- the software structure (e.g. each conditional structure as a if-then-else leads to different execution time depending on the condition; the use of execution modes may lead to part of the program that are executed only in a specific context);
- the way the software is implemented on the hardware platform (e.g. access to shared data may vary depending on if it is mapped to a local memory or global one, but also depending on which programs may access this memory region simultaneously).

Traditionally, hard real-time systems were implemented on simple single-core platforms with limited variable timing behavior. However, the massive use of multi-core platforms and the growing complexity of algorithms and collected data, leads to the use of multi or many-core platforms for hard real-time systems. This opened a set of research work to adapt the development process of such systems. In this report, we focus on *the timing analysis and the implementation (orchestration code generation) of critical applications on multi-core platforms.*

As part of the timing analysis, we focus on the estimation of *worst-case execution time* (a bound on the execution-time of a program when executed in isolation – no interruption, preemption or any interference), the estimation of the *worst-case response time* (a bound on the execution time of a program including any impact of other program executed on the same hardware platform), the estimation of *cache-related preemption delay* (a bound on the delay due to cache memory accesses during a preemption), the estimation of *interference delay* (a bound on the delay due to shared resources in multi-core platforms).

The timing analysis process was usually composed of a set of analyses that were studied separately and not really integrated to the synthesis process. This lead to an issue as regards the complexity of these steps and opened the opportunity of more integrated approaches. In this context, we worked on the *integration of the timing analysis and the synthesis of application.* The complexity may largely be reduced by a better knowledge on the hardware and software structures. In this report we focus on the *implementation of data-flow applications (such as the one generated from SCADE/Lustre) on multi-core platforms with shared banked memory (e.g. Kalray MPPA platforms).*

In this report we show how we got more precise and/or more efficient timing analysis. First, we focus separately on hardware analysis (cache memory and multi/many-core memory communication interference)–in Part I– and software semantic analysis (infeasible path)–in Part II. From what we learnt on both

sides, we show how we implement data-flow applications on multi-core platforms with integration of interference analysis during the synthesis –in Part III.

In this report, we do not go into the details of our analyses nor the state of the art that may be found in our papers, rather we focus on the way our work contributed to the timing analysis process. Our aim is to discuss future work and open questions related to our research context. For that, in each part, there are specific discussion chapters. In the last part IV we highlight where our work did change the point of view of the community (research or industry) and we enlarge the discussion on what still needs to advance to get a fully guaranteed timing analysis process that is precise and scalable.

Part I

Hardware models and analyses

The main collaborators for this part are:

Cache analysis Valentin Touzeau (PhD student [3]), David Monniaux, Jan Reineke, Bai Zhenyu and Maeva Ramarijaona.

CRPD Sebastien Altmeyer, Rob Davis, Jan Reineke, Guillaume Phavorin, Pascal Richard, Joël Goossens, Laurent George, Thomas Chapeaux.

Multi-core Hamza Rihani (PhD student [4]), Matheus Schuh (PhD student [5]), Matthieu Moy, Pascal Raymond, Juan Rivas, Joël Goossens, Sebastian Altmeyer, Rob Davis, Jan Reineke.

In the timing analysis process, we denote as hardware analysis, any analysis that relies on the behavior of a hardware component. Such an analysis is not fully separated from the software, as a timing analysis focuses on a given program. The idea is to rely on a sufficient software abstraction to get enough information on the software, but not too much that would lead to a high analysis complexity. Traditionally, in the single-core WCET analysis, the common software model was the control flow graph [6]. However, for each micro-architectural analysis, the software model level may be different. For instance, a pipeline analysis relies on each instruction and its operand. In this chapter we focus on *cache memory and interference analysis in multi-core*. For each analysis we give the convenient software model and the main ideas behind the corresponding hardware analysis, what we built upon the state-of-the-art and how our work contributes more precise and more efficient timing analysis.

Chapter 2

Cache memory

As said in the introduction, a cache memory is a small memory close to the core or pipeline to keep fast access to a subset of the memory content. Each time there is a granted access to a memory block, the block is first searched into the cache memory: in case the block is in the cache –referred to as a hit, it may be directly sent back to the core; in case the block is not cached –referred to as a miss, it needs to be loaded from the main memory. A main memory access spends ten to hundreds times more cycles than a cache access.

How a cache analysis could help the timing analysis process. If there is no cache analysis, any access needs to be considered as a miss and accounted as the worst delay. This is extremely pessimistic. A cache analysis helps in refining this pessimism by classifying as hit a part of the memory accesses for which the delay is reduced to a cache access delay instead of a main memory access delay. In this Section we focus on two kinds of cache analysis. The first one considers a program executed in isolation (no interruption, no preemption, no interference) and aims at classifying each memory access as a hit or a miss. The second one, considers the complementary case and looks for the additional misses due to a preemption: Cache Related Preemption Delay (CRPD).

Each time a block[1] is loaded in the cache, there is an eviction of one block that was in the cache. The selection of which block to be evicted is done by a replacement policy. There exist a set of replacement policies, but in this report we mainly focus on the "Least recently Used" policy (LRU). The main reason for this choice is the predictability of LRU [9].

A cache memory may be dedicated to instructions or data, or a unified cache may be used for both. A data cache analysis needs a preliminary analysis to compute the data addresses. In this report, we focus only on pure instruction cache.

2.1 Precise and efficient cache analysis

A cache analysis is based on the following model elements:

[1]To simplify we ignore the cache line and consider that a memory block is a line. For further details please refer to the corresponding papers [7, 8, 3].

- a software model with two mandatory entities:

  Memory block this is the entity that corresponds to what can be accessed from or loaded in the cache memory.

  Control flow block this is the entity that corresponds to the possible execution paths; there is a new entity when there is a conditional test in the program, or when there are two possible previous instructions.

  Both entities must be encoded in the software model.

- a cache model with:

  Cache content representation the cache content is mainly composed of the memory block identifiers.

  Evolution upon a miss or a hit this is the model of the replacement policy; this should represent an order between blocks such that the "next to be evicted" is identified, but also the evolution of block placement in this order upon a hit or a miss.

  Evolution due to the execution path this encodes the impact of the execution path upon the cache model; the whole content of the cache can not be represented in the model (complexity issue), thus, at each point where two execution paths are joined, the timing analysis may select the abstracted content to keep in the model.

Among the existing cache analyses, the main difference is the way they deal with the cache model – see [3] for more details and references. Some analyses keep the whole cache content and do not abstract so much. Even if they are very precise models, the main drawback is the complexity of the analysis that will not scale with large cache size or program due to the memory space the analysis needs to keep all cache contents. That's why some analyses focus on finding the *"good" abstracted content that limits the model size*. The model used by the approaches relying on abstract interpretation, gains in complexity by a smart combination of two abstractions: a "must" cache content that contains only blocks that are always in the cache at a program point, and a "may" cache content that contains blocks that may be cached at a point. The combination of these two abstractions lead to a good opportunity to not keep all possible cache states, but still contains enough information to classify accesses as a hit or a miss.

In this analysis based on abstract interpretation, blocks are classified as "must hit", "must miss" and "unknown". Our work is based on the observation that in the "unknown" category there are two kinds of memory accesses: accesses for which both a hit or a miss are possible depending on the execution path, and, accesses which the analysis can not classify as a hit or a miss due to the abstract model. Our work on cache analysis aims at *refining the classification* for accesses in the second category.

Based on the observation that some analyses are (i) very precise but do not scale, and others are (ii) very efficient but can still be refined, we worked first on a model improvement for the first kind of analysis (i) that inspired us to find a way to refine the second category of analysis (ii).

In both cases we introduce *a new category for the cache accesses classification* [8]. In previous analyses they use "must hit", must"miss" and "unknown"

categories. Our aim is to refine the "unknown" category to distinguish between accesses that are "unclassified" due to the fact that both *hit* and *miss* are possible from those where the category is "unknown" due to the abstraction used in the analysis. For that aim, we add the "definitely unknown" category: both a *hit* and *miss* are possible due to the execution path. This new category has been introduced in [8].

Based on this new category, both approaches presented hereafter may start upon a pre-analysis that would previously classify accesses in the three categories where no refinement is possible "definitely unknown", "must hit" and "must miss". Among the set of remaining "unclassified" accesses from this pre-analysis, accesses may be refined by a second analysis.

2.1.1 A new model for more precise analysis

Refining the set of accesses that could not be classified by the pre-analysis, is the aim of our approach [8]. In this initial approach we explored a model with a model-checker. The novelty of our approach is in *the way we model the problem.*

Our model extracts only the part of the software and cache models that have an influence on the classification of one block. This *"focused cache model"* on a given block a keeps only the previous accesses for which a new access after the access to a would have an influence on the position (age) of block a and thus on its future eviction. Furthermore, only the necessary part of the software model (control flow graph) where this set of *memory blocks influencing block a* are acessed is kept.

The "block focusing" analysis has been introduced in [8]. Even if the "block focusing" limits the size of the model, this method keeps all possible elements of this refined cache model and may lead to a scalability issue. However, it has been a first step for us to reach the next approach.

2.1.2 A new model for a precise and efficient analysis

As we have seen, we look for *analysis to refine the cache accesses classification for "unclassified" accesses.* We introduced a new method to reach that aim that is precise and efficient [7]. We have seen that with the previous method, the analysis is precise but still not so scalable when the size of the model grows. The main idea to reach this efficiency is to *reduce the model* depending on what is the target category: we use a partial order and keep only the minimal (resp. maximal) elements when looking for cache misses (resp. hits). The efficiency is also reached by *a good implementation* using zero-suppressed binary decision diagrams (ZDDs).

Again, we use the "focused model" previously introduced to limit the scope to the block under analysis. Additionally, we keep only maximal (minimal) elements to look for hit (miss) accesses. For instance, consider blocks b, c and d that may contribute to the cache classification of block a; consider also that b is encountered on one path and b, c, d are all encountered on another path: to classify a as a "always hit" it is not necessary to keep the set where only b is encountered as b is also on the second path.

This approach has been introduced in [7]. The results shows an *analysis speed up* that can reach a factor of 950 compared to our previous approach [8].

Furthermore, the average slowdown of the analysis is only of 4.12 compared to previous less precise cache analysis [9].

2.1.3 Studies for other replacement policies and other context

First, we can have a look at the complexity of analyzing other replacement strategies. In [10], David Monniaux and Valentin Touzeau formalized the classification of memory accesses as hits or misses: the theoretical complexity of finding a path leading to a hit/miss depending on the replacement policy. They have shown that this problem is NP-hard for LRU, while other common replacement policies, namely PLRU, FIFO and NMRU, lead to a PSPACE-hard classification problem. This confirms the conclusion that LRU is the most predictable replacement strategy [9]. Note that as introduced by Jan Reineke, there is a way to bound the number of misses for other replacement strategies depending on the bound on the number of misses in the case of LRU.

We made an attempt at using *our approach for PLRU replacement strategy* [3, 11]. The main idea was to find another model for the PLRU cache. The main issue with the PLRU models is that they are hard to abstract with enough precision. The collecting semantics, keeping all reachable states, is not scalable. Our model as previous ones [12] is not very precise, but its implementation is more efficient than previous ones.

We studied a use of our *exact analysis for security* [3] as part of the master Internship of Maeva Ramarijaona. This analysis aims at highlighting program points at which the execution path could be guessed from a hit/miss classification. Most of the definitely unknown accesses may lead to a potential security leak as both hit and miss are possible depending on the execution path. The idea is to highlight the program point where this access is and what it is giving as information on the execution path.

These two preliminary works reuse some concepts of our exact cache analysis: the idea of an efficient cache model implementation and the definitely unknown classification. *Our exact analysis already inspired the community and extension of our ideas* and work have been proposed, for instance: the use of ZDDs for better WCET analysis implementation [13] and a complementary exact analysis for persistence [14]. A cache block may be initially a miss and later always hit, when the cache access is in a loop for instance: the persistence analysis is an additional analysis that studies specifically this point.

2.2 Cache-related preemption delay

We have seen that cache memory analysis may be precise and efficient. However, the previously presented analyses ignore any impact of other executing tasks. In the work presented here, we focus on *the impact other executing tasks may have on the cache content*. More specifically, we focus on the case of preemptions: a task of higher priority may stop the execution of an executing task to be executed, the previously executing task continuing its execution after this higher-priority task finishes. This is referred to a preemption and during this preemption, the preempting task uses the shared cache memory. Thus, the preempting task may evict cache content of the preempted task. If this cache

content was in the cache but not reused later in the execution of the preempted task, the eviction have no effect on the preempted task. However, in case that the evicted memory blocks were reused later on in the preempted task execution, the blocks must be reloaded when necessary: this impacts the execution time of the preempted task. The upper-bound on the delay due to preemption is named the *Cache-Related Preemption Delay*. Note that there are other sources of delay in case of preemption, such as the context switch cost due to the changes of program pointer and other internal resources. We do not focus on these other delays in this report.

In our work, we studied *new cache analysis to get a more precise CRPD* estimation and how to *integrate the CRPD into schedulability analysis*. As part of schedulability analysis, we mainly took into account the estimation of the Worst-Case Response Time (WCRT) that is the upper-bound on the time from a task being released until it finishes execution taking into account the effect of other executing tasks. Our work on CRPD is not recent, that's why some subsequent work, based on our work, is also mentioned.

2.2.1 CRPD analysis

We focus here on the *cache analysis we introduced to get more precise CRPD*. The software model that is used for the CRPD estimation is based on an abstract cache state at each program point: any program point being a potential preemption point, the CRPD is estimated at each point where the cache content evolves. This is quite similar to the cache analysis and must include control flow and memory block entities.

There are two main ways of bounding the number of additional misses due to preemption:

UCB Useful Cache Block: this cache analysis identifies at each program point the memory blocks that may be in cache and may be accessed later on the execution path without being evicted. This next access is a hit and may become a miss in case of eviction of the useful cache blocks during a preemption.

ECB Evicting Cache Block: this cache analysis is on the preempting task and gives the set of memory blocks used in each cache set. These blocks may evict the useful cache blocks and thus lead to eviction.

Both UCB and ECB may be used independently or combined to better bound the number of evicted cache blocks due to a preemption. For a recent state of the art, please see [15].

The main improvement we made on the cache analysis for CRPD estimation was two fold:

1. the *definitely cached useful cache block* [16, 17]: UCB analysis and cache analysis for WCET are considered independent. Based on this observation, we have shown that some misses are accounted for as part of the WCET estimation and in the CRPD. That's why we introduced the definitely cached-useful cache block to avoid double counting the CRPD misses that were already part of the WCET estimation.

2. the *resilience analysis* [18]: the UCB and ECB analysis are two independent analyses, we observed that some misses were accounted for in the CRPD, but there were not enough cache accesses of the preempting task to evict some UCB. We introduced the resilience analysis to estimate for each UCB, the number of ECB that were necessary to evict it. From the UCB point of view, its resilience is the number of accesses that can be made before its eviction.

The CRPD analysis may be done on any cache configuration but only direct-mapped and LRU set associative caches leads to possible integration of CRPD costs in schedulability analysis. We have shown in [19] that due to the presence of timing anomalies that may be due to other replacement strategies, LRU is the only replacement strategy that is compatible with the CRPD analysis for set-associative caches.

Additionally we *studied the way the hardware could be adapted to better consider the preemption.* We essentially worked in two axes: a replacement strategy dedicated to the preemption [20], and a better memory mapping and cache or scratchpad use, for instance in [21]. Our main conclusion on this topic is that it depends not only on hardware but also on the size and properties of the software. The best adaptation should be done with hardware and software in mind.

2.2.2 Integration of CRPD into timing analysis

To work on the impact of CRPD on the Worst-Case Response Time, the software model is at higher level than the ones necessary for cache and CRPD analyses. At this level, the software is generally defined by:

- the WCET of each task,
- the deadline for each task,
- the priorities of tasks.

To add the CRPD in this software model, the main CRPD bounds we use are:

UCB the set of CRPD estimations for a preempted task (UCB-only)

ECB the CRPD estimation based on the preempting task only (ECB-only)

Comb the set of CRPD estimations when both preempting and preempted task are used (combined-UCB-ECB)

Note that the set of CRPD values is usually used to extract the worst-case value, but a multiset with the number of possible preemption points leading to a each CRPD estimation may also be used.

In our work, we look for the best *way to integrate the CRPD bound into schedulability analysis* varying on: the method to estimate the bound on the CRPD (using UCB, ECB, or combined approach) and the order of the task to be accounted for as UCB or ECB among the potential preempting tasks and preempted tasks (higher priority tasks, for instance). We studied deeply this problem depending on the category of scheduling algorithm: fixed priority [22,

23], EDF [24]. Also, we studied in [25] how to integrate the CRPD analysis into probabilistic WCET analysis.

Note that, as we have shown in [26] this way of considering the CRPD as an additional bound changes the complexity of the scheduling problem: the usual scheduling algorithms are not any more optimal (RM, DM and job-level ones such as EDF) and the problem becomes NP-hard with CRPD. We also show in [27] that no online algorithm can be optimal for the problem of scheduling a set of jobs when considering preemption delays.

2.2.3 Our work as a basis for further work

Our last work on the topic was published in 2014 [24] (2017 for the complexity [27]). Since that publication, a set of subsequent research work built upon our work. We do not give an exhaustive list here but only the main research axis as part of the CRPD analysis, and one reference to find them.

Our work has been used as a basis for extension to:

- additionally consider the persistence in the CRPD problem [15],
- integrate the CRPD to shared cache memory analyses [15],
- place the preemption points or estimate the threshold [28],
- partition cache [29].

Chapter 3

Multi-core interference analysis

As seen in previous sections, the cache memory is a shared resource that may be modelled precisely by static analysis for timing analysis. In a multi-core, there are a set of such shared resources. As in a single-core, each core is the first shared resource where some tasks may be executed or even preempted. Each core may have a private cache memory that may be analyzed using previously presented analysis. In this chapter we focus on *shared resources that can be accessed by more than one core*. For such a resource, an arbiter is generally used to decide on the shared resource access. We are mainly interested in *analyzing the interference on the bus memory access and its arbiter*.

In the multi-core we consider, each core may get a private cache memory or separated private data and instruction caches. An arbitrated memory bus is used to access a shared memory. In some of our work, this local memory may be banked: partitioned and with limited access by a subset of cores. We consider also many-core platforms where a set of multi-cores are linked by a Network-On-Chip or specific transmission ways. Our collaboration with Kalray lead to a natural application of our work to their MPPA platforms as an instance of possible platforms.

When considering interference on shared memory bus accesses, the software model considers the number of accesses to the shared memory per core: in the presence of cache, this corresponds to the number of cache misses; otherwise, it corresponds to the largest number of memory accesses. Note that this worst-case number of accesses may not be encountered along the worst-case execution path. To keep a guaranteed bound, the WCET analysis giving the worst-case path and the worst-case number of accesses must be identified in two separated steps. In our context, we consider usually that this worst-case number of accesses is given by a WCET tool such as OTAWA [2].

To start this section, we focus on *how the interference delay may be taken into account as part of the timing analysis process*. In a second section, an overview on our work on the *bus analysis* is presented. The end of this section presents an overview on the multi-core timing analysis survey we published, with the 4 categories of research work that we identified.

3.1 Integration into timing analysis

In this section, we consider that an upper-bound on the delay due to interference is given. The next step is the *integration of this delay into the timing analysis process*. To keep a guaranteed upper-bound on the timing analysis process, the separated estimation of interference delay is based on two properties: it must be correctly integrated to the other timing analysis bounds (composability), and the interference must not create any timing anomaly or unbounded domino effect [30]. The composability is quite natural and is taken into account in the work summarized in this section. The compositionality is still an open question and will be discussed in Section 4.2. Note that our work is based on a compositionality hypothesis.

The *Multi-Core Response-Time Analysis (MRTA)* [31] is our work upon which we built all work on interference analysis and multi-core timing analysis and implementation. The main idea of MRTA is based on *composability and compositionality properties*: any delay may be added to the WCET of a task. Thus, the response time of a task is composed of the WCET of a task plus any additional delay that may be incurred due to the execution of other tasks on the platform. In our original work [31], we considered the cache-related preemption delay, the delay due to scratchpad use, the DRAM memory accesses and the delay due to shared memory access (more details on this last point in the next section). Any additional detail may be found in the papers. The main take away message from this work was "let's see the multi-core implementation as an isolated implementation and a set of delays due to shared resources". It seems simple some years later but it opened a large set of work on analysis of the different delays and the way they may be integrated into response time analysis or into scheduling and mapping considerations. Also, from this work we realized that *an integration of code implementation and timing analysis* could be a good way to get more precise timing analysis and better implementation.

3.2 Bus models

As we have seen, we denote by a "bus" a way to access a shared memory and a "bus arbiter" makes the decision that must be taken on which of the processing element or core may access the shared memory as soon as there is a conflict (simultaneous access). In the original MRTA work [31], we bound the delay to access shared memory when one traditional bus arbiter is used (First In First Out, Priority, Round Robin, TDMA). This bound is based on the number of memory accesses of the task under analysis combined with the number of memory accesses of any other tasks that executes during the response time of the task under analysis and thus may generate interference delay on this response time. We reused this main ideas to *analyze the memory bus used in the Kalray MPPA Bostan and Coolidge platforms* [4, 32, 5]. We combined the one level bus arbiter equation to build an equation of the multi-level arbiters of these platforms ((i) all interference sources in a cluster, (ii) additional interference sources that comes from outside the cluster, (iii) considering a memory access that goes from a cluster to the memory of another cluster). The second level implies that we integrated a real-time calculus analysis corresponding to the NoC transfer in the Bostan platform [33]. For the Coolidge Platform, we con-

sidered and integrated a transfer inter-cluster by a crossbar (AXI) [5]. Again, those equations and analysis are used as a basis for the integration of timing analysis and implementation processes that will be seen in Part III.

3.3 About the Survey on multi-core timing analysis [1]

The idea is not to provide the survey in this report but to summarize the main take-away messages. With that aim, we present the *classification of multi-core timing analysis* and draw the main conclusions.

In the survey [1] we covered multi-core timing analysis from the first work on the topic in 2006 to 2018. The survey covers timing analysis that explicitly considers costs due to shared resource access. We identified four main categories:

Full integration This category covers works that *integrate software analyses of all tasks and the shared resource analysis.* In other words, any combination of shared resource access is considered and precisely taken into account in the timing analysis. This category of work leads to very precise bounds, but with very high complexity of the timing analysis process. This category needs a hardware/software model and our work in this category is presented in Chapter 7.

Temporal isolation In this category, *interference is avoided by use of time-division hardware* such as a TDMA bus, or by software isolation (scheduling). This isolation may be total or partial. Using isolation may be a good way to simplify the analysis process and ensure no interference. Our work on the topic considers a hardware/software model and is presented in Chapter 7.

Integrating Interference Effects into Schedulability Analysis In this category, *an interference delay is estimated and taken into account as part of the schedulability analysis.* Our work on the topic has just been presented in the previous sections (Sections 3.1 and 3.2) and also used as part of the hardware/software analysis in Chapter 8.

Mapping and scheduling In this category, *the interference delays are taken into account in the problem of finding an optimal placement of memory sections and tasks.* For a large part of our work, we consider this step as given. However, we started to study an improvement of this step to better integrate the communication costs (shared memory accesses) in Chapter 8.

Most of the discussion points are common to this report and the survey paper [1], that's why we do not summarize them here but rather summarize them in some discussion sections of the report.

Chapter 4

Discussion on hardware analyses

In this Chapter, we point out the difficulties in hardware analyses and draw a general view. Remember that our hardware analyses are not fully disconnected from software but are based on a very simplified view on the software: for instance, a control flow graph with memory block notions, for cache analysis, or a worst-case number of memory accesses for interference analysis.

Note that this chapter does not aim at giving future work that is presented in Section 10.

4.1 Complexity

A global remark on hardware analyses is that the *analysis complexity may be a real difficulty in transfering our analyses to industrial applications*. On the cache analysis side, we found a way to implement the analysis such that it scales well. However, on a large part of hardware analyses, the complexity is a big issue.

One way to reduce the complexity, is to *limit the number of analysis states/size*. This reduction may be driven by some software properties (cut or simplified model). For that aim a *better integration of software and hardware* seems a good opportunity. For instance, instead of considering each task as a potential interfering task, a better knowledge on when and where memory accesses take place may help in this complexity reduction

4.2 Timing anomaly and compositionality

All our hardware analyses are based on an additional delay in case of a bad scenario (cache miss, interference on the bus). The hypothesis that a local bad scenario leads to a longer execution time in global terms is strong and relies on the *absence of timing anomalies* (or bounded effect).

Some recent work observed that timing anomalies may happen in very simple hardware platforms. A good hope is given by research that works on how

to avoid timing anomalies by creating a new platform or only avoiding some mechanisms in the processor [34, 35].

In parallel, we did some experiments that have shown that the actual point of view on the timing anomalies may be too strong. In a platform with interference on memory accesses, timing analysis considers a large set of delays on memory bus accesses that seem to absorb the effect of timing anomaly [36]. Some complementary work also proved the absence of timing anomalies for a given application to be executed on a given platform [37]. Another way is to consider timing anomalies in the analysis [38, 13].

All that shows that *considering a timing compositional platform* where each local additionnal delay may be added to build a bound on the execution time or response time, is already possible in some platforms and on-going research should help to keep it possible.

Part II

Software models and analyses

The main collaborators for this part are:

High-level analysis Catherine Vigouroux, Pascal Raymond, Fabienne Carrier, Mihail Asavoae, Rémy Boutonnet.

LRA Catherine Vigouroux, Pascal Raymond, Nicolas Halbwachs, Erwan Jahier.

All members of the W-SEPT ANR project.

All analyses presented in the previous part consider all execution pathes as feasible. This means that there is no semantics associated with the control flow graph or other used software model. However, in a program tpically not every combination of pathes is feasible. A simple example is a program that would get a different path depending on the input value: "if $(x < 3)$" at a program point and "if $(x == 6)$" later on in sequence. On this short example we see that without any change of variable x both conditional statements will never be true on the same execution path. Considering any path as feasible,it may lead to the worst-case path executing both contradicting conditions (an infeasible path).

In this part we present some analyses that we introduce to take into account infeasible paths. Semantic analysis is generally done on higher code levels than the final binary code. The main workflow used in the approaches presented in this Part consists of analyzing the code where the information is: at high-design level –on Lustre code– and at C level. In a second step, the results are transferred to binary level where the WCET estimation is. We do not consider this transfer when presenting the analysis and discuss it in Section 6.1.

In this part, we focus on *the impact of semantic analyses on the WCET estimation*. In this context, we focus on execution paths: how to extract semantic properties and how to express them (Chapter 5). In Chapter 6, we discuss how to transfer the results to the timing analysis (binary level) and the software model level to integrate semantic analysis into timing analysis.

Chapter 5

Semantic analyses for WCET

When we mention a "feasible path" we refer to the real execution path that is on the binary level. As said previously, we do not analyze at binary level but on higher levels. Nevertheless, the binary level is where the WCET analysis is and where the final expression of our semantic properties will be. At the binary level, we saw that the semantic model is the Control Flow Graph; for C-level semantic analysis, we also work on the control flow graph and keep only the properties for which the control flow graph elements are traceable to the binary level. For higher levels, we better use the high-level structure and the traceability are not only on the control flow but also on some higher level entities.

We classify previous work in [39] based on the way they focus on 3 main steps: how the semantic properties are extracted, how they are expressed, and how they are exploited in WCET analysis. Additionally, we distinguish the software level under analysis: Model-based design, programming language level (C level generally) or binary level.

From this classification we observed that they was *a need to analyze model-based design properties as they are at binary level.* In other words, a semantic analysis that would only focus on high-level properties faces difficulties to transfer and exploit those properties on the binary level that is the target code. The following section presents our work on Lustre applications that partially answers this question.

Our second axis of work on semantic analysis focuses on the analysis of C language. We started from the observation that previous work was limited to very simple semantic properties and use Linear Relation Analysis to enlarge the scope of properties.

5.1 Infeasible executions due to high-level design

At high-level, the code usually intrinsically contains some properties on the control flow. For instance, a sequential code generated from an automaton may contain generated code corresponding to two exclusive automaton states that are

never both executed on the same execution path. However, if those two states code are transformed along the code generation and the compilation steps, it may be a useless property for the WCET estimation based on the binary code. In our work [40], the verification step uses a model-checker at Lustre level where we limited the properties to the elements we could still observe at the binary level and could lead to infeasible path properties. We extracted properties following two algorithms: (i) extract all possible properties and transfer them to binary level, (ii) formulate binary control flow properties and verify them at a high-level. All our properties are exploited as additional integer linear constraints that are used to estimate the worst-case path (Implicit Path Enumeration Technique).

Our results show that in some programs *our infeasible path analysis could largely improve the WCET analysis* (up to 50% better estimation) [40]. This result concerns mainly the automaton code generation.

5.1.1 Our work as a basis for further work

The main take-away message that has been used is "model-based design level is a good one to improve WCET". From this message, the timing analysis of synchronous programs has been more studied as in [41, 42].

5.2 Identifying infeasible paths with Linear Relation Analysis

On a programming level, we focus on C code analysis [43]. We use *Linear Relation Analysis to extract properties on the number of executions of some program points*. This counter analysis leads to the extraction of properties that were not yet exploited by previous work. The results in terms of improvement are good as soon as there is an interesting property that is nested in a loop: the impact of the property is more visible as it is valuable for all iterations of the loop. In other terms, this method is the most interesting as soon as there are a lot of loops with correlation between the number of executions of parts of the loops, or if the paths are strongly unbalanced (conditional statement lead to a path that is quite long and has much more weight in the worst-case path than its dual path).

This work has been reused in the security community in [44]: the properties on counters are used as a way to give properties that should not evolve over time and lead to attack detection.

Chapter 6

Discussion on semantic analysis and software model

The improvement due to semantic analysis is large. Thus, why not an integration of such analysis in any timing analysis? We discuss in this chapter *the issue of the traceability* that comes with any semantic analysis that is at a higher level than the executable one. Additionally, we discuss the way semantics should be taken into account in timing analysis with a trade-of between analysis complexity and improvement: what is the good software model?

6.1 Semantic analysis transfer and Traceability

Any semantic property on a higher level than binary has an impact on the timing analysis if and only if this property holds on the binary level. For such a semantic analysis, a semantic property must be extracted from the application, expressed in a way that may be used in the timing analysis and transferred to the binary level (this was the main axis of our W-SEPT ANR project [45][1]). To express our semantic properties we mainly translate them into the path analysis. The Implicit Path Enumeration Technique (IPET) is an ILP formulation whose objective function estimates the worst-case execution time solving a system of constraints. The constraints describe the control flow where the variables are a basic block number of executions along the worst-case path or its worst-case execution time bound. The type of properties that may be expressed using this method has been studied in [46].

The transfer of properties to the binary level takes place in code generator and compilation stages. In our work we used the compilation debug information to ensure that the property was still valid at the binary level, we ignore the property in the opposite case or in case of any doubt. The problem of traceability has been studied in the community [47] and some analyses have been integrated into compilers as in [48]. It is a mandatory step that is interdependent of the compiler and the selected optimization steps. This could explain why semantic analysis is generally not used in timing analysis tools.

To avoid traceability issues, a few works introduced semantic analysis at

[1] http://wsept.inria.fr/

binary level [49, 50]. It is a good opportunity to integrate semantic analysis. However, it is hard to retrieve high-level properties on the binary code, even some properties at C level may be hard to detect at this low level. This leads to an interesting open problem: how to detect, with binary code analysis, properties that may be found at higher levels?

6.2 Software models

A question still arises: *is there any software model that would be more adapted to WCET analysis ?* For hardware analysis, the main model is the CFG. For software analysis, the good model is the one that makes possible the expression of the properties to be integrated in the analysis. In any case, we notice that this level can not be independent of the complementary analysis. Thus, there may be a possible improvement by better integration of software and hardware analysis.

First, the question arises: what can be adapted? when this integration may be done? In the following Part we adapt the software to better serve the timing analysis and also be better adapted to hardware specificity. This way we found a better integration of hardware and software analysis by integrating timing analysis and implementation. As this will be done on multi-core platforms with data-flow applications, we will use a data-flow graph as the software model. This software model will be used at an entry parameter, but also modified ,if necessary, during the implementation process.

Part III

Hardware/software models, analyses and implementation

The main collaborators for this part are:

TDMA Hamza Rihani (PhD Student [4]), Matthieu Moy, Sebastian Altmeyer.

Path Julien Henry, David Monniaux, Mihail Asavoae.

MIA Hamza Rihani (PhD student [4]), Matheus Schuh (PhD student [5]), Matthieu Moy, Pascal Raymond, Amaury Graillat, Benoit Dinechin.

Models Matheus Schuh (PhD student [5]), Pascal Raymond, Joël Goossens, Benoit Dinechin.

As we have seen in previous parts, whenever one aims at a hardware timing analysis, a software model is necessary because this hardware analysis is dedicated to a specific code. Also, whenever one aims at a semantic analysis, a traceability step is mandatory to transfer the properties to the binary level, where the software is executed on a hardware platform. From these observations we guess that more precise analyses should be derived with hardware/software analysis where both analyses may be closely related.

In this part, we first focus on a *precise bus analysis that includes execution path details* to better estimate the bus delays. It uses the observation that some bus accesses are close and may be treated in the same bus slot without getting any wait on the bus for the second access. Furthermore, we show that *a feasible path analysis may be integrated into the worst-case path analysis*. The main idea is to limit the worst-case path analysis to feasible paths. The good point of both hardware/software analyses is the precision of the results. However, we also observe that a close integration of some timing analysis steps leads to poor scalability. On multi-core analysis where there is a set of hardware and software elements, such a precise analysis would not be possible on real industrial programs. That's why we looked for a higher-level integration on a less precise model but a better use of hardware and software properties.

In a second chapter, we focus on *multi-core timing analysis* and show how a higher-level hardware/software analysis can give both precise and scalable results. To obtain this result, we *integrate the timing analysis with the last step of the implementation*: this way we can play with the code generation and a precise hardware use to better know and analyze the interference. The hardware interference analysis uses the software knowledge and the software is implemented according to the hardware knowledge.

In a third chapter, we discuss the influence of the hardware architecture on all this integrated process. We also make some observations on the tradeoff between interference analysis and isolation.

Chapter 7

Local Hardware/Software models

By local, we mean a focus on a part of the hardware features or software entity. With this focus, the dedicated model is comprehensive and the corresponding timing analysis is precise.

In this Chapter, we cover two local analyses that integrate execution path and hardware analysis. The execution path leads to a refinement of the analysis: no infeasible path appears in the model and the hardware model may better integrate different execution paths.

In both cases the software model is at the level of control flow graph basic block with information about execution path conditions (if-then-else of loop conditions). The first analysis aims at better integrating semantic analysis and processor pipeline analysis. The second analysis, uses semantic analyses and processor timings to get a more precise analysis of a memory bus.

7.1 Feasible paths encoding

From the observation that usual WCET analyses do not deal with execution path analysis but integrate semantic properties, we introduced an analysis that *only considers feasible paths* [51]. For that aim, we describe the semantic of the program under analysis as a set of properties that may have an influence on the execution path. At each execution path entity (CFG basic block) there is a corresponding upper bound on its execution time. Then we look for the longest execution of the program that preserves the semantic properties.

For our proof-of-concept [51], we consider the LLVM bitcode level and generate a first-order formula that gives a model of the code semantics that influences the execution path. For that aim, the single static assignment of LLVM bitcode helps to get a simpler model. We add to this model a notion of basic block (boolean that expresses when a basic block is executed along the worst-case path) and the corresponding worst-case bounds. Using an optimization modulo theory we look for the longest path preserving the semantic properties. This initial model leads to a very complex solving, we add some cuts to help the solver: these cuts are mainly encoding that a sequence of basic blocks must not have an execution time longer than the sum of the basic blocks execution time.

With this analysis we have shown that *integrating semantic and hardware analysis may lead to good WCET bounds*. The level of semantic analysis is lower than in usual WCET analysis and leads to a more precise WCET bound. However, obtaining a more precise bound is at the price of a higher analysis complexity.

7.2 Precise bus model

With a TDMA bus, a core gets an access to the bus during a window of time that is cyclically attributed to each core. A worst-case bound for a TDMA bus considers that each access waits for the cycle length $((\#core - 1)XwindowLength$. To improve this result, from cache analysis and semantic analysis, we introduced an *analysis that encodes the path semantics and the distance between two memory accesses* [52]. With this refined model, we encode the interleaving of the bus window and the memory accesses.

As in the previous analysis, we encode the semantics and the timing that we solve with optimization modulo theory. The software model considers smaller basic blocks than the usual CFG ones, each memory access is a memory block frontier (only one memory access per block).

This analysis leads to *very precise worst-case bounds for programs executed on a multi-core with TDMA bus memory*. The main contribution for this work is to show that a more precise analysis is possible, but at the price of higher complexity. Furthermore, TDMA has a predictability property: no need of any information of what happens on other cores due to the temporal isolation that allows us to use only a local analysis. With any other bus arbiter, the interference would have much more complexity and lead to intractable complexity. This experiment shows that it is *good for the hardware analysis precision to get a detailed software model*, but this fine grained level may not be the good one for this hardware/software analysis.

7.3 Our work as a basis for further work

Our work inspired other hardware/software SMT models for WCET analysis: binary semantic analysis [53], cache analysis [54]. Furthermore, our work leads to a particular SMT model [55] that has been used as an example to improve the SMT solving : optimization modulo theory [56].

Chapter 8

Implementation and timing analysis Integration

In this chapter, we work on the *integration of implementation and timing analysis*. By implementation we mean the last step where the memory mapping takes place, the task placement and the orchestration code generation (task synchronisation and time-triggered implementation of the DAG). We observed that this last step of the implementation could deliver precious detailed information that are useful for timing analysis. Additionally, the timing analysis process could feed this implementation step to better orchestrate the execution. We are not any more estimating the WCET, that is considered as a parameter of the schedulability test that we consider here: worst-case response time as seen in Chapter 3.

On one side, there is the software to be analyzed, to get a better knowledge of the communication phases and the memory usage, we use the data-flow graph level. In the DAG we use, the useful entities are:

- each node is a program
- each edge represents:
  - a precedence: the target node can not start before the end of the source one
  - a communication: the source node may write some data to communicate to the target node.

For our work, we use DAG generated from Lustre or SCADE applications. However, our process may be used for any application with the above characteristics.

Furthermore, we use an additional property of such DAG code: the code phase. This code is usually generated with a first phase that is used to read the data; followed by a phase that executes the node code working with the data read; in a last phase, the computed values are written to be communicated to the next nodes in the DAG. The good property of a phased execution model is its separation of shared memory accesses and local memory accesses. We use it to refine the interference delay estimation and also to better avoid them during implementation. Also. we adapt this phased execution model to better fit the hardware we target and the property of the DAG.

On the other side, there is the hardware where the program will be executed, a better usage of this hardware platform is based on some of its characteristics:

- *a banked memory*: each core may access a dedicated bank, or may communicate by accessing other banks (note that it generalizes to any partitioned memory);
- *a compositional core*: interference delay may be integrated by additional delay;
- *a good knowledge of any memory access delay and hardware specificity* that may impact the execution timings.

For our work, we target the Kalray MPPA platforms (Bostan and Coolidge) as instances of platforms with these characteristics. The main reason is the good knowledge of the platforms details due to our close collaborations with the Kalray company.

From this software model and hardware knowledge, we may proceed to aim at a good hardware/software analysis by the integration of the last implementation phase and the timing analysis (interference delay, schedulability test, scheduling and mapping).

First, we present the way we integrate these analyses. Second, we show different implementation possibilities and compare them to get feedback on our integrated process. Last, we give some hint about how to improve the mapping of the DAG to a clustered many-core platform.

8.1 Multi-Core Interference Analysis: MIA

The integration of the last implementation step and the timing analysis leads to a better knowledge on the configuration of hardware and software to be analyzed, and a better implementation that takes into account the software characteristics and the way to avoid some interference.

On the last implementation step, we can modify/act on:

- *where to place shared memory and private memory*: any access to the same memory bank has a potential for interference due to the shared memory bus;
- *where to place each task to be executed*: which node on which core;
- *when to start the task execution*: preserving the precedence and software properties, the release date plays a role on the time-triggered implementation, and is a key point to get a precise interference delay estimation.

We introduced a *2-phased implementation* [57], instead of the traditional 3-phased (read-execute-write). As the DAG gives information on when and which communication are played, we observed that the implementation does not benefit from having both a read and a write phase, rather one may be enough. In fact, the knowledge of which nodes communicate, may duplicate the interference observed on the communications phases if the read and write phases are used. Instead, we use a remote write where any read to shared resource is

local, only the communication write phases accesses another bank, the bank associated with the node that is the target of the communication.

On the question of the interference analysis, using the 2-phased implementation, the interference may be observed when a write happens during the execution of a node on the target core, or when two write phases may happen simultaneously. The most used implementation method consists of modifying the phases scheduling such that those phases may never happen simultaneously. A second way is to add some delay to the timing bound of nodes to integrate the interference delay, but at the price that the global response time may be very large with the integration of any potential interference. We observed that the interference in our context does not impact so much the timing as it is limited in time and number of memory accesses. We further introduce *a method to evaluate the interference precisely and adjust the global timing* [57, 4]. For this, we analyse one period of the DAG execution. The interference is evaluated chronologically and a delay is added to the node WCET. The release date of the communicating nodes (the one that is impacted by a precedence) are estimated finely depending on the interference delay. This way instead of adding isolation delay everywhere or adding interference delay that impacts the whole DAG, the interference delay is precise and the impact minimized.

This method has shown very good results and was the first step to evaluate further and integrate additional timing analyses stages (scheduling, mapping). We applied this method to multi-core and multi-cluster (many-core) platforms [57, 33]. This work lead to the MIA tool [4] that from a DAG and timings evaluates the interference delay and assigns the release dates. This tool already progressed, the initial version [4] has been re-implemented with a more scalable algorithm [58] and additional features [5].

8.2 Predictable Execution models

To study the behavior of our integrated approach, we compare in [59] a different memory mapping combined with 3- or 2-phased implementation and the use of isolation vs. interference delay integration. As memory mapping and orchestration we compare:

- *a memory-centric approach* where any shared memory access is managed by a dedicated task that is mapped to a dedicated core and a dedicated bank. Interference is limited to the read by the shared-memory task of each produced data.
- *a dedicated shared memory bank* accessed by each node during the read and/or write phases. The interference is limited to all accesses to this shared memory bank.
- *a local memory model* where each node executes locally, reads in its predecessors memory and writes to its successors memory.

As implementation model we compare:

- *isolation vs. interference*: full isolation of the phases that could interfere vs. integration in MIA and estimation of release date depending on the interference delay.

- *3-phased vs. 2-phased model*: our remote write model is compared against the traditional 3-phased model in all configurations we use.

We compare all configurations picking a memory model, interference/isolation and 2/3 phases. We introduced some scheduling algorithm when a choice is possible in the phases order (for instance, which of the following read phases is scheduled when the previous node ends its execution).

This study has shown that for our context (DAG implementation, multi-core with fast access banked memory, application that fits in this memory), our 2-phased remote-write with interference delay integration was more efficient and the *interference delays cause less waiting than the full isolation.* About the two other memory mappings, the memory centric model has been introduced for distant memory and do not behave well in our context: sequential communication phases, only the execute phase may benefit from concurrent execution. The dedicated shared memory bank generates also a large set of interference on this shared bank. Both models are incomparable as each one behaves better in a specific context. We even implemented our study on two platforms and the trends were similar [5]. Our 2-phased model with integration of interference delays always behaves better.

The fully isolated model is generally used in industry, it gives the feeling that no problem may be caused due to interference. With our work we show that an intermediate approach with *orchestrated code to limit interference seems more convenient in terms of efficiency and gives the expected guarantees in term of safety.*

8.3 Mapping nodes to cores and clusters

For all work mentioned in this chapter, we consider given the mapping of task to core and clusters. Lately, we observed that among the mapping algorithms which focus on the placement of task on cores and clusters, none of them were guided by the communication size and placement. As the communication is a key point to better limit interference, we aim at a new mapping algorithm guided by the communication costs. Our first attempt [5] showed that the idea of guiding the task placement by the communication weight and the memory size is a good idea.

8.4 Our work as a basis for further work

Our integrated process has been used as proof-of-concept in two industrial projects[1] [60]. Our work as been integrated in the SCADE tool chain as a last implementation step. As part of the research community, our integrated process inspired other ones on the implementation process [61, 60, 62], the memory mapping [63, 64], the interference limitation [65] and the execution model [66, 67, 68].

[1] CAPACITES and ES3CAP

Part IV

Conclusion and perspectives

In this report, we summarize our work in timing analysis and its integration into the implementation process. From hardware model for timing analysis and software analysis for timing analysis, we observe that both suffer a high complexity to go further into analysis precision. Thus we combined them such that both benefit from details from the other one. Finally, for the implementation of DAG applications to multi-core platforms, we integrate timing analysis into the last implementation stage to get a timing analysis that benefit from implementation details and a time-triggered implementation that takes precisely timing analysis into account, including interference delays.

On the *hardware models and analysis*, we focus on two aspects, cache analysis and interference analysis on the memory bus. We introduced a *new way to analyze cache memory by a better efficiency of the analysis* in two ways: larger classification (less unclassified accesses) and good implementation of the analysis. Also, we study cache analysis in the context of preemptions, introducing analyses and different approaches to integrate it into the timing analysis process. We introduce a way to *integrate interference delay into worst-case response time analysis* and add some *interference models for the memory bus*.

On the software analysis, we introduce some *infeasible path analyses* to extract semantic properties that refine the worst-case execution time bound.

We introduce two *local hardware/software analyses*. We integrate (i) an infeasible path analysis to a hardware model and (ii) a path analysis to a bus model. Finally, we integrate hardware/software timing analysis to the last implementation step. This way we get *more precise interference analysis by a good knowledge of hardware and software* to which we add details about the exact release dates from the implementation; also, we get *a better implementation* with release dates that are fixed depending on interference delays.

Chapter 9

Where our work did change the point of view

Our work can be seen as a set of bricks in the timing analysis house. Some of them changed the communities point of view on the analysis and methods.

Our cache analysis has been introduced when the community thought that this topic could not be improved and was an almost closed topic. With our analysis we contributed to change the point of view on cache analysis: we have shown that *the analysis could still gain in precision* and above all we have shown that *cache analysis could be very efficient with a good implementation.*

Our work on infeasible paths has shown that there is room for large improvements with a *better integration of semantic properties.* At the design high-level, industry has shown a large interest in our work where they look for a better consideration of the characteristics of applications. Our work did open the door for future work on this topic where taking into account application modes, for instance, could be a good start.

Our work on cache-related preemption delay did open the door to a large set of work on a *better integration of hardware cost into timing analysis.* This open door lead us to the integration of interference analysis on the same model. This work on multi-core response-time analysis has been a pioneer work to *better understand and integrate interference into timing analysis.*

Finally, our integration of timing analysis and implementation has been a good proof-of-concept to *show that isolation is not the only way to consider for critical application implementation.* Our interference model combined with our 2-phased implementation led to efficient and precise analysis that starts to change the timing analysis process and its integration with implementation.

Chapter 10

The future opportunity to make all in one: where we are

We have shown good progress in the timing analysis process. All that raises open questions on the combination of analysis and improvement on the general process.

In our research community, work usually focus on one timing analysis step and there have been few progress on the combination of them. There is room for improvement by a better combination of the timing analyses. For instance, the following analyses could be better integrated:

- *cache analysis and CRPD*: we have shown with our "definitely cached useful cache block" that both analyses get complementary results, however, as far as I know, there have been very few advances in their integration. Could there be one cache analysis that would serve both for WCET and WCRT and give complementary and more precise results?

- *cache analysis and interference analysis*: in our work we consider the "worst-case number of accesses" that is the number of misses on the cache memory. However, integrating a better knowledge on which accesses they are, when they happen and how their "miss delay" is accounted as part of the WCET combines with the "interference delay" accounted as part of the WCRT could lead to more precise timing analysis and better integration of these separated steps.

- *semantic analysis and timing analysis*: as we have shown, semantic analysis refines the software model used in hardware analysis. This opens the door for improvement of all these analyses as far as a way for a good integration (traceability vs. low-level analysis) of these properties is found. For instance, high-level properties give information on reused code from one period to the next one that could be integrated into the cache analysis.

- *semantic properties and interference*: similarly, interference analysis could gain from a better knowledge of high-level properties. For instance, a

knowledge of modes could lead to infeasible path analysis in the DAG that could refine our analyses and implementation steps.

The integration of timing analysis steps in industrial processes could be eased by a *refinement of the timing analysis method*. The big picture on the different steps to combine is hard to get. There is a large door opened to better clarify timing analyses and better integrate into the industrial implementation process.

With the start of the CAOTIC project (Collaborative Action On Timing Interference), we will work in the next years on a better integration of interference analysis in timing analyses, including the timing compositionality. With the help of an industrial committee of 10 partners we will get feedback on our methods and try to go into a better integration of all timing analysis steps and implementation.

Bibliography

[1] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Comput. Surv.*, vol. 52, pp. 56:1–56:38, June 2019.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)* (S. L. M. R. P. P. P. T. Ungerer, ed.), vol. LNCS-6399 of *Software Technologies for Embedded and Ubiquitous Systems*, (Waidhofen/Ybbs, Austria), pp. 35–46, Springer, Oct. 2010.

[3] V. Touzeau, *Static analysis of least recently used caches: complexity, optimal analysis, and applications to worst-case execution time and security.* Theses, UGA - Université Grenoble Alpes, Oct. 2019.

[4] H. Rihani, *Many-Core Timing Analysis of Real-Time Systems.* Theses, Université Grenoble Alpes, Dec. 2017.

[5] M. Schuh, *Safe Implementation of Hard Real-Time Applications on Many-Core Platforms.* PhD thesis, Université Grenoble Alpes, May 2022.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, may 2008.

[7] V. Touzeau, C. Maiza, D. Monniaux, and J. Reineke, "Fast and exact analysis for lru caches," *Proc. ACM Program. Lang.*, vol. 3, pp. 54:1–54:29, Jan. 2019.

[8] V. Touzeau, C. Maiza, D. Monniaux, and J. Reineke, "Ascertaining uncertainty for efficient exact cache analysis," in *Computer Aided Verification - 29th International Conference* (V. K. Rupak Majumdar, ed.), vol. 10427 of *Lecture notes in computer science*, (Heidelberg, France), pp. 20 – 40, Rupak Majumdar and Viktor Kuncak, Springer, Jul 2017.

[9] J. Reineke, *Caches in WCET Analysis.* PhD thesis, Universität des Saarlandes, Nov 2008.

[10] D. Monniaux and V. Touzeau, "On the complexity of cache analysis for different replacement policies," *Journal of the ACM*, vol. 66, pp. 41:1–41:22, nov 2019.

[11] Z. Bai, D. Monniaux, and C. Maïza, "Plru cache analysis," in *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC19*, 2019.

[12] D. Grund and J. Reineke, "Toward precise PLRU cache analysis," in *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium* (B. Lisper, ed.), vol. 15 of *OASIcs*, pp. 23–35, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[13] F. Brandner and C. Noûs, "Precise, efficient, and context-sensitive cache analysis," *Real-Time Systems*, June 2021.

[14] G. Stock, S. Hahn, and J. Reineke, "Cache persistence analysis: Finally exact," in *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pp. 481–494, IEEE, 2019.

[15] S. A. Rashid, *Another step towards the applicability of multi-core platform to hard real-time systems.* PhD thesis, Porto University, Apr. 2021.

[16] S. Altmeyer and C. Maiza-Burguière, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, pp. 109–118, IEEE Computer Society, July 2009.

[17] S. Altmeyer and C. Maiza, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, pp. 707–719, 2010.

[18] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: Tightening the crpd bound for set-associative caches," in *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, (New York, NY, USA), pp. 153–162, ACM, April 2010.

[19] C. Maiza-Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches—pitfalls and solutions," in *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[20] J. Reineke, S. Altmeyer, D. Grund, S. Hahn, and C. Maiza, "Selfish-lru: Preemption-aware caching for predictability and performance," in *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*, April 2014.

[21] J. Whitham, R. I. Davis, N. C. Audsley, S. Altmeyer, and C. Maiza, "Investigation of Scratchpad Memory for Preemptive Multitasking," in *RTSS*, 2012.

[22] S. Altmeyer, R. I. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[23] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[24] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related pre-emption delay analysis into edf scheduling," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pp. 75–84, IEEE Computer Society, 2013.

[25] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of Probabilistic Cache Related Pre-emption Delays," in *25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pp. 168–179, July 2013.

[26] G. Phavorin, P. Richard, and C. Maiza, "Complexity of scheduling real-time tasks subjected to cache-related preemption delays," in *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pp. 1–8, 2015.

[27] G. Phavorin, P. Richard, J. Goossens, C. Maiza, L. George, and T. Chapeaux, "Online and offline scheduling with cache-related preemption delays," *Real-Time Systems*, vol. 54, no. 3, pp. 662–699, 2018.

[28] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 177–188, 2014.

[29] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Outstanding paper: Evaluation of cache partitioning for hard real-time systems," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 15–26, 2014.

[30] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany* (F. Mueller, ed.), vol. 4 of *OASIcs*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[31] R. I. Davis, S. Altmeyer, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, 2018.

[32] B. D. de Dinechin, "Consolidating high-integrity, high-performance, and cyber-security functions on a manycore processor," in *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[33] A. Graillat, C. Maiza, M. Moy, P. Raymond, and B. D. de Dinechin, "Response time analysis of dataflow applications on a many-core processor with shared-memory and network-on-chip," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019, Toulouse, France, November 06-08, 2019* (J. Ermont, Y. Song, and C. D. Gill, eds.), pp. 61–69, ACM, 2019.

[34] A. Gruin, T. Carle, H. Cassé, and C. Rochange, "Speculative execution and timing predictability in an open source risc-v core," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 393–404, 2021.

[35] S. Hahn and J. Reineke, "Design and analysis of SIC: a provably timing-predictable pipelined processor core," *Real-Time Systems*, Nov 2019.

[36] C. Allioux, "Exploration of timing anomalies on simplistic processor with model-checking," in *13th Junior Researcher Workshop on Real-Time Computing (JRWRTC19*, 2019.

[37] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, "Formal modeling and verification for amplification timing anomalies in the superscalar tricore architecture," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 3, pp. 415–440, 2022.

[38] C. Rochange and P. Sainrat, *A Context-Parameterized Model for Static Analysis of Execution Times*, pp. 222–241. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[39] M. Asavoae, C. Maiza, and P. Raymond, "Program semantics in model-based wcet analysis: A state of the art perspective," in *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France* (C. Maiza, ed.), vol. 30 of *OASICS*, pp. 32–41, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[40] P. Raymond, C. Maiza, C. Parent-Vigouroux, F. Carrier, and M. Asavoae, "Timing analysis enhancement for synchronous program," *Real-Time Systems*, pp. 1–29, 2015.

[41] M. Mendler, J. Aguado, B. Bodin, P. Roop, and R. von Hanxleden, *Logic Meets Algebra: Compositional Timing Analysis for Synchronous Reactive Multithreading*, pp. 45–67. Cham: Springer International Publishing, 2019.

[42] I. Fuhrmann, D. Broman, R. von Hanxleden, and A. Schulz-Rosengarten, "Time for reactive system modeling: Interactive timing analysis with hotspot highlighting," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, (New York, NY, USA), p. 289–298, Association for Computing Machinery, 2016.

[43] P. Raymond, C. Maiza, C. Parent-Vigouroux, E. Jahier, N. Halbwachs, F. Carrier, M. Asavoae, and R. Boutonnet, "Improving WCET evaluation using linear relation analysis," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, pp. 02–1–02:28, 2019.

[44] G. Barthe, B. Grégoire, V. Laporte, and S. Priya, "Structured leakage and applications to cryptographic constant-time and cost," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, (New York, NY, USA), p. 462–476, Association for Computing Machinery, 2021.

[45] A. Bonenfant, F. Carrier, H. Cassé, P. Cuenot, D. Claraz, N. Halbwachs, H. Li, C. Maiza, M. De Michiel, V. Mussot, C. Parent-Vigouroux, I. Puaut, P. Raymond, E. Rohou, and P. Sotin, "When the worst-case execution time

estimation gains from the application semantics," in *8th European Congress on Embedded Real-Time Software and Systems*, (Toulouse, France), Jan 2016.

[46] P. Raymond, "A general approach for expressing infeasibility in implicit path enumeration technique," in *International Conference on Embedded Software (EMSOFT 2014)*, (New Dehli, India), Oct. 2014.

[47] H. Li, I. Puaut, and E. Rohou, "Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation," in *RTNS - 22nd International Conference on Real-Time Networks and Systems*, (Versailles, France), Oct. 2014.

[48] S. Blazy, A. Maroneze, and D. Pichardie, "Formal verification of loop bound estimation for WCET analysis," in *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers* (E. Cohen and A. Rybalchenko, eds.), vol. 8164 of *Lecture Notes in Computer Science*, pp. 281–303, Springer, 2013.

[49] J. Ruiz, H. Cassé, and M. de Michiel, "Working around loops for infeasible path detection in binary programs," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 1–10, 2017.

[50] A. Mangean, J.-L. Béchennec, M. Briday, and S. Faucou, "BEST: a Binary Executable Slicing Tool," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)* (M. Schoeberl, ed.), vol. 55 of *OpenAccess Series in Informatics (OASIcs)*, (Toulouse, France), pp. 7:1–7:10, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2016.

[51] J. Henry, M. Asavoae, D. Monniaux, and C. Maiza, "How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics," in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14*, pp. 43–52, june 2014.

[52] H. Rihani, M. Moy, C. Maiza, and S. Altmeyer, "WCET analysis in shared resources real-time systems with TDMA buses," in *RTNS 2015*, 23rd International Conference on Real-Time Networks and Systems, (Lille, France), Nov 2015.

[53] J. Ruiz and H. Cassé, "Using smt solving for the lookup of infeasible paths in binary programs," in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, (Lund, SE), pp. 95–104, OASICs, Dagstuhl Publishing, 2015. Thanks to Dagstuhl Research Online Publication Server (DROPS)Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0. This paper appears inOpenAccess Series in Informatics (OASIcs)ISBN 978-3-939897-95-8 ISSN 2190-6807 The definitive version is available at: http://drops.dagstuhl.de/opus/volltexte/2015/5260/.

[54] D.-H. Chu, J. Jaffar, and R. Maghareh, "Precise cache timing analysis via symbolic execution," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–12, 2016.

[55] D. Monniaux, "A survey of satisfiability modulo theory," in *Computer Algebra in Scientific Computing* (V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, eds.), (Cham), pp. 401–425, Springer International Publishing, 2016.

[56] R. Sebastiani and P. Trentin, "Optimathsat: A tool for optimization modulo theories," *Journal of Automated Reasoning*, vol. 64, pp. 423–460, Mar 2020.

[57] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor," in *RTNS 2016*, 24th International Conference on Real-Time Networks and Systems, (Brest, France), Oct 2016.

[58] M. Dupont De Dinechin, M. Schuh, M. Moy, and C. Maiza, "Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems," in *DATE 2020 - Design, Automation and Test in Europe Conference*, (Grenoble, France), pp. 1–4, Mar 2020.

[59] M. Schuh, C. Maiza, J. Goossens, P. Raymond, and B. Dinechin, "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory," in *RTSS 2020, Real-Time Systems Symposium*, 2020.

[60] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat, "Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware," *ACM Trans. Archit. Code Optim.*, vol. 16, jul 2019.

[61] M. Lo, N. Valot, F. Maraninchi, and P. Raymond, "IMPLEMENTING A REAL-TIME AVIONIC APPLICATION ON A MANY-CORE PROCESSOR," in *42nd European Rotorcraft Forum (ERF)*, (Lille, France), Sept. 2016.

[62] C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold, "Automated generation of time-predictable executables on multicore," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, (New York, NY, USA), p. 104–113, Association for Computing Machinery, 2018.

[63] A. Tretter, G. Giannopoulou, M. Baer, and L. Thiele, "Minimising access conflicts on shared multi-bank memory," *ACM Trans. Embed. Comput. Syst.*, vol. 16, sep 2017.

[64] R. Koike and T. Azumi, "Federated scheduling in clustered many-core processors," in *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–8, 2021.

[65] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21853–21882, 2022.

[66] B. Pourmohseni, F. Smirnov, S. Wildermann, and J. Teich, "Isolation-aware timing analysis and design space exploration for predictable and composable many-core systems," *CoRR*, vol. abs/1905.13503, 2019.

[67] R. Meunier, T. Carle, and T. Monteil, "Correctness and Efficiency Criteria for the Multi-Phase Task Model," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)* (M. Maggio, ed.), vol. 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 9:1–9:21, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[68] P. Wu, C. Fu, T. Wang, M. Li, Y. Zhao, C. J. Xue, and S. Han, "Composite resource scheduling for networked control systems," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 162–175, 2021.