Execution time prediction for applications running on multi-core architectures

Rémi Meunier

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut National des Sciences Appliquées de Toulouse

---

Présentée et soutenue par

Rémi MEUNIER

Le 30 novembre 2023

Prédiction du temps d'exécution d'applications dans des architectures multi-coeur

---

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Thierry MONTEIL et Thomas CARLE

Jury

M. Emmanuel GROLLEAU, Rapporteur
Mme Angeliki KRITIKAKOU, Rapporteure
Mme Christine ROCHANGE, Examinatrice
M. Julien FORGET, Examinateur
M. Thierry MONTEIL, Directeur de thèse
M. Thomas CARLE, Co-directeur de thèse

Remerciements

Je remercie d'abord mes deux directeurs de thèse Thierry Monteil et Thomas Carle. Nos points réguliers dès le début de la thèse, alors que nous étions en pleine crise sanitaire, ont été essentiels à la réussite de ma thèse. Durant nos échanges je me suis toujours senti écouté et encouragé, le tout avec bienveillance et souvent avec humour. Je leur suis reconnaissant pour la confiance qu'ils m'ont accordé, leur patience, et parce qu'ils m'ont beaucoup appris. Le co-encadrement a été je trouve très bénéfique grâce à l'alliage entre les expériences, compétences et caractères de chacun qui étaient assez différents.

Je tiens aussi à remercier Angeliki Kritikakou et Emmanuel Grolleau d'abord pour avoir accepté d'être rapporteurs puis pour leurs commentaires qui ont permis d'améliorer la qualité du manuscrit. Merci aussi à Julien Forget, Christine Rochange et Julian Thévenard qui m'ont fait l'honneur de participer à mon jury de thèse.

Dès que j'ai pu me rendre régulièrement à l'IRIT, j'ai beaucoup apprécié les moments que j'y ai passés en compagnie des deux équipes auxquelles j'appartenais (TRACES et SEPIA). Je me suis senti bien accueilli, et pour cela je remercie les deux cheffes d'équipe Patricia Stolf et Christine Rochange. Je prenais aussi plaisir à participer aux évènements d'équipe. Durant mes journées sur place, j'étais ravi de partager un bureau avec Alban, Mickaël puis Louison, Noïc et Jérémie qui sont arrivés plus récemment. J'ai toujours pu y travailler sérieusement tout en pouvant compter sur leur bonne humeur, ce qui était précieux pour moi. Je n'oublierai pas non plus les pauses repas où nous rejoignions le plus souvent Christine et Thomas pendant lesquelles j'ai beaucoup ri.

J'ai aussi passé du temps dans l'entreprise Randstad Digital du fait de mon contrat Cifre. J'ai connu Xingyu dès mon arrivée à Toulouse et nous avons eu beaucoup d'occasions pour discuter tout au long de la thèse. Il m'a souvent rassuré en me transmettant son expérience de doctorant Cifre. Joséphine a aussi été d'une grande aide au début de la thèse. Elle a tout fait pour me faire démarrer dans de bonnes conditions, et je mesure à quel point ce n'était pas évident. Je remercie également beaucoup Julien et Inès, qui ont pris le train en route mais m'ont très bien accompagné pour la deuxième partie de la thèse. Eux aussi m'ont fait profiter de leur expérience de doctorant. Nos points de suivi resteront pour moi d'excellents moments.

Mes proches ont joué un rôle essentiel pendant ces trois années. J'ai eu la chance de pouvoir compter sur leur soutien indéfectible et leur enthousiasme dès que je leur ai parlé de mon projet de revenir dans les études (que j'avais quittées alors moins d'un an auparavant). Je parle ici de mes parents, ma soeur et mon frère, et de ma compagne Rosanne qui a en plus dû me supporter au quotidien et parfois me réconforter. Elle m'a donné beaucoup d'élan et de motivation par l'attention qu'elle portait à mon bien être et même à mes travaux, malgré le peu de points communs entre nos domaines de recherches.

Je voudrais aussi remercier mes amis qui m'ont encouragé. Mon éloignement géographique a rendu les occasions de les retrouver rares mais elles en sont devenues plus précieuses pour moi.

Contents

List of acronyms

HRT Hard Real-Time

WCET Worst-Case Execution Time

WCRT Worst-Case Response Time

BCET Best-Case Execution Time

ACET Average-Case Execution Time

CFG Control Flow Graph

ISA Instruction Set Architecture

SESE Single-Entry Single-Exit

IPET Implicit Path Enumeration Technique

DMA Direct Memory Access

TDMA Time Division Multiple Access

RR Round Robin

FIFO First-In First-Out

EDF Earliest Deadline First

TIPs Time Interest Points

StAMP Static Analysis of Memory Access Profiles

KDE Kernel Density Estimation

PREM PRedictable Execution Model

AER Acquisition Execution Restitution

ACO Ant Conly Optimization

PSO Particle Swarm Optimization

SA Simulated Annealing

GA Genetic Algorithm

MCTS Monte-Carlo Tree Search

ML Machine Learning

RL Reinforcement Learning

DRL Deep Reinforcement Learning

ILP Integer Linear Programming

SDE Starting Date Enumeration

IPH Iterative Priority Heuristic

ASAP As Soon As Possible

DAG Directed Acyclic Graph

EASA European union Aviation Safety Agency

Thesis summary in English

The execution time of a task depends both on the input data, which define the sequence of instructions executed, and on the execution platform, which determines the duration of these instructions. In the context of real-time critical systems (e.g. avionics, spatial), it is crucial to know the Worst-Case Execution Time (WCET) of the tasks in order to guarantee that they all meet their timing constraints, which ensure a safe execution of the system (e.g. deadline). The WCET problem has already been studied in a lot of research works but they must be updated to take into account the recent hardware evolutions. Therefore, for more than a decade, multi-core architectures have been at the center of many research works. Indeed, manufacturers prefer them over single-core architectures because they offer performance, energy and space gains. However, timing verification is more challenging for these architectures due to the presence of shared resources that are sources of interference. Contentions appear when at least two cores simultaneously attempt to access a same resource. In this case, some tasks may take longer than expected to use the shared resource. This additional delay must be taken into account in the WCET of the tasks although it is difficult to predict the number of contentions that may occur and their moment.

The first part of the thesis focuses on the multi-phase task model, which can be used during the scheduling and the interference analyses to compute the contentions with more precision. Indeed, tasks are no longer represented as a single temporal block but divided into several phases, each of which represents a portion of the task execution. With this model, accesses are no longer considered to be performed from the beginning to the end of the task but rather in a subset of the phases. This model was introduced prior to the thesis but we propose a more generic multi-phase model that we use to conservatively compute the number of accesses that may be performed in the phases for any multi-phase profile. We present 3 formal correctness criteria which guarantee that the method to compute the accesses in the phases is safe, and that the implementation of the model in the code, using synchronizations, is correct given a static schedule.

The second part proposes techniques to design and optimize multi-phase profiles from their execution traces. The first technique is based on Kernel Density Estimation (KDE) that is used to cluster the instructions performing accesses in phases. We also present optimization and correction methods to maximize the efficiency of the profile during the interference analysis, along with a method to select synchronizations. Then, as our problem is multi-criteria and admits a lot of solutions, we propose two methods based on Genetic Algorithms (GA) respectively to create and to optimize multi-phase profiles. The design techniques presented are compared and evaluated using two case studies.

The last part addresses the static scheduling of multi-phase tasks in multi-core architectures. The problem is presented with an ILP formulation that considers both the tasks dependencies and the effects of contentions to minimize the worst-case response time of the system. Then, we propose heuristics with different strategies to take advantage of the multi-phase model and to include the effects of contentions. We performed a statistical study over a large number of synthetic multi-phase tasks systems in order to compare the methods in different configurations and to prove the efficiency of the multi-phase model. The experiments end with two case studies that confront the heuristics to more realistic multi-phase profiles and bigger systems.

Résumé de la thèse en français

Le temps d'exécution d'un programme varie en fonction de ses entrées, qui définissent la séquence des instructions exécutées et de la plateforme d'exécution qui détermine la durée de ces instructions. Dans le contexte des systèmes temps-réel critiques (avionique, spatial...), il est crucial de connaître le temps d'exécution pire-cas des tâches, ou Worst-Case Execution Time (WCET), afin de garantir qu'elles respectent les contraintes temporelles qui régissent le bon fonctionnement du système comme la date d'échéance. De nombreuses recherches ont déjà étudié la prédiction du temps d'exécution et le WCET mais elles doivent être mises à jour pour tenir compte des évolutions matérielles. Ainsi, depuis plus d'une décennie, beaucoup de ces recherches ciblent les architectures multi-cœur qui sont plébiscitées par les industriels (meilleures performances, efficacité énergétique, gain de place...). Cependant, la présence de ressources partagées entre plusieurs cœurs fait apparaître des interférences lorsqu'ils tentent d'y accéder de façon concurrente. Certaines tâches peuvent alors attendre plus longtemps que prévu pour être servies par la ressource. L'analyse temporelle doit prendre en compte ces délais bien que la quantité d'interférences et leurs dates soient compliquées à prédire.

La première partie de la thèse s'intéresse à l'utilisation d'un modèle de tâche plus précis qui peut être utilisé durant l'ordonnancement et l'analyse d'interférences appelé modèle multi-phase. Il consiste à représenter une tâche non plus sous la forme d'un seul bloc temporel mais divisée en plusieurs phases. Par conséquent, au lieu de considérer que les accès d'une tâche peuvent s'effectuer du début à la fin de cette tâche, on peut compartimenter ces accès dans les phases et ainsi calculer les interférences à l'échelle plus fine des phases. Ce modèle a déjà été utilisé avant la thèse mais nous proposons une formalisation plus générique ainsi qu'une méthode pour calculer de façon conservative le nombre d'accès dans les phases de n'importe quel profil multi-phase d'une tâche. Nous introduisons aussi 3 critères de correction pour garantir que la méthode de calcul des accès est sûre et que l'implémentation du modèle multi-phase dans le code, à l'aide de synchronisations, est correcte d'après un ordonnancement statique donné.

La deuxième partie propose des techniques de construction et d'optimisation de profils multi-phase à partir des traces d'exécution d'une tâche. La première technique utilise l'estimation de densité par noyau avec pour objectif de grouper les instructions effectuant des accès dans des phases. À cette technique s'ajoutent des optimisations et corrections pour maximiser l'efficacité du profil durant l'analyse d'interférences, ainsi qu'une méthode pour sélectionner des synchronisations. Par la suite, nous proposons 2 méthodes basées sur des Algorithmes Génétiques (GA) respectivement pour créer et optimiser des profils. En effet, les GA sont adaptés aux problèmes multi-critères avec un large espace de solutions. Nous utilisons deux cas d'études pour comparer et évaluer les méthodes présentées.

La dernière partie s'intéresse à l'ordonnancement statique des tâches multi-phase sur les architectures multi-cœur. Le problème est présenté avec une formulation ILP considérant des dépendances entre tâches et les possibles effets des interférences pour minimiser le pire temps de réponse du système. Ensuite, nous proposons des heuristiques avec différentes stratégies pour tirer profit du caractère multi-phase des tâches et inclure les effets des interférences. Enfin, nous menons une étude statistique avec des systèmes synthétiques pour comparer les méthodes dans différentes configurations et mesurer l'efficacité du modèle multi-phase. L'étude se poursuit avec 2 études de cas pour confronter les heuristiques à des formes de profils plus réalistes et de plus gros systèmes.

Publications

- *Correctness and Efficiency Criteria for the Multi-Phase Task Model.* Rémi Meunier, Thomas Carle, Thierry Monteil. Euromicro Conference on Real-Time Systems ECRTS 2022.
- *A statistical study of the multi-phase representation for real-time tasks running in multi-core processors.* Rémi Meunier, Thomas Carle, Thierry Monteil. IFSE: journées FAC 2021.

Chapter 1

Introduction

Contents

1 Context

A real-time system is a system whose correctness depends both on the logical results it produces and on the time at which the results are produced [1]. For example, the pressure sensor of an avionic system must be re-executed periodically because the pressure changes over time. Hence, the data read at a given time instant will be considered unreliable at some point in the future. For this reason, real-time tasks are generally subject to timing constraints expressed by their *release date*, i.e. the date from which the task can be executed, their *deadline*, i.e. the time instant at which the results of the tasks must be produced and their *period*, i.e. the interval between two release dates. In the context of a critical application, it is required to verify that each task will always meet its deadline in order to guarantee its correctness. This guarantee can be verified by computing the Worst-Case Execution Time (WCET) of the tasks [2]: an upper bound on their execution time. The execution time depends on the inputs of the task that define which sequence of instructions (i.e. execution trace) is executed and on the execution platform that determines the set of instructions and their execution time. The difficulty to compute a tight (i.e. close to the real value) and safe (i.e. that is not under-estimating the real value) WCET has increased over the years due to the addition of complex micro-architectural features [3]. These features are mainly designed to improve the average execution time because performance gain is important for most of the usages. Nonetheless, it is prejudicial to WCET analysis. The reason is that the modelling of such mechanisms adds complexity and deteriorates the predictability. Still, WCET analysis is a necessary step in the development of safety-critical real-time systems to guarantee that a deadline miss cannot occur. At the same time, these systems are required to carry out an increasing number of functionalities which justifies the employment of accelerator mechanisms.

In this context, single-core platforms are outdated and industry looks towards multi-core architectures [3, 4] even for critical applications [5, 6, 7]. The main benefits of using multi-core architectures are that they allow multiple tasks to run simultaneously so their computational capacity is better and they also improve energy efficiency and space utilization. Moreover, as multi-core is now the standard architecture, they represent most of the large series manufactured so they are also more interesting economically. Regarding WCET analysis, these architectures pose additional challenges compared to single-core ones due to the presence of shared resources. Indeed, in multi-core platforms, several tasks can be executed in parallel and have access to shared hardware components such as the main memory, caches or I/O peripherals. Simultaneous accesses from different tasks to the same resource may cause its saturation and lead to interference when a task has to wait until

the resource can serve its request. This situation inevitably affects the execution time of the task but is not taken into account in usual WCET analyses. Moreover, the timing of interference events is highly unpredictable if no mitigation strategy is implemented. Hence, considering the effects of interference in the WCET is challenging. This is pointed out by several certification authorities in a common document named *CAST-32A* [8] published in 2016.

2 Contributions

Multi-core interference induces additional delays to take into account during WCET analysis due to the presence of shared resources. Some experiments showed that these delays can increase the WCET by a factor of up to 2.96 [9]. Their mitigation is a difficult problem because there exists many sources of such interference (caches, interconnect, memory) that depend on the considered architecture, and a variety of approaches can be explored to either eliminate or limit them.

A first distinction needs to be made between a dynamic and a static timing analysis approach. On the one hand, dynamic timing analyses are executing the program under study on the target architecture or a simulator to measure the execution time of the program. As the worst-case path is generally not known beforehand, a safety margin is applied using probability techniques because there is no guarantee that the real WCET has been measured. On the other hand, static timing analyses are not executing the software but they combine information from the code of the program and from the hardware to provide the WCET estimate. This thesis relies on static timing analysis approaches since, as opposed to the dynamic approach, it is possible to guarantee that the WCET estimate is safe (i.e. equal to or higher than the real WCET). However, the main drawback of static methods is the tightness of their estimations as they tend to over-estimate the maximum execution time in order to guarantee the system safety in any execution scenario. Indeed, static analyses generally consider a lot of cases that can appear throughout the system execution based on the combination of software and hardware states. Abstract models are used to capture a conservative approximation of these states so the combination of the approximations aggravate the final over-estimation.

One method to cover the worst-case interference scenario in a schedule is to multiply the WCET of the tasks by a factor which accounts for the effect of the possible contentions. Otherwise, the analyses must compute a bound on the number of possible accesses that tasks may perform to each shared resource and derive the maximum number of contentions that the tasks can suffer given a schedule. A solution to increase the precision of the interference computation is to consider new abstraction models to represent the tasks with a finer grain. This is the objective of the multi-phase representation of tasks that splits the tasks into multiple phases so that the interference can be computed at the phase level instead of the task level. Initially, tasks were only cut into 2 or 3 phases and required substantial code modifications to enforce the model during execution [10, 11]. More recently, some works have proposed to lift these restrictions and to rely directly on existing code [12, 13] to build the phases. This thesis is based on this approach to address the multi-core interference problem. The contributions rely on a generic multi-phase model of tasks from which we derive properties to guarantee a safe analysis and a conservative computation of the worst-case interference scenario. In addition, we propose new methods to create and optimize the multi-phase representation of tasks from their binary code.

We also studied several scheduling techniques to find how to benefit the most from the multi-phase representation in order to reduce the makespan of a schedule. Again, both dynamic and static approaches exist for this problem. Dynamic approaches make scheduling decisions during the execution of the systems, based on the current date and state of the tasks. Doing so allows the system to efficiently adapt to online events and to task execution time variability. However, providing timing guarantees is challenging in this case (even with single-core architectures) because the behavior of the system is hard to predict in advance. As a consequence, static solutions are generally preferred over dynamic solutions for critical systems. We also chose to use a static ap-

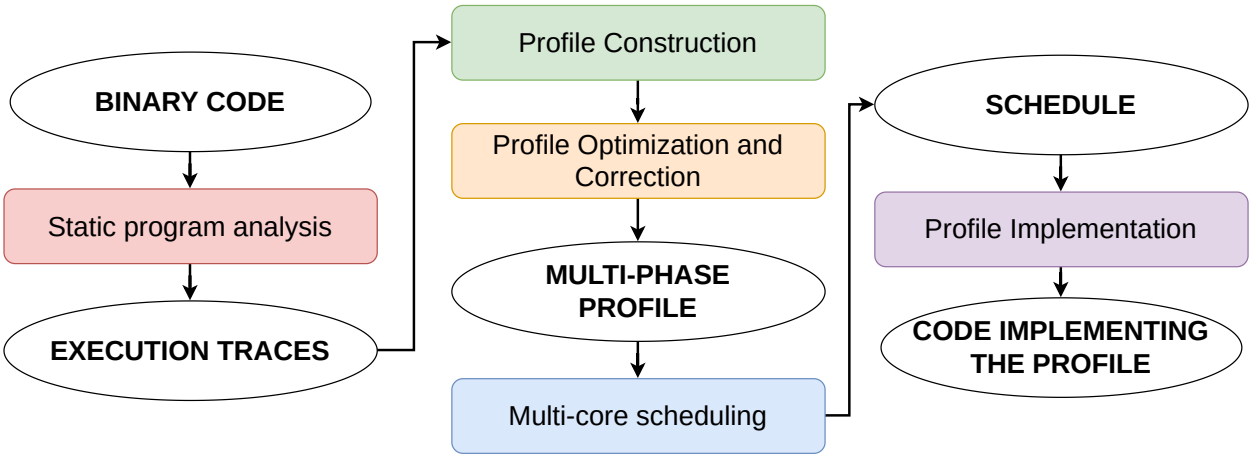


Figure 1.1: Main steps from the static analysis of the program to the computation of the system schedule

proach to this problem. Most of the existing static scheduling techniques are either based on the usual single-phase task model or rely on the 2 or 3-phases models [10, 11] that lack generality and are mainly used to completely eliminate interference. Accounting for contentions in a schedule in a conservative manner can actually yield better makespan improvements than eliminating interference, according to [14]. We adopted this philosophy to design our scheduling methods because it is more suitable to a generic multi-phase model where the presence of phases without accesses is not constrained nor guaranteed, unlike in the 2 and 3-phases models.

Figure 1.1 presents the main steps a framework can perform in order to implement a multi-phase task system according to a schedule based on the binary code of the tasks composing the system. The contributions of the thesis are located between the Profile Construction step in green and the Profile Implementation step in purple. Chapter 2 introduces the WCET problem with the main techniques to compute the WCET in single-core architectures and explains why they need to be adapted for multi-core platforms due to the apparition of new sources of interference (see red rectangle, Figure 1.1). The chapter ends with the state of the art approaches to mitigate the effects of multi-core interference during temporal analyses. Chapter 3 begins with an informal explanation of the benefits and challenges of the multi-phase task model before presenting a formal and generic definition. Then, it proposes a set of properties that allow to safely account for the accesses that may occur in each phase of a task (see green rectangle, Figure 1.1) and to correctly implement the multi-phase model. The properties are independent from the method used to generate the multi-phase profile of the task under study. Following this contribution, Chapter 4 introduces new techniques to build a multi-phase profile. In addition, it presents optimizations to improve the trade-off between the effort to enforce the execution of the model and the reduction of the over-estimation of contentions in the schedule (see orange rectangle, Figure 1.1). Chapter 5 compares the design techniques and optimizations using two case studies. Chapter 6 presents the contributions of the thesis to the multi-core scheduling problem using the multi-phase model (see blue rectangle, Figure 1.1). We propose an ILP formulation of the problem as well as a list of heuristics with different strategies. Finally, Chapter 7 begins by evaluating the potential of the multi-phase model over the single-phase model using the ILP formulation to compute optimal schedules. Then, a comparative study of the scheduling techniques of Chapter 6 is conducted in which we also discuss the influence of the shape of the profile on the computation of interference. The chapter ends by applying the heuristics on two case studies.

Chapter 2

State of the art: worst-case execution time problem and multi-core scheduling

Contents

This chapter introduces the Worst-Case Execution Time (WCET)problem and the state of the art approaches to address it. The first section is dedicated to static WCET analysis with single-core architectures where no inter-core interference can occur. It presents the main approaches to compute the WCET of a task and mentions some issues to understand the limitations and the difficulty of WCET analysis. This analysis is sometimes used as a baseline before extending it to multi-core architectures in the literature. The second section highlights the limitations of the existing WCET analyses when the system is executed on a multi-core architecture. Indeed, most of these analyses target single-core architectures. Then, we present a list of existing approaches, including multi-core scheduling techniques, to handle interference on shared components, as well as their limitations in order to understand how the work of this thesis addresses some of them.

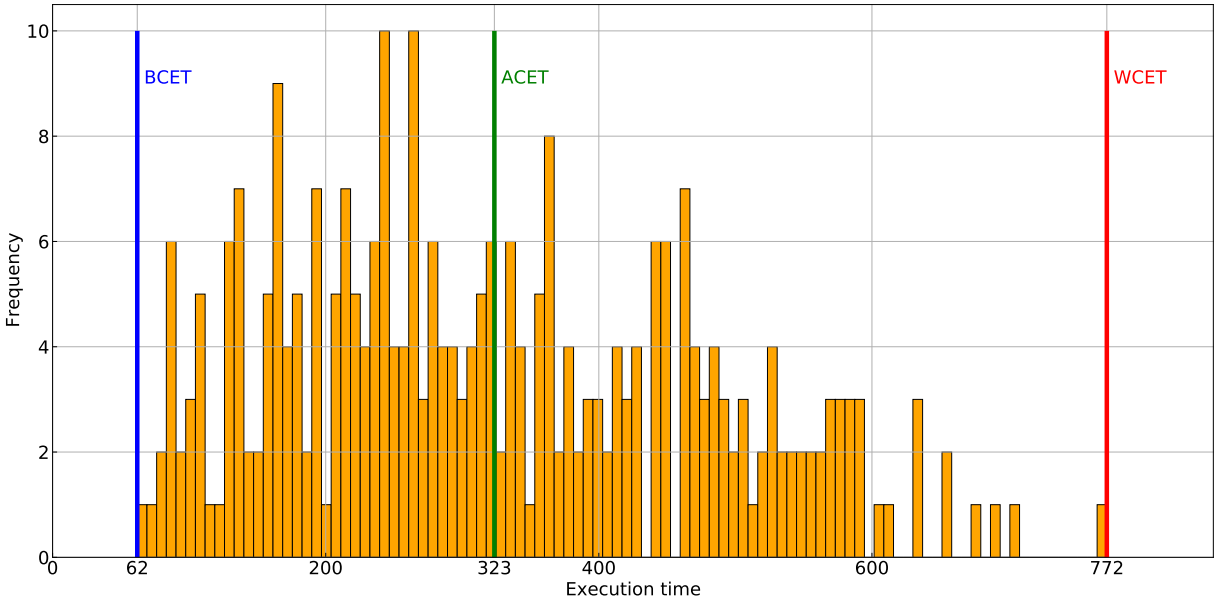


Figure 2.1: Distribution of execution times of a dummy program in number of processor cycles.

1 Single-core architectures

1.1 Correctness of safety-critical systems

Safety-critical systems are systems whose failure may provoke unacceptable and catastrophic events that can severely endanger humans or the environment (e.g. in avionics, space or nuclear contexts). In order to be commercialized, these systems must comply with regulations managed by certification authorities such as the European union Aviation Safety Agency (EASA) to demonstrate their safety and reliability. In the context of real-time systems, correctness not only depends on the results produced but also on the time at which they are produced. The real-time constraints are generally defined with deadlines derived from the system they control and systems are categorized according to the consequence of missing a deadline. In this thesis, we only consider Hard Real-Time (HRT) systems in which missing a single deadline results in a total failure of the system and is prohibited. For such systems, it is necessary to compute the WCET of the tasks, an upper bound on their maximum execution time, in order to later prove that they cannot miss their deadline.

1.2 Execution time of a task

The execution time of a task depends both on the input data and on the execution platform. A naive way to determine the WCET of a task is to execute it with all the possible inputs for all the possible initial states of the architecture on which it will execute and to store the end-to-end measurements. Then, it is possible to represent the distribution of the execution times measured as depicted in Figure 2.1 for an arbitrary program (i.e. the distribution does not come from a real case). From this distribution we can determine the Best-Case Execution Time (BCET), i.e. the lower bound of the execution time, the Average-Case Execution Time (ACET) and the Worst-Case Execution Time (WCET) of the task. However, such an exhaustive test is intractable in practice for real systems not only because it is very complicated to derive all the possible inputs but also because these input data, multiplied by the number of initial hardware states to consider, represent an intractable number of tests to perform. Moreover, one must consider the complexity of setting an initial hardware state and feeding input data with precision.

Still, there exists measurement-based approaches to WCET estimation that execute the tasks on the target hardware or on a simulator. Only a subset of inputs are executed and then, the

maximum observed execution times can be combined with safety margins in order to obtain the overall WCET estimation. However, there is no guarantee that the WCET is not underestimated because we generally do not know the worst-case execution path so we cannot know if the worst-case execution time has been measured.

Such a guarantee can be offered by a static analysis, which does not execute the task but combines software and hardware analyses to cover all the possible execution paths of the task and derive an upper bound of the WCET. However, static analyses are usually more difficult to conduct because of the complexity of the code and of the hardware that may also contain advanced features (e.g. out-of-order execution, branch predictor...) that are challenging to model. A trade-off must be found between precision and complexity. In this thesis, we focus on static methods to conduct the timing analysis of programs.

1.3 Static analysis

Static analysis methods aim at determining the WCET of a task without executing the code. The analysis relies on a abstract model of the hardware architecture and on the set of possible control flow paths of the task. The two are combined to determine how the architecture may impact the code execution. Then, it is possible to derive an upper bound of the real WCET (the worst-case execution path is potentially infeasible). The main drawback of static analysis is its computational complexity because all the possible hardware states must be considered at each point of the control flow and for any set of input data in order to guarantee conservative bounds on the execution time. This requires efficient yet conservative abstract models of the code and the hardware components. Indeed, such models must consider a trade-off between keeping as much useful information as possible to ensure tightness and approximating some data to limit the analysis complexity. A static analysis relies on several sub-analyses. We can divide them into three steps:

1. The control flow analysis that retrieves the possible execution paths of the program, possibly including loop bounds.

2. The processor behaviour analysis that computes the impact of each hardware component on the execution time of the instructions.

3. The bound computation that combines the information of the two former steps to estimate the WCET.

Listing 2.1: a C function as example

```
int example (int a)
{
    if (a < 10){
        a ++;
    }
    else if(a > 10){
        a--;
    }
    return a;
}
```

Control-flow analysis The control-flow analysis retrieves the necessary information on the execution paths of the task. Most of the time, this analysis is based on the Control Flow Graph (CFG) of the program. This is a directed graph that represents the possible execution paths of the program. Each node of the CFG is a Basic Block (BB), which is a sequence of instructions with exactly *one* entry point and *one* exit point. The edges represent the possible control flow of the program between the BBs. It is better to perform the control flow analysis from the disassembled

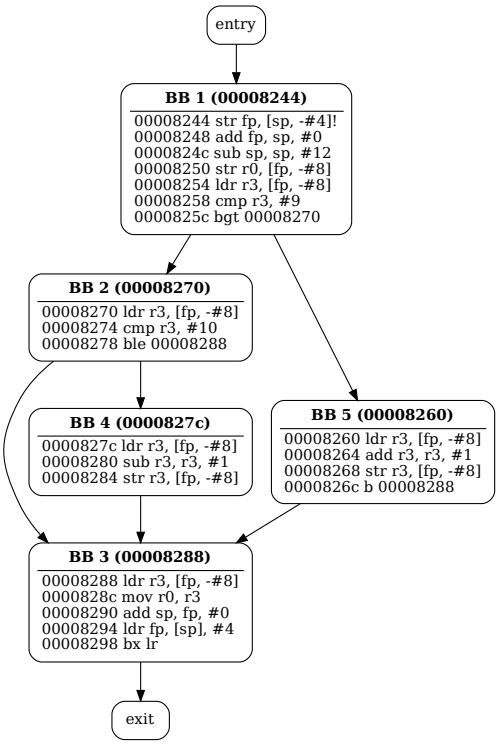


Figure 2.2: Control Flow Graph of function *example*

Listing 2.2: assembly code of function *example*

```
1
2       str fp, [sp, #-4]!
3       add fp, sp, #0
4       sub sp, sp, #12
5       str r0, [fp, #-8]
6       ldr r3, [fp, #-8]
7       cmp r3, #9
8       bgt .L2
9       ldr r3, [fp, #-8]
10      add r3, r3, #1
11      str r3, [fp, #-8]
12      b .L3
13  .L2:
14      ldr r3, [fp, #-8]
15      cmp r3, #10
16      ble .L3
17      ldr r3, [fp, #-8]
18      sub r3, r3, #1
19      str r3, [fp, #-8]
20  .L3:
21      ldr r3, [fp, #-8]
22      mov r0, r3
23      add sp, fp, #0
24      ldr fp, [sp], #4
25      bx lr
```

code of the program rather than from the source code since the latter does not take into account the potential compiler optimizations so the real control-flow may differ from what is analyzed. An example function in C is given in Listing 2.1. The corresponding ARM disassembly code is written in Listing 2.2 and Figure 2.2 is the CFG of the program obtained using OTAWA [15], a static WCET analysis tool. From this CFG it is possible to retrieve all the execution paths of the program. Additional information can be added to tighten the WCET estimates by limiting the set of paths to explore:

- Loop bounds: it is necessary to set an upper bound on all loops, otherwise an infinite WCET will be derived that corresponds to the infinite repetition of a loop. The more precise the bounds, the tighter the WCET estimation.
- Targets of indirect branches: for example, when using pointer calls, the possible paths to consider depend on the possible targets of the call. Failure to detect the targets of a call may lead to the underestimation of the WCET.
- Infeasible or mutually exclusive paths: these are control flow paths that cannot be taken by the program during its execution. For example, when two *if-then-else* structures follow each other and have conditions relying on the same variables, it is interesting to study the possible combinations of their variables values. If a particular combination of condition values (e.g; true for the first if and false for the second if) cannot be obtained by varying the variables values, then the corresponding path is infeasible. An infeasible path can also be a path through dead code.

This information can be obtained automatically by analyzing the code [16, 17, 18] but the analysis may lack precision when the control-flow structures are complex (e.g. nested loops with if-then-else

structures). Therefore, most of the WCET frameworks allow user annotations to specify e.g. loop bounds where analysis is unable to derive automatically produce their results.

In the thesis, we consider that no infeasible paths analysis is performed. This assumption has implications that are discussed in Chapter 4 Section 1.2.3.

Processor behaviour analysis The computation of the execution time for an individual instruction requires to analyze the processor behaviour. The hardware state is given by the state of its components (e.g. memories, bus, pipeline) that changes throughout the execution paths. Abstract interpretation allows to model the state of hardware components with conservative approximations such that the successive states are computed with limited computations. For example, we can conduct a cache analysis to determine the instructions guaranteed to be present or absent in the cache at each point of the program [19]. Usually, such an analysis is composed of 3 sub-analyses:

- The *MUST* analysis checks that a memory block is always present in the cache at a specific program point. If it is the case, then the contents (instruction or data) in this memory block are classified as *Always-Hit* (AH) meaning that the access to this instruction will always result in a hit.
- The *MAY* analysis checks if a memory block has a possibility to be in the cache. This analysis allows to classify instructions or data as *Always-Miss* (AM) if its memory block cannot be in the cache.
- The *PERSISTENCE* analysis is not always conducted but increases the precision of the analysis. It checks if, after being loaded, a memory block is not evicted before it is needed again and classifies instructions as *persistent*. This is useful in the context of loops for example when an instruction is not present at the first iteration but will always result in a hit in the subsequent iterations.

The instructions that can not be classified as AH, AM or persistent are set to *Not Classified* (NC).

In order to compute the execution time of the basic blocks, an analysis of all the components involved in the execution (e.g. pipeline, branch prediction...) is necessary. As processors become more complex, the analyses make more conservative hypotheses, which tend to reduce the tightness of the WCET estimation.

WCET computation The most common techniques to combine the flow and timing information gathered during the two former steps are:

- Tree-based Technique [20, 21]: this technique builds a syntax tree that represents the structure of the program. The leaves of the tree are basic blocks while the other nodes are either "sequence", "loop" or "if" control-flow structures. The syntax tree is traversed in a bottom-up manner: the execution time of the leaves (computed during the processor behaviour analysis) are combined according to rules defined for each control-flow structure until the root of the tree. For example, the resulting execution time bound of an "if" node is the maximum execution time of its branches. A limitation of the method is that the control-flow structure of the binary code may not correspond to the structure of the source code due to compiler optimizations.
- Path-based Technique [22, 23]: the longest path is searched among all the possible paths of the program. In the presence of many conditional statements, the number of paths grows exponentially. Hence, the analysis becomes intractable unless using some heuristic search to only study a subset of the paths but then there is no guarantee to find the longest path.
- Implicit Path Enumeration Technique (IPET) [24, 25]: this technique builds a set of constraints from the CFG of the program that are then used to formulate an Integer Linear

Programming (Integer Linear Programming (ILP)) problem. Basically, each basic block i has a time constant t_i which is the upper bound of its execution time (determined by the processor behaviour analysis) and a count variable x_i that expresses the number of time it is executed. The latter variables are subject to constraints that express the control flow of the program, and the objective function maximizes $\sum_{i \in BBs} x_i \times t_i$ to obtain an upper bound of the WCET. The major drawback of this method is the use of ILP, whose resolution time grows quickly with the number of constraints and variables.

1.4 Limitation of static analyses

Although static methods provide a safe WCET estimate, a recurrent issue is the tightness of the estimate, i.e. how close the estimate is to the real WCET. The over-estimation comes partially from the abstract interpretation that inevitably makes conservative assumptions in order to cover all the possible paths (the abstract states are approximations of possible real states). However, it also comes from the processor behaviour model that can be based on partial information about the hardware characteristics (e.g. latency of some operations, arbitration policies...) and incomplete or imprecise flow facts (e.g. loop bounds, branch targets...). In summary, the tightness of a WCET estimation is degraded by the accumulation of conservative assumptions that are necessary to guarantee its soundness.

2 Multi-core architectures

2.1 Emergence of the multi-core architectures

For more than a decade, single-core architectures improvements have been slowing down: performance gains from miniaturization requires a lot of effort and it becomes difficult to continue increasing the clock frequency due to important power issues (consumption and heat dissipation) [4]. At the same time, the demand for performance continues as industrials want to integrate more and more features in their systems (e.g. automotive industry with autonomous vehicles), with a growing interest for energy efficiency. Multi-core architectures offer a good solution to address the performance needs in this context. A multi-core architecture is a processor with several independent processing units, named cores. The advantages of multi-core compared to single-core architectures are:

- Parallelism: several programs can be executed simultaneously, so more instructions can be executed per second.
- Power efficiency: more instructions per watt can be executed.
- Space utilization: the cores coexist on the same chip and can share some components (e.g. memory).
- Heterogeneity: several types of cores can be placed so that the trade-off between energy and computation efficiency can be optimized. For example, a high performance core can execute computationally expensive tasks while other tasks can use energy efficient cores.
- Centralization: the coexistence of cores in the same chip allows a centralized control over them.

Therefore, multi-core architectures offer a performance leap and are often preferred over single-core, particularly for complex embedded systems where numerous single-core platforms are required to execute the applications. The adoption of multi-core platforms is then also promising for critical systems such as avionics [6] where multiplying the number of single-core platforms is an inefficient

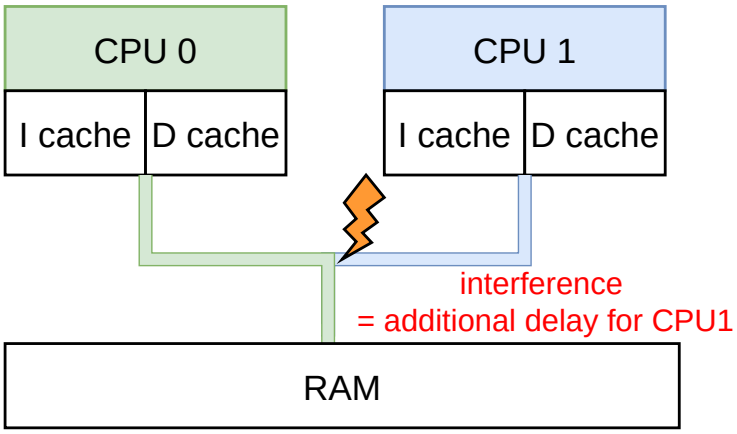


Figure 2.3: A simultaneous access to the main memory bus causing an interference.

solution to face the increasing number of requirements and functionalities. Moreover, chips manufacturers encourage the use of multi-core architectures for critical systems because the single-core processors now represent a small proportion of their production.

2.2 Challenges of WCET analysis for multi-core systems

Although multi-core architectures have already replaced single-core architectures in many embedded systems, their adoption for safety-critical systems faces important barriers due to predictability issues [5]. Indeed, different programs are executed on several cores that share some hardware components such as caches, memories or buses. Hence, they may try to access the same component at the same time, creating *contentions* (also referred to as *interference*). In this case, an arbitration mechanism decides which core is served first and delays the others. We call this delay an *interference penalty* or a *memory contention duration* in the remainder of the thesis. An example of contention is depicted in Figure 2.3. CPU0 and CPU1 both use the interconnect to access the main memory, but only one can be served at a time. In this example, CPU1 is waiting for CPU0 to complete its memory transaction. Such a delay is not taken into account when computing the WCET of the task running on CPU1 but the total additional delay that can occur must be bounded in order to guarantee the safety of the system. Precisely determining the amount and the moment when contentions happen is difficult. This uncertainty is prejudicial to the predictability of the system and then to the tightness of its timing analysis. Several certification authorities from Europe, America and Asia have published together an informational position paper *CAST-32A* [8] (originally in 2014 and revised in 2016) and then their own guidelines such as [26] dedicated to the use of multi-core processors in avionic systems. They argue that an interference-aware safety analysis must be conducted for HRT systems: the interference channels must be identified, classified according to the acceptability of the interference they may produce, and then mitigation mechanisms must be implemented to guarantee that these channels cannot cause system failure.

3 Considering and mitigating the effects of interference

Research on timing verification techniques for multi-core platforms is not recent according to a survey [27] whose oldest papers are from 2006. The number of papers addressing this topic has increased throughout the years showing that many questions remain open.

3.1 Impact of multi-core interference on the execution time

Several papers have investigated the sources of interference in multi-core architectures [28, 9, 29]. The main sources are: shared buses, I/O peripherals, shared caches or main memories where

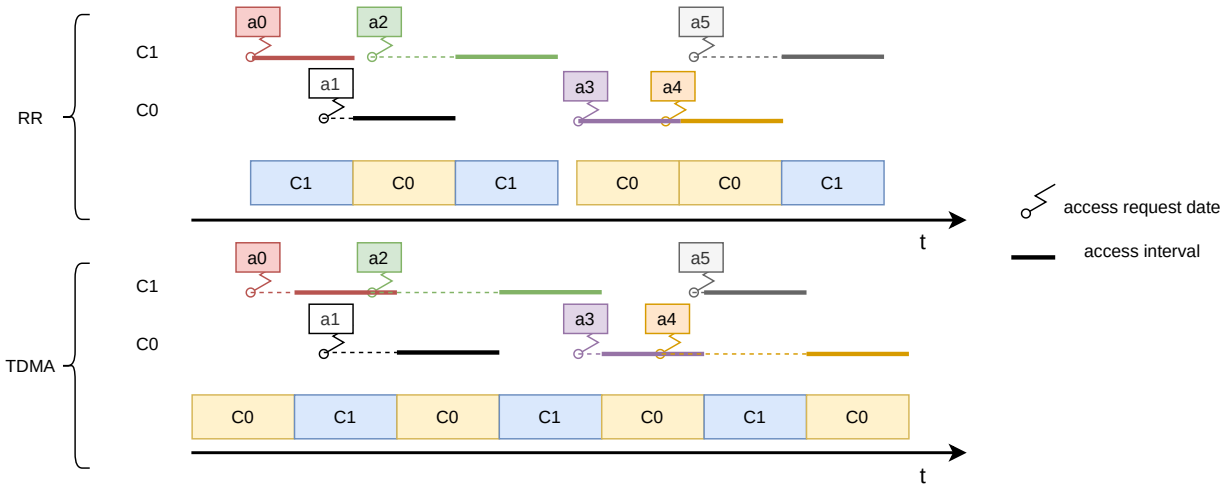


Figure 2.4: Example highlighting the difference between Time Division Multiple Access (TDMA) and Round Robin (RR) bus arbitration.

different cores may compete to access the resource. In addition to interference from simultaneous requests served in sequence, shared caches are subject to another type of interference where task may evict the data of another task executing on a different core. In this thesis we consider multi-core architectures such as the one presented in Figure 2.3 where cores have local private memories (e.g. private L1 caches or scratchpads) and are connected to the main memory by a shared bus. We focus on shared memory bus interference but the techniques that we present can be adapted and applied to any type of interference.

In [9], Pellizzoni et al. compare the WCET of applications in isolation and in the presence of interference. They measure an increase of up to 2.96 times the maximum time observed in isolation by saturating the main memory using a Direct Memory Access (DMA) component. The difference between the WCET in isolation and the one in the presence of interference may be even greater using static analyses because it is very challenging to predict the behaviour of tasks executing in parallel on other cores. For this reason, some pessimism is introduced in order to cover the worst-case scenario. Such pessimism is detrimental to the schedulability of a system because it increases the amount of time that must be provisioned for each task to be executed.

In [30], the authors model a variety of possibly shared resources (caches, interconnect, main memory...) with different possible arbitration policies. They directly compute a response time bound for each task based on their execution traces without considering the WCET in isolation. This work, later extended in [31] and [32], is a comprehensive study with explicit interference modeling that provides precise and safe estimates. Nonetheless, relying on all the execution traces of all the tasks in a system is intractable for realistic applications. Therefore, the traditional two-step approach is often preferred: first, tasks are analyzed in isolation to compute their WCET and the number of access they may perform and then a response time analysis that bounds the effects of interference is conducted with respect to a given schedule.

3.2 Isolation-based techniques

One approach to completely eliminate contentions in a system is the use of software or hardware isolation mechanisms. The advantage of these methods is that they are independent from the schedule of the task system because the worst-case scenario is identical whether or not another task is executing in parallel (i.e. the WCET is the same as computed by a static analysis in isolation). This isolation can be ensured by using bus arbitration policies such as Time Division Multiple Access (TDMA) where each core periodically has a dedicated time slot to access the bus as in [33, 34, 35], Round-Robin (RR) [36, 37], or compatible with both policies [38]. Although

bounding additional delays is easier with TDMA arbitration, it is not work-conserving as opposed to RR. Thus, when a core has no pending request the bus is idle even if the next core is waiting to access the bus. Figure 2.4 depicts a scenario of access requests from two cores and how they are served by RR and TDMA at the top and at the bottom respectively. We use the granularity of the requests in this example but one core may actually perform several requests per slot depending on the time to perform one access and the duration of a time slot. We see that the work-conserving nature of RR allows it to serve request a0 and a3 immediately and to serve a4 just after a3 even if it comes from the same core because C1 has not yet issued a5 when the slot serving a3 is finished. As a result, a5 is served faster by TDMA than by RR. Generally, these arbitration policies improve the predictability and facilitate the timing analysis but they also result in an important under-utilization of the resources due to their context-independent nature. Indeed, by default the worst-case analyses pessimistically assume that each access suffers the worst-case delay. To address this issue, isolation is combined with other techniques to provide more context and more precise bounds on the number of access requests performed in a time window. For example, [36] proposes a method to compute this bound for each core assuming that tasks have already been assigned to cores. In addition, [9] uses a more elaborated task model to derive this bound from arrival curves (this model is described in more details in the next section).

The bound on the number of access requests can also be enforced directly by dividing the memory bandwidth among the cores. The principle is that each core has an initial memory budget that is consumed each time it performs an access and that is replenished periodically. The memory budget can be updated by monitoring performance counters for example. If a task requests an access but the budget of its core is exhausted then the task is suspended and resumed at the beginning of the next regulation period. MemGuard [39] implements such a reservation system that is also able to redistribute unused bandwidth from a core to the others in order to efficiently use the memory bandwidth. However, the system relies on dynamic predictions so this redistribution may not be adapted to HRT systems. Indeed, a misprediction can cause a shortage of bandwidth budget for a task that may then miss its deadline. In [40], the bandwidth reservation servers consider both memory and computation time bandwidths. Therefore, the execution in a core is suspended whenever one of its budgets is empty. The schedulability analysis is performed in 2 steps: first a local analysis ensures for each server that its tasks are schedulable and then the analysis is performed on the entire system considering all the servers. In spite of an increased precision and more flexibility than TDMA, the main drawback of these methods still are the performance of the system that cannot fully benefit from the advantage of multi-core architectures.

3.3 Advanced task models

Another approach is to model tasks as a succession of temporal slots, called *phases*, representing a portion of the code task execution. By analyzing the code portion that may be executed in a phase it is possible to bound its execution time and the number of access requests that can occur during its execution. This *multi-phase task model* increases the precision of the interference analysis because individual accesses are no longer considered to occur at anytime during the task execution but only in one or a subset of the phases. In [9], the phases are called superblocks and the authors derive arrival curves to bound the memory access requests for sequences of superblocks that can cover several tasks. These arrival curves are then used to compute the maximum delay due to interference of the superblocks with a dynamic programming approach and according to different bus arbitration policies.

The PRedictable Execution Model (PREM) is introduced in [10]. In its initial form, PREM proposes to decompose the tasks in two phases: first a *memory phase* that performs all the memory transfers between the memories local to the core (caches or scratchpads) and the shared memory such that the results of the previous computation are written in the shared memory and all the instructions and data that can be used during the execution are loaded in the local memory, and second an *execution phase* that executes the task without performing any memory access (outside

the core private memory). This model was originally designed for single-core platforms to prevent interference between I/O peripherals and the core in a shared interconnect during data transfers. In short, the peripherals are only allowed to use the interconnect when the core is executing an execution phase or if it is idle. However, as pointed out by the authors, the model can be applied to multi-core systems to avoid that cores use the interconnect at the same time. The PREM model was later extended in [11] with the Acquisition Execution Restitution (AER)) model: in addition to the acquisition and execution phases that correspond to the PREM phases, an extra memory phase (restitution) at the end of the task writes the computation results into the shared memory. This new model specifically targets multi-core architectures.

Initially, the phased models were only used to statically compute an interference-free schedule by preventing memory phases (acquisition and restitution) from executing simultaneously. Some of these works are listed in Table 2.1 (see page 25) but the list is not exhaustive. Different strategies have been studied over the years. Many works use this model to extend previous works that used TDMA. They propose strategies to efficiently fill the slots attributed to each core with the memory phases in order to improve the schedulability of the system under study. For example, in [41], several scheduling strategies are compared and they observe that prioritizing the scheduling of memory phases over computation phases yields the best results. They add that the results are even better when computation phases are scheduled with Earliest Deadline First (EDF) on each core afterwards. They also try schedulers without TDMA bus arbitration where memory phases are scheduled globally (and EDF is used to schedule tasks locally on the cores). For this scheduling scheme, they conclude that least-laxity first is the best policy to schedule the memory phases. A further proposition to efficiently use the memory bus is to create two partitions in the local memory such that, while a computation phase executes with one partition, the data to execute the next computation phase can be loaded using a DMA. The idea is firstly proposed for a single-core processor but the authors argue that their work can be extended to multi-core architectures using a TDMA bus [42] such that the data transfers cannot interfere with each other. Later, they directly address multi-core architectures without using TDMA but presenting a global scheduling algorithm [43].

In [44], the authors transform PREM tasks with 3 phases into a Resource Constrained Project Scheduling (RCPS) problem. This type of problem organizes the completion of activities (e.g. in a project or an industrial production process) with limited resources, resource usage constraints (e.g. capacity), and precedence constraints between activities. As the phases execute on the same processor non preemptively, the authors propose to view the set of cores as one take-give resource with a capacity equal to the number of cores in the considered architecture. When a task executes its first memory phase it takes one unit of the take-give resource (i.e. one core of the execution platform). When the last memory phase has completed its execution, the resource unit is released (i.e. the core either is idle or begins the execution of another task). This allows to extend an existing technique [45] addressing the RCPS problem whose results are close to optimal solutions retrieved using an ILP formulation.

The contentions-free constraint is sometimes lifted such as in [14] that compares both 2 and 3-phases models scheduled with different configurations and with or without tolerating contentions. Their results show that letting memory phases overlap and accounting for the worst-case delays due to contentions actually reduces the Worst-Case Response Time (WCRT) of the system compared to when the memory phases are isolated. Likewise, [46] proposes to schedule AER tasks with an RR bus using an ILP and a heuristic by considering both the case with and without interference. As explained in [47], performing the interference analysis of a schedule is complicated by the interdependence between the execution intervals of the tasks, that define which tasks scheduled in parallel can interfere, and the number of contentions that the tasks can suffer. Indeed, during an interference analysis, the duration of the tasks are inflated by a timing penalty corresponding to the maximum number of their accesses that can be interfered. However, by extending the duration of the tasks we may also change the set of tasks that can contend with them and delay

their successors. Hence, the interference analysis must be recomputed until a fix point is reached. Instead, [47] presents an algorithm to compute the contentions faster by adding tasks progressively in the schedule. It uses a time cursor starting at date 0 that jumps to the next end of task at each iteration to compute the contentions so that it never has to go backwards to recompute some contentions. Although developed independently, we use a similar algorithm in this thesis in order to perform interference analyses.

Although PREM and AER models are convenient to suppress interference and ensure predictability, they have important drawbacks. First, the code must comply with the model of memory and computation phases: either the code is written specifically to use this model or it must be transformed. Numerous techniques have been proposed to transform functions to their multi-phase model using compilation tools [48, 49, 50, 51, 44, 14, 52] but they still require important modifications, which is an issue for legacy code. Moreover, the first memory phase has to load all the instructions and data that could be used during the execution phase. Therefore, the data loaded that is not used during the task execution may occupy a large part of the available storage space (and may be too large to fit into the local memory) and still occupies the bus during the memory transfers. Recently, two models have emerged that are more adapted to legacy code because they propose to derive a multi-phase representation from the binary code directly: the Static Analysis of Memory Access Profiles (StAMP) [13] method builds phases that correspond to "well-formed" regions in the code and the Time Interest Points (TIPs) method [53, 12] builds phases around instructions that may perform memory accesses. In [53], an ILP formulation is presented to schedule multi-phase tasks so that the makespan is minimized in the presence of contentions. However, there is only one task per core and no dependencies between the tasks.

3.4 Creating a multi-phase task model from its code

In addition to their suitability for legacy code, the TIPs and StAMP methods do not restrict the number of phases or force the alternation of memory and execution phases. They allow to build other profile shapes that can be adapted for scheduling optimizations that tolerate memory contentions. In order to take into account the contentions in the schedule, these methods require a safe and precise accounting of the potential accesses performed in each phase to conduct an interference analysis.

In the StAMP method [13], phases are directly derived from Single-Entry Single-Exit (SESE) regions of the code that are identified after an analysis of the CFG. The authors propose to use a node-centric definition of SESE instead of the usual edge-centric version because this allows a more fine-grained division of the code. The Worst-Case number of Memory Accesses (WCMA) and the partial WCET of each phase are both computed using IPET. The direct link between code regions and phases is both a strength and a weakness. Indeed, the authors note that dynamic scheduling strategies could benefit from knowing the code portion under execution. In the meantime, some control-flow structures strongly limit the possibilities to create phases. In their experiments, the authors found tasks where only one very long phase was surrounded by very short other phases so the multi-phase model may not behave differently than the single-phase model in this case.

The creation of phases involves more complex computations with the TIPs method [53, 12] because it is based on the analysis of the execution traces of the tasks. Since such an analysis may be intractable for real programs, the authors propose to enumerate the traces from a lightweight version of the CFG called TIPsGraph. The TIPsGraph represents only the instructions that may perform an access, named Time Interest Points (TIPs), along with the possible control flow between them. The TIPs can be extracted from a cache analysis with a static analysis tool such as OTAWA [15]. The other instructions are abstracted into the edges of the TIPsGraph by computing partial WCETs indicating the maximum execution time between the source node to the destination node of the edge. Figure 2.5 shows the TIPsGraph of the example function of Listing 2.2. The numbers next to each edge indicate the WCET of the transitions computed from the instructions that are not represented.

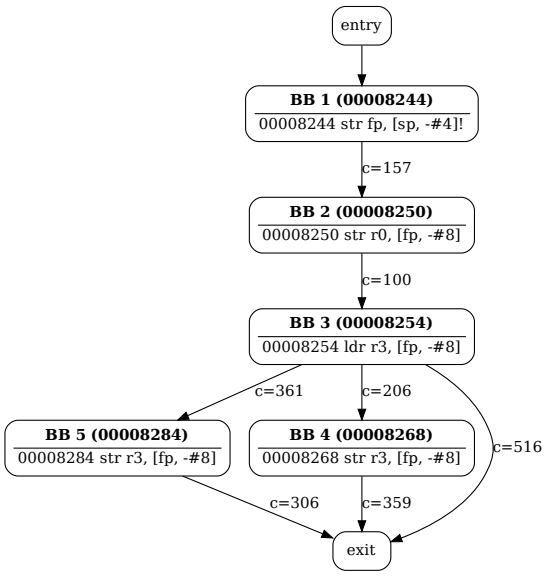


Figure 2.5: TIPsGraph of the example program in Listing 2.2

This representation is later used to enumerate the execution traces of the task, resulting in an abstraction that exhibits only the necessary information to compute the accesses in each phase: the possible sequences of instructions that may perform memory accesses (TIPs) and their worst-case date. In order to create phases from this abstraction, it is proposed to extract phases that are guaranteed to perform no memory access in each trace based on the worst-case date of TIPs and their worst-case memory access latency. Then, the phases created for each trace are combined and adjacent phases smaller than a parameter δ are merged together (whether they perform accesses or not). This generates profiles in which phases of size around δ perform accesses and larger phases do not perform accesses. By varying δ, one is able to generate varied profiles for a given task.

4 Employing meta-heuristics and machine learning techniques to schedule tasks

Due to the potentially high number of possibilities to map and schedule tasks in a multi-core platform, ILP-based methods are fit only for small systems with few tasks and cores while the proposed heuristics are greedy and tend to exploit one solution rather than exploring other possibilities. In order to cover a larger space of solutions, some works have employed meta-heuristics and machine learning methods.

4.1 Meta-heuristics

As opposed to heuristics, meta-heuristics techniques do not depend on the considered problem. Many of them are nature-inspired. For example, Ant Conly Optimization (ACO) mimics the way ants communicate with pheromones using several agents that cooperate to explore a space. Also, Simulated Annealing (SA) simulates a process used in metallurgy to slowly cool metals such that their crystalline structure reaches a near optimal energy state. A recent survey [54] points out the proliferation of these algorithms in research works. They identified 540 different meta-heuristics, 385 of which have appeared in the last ten years. They also observe that several of these meta-heuristics are very similar. According to them, the most popular meta-heuristics are Particle Swarm Optimization (PSO), Genetic Algorithm (GA) and SA.

Table 2.1: List of approaches to tackle interference through schedulability analysis

Source	Task system	Task model	Contentions-free	Method	Remarks
[60]	DAG	3 phases	yes	scheduling	Different bus access policies compared with TDMA
[61]	periodic	PREM	yes	scheduling	TDMA bus, promotion of memory phases over computation phases
[41]	periodic	PREM	yes	scheduling	TDMA bus, separate scheduling policies for memory phases and execution phase
[42]	sporadic	PREM (3 phases)	yes	scheduling	TDMA bus, partition of local memory to anticipate the load of next tasks to execute using DMA
[43]	sporadic	PREM (3 phases)	yes	scheduling	Extends [42] by global scheduling without TDMA arbitration
[62]	periodic	PREM (3 phases)	yes	scheduling	ILP + heuristic that gives priority to write phases over read phases
[63]	periodic	PREM (3 phases)	yes	scheduling	Promotion of memory phases over computation phases
[46]	DAG	AER	yes/no	scheduling	Round-Robin bus, ILP + heuristic, scenario with or without memory phases overlapping are both considered
[44]	take-give	PREM (3 phases)	yes	scheduling	Converts the problem to a resource-constrained project scheduling problem
[64]	DAG	single-phase	not considered	scheduling	Uses Monte-Carlo Tree Search to explore the scheduling possibilities
[65]	DAG	single-phase	not considered	scheduling	Uses Monte-Carlo Tree Search combined with Deep Reinforcement Learning
[51]	sporadic	PREM	yes		TDMA bus, tasks are optimally split into a DAG of intervals based on the PREM model according to the priorities assigned
[66]	periodic	single-phase/PREM	no	WCRT analysis	ILP to reduce the pessimism when computing interference delays
[67]	sporadic	PREM (3 phases)	no	WCRT analysis	Compares different arbitration policies for the memory controller
[68]	DAG	single-phase	not considered	scheduling	List scheduling with rules that prioritize tasks along the critical path
[14]	DAG	PREM/AER	yes / no	scheduling	Comparison of different scheduling schemes
[69]	periodic	single-phase	no	mapping	ILP + heuristics to define allocation strategies that reduce interference
[70]	DAG	single-phase	not considered	scheduling	Scheduling with Deep Reinforcement Learning
[71]	periodic	AER	no	WCRT analysis	Bounds the number of bus contentions suffered by AER tasks knowing the tasks that may execute in parallel.
[53]		TIPs	no	scheduling	ILP considering one task per core
[72]	periodic	PREM	yes	scheduling	Memory phases are scheduled by an ILP or a heuristic, computation phases are scheduled with EDF
[73]	DAG	single-phase	not considered	scheduling	Scheduling with reinforcement learning
[37]	sporadic	AER	no	WCRT analysis	Round-Robin bus

Meta-heuristics have been employed in many forms of scheduling problems even before the 2000's including multi-core scheduling problems. For example, [55] uses a GA to assign tasks to cores and schedule them in an heterogeneous architecture with the objective to minimize the makespan of the system (i.e. the end date of the last task executed). A GA is a population-based method that reproduces natural selection processes (crossover, mutation) to explore and select promising solutions, called chromosomes. In [56] the GA is only used to assign priorities to tasks and a heuristic is managing the assignation of tasks to cores. In [57] and [58] some heuristics are directly integrated into respectively a GA and a PSO algorithm in order to accelerate their convergence. The meta-heuristics are also used for multiple objective optimization problems such as in [59] where the objective is to find a trade-off between the makespan of an application, and the resources needed to execute it using ACO.

4.2 Machine Learning

Another alternative to traditional heuristics is the use of Machine Learning (ML) techniques. They allow to adapt easier to changing environments by learning from past decisions and their effects. A survey on machine learning techniques to improve energy efficiency can be found in [74] but they also appear to minimize the schedule makespan in DAG scheduling problems. In [75], given a task order, a Deep Reinforcement Learning (DRL)-based method is used to assign the tasks to cores in an heterogeneous architecture. The state of the system is represented by the earliest start time of the tasks on each processor. Capturing information of the DAG is important for the performance of the ML-based methods so [70, 76, 73] use Graph Neural Networks (GNN) that can directly work on graphs and extract their information.

ML techniques can also be combined with meta-heuristics. Yano et al. [77] study how to schedule a DAG in a clustered many-core processor by taking into account the different communication times if tasks are scheduled in the same cluster (using a bus) or not (using a Network-on-Chip). The GA encodes the communication channel used by the tasks to communicate. Each solution of the GA is then used by a Reinforcement Learning (RL) technique to determine the best scheduling order. Finally, the tasks are assigned to cores with a heuristic and the makespan of the solution under study is computed to assess its quality. In [65], a RL technique is used in a Monte-Carlo Tree Search (MCTS). This meta-heuristic explores a decision tree and uses simulations to reach final states (i.e. a compete schedule in our problem) from selected nodes in order to evaluate their quality with respect to the considered problem. Thanks to the statistics from simulations results, the algorithm can progressively choose only the best solutions. In order to make MCTS converge faster, a DRL technique is added to select the most promising branches when expanding the tree and for the simulations.

5 Conclusion

Extending single-core WCET analyses to multi-core platforms is challenging due to the presence of shared components that are the source of interference between the cores. Static methods for single-core architectures already suffer from precision issues and the unpredictability of interference aggravates the over-estimation for multi-core architectures.

For this reason, numerous research works address the mitigation of the effects of interference. Some of them consider isolation approaches but they result in an important under-utilization of the resources. In order to resolve this issue, advanced tasks models such as PREM and AER have been developed that represent tasks with an alternation of memory and execution phases (2 or 3 phases). It is then possible to build interference-free schedules by forbidding memory phases from being executed at the same time on different cores.

More recently, new multi-phase models directly built from the code of the tasks have been proposed. The idea is to generalize the PREM and AER models by building tasks with an arbitrary

number of phases. The worst-case number of memory accesses in the phases must be accounted for in a conservative manner so that an interference analysis can be conducted on the schedule to compute the possible effects of interference. We follow this approach in this thesis.

Many works have proposed solutions to schedule tasks systems on multi-core platforms. However, few of them consider a generic multi-phase model to represent the tasks and take into account the effects of interference while tolerating contentions in the schedule. The solution space to schedule tasks in multi-core platforms is potentially very large as it includes both the mapping of tasks to cores and the choice of the order or the dates of the tasks. Therefore, some works employed meta-heuristics or machine learning methods to explore the solution space more efficiently. To the best of our knowledge, none of them considered multi-phase tasks and the effects of interference, which further enlarge the solution space.

Chapter 3

The multi-phase task model

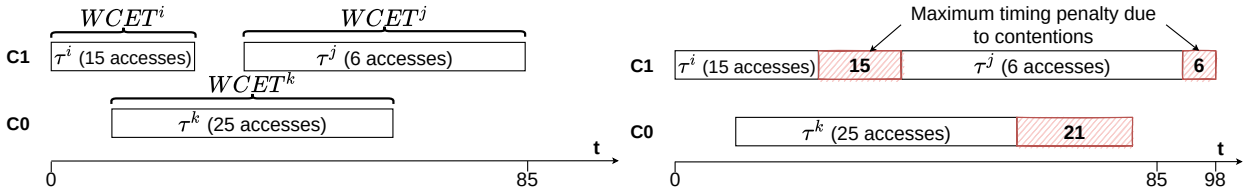
Contents

The previous chapter introduced the WCET problem, the challenges posed by multi-core platforms that limit the transposition of the methods initially designed for single-core architectures, and gave an overview of the state of the art techniques to mitigate the effect of the memory interference during the interference analysis. This chapter initiates the focus of the thesis on the multi-phase task model as an approach to address the memory interference problem. The first part provides some insights about the benefits of using this model with a preliminary discussion about its limits, then the model is defined formally and a method to correctly account for the number of accesses in the phases is presented along with a description of the consequences of the interference analysis on the model.

1 Introduction to the multi-phase representation

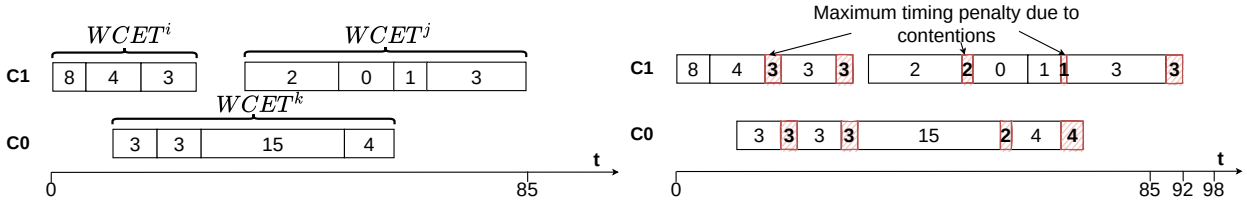
1.1 Advantage of the multi-phase representation

The occupancy of a core by a task is traditionally represented as a single time slot that covers its WCET as shown by Figure 3.1a with a system of three tasks τ^i, τ^j and τ^k scheduled on two cores. In the following, we call this representation the single-phase representation. The interference analysis is conducted after the schedule of tasks is built (or after their priorities have been chosen) and bounds the maximum amount of interference that each task may suffer according to the tasks that may execute in parallel and their respective number of accesses. Using this model, the amount of interference is computed at the task granularity level: an access can be performed from the



(a) Before the interference analysis. (b) After the interference analysis.

Figure 3.1: Task system represented with the single-phase model.



(a) Before the interference analysis. (b) After the interference analysis.

Figure 3.2: Task system represented with the multi-phase model.

beginning to the end of its task because the model does not keep track of more precise information. The result of the interference analysis on the three tasks is given by Figure 3.1b: the 15 accesses of task τ^i and the 6 accesses of τ^j may all be interfered by the accesses of τ^k, but at most 21 accesses of τ^k (15+6 from the two other tasks) may suffer contentions. The red rectangles at the end of the tasks represent the worst-case timing penalty due to the potential interference suffered by each task.

The multi-phase model proposes to represent the tasks as a sequence of phases, called a *profile*, to improve the precision of the analysis. Indeed, it is possible to restrict the possible timing of an access to a subset of the phases composing the tasks so that it is not accounted for in all the task. It is worth noting that each phase represents a portion of the task execution but is not directly representing sequences of instructions. Therefore, the number of accesses for a phase is the worst-case number of accesses that may be performed within the timing interval defined by this phase, whatever the execution trace of the task. Figure 3.2a displays the same tasks as in Figure 3.1a but using the multi-phase model: the accesses are distributed in the different phases and in this example the sum for each task equals the total number of accesses accounted for with the single-phase model. Using this abstraction, the interference analysis can be performed at the phase level in a fine-grained manner as depicted in Figure 3.2b. The eight accesses of the first phase of τ^i are no longer interfered because no phase is executing in parallel, and some phases have a worst-case interference scenario with less contentions than the accesses that they perform (e.g. 2 possible contentions over 15 accesses for the first phase of τ^j). In this case, the multi-phase representation allows to reduce the total amount of contentions and the makespan of the system (42 contentions and 98 time units with the single-phase model in Figure 3.1b and 24 contentions and 92 time units with the multi-phase model in Figure 3.2b).

1.2 Computing the worst-case number of accesses in each phase

Existing methods such as TIPs [12] or StAMP [13] divide the tasks according to the program structure, based on the worst-case date of instructions performing accesses for the former and on SESE regions for the latter (see chapter 2 section 3.4 for more details). Another approach to the problem is to arbitrarily divide the task into phases and then to safely account for the accesses that can be performed in each phase. Regardless of the chosen approach, this chapter provides formal correctness criteria to ensure that the accesses are safely accounted for in the phases where they

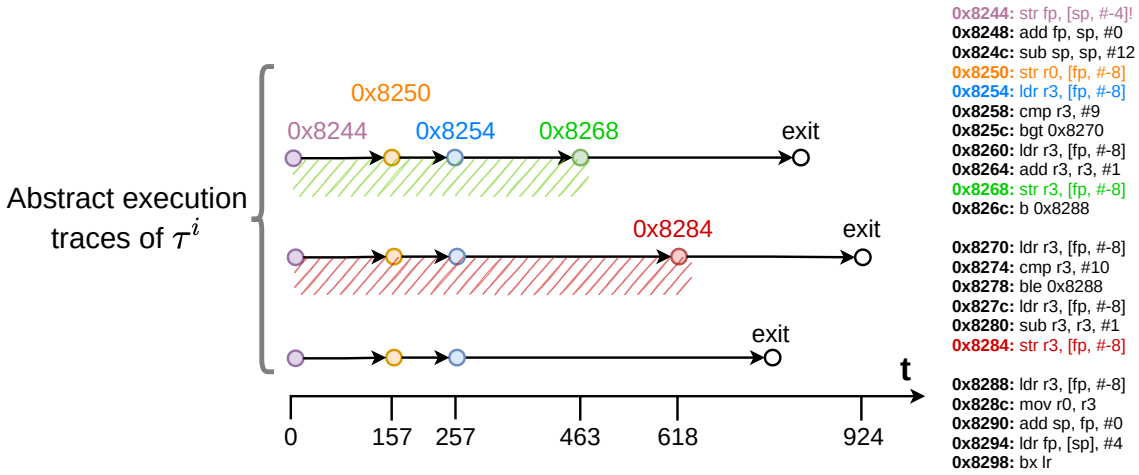


Figure 3.3: A code and its corresponding traces representation.

can occur and that the actual system implementation corresponds to the model and upholds its hypotheses.

The computation of the accesses in each phase relies on information gathered during a static analysis of the binary code of the task: firstly the list of instructions that may perform an access and secondly the time intervals covering the possible execution dates of these instructions. Such information can be obtained by retrieving all the execution traces of the task from its CFG. However, relying on the complete set of execution traces may be intractable in practice. As an alternative, the TIPs framework [12] proposes to enumerate the traces of a task from a light version of the CFG called TIPsGraph. A TIPsGraph only represents the sequences of instructions performing accesses and their worst-case date. In the following, we call *abstract execution traces* the paths enumerated from the TIPsGraph because one abstract trace may represent several execution traces of the task (due to the instructions that are not represented in the TIPsGraph). We use these abstract execution traces in the remaining of this chapter because they exhibit only the information we require.

As an example, we continue to work on the simple task of Listing 2.2, whose TIPsGraph was already depicted by Figure 2.5 in the last chapter. Figure 3.3 represents the abstract execution traces of the task derived from its TIPsGraph as proposed in the TIPs framework on the left hand side, and the code of the task on the right side. In our traces abstraction, each node represents the execution of an instruction that may perform a memory access in a particular trace. The colors link the nodes to the instruction they represent in the code using different colors: nodes 0x8244, 0x8250 and 0x8254 are executed in all the traces while 0x8268 and 0x8284 are executed in two distinct traces only. Moreover, the duration of an edge corresponds to the WCET of all the possible paths between its source and its destination nodes. Therefore, the nodes are placed at their worst-case execution date and without additional information from the model, we must assume that they can be executed at any moment before this date. As an example, the green and red rectangles in the figure cover the interval where respectively the green and the red node can be executed.

In Figure 3.4, 2 additional representations are introduced for the same example of task:

- 1 possible execution of the task for each trace, where nodes are depicted with crosses instead of circles. The correspondence between the circles and crosses are depicted with dashed lines. Notice that each cross is placed before its corresponding circle, as these denote the worst-case date of the access.

- a multi-phase representation composed of 4 phases that are annotated with the worst-case number of accesses that can be performed during their execution.

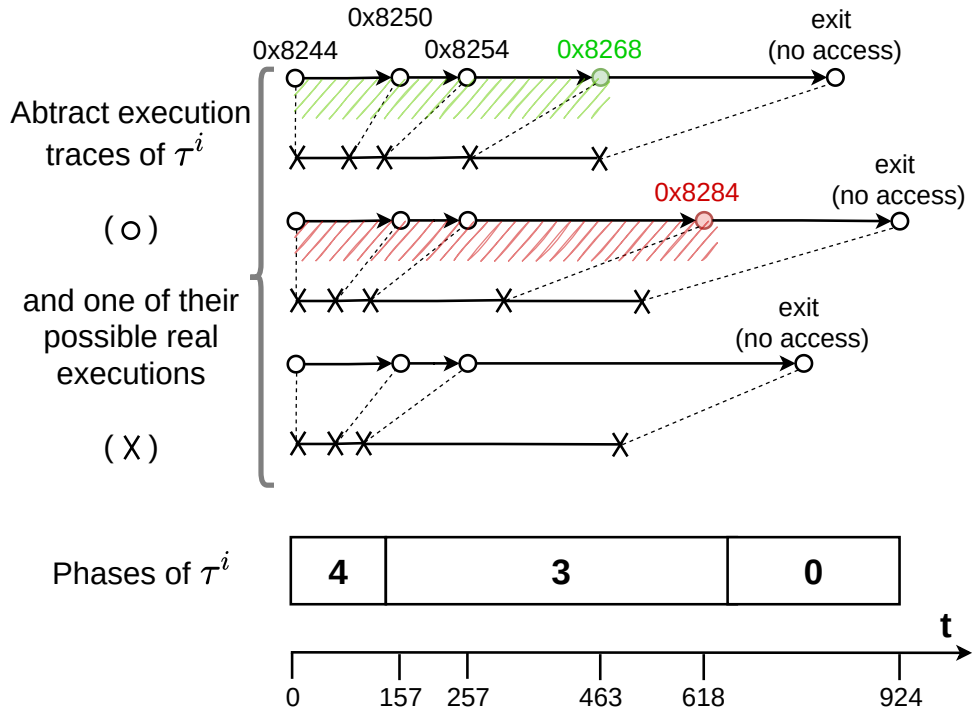


Figure 3.4: Traces of the TIPsGraph in Figure 2.5 and a possible multi-phase profile for the task

In order to compute the accesses in a phase, we first compute the number of accesses that each trace can perform within this phase. Then, as only one trace can be executed at a time, the number of accesses in the phase is the maximum of the accesses that can occur during the execution of a trace. Let's take the second phase as an example, we assume that each represented node performs at most one access except the exit node. The nodes cannot be executed after their worst-case date. Hence, the trace at the top can perform up to 3 accesses in this phase because as opposed to 0x8244, instructions 0x8250, 0x8254 and 0x8268 have a date higher than the start date of the phase. Similarly, the trace in the middle can perform at most 3 accesses and the bottom one at most 2 accesses in this phase. Therefore, we conservatively assume that 3 accesses can be performed in the second phase.

This abstraction causes a consequent over-approximation of the number of accesses. For instance, the last node of each trace must be accounted for in each phase. Hence, the total number of accesses using the multi-phase model is 6 while if we were using the single-phase model, at most 4 accesses would be performed by the task (by the middle or top traces). This undermines the performance of the multi-phase model. In order to reduce the execution span of the nodes, so that the number of phases where they are accounted for is limited, we must lower bound their execution date. A lower bound on the execution date could be provided by computing the best-case execution date of the instructions, but such information cannot be computed directly with the results of a WCET analysis so we adopt the solution proposed in [53] based on synchronizations. Basically, we decide to set minimum dates for some selected nodes in the traces.

This is depicted in Figure 3.5 where the synchronizations, represented by black nodes, fix the execution date of the synchronized instructions. Here, the nodes are synchronized on their worst-case execution date so the crosses are at the same date as their corresponding circle nodes. This allows to reduce the set of phases where the red and green instructions may be executed (along with their successors). This time, only 1 access can be performed in the first phase because thanks to the synchronization of instruction 0x8250, only 0x8244 may perform an access in this phase for all the traces. Therefore, the multi-phase task performs at most 4 accesses, which is the same amount as if we used the single-phase model. Note that it is guaranteed that no access can be performed in the

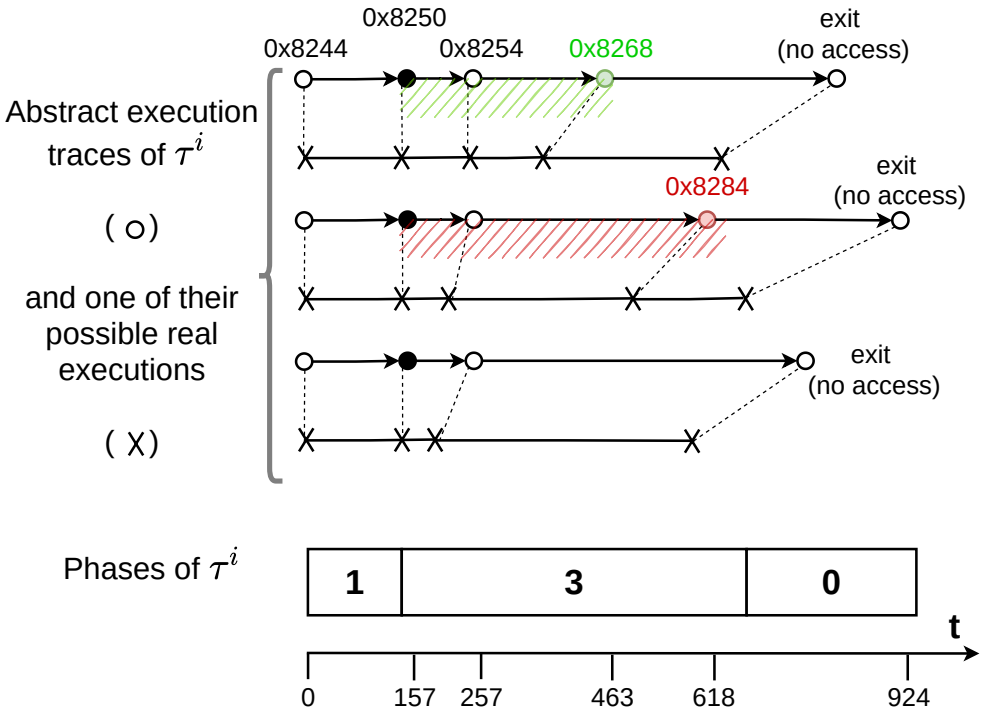


Figure 3.5: Traces of the TIPsGraph in Figure 2.5 and a possible multi-phase profile for the task, synchronizations are represented with black nodes.

last phase, so this phase cannot cause or suffer from contentions. Some experiments in Chapters 5 and 7 will demonstrate that even when the count of accesses is over-approximated, the multi-phase model can still lead to a tighter worst-case interference prediction than when using the single-phase model because the interference analysis is performed at the phase granularity, which offers more precision than the task granularity.

1.3 Summary

To summarize, the multi-phase model is able to reduce the over-estimation on the number of contentions for a task system by lowering the granularity at which the memory accesses are represented. However, accounting for the number of accesses in each phase requires to lower bound the execution date of certain nodes and limit the number of phases where they must be accounted for, using either their BCET or synchronizations. An efficient selection of these synchronizations reduces and may eliminate the access over-approximation in the task compared to if the task was represented using the single-phase model.

2 Formal model

The previous section introduced the core concepts of the multi-phase model and gave a first insight of the challenges it poses in order to provide a correct, safe and yet more precise interference analysis. In the following, we formalize the multi-phase model in order to provide criteria that guarantee the correctness of its implementation. In other words, the correctness criteria that are proposed throughout this part ensure that the program will always execute in conditions that are covered by our analysis hypotheses.

Notation	Definition
τ^i	task i
ϕ^i_k	phase k in the representation of τ^i
$\phi^i_k.d$	start date of ϕ^i_k without interference
$\phi^i_k.dur$	duration of ϕ^i_k without interference
$\phi^i_k.m$	maximum number of memory accesses performed within ϕ^i_k
t^i_j	abstract execution trace j of task τ^i
$\eta^i_{j,k}$	node k in trace t^i_j
$\eta^i_{j,k}.it$	instruction represented by $\eta^i_{j,k}$
$\eta^i_{j,k}.d$	worst-case execution date of $\eta^i_{j,k}$ without interference
$\eta^i_{j,k}.m$	maximum number of memory accesses performed by $\eta^i_{j,k}$
$\eta^i_{j,k}.sync$	True if the node is synchronized, i.e. cannot be executed before $\eta^i_{j,k}.d$
$wced(ins)$	the worst-case execution of the instruction ins across all traces
$s_{last}(\eta^i_{j,k})$	last synchronized node before $\eta^i_{j,k}$ in trace t^i_j
$t^i_j\|_{\phi^i_k}$	restriction of trace t^i_j to ϕ^i_k, i.e. the set of nodes in t^i_j that may execute during ϕ^i_k

2.1 Architecture model

We consider a multi-core architecture $\mathbb{C} = \{C_k | 0 \leq k < N_c\}$ composed of N_c homogeneous cores. The cores access a shared memory using a shared bus that can serve one request at a time with a FIFO arbitration policy. Additionally, we assume that an ongoing access to the bus cannot be preempted. Therefore, the work focuses on the bus as the only source of interference.

2.2 Tasks model

We consider a system T of real-time tasks τ^i ($i \geq 0$). Each task has two distinct representations:

- the *multi-phase* representation, or *profile*: the task is described as a sequence of phases, whose duration is at least equal to the WCET of the task. Moreover, each phase has a maximum number of memory accesses.
- the *traces* representation: the execution traces are the possible sequences of instructions executed by the task.

The multi-phase representation is convenient to schedule the tasks and perform the interference analysis while the traces are used as an intermediate representation between the code and the multi-phase representation. It is used to compute the worst-case number of accesses in the phases and select synchronizations in order to implement the model in the code of the task.

The multi-phase representation of τ^i is denoted $\mathbb{P}^i = \{\phi^i_k \ | 0 \leq k < \Phi^i\}$ with Φ^i the number of phases. Each ϕ^i_l is defined by:

- $\phi^i_l.d$: its start date.
- $\phi^i_l.dur$: its worst-case duration in isolation (without interference).
- $\phi^i_l.m$: the worst-case number of memory accesses that may be performed within $[\phi^i_l.d, \phi^i_l.d + \phi^i_l.dur[$.

The date of ϕ^i_0, which is also the start date of task τ^i without interference, is set when the static schedule of the system is built or the priorities of the tasks are set. Then, for each ϕ^i_l ($l > 0$) the start date is defined by:

$$\phi^i_l.d = \phi^i_0.d + \sum_{0 \leq q < l} \phi^i_q.dur \tag{3.1}$$

Alternatively, we can define recursively the start date of each phase (except the first one) by:

$$\forall l > 0, \phi_l^i.d = \phi_{l-1}^i.d + \phi_{l-1}^i.dur$$

In order to compute the worst-case number of memory accesses performed during a given phase (i.e. $\phi_k^i.m$), the code portions of τ^i that may be executed during ϕ_k^i must be identified and analyzed. To do so, we introduce $\mathbb{T}^i = \{t_j^i | 0 \leq j < T^i\}$ the set of *abstract execution traces* of τ^i. Each trace corresponds to a possible execution of τ^i (corresponding to a particular set of inputs) and is a sequence of nodes $\eta_{j,k}^i$ representing instructions with $0 \leq k < N_j^i$ the node's index in its sequence. $\eta_{j,0}^i$ is the *entry point* of task τ^i and each node is defined by:

- $\eta_{j,k}^i.it$: the instruction represented by $\eta_{j,k}^i$.
- $\eta_{j,k}^i.m \in \mathbb{N}$: the worst-case number of memory accesses performed by one execution of $\eta_{j,k}^i.it$.
- $\eta_{j,k}^i.d$: the worst-case execution date of $\eta_{j,k}^i.it$ in trace t_j^i.

An instruction is not just understood as an element of the core Instruction Set Architecture (ISA) (e.g. the ADD instruction), but as a particular instruction in the binary code of the task. Thus, nodes from different traces may reference the same instruction *instr* in the code. We say that such nodes are equivalent: $\eta_{j,k}^i \sim \eta_{j',k'}^i \iff \eta_{j,k}^i.it = \eta_{j',k'}^i.it = instr$. Moreover, we define the equivalence class of any instruction *instr* as $\mathbb{N}_{equiv}^i(instr) = \bigcup_{t_j^i \in \mathbb{T}^i} \{\eta_{j,k}^i \in t_j^i | \eta_{j,k}^i.it == instr\}$ and $wced(instr)$ as the worst-case execution date of the instruction *instr* across all the traces so $wced(instr) = max_{\eta_{j,k}^i \in \mathbb{N}_{equiv}^i(instr)}(\eta_{j,k}^i.d)$.

2.3 Synchronizations

As it stands, the model guarantees that a node $\eta_{j,k}^i$ can execute in the interval $[\eta_{j,0}^i.d, \eta_{j,k}^i.d]$, i.e. from the start date of the task to its worst-case execution date, because no minimum execution date is specified. Therefore, its accesses ($\eta_{j,k}^i.m$) must be accounted for in all the phases that start before the worst-case date of the node so all the phases ϕ_l^i such that $\phi_l^i.d \leq \eta_{j,k}^i.d$. Although it ensures a safe account of accesses in the multi-phase model, it also induces huge access over-approximations. Therefore, we propose to *synchronize* some nodes to limit this over-approximation: a code inserted in the program before the instruction represented by the node ensures that the instruction cannot be executed before its worst-case date. The synchronization code can be added by the programmer directly in the source code of the tasks, by the compiler as part of a low-level compilation pass, or during an automatic code re-engineering process to adapt legacy code to the multi-phase model. Because the synchronization of a node $\eta_{j,k}^i.d$ actually consists in applying the synchronization to the instruction it represents, we must consider all the nodes of the equivalence class of this instruction to implement the synchronization. Moreover, since the model uses worst-case dates, the date chosen for all nodes in an equivalence class must be the maximum date amongst them.

To keep track of the synchronized nodes, we add the boolean attribute $\eta_{j,k}^i.sync$ which is true if the node is synchronized and false otherwise.

Using these synchronizations, the accesses performed by any node must only be accounted for in the phases that:

1. finish after the last synchronization prior to the node, **AND**
2. start before the worst-case date of the node.

This is illustrated in Figure 3.6, which depicts 3 execution traces (t_0^i, t_1^i and t_2^i) and 2 possible profiles for a task τ^i. Synchronized nodes are depicted in black in the traces. The red (resp. green) rectangle shows the time window in which the accesses of node $\eta_{2,3}^i$ (resp. $\eta_{1,7}^i$) must be accounted

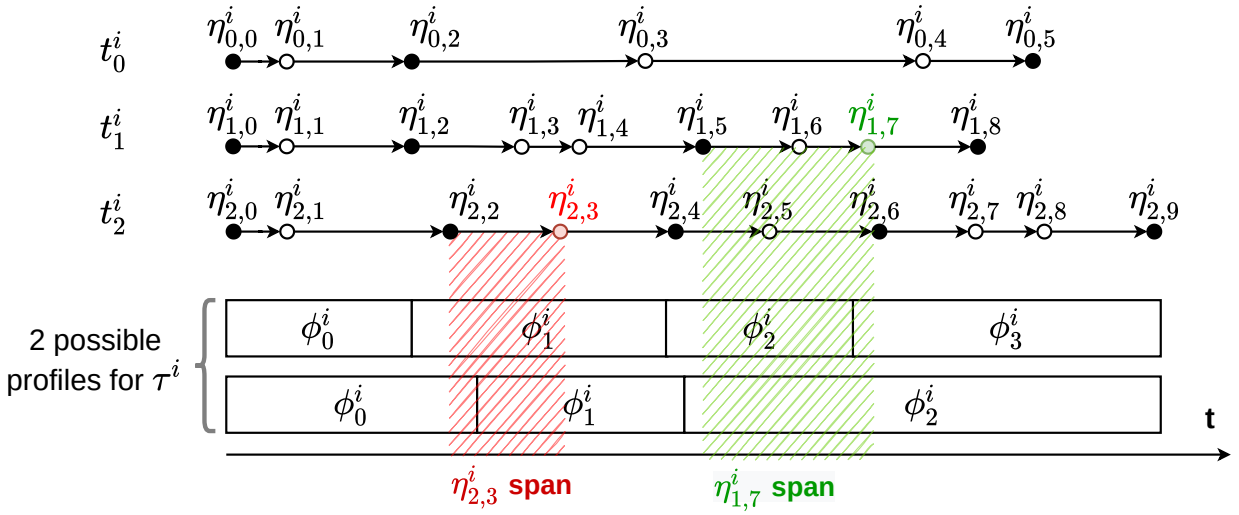


Figure 3.6: Three traces and two profiles for task τ^i

for. In the first profile, the accesses of $\eta_{1,7}^i$ must be considered in phases ϕ_2^i and ϕ_3^i, whereas in the second profile, they would only be considered in ϕ_2^i.

It is important to note that since $\eta_{j,k}^i.d$ is a worst-case date, if node $\eta_{j,k}^i$ is synchronized, then its execution date is exactly[1] $\eta_{j,k}^i.d$. We denote $s_{last}(\eta_{j,k}^i)$ the last synchronized node before $\eta_{j,k}^i$ in trace t_j^i. By convention, we set $s_{last}(\eta_{j,k}^i) = \eta_{j,k}^i$ when $\eta_{j,k}^i.sync$.

To account for the tasks schedule, for all tasks τ^i, the entry node (on any trace t_j^i) is synchronized and its worst-case execution date is set to the start of the first phase of the profile:

Property 1. $\forall i, j : \eta_{j,0}^i.sync \wedge (\eta_{j,0}^i.d = \phi_0^i.d)$

The worst-case date of any other node $\eta_{j,k}^i$ with $k > 0$ is defined according to the date of the last synchronized node on its trace:

Property 2. $\eta_{j,k}^i.d = \eta_{j,s}^i.d + \sum_{s \leq t < k} wcet(\eta_{j,t}^i.it, \eta_{j,t+1}^i.it)$

where $wcet(\eta_{j,t}^i.it, \eta_{j,t+1}^i.it)$ is the WCET between instructions $\eta_{j,t}^i.it$ and $\eta_{j,t+1}^i.it$, and $\eta_{j,s}^i$ is $s_{last}(\eta_{j,k}^i)$ if $\neg\eta_{j,k}^i.sync$ and $s_{last}(\eta_{j,k-1}^i)$ otherwise.

A node $\eta_{j,k}^i$ can only be executed in the interval $[s_{last}(\eta_{j,k}^i).d, \eta_{j,k}^i.d]$. As we saw in the example of Figure 3.6, this interval may overlap with several phases of the task profile.

We denote $t_j^i|_{\phi_l^i}$ the set of nodes in trace t_j^i that may be executed within $[\phi_l^i.d, \phi_l^i.d + \phi_l^i.dur[$, called the *restriction* of trace t_j^i to phase ϕ_l^i:

$$t_j^i|_{\phi_l^i} = \{\eta_{j,k}^i | (\eta_{j,k}^i.d \geq \phi_l^i.d) \wedge (s_{last}(\eta_{j,k}^i).d < \phi_l^i.d + \phi_l^i.dur)\}$$

The notion of restriction of a trace to a phase is illustrated in Figure 3.7 on 3 traces over phase ϕ_1^i. Lets focus on t_0^i and identify the nodes that can be executed in ϕ_1^i, which compose the restriction of t_0^i to ϕ_1^i. Firstly, node $\eta_{0,2}^i$ is synchronized at a date covered by ϕ_1^i, so this node will execute only in ϕ_1^i. Moreover, $\eta_{0,3}^i$ and $\eta_{0,4}^i$ are not synchronized but may execute from after $\eta_{0,2}^i$ to their worst-case date. Hence, they are accounted for in ϕ_1^i. Finally, $\eta_{0,5}^i$ is synchronized after the end of ϕ_1^i so it cannot be executed in this phase. Therefore, the restriction of t_0^i to ϕ_1^i is $t_j^i|_{\phi_1^i} = \{\eta_{0,2}^i, \eta_{0,3}^i, \eta_{0,4}^i\}$.

[1] With a precision of a few cycles depending on the implementation of the synchronization mechanism.

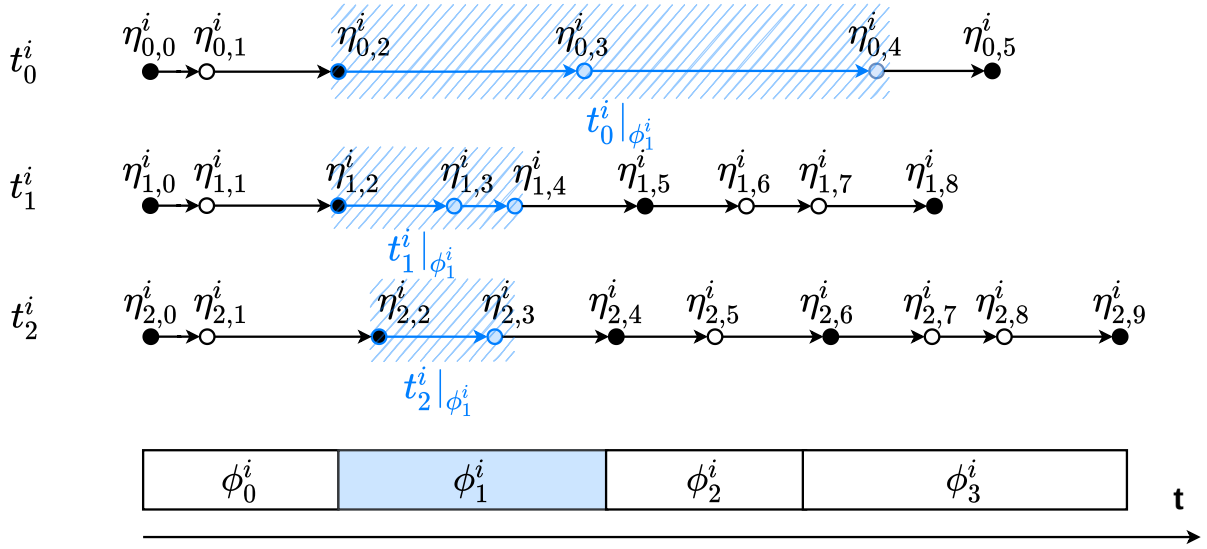


Figure 3.7: Restrictions of traces t_0^i, t_1^i and t_2^i to phase ϕ_1^i.

2.4 Maximum number of accesses in a phase

The number of accesses that may be performed during a phase for an individual trace is equal to the sum of the accesses of the nodes from this trace that may be executed in the phase. During the execution of a task, only one trace executes (which one depends on the execution context): as a consequence, the worst-case number of accesses performed during a phase is equal to the maximum number of accesses that may be performed by any execution trace during that phase.

Property 3. *The worst-case number of accesses that may be performed during phase ϕ_l^i, denoted $\phi_l^i.m$, is equal to the maximum of accesses per trace during phase ϕ_l^i:*

$$\phi_l^i.m = \max_{0 \leq j < T^i} \Big(\sum_{\eta_{j,k}^i \in t_j^i|_{\phi_l^i}} \eta_{j,k}^i.m \Big)$$

Correctness criterion 1. *The formula of Property 3 provides a conservative estimation of the number of memory accesses that can occur during the phases of a multi-phase profile.*

Since nodes may span over multiple phases, the number of accesses counted task-wise may be overestimated, even when some nodes are synchronized. However, nodes from a trace which span over multiple phases may be "covered" by other nodes from another trace performing more accesses on a given phase. For example, in Figure 3.7, if we consider that each node performs 1 access, trace t_2^i is the local worst trace on ϕ_3^i with 4 nodes performing accesses and trace t_1^i is the local worst trace on ϕ_2^i with 3 nodes performing accesses. On phase ϕ_1^i, traces t_0^i and t_1^i both have 3 nodes performing accesses. In such circumstances, although node $\eta_{0,4}^i$ spans over ϕ_3^i, ϕ_2^i and ϕ_1^i, it does not contribute to any over-approximation.

We quantify the task-wise over-approximation of memory accesses compared to the 1-phase model, by computing the difference between the sum of accesses accounted for in each phase, and the worst trace-wise number of accesses.

Property 4. *The memory access over-approximation in a multi-phase profile of a task τ^i compared to its 1-phase representation is equal to:*

$$M = \Big(\sum_{0 \leq l < \Phi^i} \phi_l^i.m \Big) - \max_{0 \leq j < T^i} \Big(\sum_{0 \leq k < N_j^i} \eta_{j,k}^i.m \Big)$$

The access over-approximation rate of a profile is defined by:

$$\Delta = (M/ \max_{0 \leq j < T^i} (\sum_{0 \leq k < N^i_j} \eta^i_{j,k}.m))$$

3 Interference analysis

In this section, we consider a task system for which an analysis has provided a multi-phase model as well as a selection of synchronized nodes for each task. We assume that this task system is scheduled statically (the $\phi^i_0.d$ for each τ^i are selected and the start dates of the other phases are computed using equation 3.1), and that an interference analysis is applied to compute and account for the effect of potential interference between the tasks phases, assuming the timing-compositionality of the target processor [78].

3.1 Consequences of the interference analysis

In practice, during the interference analysis, each phase that potentially suffers from interference is extended using a time penalty, and the next phases are postponed accordingly. This extension may violate assumptions that were made on the correspondence between phases and traces: in particular the restrictions of traces to phases that were computed prior to the interference analysis may no longer be correct, resulting in the possibility that some contentions between cores may happen in phases in which they were not accounted for.

A simple example is given in Figure 3.8 which represents in (a) the trace and a multi-phase profile of τ^i in isolation (we suppose that this task has only one trace in the example). Node $\eta^i_{0,4}$ is synchronized so ϕ^i_0 may perform up to 4 accesses and ϕ^i_1 5 accesses. The task is scheduled on core 0 (C0) in parallel with task τ^j scheduled in core 1 as shown in (b) and an interference analysis is performed to bound the possible delays due to contentions for each phase: ϕ^i_0 may suffer 4 contentions and ϕ^i_1 5 contentions in the worst-case. The maximum contention delays corresponding to the results of the interference analysis are represented with red hatched rectangles. Then, (c) represents the trace (nodes as circles) and a possible execution of this trace (nodes as crosses). In this possible execution, it happens that in the end only $\eta^i_{0,2}$ is interfered by τ^j but $\eta^i_{0,3}$ still occurs before its worst-case date. We can observe that although at most 4 accesses were supposed to occur in ϕ^i_0 according to the interference analysis in (b), three additional accesses of $\eta^i_{0,4}$, $\eta^i_{0,5}$ and $\eta^i_{0,6}$ are actually performed in this phase. These 3 additional accesses were supposed to occur only in ϕ^i_1 so we did not account for them when computing the maximum penalty of ϕ^j_0. Therefore, the result of the interference analysis is unsafe because ϕ^j_0 can actually suffer from (and inflict to other tasks) 3 additional contentions in the worst case.

One solution is to recompute the number of accesses in the phases and the potential contentions until a fix point is reached. However, this solution is computationally expensive and degrades the over-approximation of accesses if the synchronizations are not modified. In the example, ϕ^i_0 now needs to account for the 9 accesses of the trace but 4 of these accesses are already accounted for in ϕ^i_1 so we must consider that they can interfere with the accesses of both ϕ^j_0 and ϕ^j_1. Another solution that we propose is to change the synchronization date of $\eta^i_{0,4}$ so that it remains in ϕ^i_1 and the accesses accounted for in each phase stays as in (a) and (b). The disadvantage of this solution

Table 3.1: Notations introduced in the section

Notation	Definition
$\phi^i_l.p$	timing penalty added to ϕ^i_l due to potential interference
$\phi^i_l.d^\#$	*post-analysis* date of ϕ^i_l, i.e. date in the presence of interference
$\eta^i_{j,k}.d^\#$	*post-analysis* date of $\eta^i_{j,k}$, i.e. worst-case execution date in the presence of interference

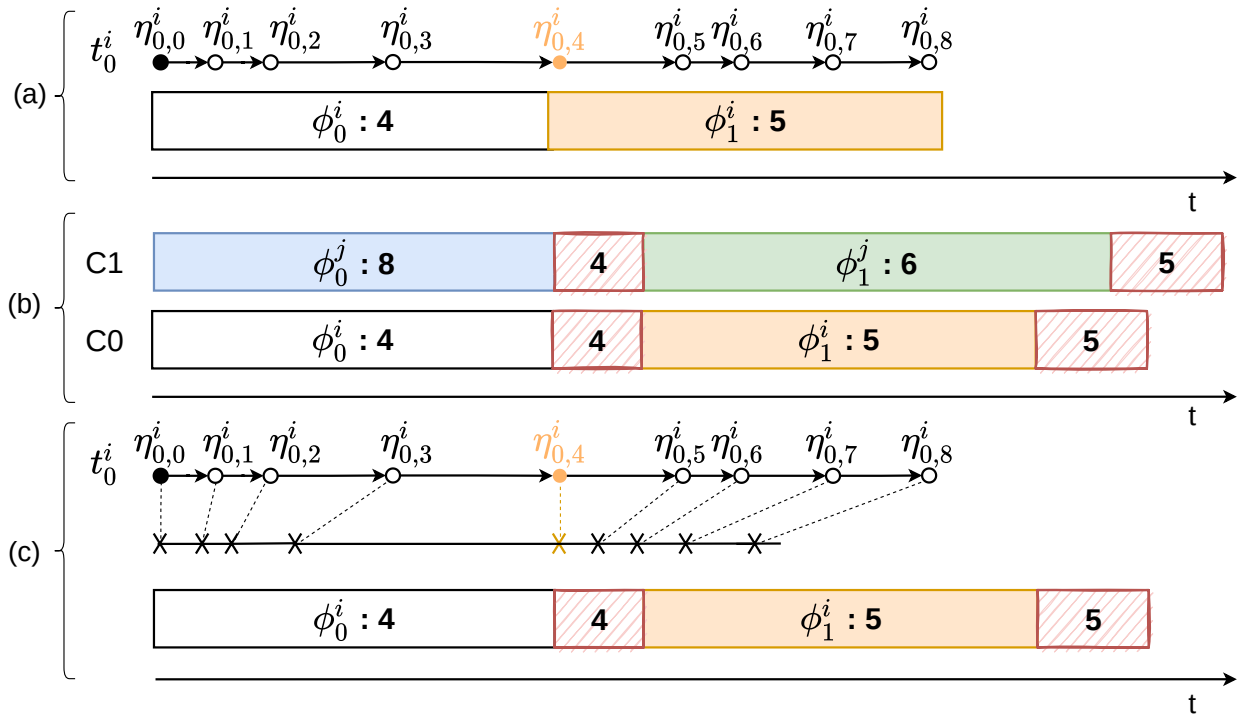


Figure 3.8: Consequences of the interference analysis on the model of task τ^i.

is that although it preserves the over-approximation of accesses, by delaying the synchronization of $\eta_{0,4}^i$ we also degrade the average execution time of the task. However, promoting the worst-case scenario over the average-case performance can be acceptable for HRT systems.

The solution is explained with another example depicted in Figure 3.9 that displays trace t_2^i and the profile from Figure 3.7, at three stages of the analysis:

- (a) depicts the trace and phases before the interference analysis. We have:
  $t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}$; $t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i\}$; $t_2^i|_{\phi_2^i} = \{\eta_{2,4}^i, \eta_{2,5}^i\}$; $t_2^i|_{\phi_3^i} = \{\eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i, \eta_{2,9}^i\}$
  Additionally, we consider that for this task, $\phi_1^i.m = 2$ and $\phi_2^i.m = 2$.

- (b) shows the same trace and profile after the interference analysis (assuming other tasks in the system): the effect of interference is materialized by timing penalties on the phases (the red rectangles after each phase). $t_2^i|_{\phi_1^i}$, $t_2^i|_{\phi_2^i}$ and $t_2^i|_{\phi_3^i}$ are different than in (a):
  $t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}$; $t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i, \eta_{2,4}^i, \eta_{2,5}^i\}$; $t_2^i|_{\phi_2^i} = \{\eta_{2,5}^i, \eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i, \eta_{2,9}^i\}$;
  $t_2^i|_{\phi_3^i} = \{\eta_{2,8}^i, \eta_{2,9}^i\}$
  As a consequence, the worst-case amount of accesses that can happen during phases ϕ_1^i and ϕ_2^i is higher than what was assumed and therefore their interference penalty and those of the tasks scheduled in parallel are no longer conservative.

- (c) represents a solution to respect the model's assumptions of (a): the synchronized date of $\eta_{2,4}^i$ (resp. $\eta_{2,6}^i$) is set to the new starting date of ϕ_2^i (resp. ϕ_3^i), which is the unique phase in which it was accounted for in (a). With this slight modification, the restrictions of t_2^i to each phase are identical to the ones in (a) and the $\phi_l^i.m$ that was computed in isolation for each phase remains correct.

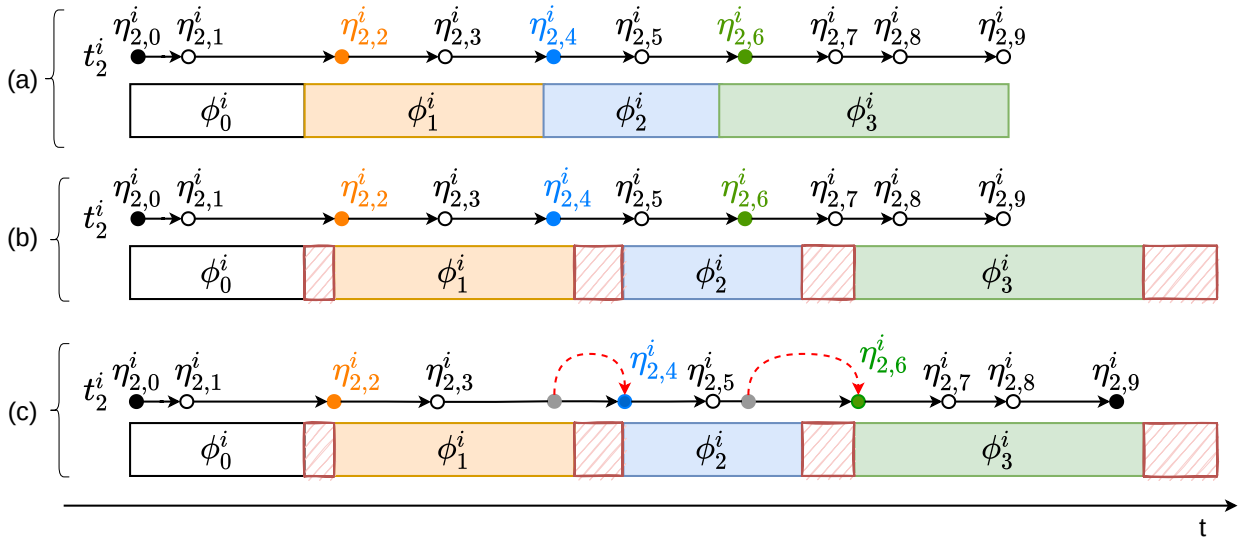


Figure 3.9: A trace and its corresponding phases representation : (a) in isolation, (b) after the interference analysis, red rectangles are the timing penalty added for each phase, (c) after a correction on nodes dates.

3.2 Enforcing the model's assumptions and the analysis results

Since the duration and start dates of phases can be changed as a result of the interference analysis, new attributes are added to the formal model of the phases:

- $\phi_l^i.p \geq 0$ is the timing penalty added to ϕ_l^i due to potential interference. It is a conservative bound computed during the interference analysis.
- $\phi_l^i.d^{\#}$ is the *post-analysis* date of ϕ_l^i, i.e. its start date taking into account the potential interference in the system.

After the interference analysis, the start date of some tasks may be postponed due to interference that delays previous tasks. $\phi_0^i.d^{\#}$ is thus fixed by applying the interference analysis results to the initial schedule. The start dates of all other phases ϕ_l^i describing the execution of τ^i are computed as:

$$\phi_l^i.d^{\#} = \phi_0^i.d^{\#} + \sum_{0 \leq q < l} (\phi_q^i.dur + \phi_q^i.p) \tag{3.2}$$

Correctness criterion 2. *The synchronization dates in the final implementation of tasks must at least be equal to the start date of the corresponding phase: for each synchronization node* $\eta_{j,k}^i \in t_j^i|_{\phi_n^i}$, *the synchronization date is set to at least* $\phi_n^i.d^{\#}$. *This way it is guaranteed that nodes after* $\eta_{j,k}^i$ *cannot execute and thus produce accesses before the start of* ϕ_n^i.

It seems straightforward that, by construction, a task set implemented using this rule is guaranteed to fulfill the assumptions made during the interference analysis. Indeed, during the execution of the system, memory accesses will only occur at times that were accounted for during the analysis, and thus the amount of interference cannot be larger in practice than what was accounted for. However, although this implementation rule directly guarantees that accesses are not performed before the phases in which they are accounted for, it may be harder to convince oneself that they cannot occur later than the end of these phases. Consequently, and given the potentially critical nature of the tasks modelled in the multi-phase representation, we provide in the remainder of the section a formal proof of the correctness of this implementation scheme w.r.t. the result of the interference analysis. Once again, this is completely agnostic of the analysis method, as long as it correctly provides a conservative bound on the interference level.

We denote $\eta^i_{j,k}.d^\#$ the *post-analysis* worst-case date of node $\eta^i_{j,k}$. The post-analysis dates of nodes are upper bounds on the worst-case execution dates of nodes in the presence of interference. We start by characterizing those bounds in our formal model (Properties 5, 6 and 7), and then use them to prove the correctness of the implementation of a multi-phase model of tasks.

First, the post-analysis execution date of the entry point of each task τ^i is the post analysis start date of its first phase ϕ^i_0.

Property 5. *For any task τ^i: $\forall j < T^i, \eta^i_{j,0}.d^\# = \phi^i_0.d^\#$*

Second, correctness criterion 2 has the following consequences for the post-analysis execution date of any synchronized node $\eta^i_{j,k}$ (except the entry point) of any task τ^i:

- If the phase ϕ^i_n in which the node was supposed to be executed is postponed due to interference penalties on previous phases, the node cannot be executed before the post-analysis start date of ϕ^i_n.
- If previous synchronized nodes see their execution dates postponed, the synchronization date of $\eta^i_{j,k}$ must be postponed accordingly, and thus computed from the post-analysis date of the previous synchronized node $\eta^i_{j,s}$. In this case, we must consider the interference that can take place between $\eta^i_{j,s}$ and $\eta^i_{j,k}$. If there exists one or more phases that span entirely between the two nodes, their penalties are added to the post-analysis date of $\eta^i_{j,k}$ (which is conservative). Moreover, by convention we count in the post-analysis date of $\eta^i_{j,k}$ the entire amount of penalty of the phase to which it belongs (which is also conservative since it accounts for the interference that can occur on each access in the phase prior to the synchronization node, and on each access that may occur until the next synchronization node).

Property 6. *For any synchronized node $\eta^i_{j,k}$ of any trace t^i_j of task τ^i:*
$(k > 0 \wedge \eta^i_{j,k}.sync \wedge (\eta^i_{j,k} \in t^i_j{}_{|_{\phi^i_n}}) \wedge (\eta^i_{j,s} = s_{last}(\eta^i_{j,k-1})) \wedge (\eta^i_{j,s} \in t^i_j{}_{|_{\phi^i_m}}))$
$\Rightarrow \eta^i_{j,k}.d^\# = max(\phi^i_n.d^\#, \eta^i_{j,s}.d^\# + \sum_{s \le l < k} wcet(\eta^i_{j,l}.it, \eta^i_{j,l+1}.it) + \sum_{m < b \le n} \phi^i_b.p)$

Correctness criterion 3. *The synchronization dates in the final implementation of tasks must not be set to a value higher than the date computed in Property 6.*

Finally, for any non-synchronized node, its post-analysis date accounts for the possible postponing of the previous synchronized node $\eta^i_{j,s}$. Note that the potential interference occurring between them has been accounted for entirely in the post-analysis date of the previous synchronized node.

Property 7. *For any non-synchronized node $\eta^i_{j,k}$ of any trace t^i_j of task τ^i:*
$(\neg\eta^i_{j,k}.sync \wedge (\eta^i_{j,s} = s_{last}(\eta^i_{j,k}))) \Rightarrow \eta^i_{j,k}.d^\# = \eta^i_{j,s}.d^\# + \sum_{s \le l < k} wcet(\eta^i_{j,l}.it, \eta^i_{j,l+1}.it)$

4 Proof of correctness

We now prove that any task system which respects the 3 correctness criteria is correct w.r.t. the results of the interference analysis i.e. cannot generate interference that was not accounted for.

First, the difference between the start date of a synchronized node $\eta^i_{j,k}$ before and after the interference analysis is bounded by the difference between the start date of the phase ϕ^i_l in which it is executed, before and after the interference analysis, added to the maximum amount of interference that can occur in ϕ^i_l.

Lemma 1. $\forall \eta^i_{j,k}$: $(\eta^i_{j,k}.sync \wedge (\eta^i_{j,k} \in t^i_j{}_{|_{\phi^i_l}})) \Rightarrow \eta^i_{j,k}.d^\# - \eta^i_{j,k}.d \le \phi^i_l.d^\# - \phi^i_l.d + \phi^i_l.p$

Proof. We will prove by induction that the property is true for all synchronized nodes. If $\eta^i_{j,k}$ is the entry node of τ^i, the proof is direct using Properties 1 and 5. Otherwise, using Property 6, $\eta^i_{j,k}.d^\#$ is either equal to $\phi^i_l.d^\#$ or must be computed from the previous synchronized node on trace t^i_j. Let $\eta^i_{j,s} = s_{last}(\eta^i_{j,k-1})$, and assume that the property is true for $\eta^i_{j,s}$. Then,

- If $\eta^i_{j,k}.d^\# = \phi^i_l.d^\#$:
  since $\eta^i_{j,k} \in t^i_j|_{\phi^i_l}$, by definition $\eta^i_{j,k}.d \geq \phi^i_l.d$, and thus $\eta^i_{j,k}.d^\# - \eta^i_{j,k}.d \leq \phi^i_l.d^\# - \phi^i_j.d$.

- Otherwise:
  $\eta^i_{j,k}.d^\# = \eta^i_{j,s}.d^\# + \sum_{s \leq a < k} wcet(\eta^i_{j,a}.it, \eta^i_{j,a+1}.it) + \sum_{m < b \leq l} \phi^i_b.p$. Using Property 2, we get: $\eta^i_{j,k}.d^\# - \eta^i_{j,k}.d = \eta^i_{j,s}.d^\# - \eta^i_{j,s}.d + \sum_{m < b \leq l} \phi^i_b.p$. The induction hypothesis gives us: $\eta^i_{j,s}.d^\# - \eta^i_{j,s}.d \leq \phi^i_m.d^\# - \phi^i_m.d + \phi^i_m.p$, where ϕ^i_m is the phase in which $\eta^i_{j,s}$ executes. If $m = l$ (i.e. both nodes execute in the same phase) the property is directly proven for node $\eta^i_{j,k}$. Otherwise, $m < l$ and then $\phi^i_l.d^\# - \phi^i_l.d = \phi^i_m.d^\# - \phi^i_m.d + \sum_{m \leq b < l} \phi^i_b.p$ (using Equations 3.1 and 3.2), and thus the property is also proven.

By induction, we just proved that the property holds for all synchronized nodes. □

We are now ready to prove the correctness property:

Theorem 2. *For any task system that respects correctness criteria 1, 2 and 3, for any $\eta^i_{j,k}$ of any task τ^i, if $\eta^i_{j,k}$ spans over a phase ϕ^i_l after the interference analysis, then $\eta^i_{j,k}$ was necessarily accounted in the restriction of trace t^i_j to ϕ^i_l before the analysis:*
$\forall 0 \leq j < T^i, \forall 0 \leq k < N^i_j, \forall 0 \leq l < \Phi^i$:
$[s_{last}(\eta^i_{j,k}).d^\#, \eta^i_{j,k}.d^\#] \cap [\phi^i_l.d^\#, \phi^i_l.d^\# + \phi^i_l.dur + \phi^i_l.p[\neq \emptyset \Rightarrow \eta^i_{j,k} \in t^i_j|_{\phi^i_l}$

Proof. The case where $\eta^i_{j,k}$ is the entry node is direct. For all other nodes we consider separately the case of synchronized nodes and of non-synchronized nodes.
Case 1: $\eta^i_{j,k}.sync$ is true:
By convention, $s_{last}(\eta^i_{j,k}) = \eta^i_{j,k}$. Let us assume ϕ^i_l such that $\eta^i_{j,k}.d^\# \in [\phi^i_l.d^\#, \phi^i_l.d^\# + \phi^i_l.dur + \phi^i_l.p[$. Let us denote ϕ^i_z the phase such that $\eta^i_{j,k} \in t^i_j|_{\phi^i_z}$ (z is unique because $\eta^i_{j,k}$ is synchronized). We want to prove that $l = z$. Using Property 6, either $\eta^i_{j,k}.d^\# = \phi^i_z.d^\#$ or it is greater. If it is equal, then directly $\phi^i_l = \phi^i_z$ because phases of the same task do not overlap. Otherwise, if $z > l$ then $\eta^i_{j,k}.d^\# > \phi^i_z.d^\# \geq \phi^i_l.d^\# + \phi^i_l.dur + \phi^i_l.p$ which contradicts the assumption. So z would have to be less than l. Now, since $\eta^i_{j,k} \in t^i_j|_{\phi^i_z}$, $\eta^i_{j,k}.d - \phi^i_z.d < \phi^i_z.dur$. At the same time, $\eta^i_{j,k}.d^\# \geq \phi^i_l.d^\# \geq \phi^i_z.d^\# + \phi^i_z.dur + \phi^i_z.p$, so $\eta^i_{j,k}.d^\# - \phi^i_z.d^\# \geq \phi^i_z.dur + \phi^i_z.p$. This contradicts Lemma 1, from which we conclude that $l = z$. This concludes the proof for case 1.
Case 2: $\eta^i_{j,k}.sync$ is false:
Let ϕ^i_l such that $[s_{last}(\eta^i_{j,k}).d^\#, \eta^i_{j,k}.d^\#] \cap [\phi^i_l.d^\#, \phi^i_l.d^\# + \phi^i_l.dur + \phi^i_l.p[\neq \emptyset$. Let us denote ϕ^i_m the phase to which $s_{last}(\eta^i_{j,k}).d^\#$ belongs, and assume by absurd that $\eta^i_{j,k} \notin t^i_j|_{\phi^i_l}$. Then by definition either $(s_{last}(\eta^i_{j,k}).d > \phi^i_l.d + \phi^i_l.dur)$ or $(\eta^i_{j,k}.d < \phi^i_l.d)$.
If $s_{last}(\eta^i_{j,k}).d > \phi^i_l.d + \phi^i_l.dur$: then $m > l$, and thus using Property 6: $s_{last}(\eta^i_{j,k}).d^\# \geq \phi^i_m.d^\# \geq \phi^i_l.d^\# + \phi^i_l.dur + \phi^i_l.p$, which contradicts the original assumption.
If $\eta^i_{j,k}.d < \phi^i_l.d$, then using Property 2: $s_{last}(\eta^i_{j,k}).d + \sum_{s \leq t < k} wcet(\eta^i_{j,t}.it, \eta^i_{j,t+1}.it) < \phi^i_l.d$. Then, we can deduce:

$$
\begin{aligned}
s_{last}(\eta^i_{j,k}).d^{\#} + \sum_{s \le t < k} wcet(\eta^i_{j,t}.it, \eta^i_{j,t+1}.it) &< \phi^i_l.d + s_{last}(\eta^i_{j,k}).d^{\#} - s_{last}(\eta^i_{j,k}).d \\
\overset{Prop.\ 7}{\Rightarrow} \quad \eta^i_{j,k}.d^{\#} &< \phi^i_l.d + s_{last}(\eta^i_{j,k}).d^{\#} - s_{last}(\eta^i_{j,k}).d \\
\Rightarrow \quad \eta^i_{j,k}.d^{\#} &< \phi^i_l.d + s_{last}(\eta^i_{j,k}).d^{\#} - s_{last}(\eta^i_{j,k}).d + \sum_{b=m+1}^{l-1} \phi^i_b.p \\
\overset{Lemma\ 1}{\Rightarrow} \quad \eta^i_{j,k}.d^{\#} &< \phi^i_l.d + \phi^i_m.d^{\#} - \phi^i_m.d + \phi^i_m.p + \sum_{b=m+1}^{l-1} \phi^i_b.p \\
\Rightarrow \quad \eta^i_{j,k}.d^{\#} &< \phi^i_m.d^{\#} + \phi^i_m.p + \sum_{b=m+1}^{l-1} \phi^i_b.p + \sum_{b=m}^{l-1} \phi^i_b.dur \\
\Rightarrow \quad \eta^i_{j,k}.d^{\#} &< \phi^i_l.d^{\#}
\end{aligned}
$$

which contradicts the initial hypothesis. We conclude that necessarily $\eta^i_{j,k} \in t^i_j|_{\phi^i_l}$.

□

5 Conclusion

This chapter introduced the core concepts of the multi-phase model and then formally defined it. We proposed 3 correctness criteria that guarantee that the implementation of a task system described in the multi-phase model is correct w.r.t. a chosen interference-aware static schedule. These criteria are very simple and are agnostic to the method used to create the profile. It makes them easy to verify and offers room for optimizations in the analysis of tasks in order to derive a profile as we will see in Chapter 4.

These two aspects can be presented as optimization problems. Indeed, we observed in our experiments that the results of the interference analysis on a task system is highly impacted by the shape of the multi-phase profiles representing the tasks. Hence, the multi-phase model can both improve or degrade the worst-case interference scenario computed with the single-phase model, depending on the way the tasks have been divided in phases. However, this impact is not known when the profile is under construction so it is only possible to favor some characteristics that generally yield gains.

At the same time, the implementation of a synchronization mechanism has not been addressed in this thesis, but it may represent an obstacle to the implementation of the multi-phase model. The three correctness criteria that allow to bound the number of accesses in each phase do not make assumptions on the way synchronizations must be implemented, which leaves room for many solutions. In this chapter, we defined equivalent nodes as all the nodes that represent the same instruction and we stated that synchronizations are applied on equivalence classes. Therefore, our definition of equivalent nodes complies with synchronization mechanisms that are not aware of context information (e.g. the trace that is being executed or the current iteration in a loop). By extension, it also complies with context-aware mechanisms. However, in order to fully benefit from a context-aware mechanism, the definition of equivalent nodes should be extended in order to define different synchronization dates for the same instruction.

The definition of equivalence also rules the impact of synchronizations on the traces. Indeed, when one of the equivalent nodes synchronized is not already at the synchronization date, its trace requires modifications on the date of certain nodes. Another aspect to consider regarding the synchronization mechanism is the impact of the synchronization code. If the binary code is modified then the static analysis of the code is no longer valid: we need to account for the additional instructions in the WCET and perform a new instruction cache analysis if needed. A solution to avoid the new instruction cache analysis is to store all the instructions of the task in a scratchpad before its execution. In the same way, if the code or the synchronization date must be loaded from a shared memory, the additional accesses must be accounted for. We let these implementation questions open: in the following we only consider the adjustments required on the traces that may

be necessary before re-applying the correctness criteria.

Following the method to correctly compute the accesses in the phases of a task, there are still three interdependent aspects to address: choosing a multi-phase profile, selecting synchronizations such that their number is acceptable with respect to their impact on the code, and trying to reduce the over-estimation of the worst-case number of contentions. These issues are addressed in the next chapter.

Chapter 4

From the binary code to the multi-phase representation of a task

Contents

The last chapter introduced the formal definitions of the multi-phase model. Then, we presented a conservative method to account for the accesses in each phase, provided a technique is able to derive a profile and a selection of synchronizations for the task. In this chapter, we propose different techniques to create phases from traces, select synchronizations, and optimize the resulting profile. We only suppose that the traces of the task have been retrieved using a static analysis of its binary code. A comparative study of the techniques presented in this chapter is conducted in Chapter 5.

A simple heuristic method based on Kernel Density Estimation (KDE) is presented in the first part of the chapter. The advantage of this method is that it is able to cluster the accesses of a task in phases. Once the phases have been created, the selection of synchronizations is a crucial step because it rules the number of accesses that each phase must account for by limiting the interval in which each memory access can occur. Hence, we propose a method to select synchronizations that reduces the over-approximation of the number of accesses in each phase. This process must be conducted carefully because implementing the synchronizations impacts the code in several ways. As a result, we adopt a generic approach to select synchronizations, compliant with a variety of implementation schemes.

In addition, after some preliminary experiments, some optimization passes have been developed to increase the efficiency of the created profiles, with respect to the interference analysis. In particular, the optimization methods target three means to optimize a profile: increasing the time when no accesses can be performed, packing the accesses together in phases to increase the precision of the interference analysis and reducing the access over-approximation inherent to the shape of the profile.

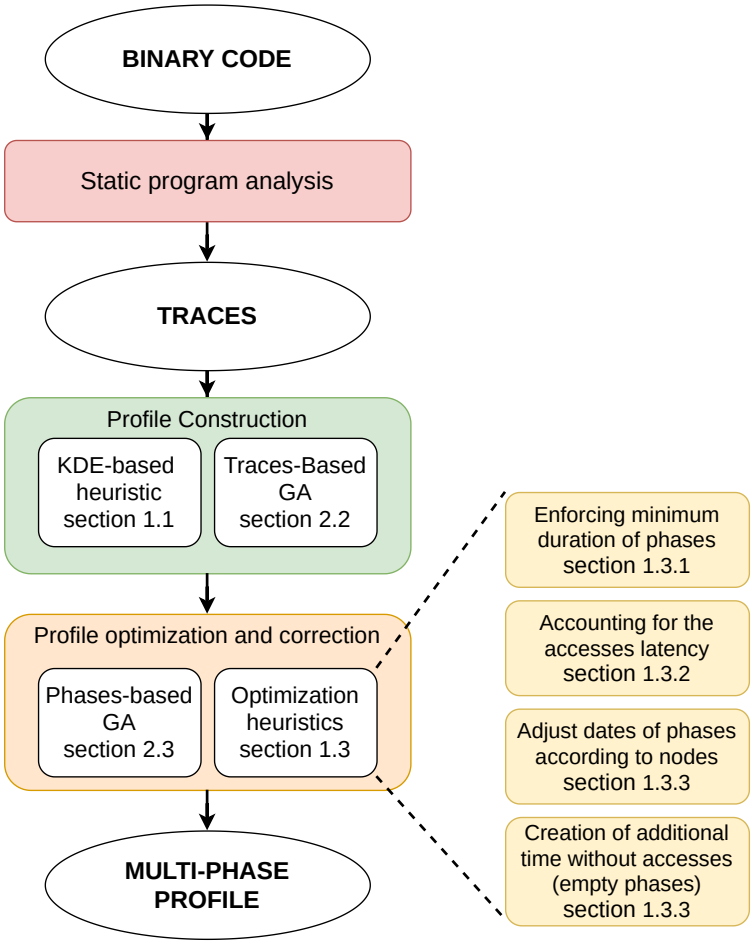


Figure 4.1: Construction of a multi-phase profile from its binary code.

Finally, as several interdependent parameters are at play when creating a profile, this process can be viewed as a multi-objective optimization problem with a large solution space if we consider both the creation of phases and the selection of synchronizations. Hence, two new methods to build a profile are proposed based on a Genetic Algorithm (GA), which allow to explore more solutions and to compute a trade-off between the objectives. The first method starts from the traces of the tasks as for the KDE-based technique. On the other hand, the second method starts from an initial profile, so it is proposed as a means to optimize existing profiles while keeping the original characteristics of the profile.

The whole process of building a multi-phase profile presented in this chapter is described in Figure 4.1. The numbers are referring to the sections where each technique is described. Firstly, the static analysis of the program allows to derive the traces representation of the task. This step is based on state of the art techniques of the literature and is not discussed in this chapter. Then, either a KDE heuristic or a GA presented in this chapter can be applied on the traces to obtain a first sequence of phases. Finally, the optimization and correction step further improves the initial profile and checks that it is correct. It is also conducted either by simple heuristics or by another GA.

1 A simple heuristic to compute a multi-phase profile from the traces representation of a task

This section proposes a technique to compute a multi-phase profile from the set of traces of a task. The core of the technique relies on the application of *Kernel Density Estimation (KDE)* on

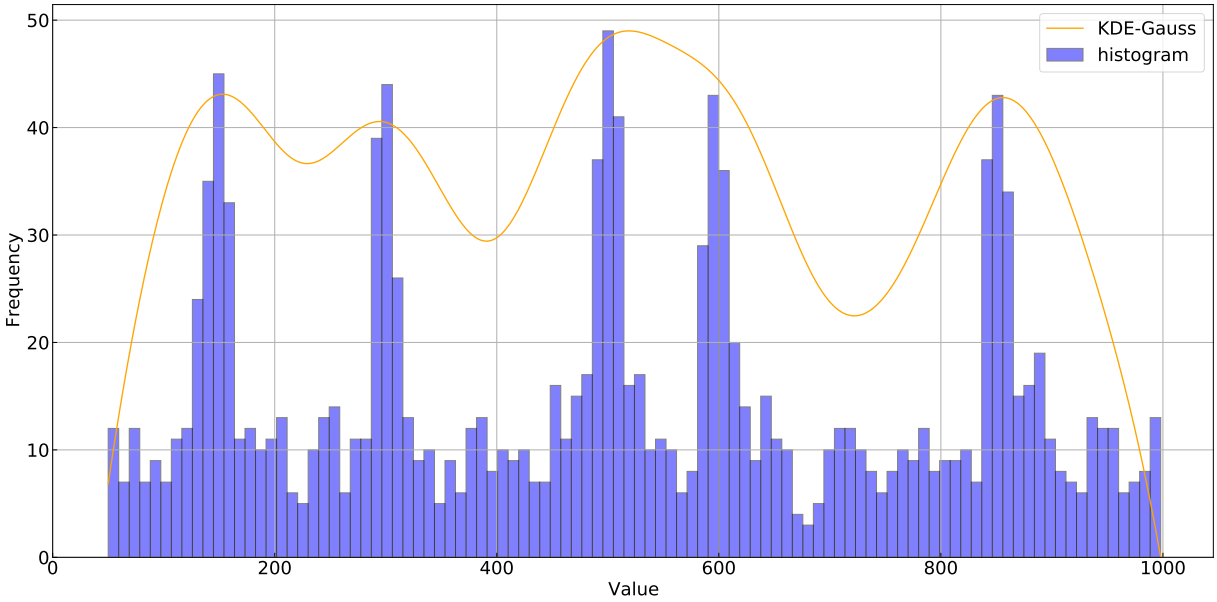


Figure 4.2: Kernel Density Estimation applied on a random distribution.

a distribution of the worst-case dates at which the accesses of a task can be performed.

1.1 Creating the phases from nodes using Kernel Density Estimation

We start by presenting the basic concepts of KDE in section 1.1.1, and then explain how we applied it to create phases from the traces of a task in the subsequent sections.

1.1.1 Kernel Density Estimation (KDE)

Let's consider a set of n independent and identically distributed (i.e. with the same probability distribution) observed data points forming a sample $\{x_0, x_1, ..., x_n\}$ from a distribution X with unknown density probability function $f(x)$. We can estimate $f(x)$ using the KDE method. This estimator is defined by:

$$\widehat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big) \tag{4.1}$$

with K a kernel (e.g. uniform, Gaussian or Epanechnikov) and $h > 0$ the bandwidth or smoothing parameter.

KDE is sometimes called continuous histogram, the reason is visible in Figure 4.2. The idea is to sum the kernels of the observations to obtain the continuous underlined probability density function.

1.1.2 Application of KDE to build phases

The intuition behind the use of KDE is that, using the dates of instructions that may perform accesses, we can derive a continuous function that compiles all these dates and describes clusters of accesses. The clusters can be directly used to create phases and form a profile.

In practice, the set of worst-case dates of accesses in the traces is used as an input sample to obtain the associated continuous function describing when the accesses of the task may occur (in the worst-case). Then, the phases can simply be delimited by the position of the local extrema of this function (defining the positions of the clusters).

Algorithm 1 KDE-based algorithm to build a multi-phase profile

Require: h ; $kernel$
1: $sample = []$
2: $\mathbb{P}^i = []$
3: **for** $t^i_j \in \mathbb{T}^i$ **do**
4: **for** $\eta^i_{j,k} \in t^i_j$ **do**
5: $push(sample, \eta^i_{j,k}.d)$
6: **end for**
7: **end for**
8: $\widehat{f_h}(x) = KDE(sample, h, kernel)$
9: $extrema = getExtremaDates(\widehat{f_h}(x), 0, WCET(\tau^i))$
10: $date = 0$
11: $phaseIdx = 0$
12: **for** $e \in extrema$ **do**
13: $phaseBudget = e - date$
14: $\phi^i_{phaseIdx} = createPhase(date, phaseBudget)$
15: $\mathbb{P}^i.push(\phi^i_{phaseIdx})$
16: $phaseIdx = phaseIdx + 1$
17: $date = date + phaseBudget$
18: **end for**
19: **return** $\mathbb{P}^i$

Algorithm 1 presents the function transforming the traces representation of a task τ^i to a multi-phase profile using KDE. The parameters are the bandwidth h and the kernel function to be used by the KDE method. By default, we use a Gaussian kernel (this parameter is recognized to have a minor effect on the estimation [79, 80] and the choice of h is discussed in the next paragraph). In lines 3 to 7, the algorithm collects the worst-case dates of nodes from all traces of τ^i. These dates are then used to compute the corresponding probability density function $\widehat{f_h}(x)$ (line 8). In line 9, a function retrieves the extrema of $\widehat{f_h}(x)$ in the interval $[0, WCET(\tau^i)[$. Eventually, a phase is created between each consecutive extrema and added to the new profile of the task (lines 12 to 18).

Figure 4.3 shows the application of the KDE-based method on a task with 5 traces, with a bandwidth factor of 50 cycles (which corresponds to the duration of a memory access in the analysis of the task). The 5 traces of the task are depicted by horizontal lines with nodes representing the memory instructions (each instruction is assigned an index visible in the figure), the KDE function is plotted in blue with vertical lines to indicate the positions of minimum (in red) and maximum (in green) extrema that were used to delimit the phases at the bottom of the figure.

Influence of the smoothing parameter h. The smoothing parameter highly impacts the results of KDE, in particular it impacts the bias and the variance of the estimator. A high value (*over-smoothing*) hides a lot of information, because it makes $\widehat{f_h}(x)$ cover a large neighborhood of x, inducing a high bias and a low variance. On the contrary, a low value (*under-smoothing*) may create artificial variations on the curve. For example, Figure 4.4 shows the KDE-based profile of the same task as Figure 4.3 but with a bandwidth of 30 cycles instead of 50 cycles. We observe that the number of phases increases due to the higher number of extrema in the function.

There exists some automatic methods to define h such as Scott's rule where $h = n^{(-1/(d+4))}$ or Silverman's rule where $h = (n(d+2)/4)^{(-1/d+4)}$ [79] with n the number of observations and d the number of dimensions (1 in our case). For our problem, the h value determines the number of clusters and phases. On the one hand, it is important to highlight the details of the distribution to isolate accesses so h has to be relatively low. On the other hand, highlighting too many details is unnecessary as in Figure 4.4 where some phases are not useful and only increase the number of required synchronizations.

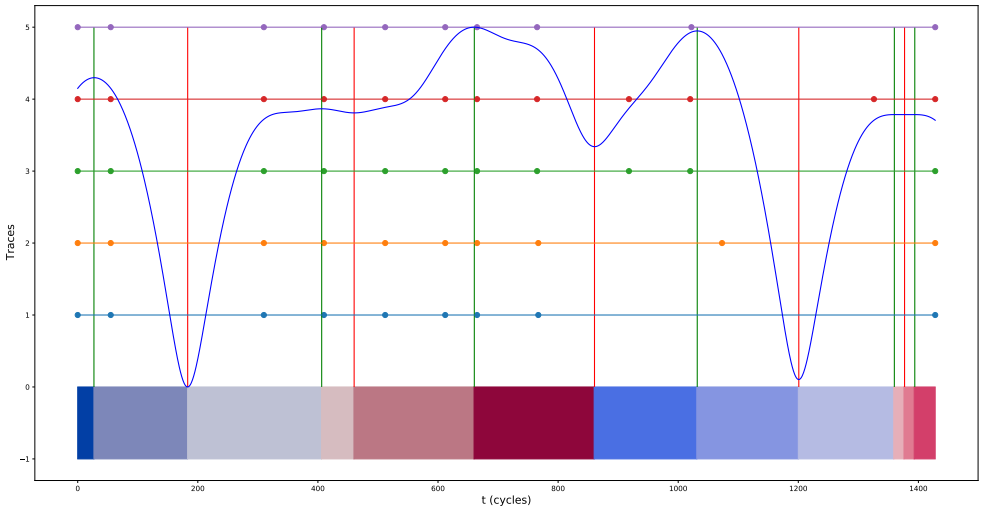


Figure 4.3: KDE-based profile and the traces of a task with bandwidth 50 cycles.

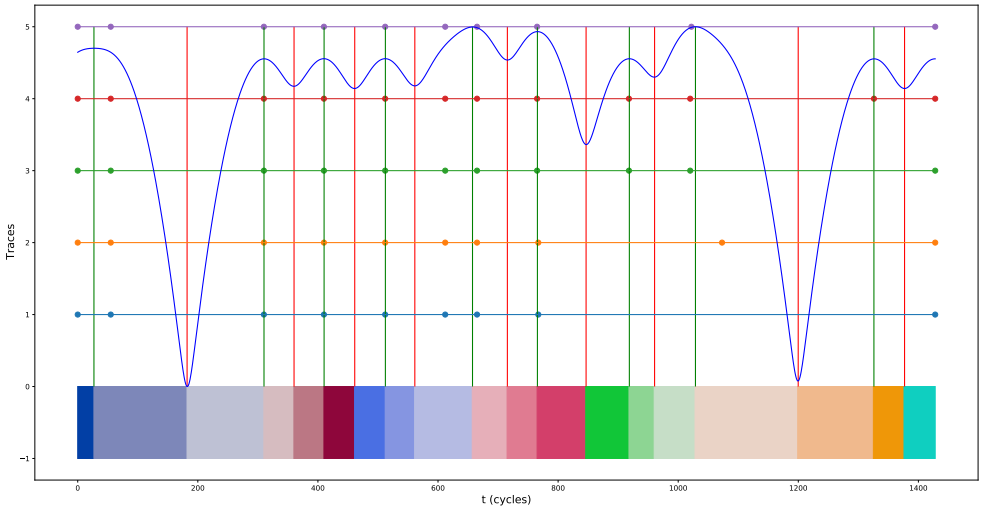


Figure 4.4: KDE-based profile and the traces of a task with bandwidth 30 cycles.

Imbalance of the weight of instructions in the sample of dates It is worth noting that an instruction represented n times across the traces is also represented n times in the sample of dates used to apply KDE, even if the dates across the traces are identical. Consequently, if an instruction appears at the same date in different traces, it increases the number of observations of the sample and the chances that a phase is created at this date. Such situation is beneficial when the over-represented instruction is synchronized because it prevents many nodes from being executed in several phases, with only one synchronization added to the code.

1.2 Selecting synchronizations

The selection of synchronizations is a necessary step to control the maximum number of accesses in each phase of a profile, hence it is also a good leverage to limit the access over-approximation of the multi-phase model. Ideally, each node performing an access could be synchronized so that the accesses would only be feasible in a unique phase of the profile. However, it is important to reduce the number of synchronizations so that the size of the corresponding additional code remains limited. This section presents and discusses the rules applied to select the synchronizations. The first rule is applied before creating the phases, in order to reduce the complexity of the analysis. The others are applied once the phases have been computed with the KDE-based algorithm. These

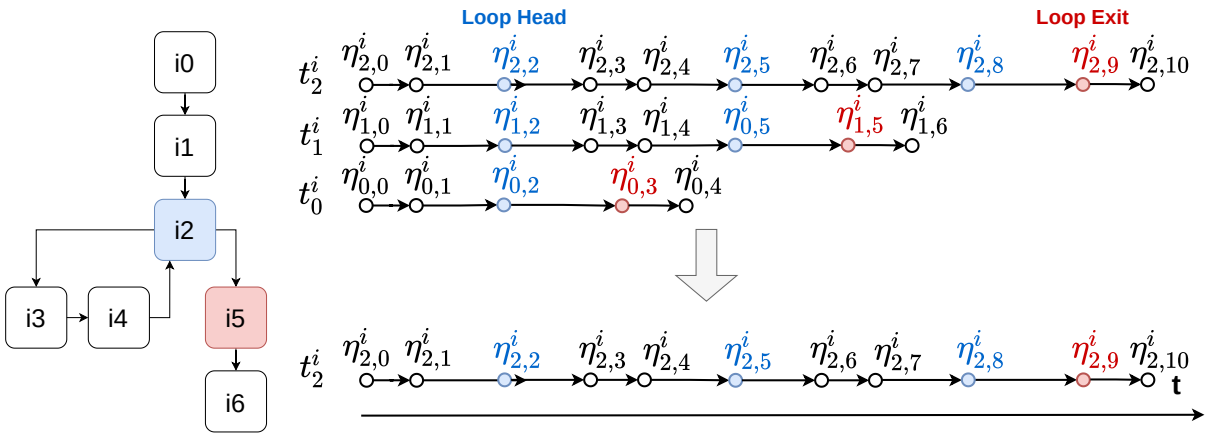


Figure 4.5: A simple CFG with a loop that has 0 to 2 iterations, and the corresponding traces in the right side. At the top the traces considering all the loop iterations and at the bottom the only trace that will be considered during the analysis.

rules can also be applied to any existing profile, obtained with any other method.

1.2.1 Simplification of the traces through synchronization

The presence of loops (resp. if-statements) increases the number of traces to consider because all the possible number of iterations (resp. all the possible branches) must be considered. We propose to simplify the traces to reduce the analysis complexity. The simplification is based on the synchronization of the node that post-dominates the considered loop or if-statement, i.e. the node located after the considered control-flow structure, and through which all the traces with some nodes in this structure pass.

Let's take Figure 4.5 as an example. The top of the figure depicts three traces t_2^i, t_1^i and t_0^i that represent the execution of the loop in the left side of the figure with respectively 2, 1 and 0 iterations. The loop exit instruction i5, represented by $\eta_{2,9}^i$, $\eta_{1,5}^i$ and $\eta_{0,3}^i$, post-dominates instructions i3 and i4 in the loop and the loop head instruction i2 (i.e. any path from the first instruction i0 including i2, i3 or i4 also passes through i5 to reach the exit instruction i6). If we synchronize instruction i5 to its worst-case date ($\eta_{2,9}^i.d$), then it is possible to only consider trace t_2^i that has the maximum number of iterations because:

- t_2^i represents the information of the loop contained in the other traces.
- Instruction i5 is executed at the same date in the three traces (because we synchronize it at $\eta_{2,9}^i.d$) and also has the same successors with the same dates.

In the following we systematically apply this simplification when the traces are enumerated. Some synchronizations selected during subsequent steps can be desynchronized, i.e. the lower bound on the execution date specific to the instruction is removed. However the synchronizations defined during the traces enumeration are never desynchronized. Otherwise, the traces that can be ignored using the synchronizations would no longer be represented, so it would not be possible to guarantee that the model encompasses all possible executions of the task.

1.2.2 Systematic synchronization algorithm

A variety of methods can be employed to implement synchronizations, as the three correctness criteria of Chapter 3 do not make any assumptions. In order to remain as generic as possible in our hypotheses we propose to assume a context-independent mechanism, i.e. a mechanism that always

Algorithm 2 syncNodesHeuristic

Require: $\mathbb{T}^i$; $\mathbb{P}^i$
1: $phasesSyncs = getSynchroCandidates(\mathbb{T}^i, \mathbb{P}^i)$ ▷ get the nodes complying with rule 3
2: $phIdx = 0$
3: **while** $phIdx < \Phi^i$ **do**
4: $syncCandidates = phasesSyncs[phIdx]$
5: **for** $\eta^i_{j,k}$ **in** $syncCandidates$ **do**
6: **if** $\eta^i_{j,k}.sync$ **then**
7: **continue** ▷ rule 2
8: **else**
9: $synchronizeNode(\eta^i_{j,k})$
10: **end if**
11: **end for**
12: **end while**

synchronizes at the same date, regardless of the execution context. Such a mechanism could be implemented with a simple busy-wait loop as mentioned in [53].

In this section we propose a set of rules and a heuristic to select synchronizations nodes and synchronization dates such that the access over-approximation (see definition in Property 4 of Chapter 3) is limited and the resulting traces remain correct. Moreover, we present a basic synchronization heuristic that aims at preventing a maximum of nodes from being accounted for in several phases to limit the access over-approximation while respecting the synchronizations that were selected during the enumeration of traces. For this purpose, the three following rules are proposed:

Synchronization Rule 1. *The synchronizations selected prior to the design of the profile or retrieved from the previous analyses (e.g. during the traces enumeration) cannot be desynchronized.*

Synchronization Rule 2. *The selection of synchronizations is restricted to nodes that are outside loops because the synchronization of instructions repeated in a same trace requires a context-aware mechanism.*

Synchronization Rule 3. *For each trace, at least the first node in each phase is synchronized.*

Algorithm 2 describes the basic heuristic to select synchronizations in compliance with the rules. Firstly, the nodes complying with rule 3 are retrieved by calling the *getSynchroCandidates* function (line 1). Then, we iterate through the phases to get their list of nodes to synchronize (line 4). For each node of *syncCandidates* the if-then-else structure beginning at line 6 processes the node under study ($\eta^i_{j,k}$): if the node is already synchronized (by previously processing an equivalent node) the synchronization is removed, otherwise the node and all its equivalents are synchronized on the same date (see Algorithm 4).

Algorithm 3 is used to get the minimal list of nodes complying with rule 3 for each phase. Formally, it selects for each phase ϕ^i_l and for each trace t^i_j, the node $\eta^i_{j,n}$ whose date is within the phase and the closest to $\phi^i_l.d$. It returns a list made of these nodes. The algorithm scans each trace to detect the first node in each phase that is not in a loop (rule 2). The variables n and p are respectively the index of the current node and the current phase. Until the last node of the current trace, the first **if** ensures that p points to the right phase (this could be a predecessor of this phase if no node could be synchronized in it). The **else if** (at line 11) checks that the current node is within the phase and not in a loop (at line 11). If so, the node is pushed to the list of synchronized nodes of the current phase and we go to the next phase by increasing p so that only one node per trace is appended to $synchros[p]$. The final *else* (line 14) simply goes to the next node in the case where the current node cannot be synchronized.

Algorithm 3 getSynchroCandidates

Require: $\mathbb{T}^i$; $\mathbb{P}^i$
1: $synchros = []$
2: **for** ϕ^i_p **in** $\mathbb{P}^i$ **do**
3: $synchros.push([])$
4: **end for**
5: **for** t^i_j **in** $\mathbb{T}^i$ **do**
6: $n = 0$ ▷ Index of the current node
7: $p = 0$ ▷ Index of the current phase
8: **while** $n < N^i_j$ **do**
9: **if** $\eta^i_{j,n}.d \geq \phi^i_{p+1}.d$ **then**
10: $p++$
11: **else if** $(\phi^i_p.d \leq \eta^i_{j,n}.d < \phi^i_{p+1}.d) \wedge (\neg isInLoop(\eta^i_{j,n}))$ **then**
12: $synchros[p].push(\eta^i_{j,n})$
13: $p++$
14: **else**
15: $n++$
16: **end if**
17: **end while**
18: **end for**
19: **return** *synchros*

1.2.3 Synchronizing a node

When a node is selected to be synchronized, synchronization code is inserted before the instruction it represents in the binary. Because we assume a context-independent synchronization, all the nodes representing a synchronized instruction (i.e. equivalent nodes) must have the same synchronization date. As a result, a synchronization impacts the date of nodes in the traces where the instruction is present. Therefore, we introduce some additional synchronization rules to correctly manage the consequences of a synchronization in the traces:

Synchronization Rule 4. *Equivalent nodes are always synchronized or desynchronized together so:* $\forall \eta^i_{l,m} \in \mathbb{N}^i_{equiv}(\eta^i_{j,k}.it), \eta^i_{l,m}.sync = \eta^i_{j,k}.sync$

Synchronization Rule 5. *The synchronization date of a node* $\eta^i_{j,k}$ *is the worst-case date of the instruction it represents in all traces:* $\eta^i_{j,k}.sync \implies \eta^i_{j,k}.d = wced(\eta^i_{j,k}.it)$.

Moreover, we recall that if a node $\eta^i_{j,k}$ is not synchronized then the duration between its date and the date of its predecessor $\eta^i_{j,k-1}$ is always equal to the partial WCET between the two instructions (see Property 7 in Chapter 3):

$$\forall \eta^i_{j,k},\ s.t.\ \neg\eta^i_{j,k}.sync : \eta^i_{j,k}.d - \eta^i_{j,k-1}.d = wcet(\eta^i_{j,k-1}.it, \eta^i_{j,k}.it)$$

Figure 4.6 represents a single trace t^i_0 with $\eta^i_{0,2}$ synchronized in two situations: the top and bottom representations depict respectively the trace before and after the synchronization of $\eta^i_{0,4}$. We assume that a node equivalent to $\eta^i_{0,4}$ has a later date in another trace ($wced(\eta^i_{0,4}.it) > \eta^i_{0,4}.d$) so $\eta^i_{0,4}$ must be shifted to the right when it is synchronized. This date modification also affects $\eta^i_{0,5}$ because as it is not synchronized, the WCET between $\eta^i_{0,4}.it$ and $\eta^i_{0,5}.it$ remains identical.

Algorithm 4 presents how a node and its equivalent are synchronized in compliance with the synchronization rules. The synchronization date retrieved at line 1 is the worst-case execution date of the instruction across all traces. If the synchronization date is superior to the current date of the node (line 3), then we need to update the execution date of the node and of its successors to

Algorithm 4 synchronizeNode

Require: $\eta^i_{j,k}$
1: $sync\_date = wced(\eta^i_{j,k}.it)$
2: **for** $\eta^i_{j',k'}$ **in** $\mathbb{N}^i_{equiv}(\eta^i_{j,k}.it)$ **do**
3: **if** $\eta^i_{j',k'}.d < sync\_date$ **then**
4: $diff = sync\_date - \eta^i_{j',k'}.d$
5: $t = k'$
6: **while** $t < N^i_{j'} \wedge \neg(\eta^i_{j',t}.sync)$ **do**
7: $\eta^i_{j',t}.d = \eta^i_{j',t}.d + diff$
8: **end while**
9: $\eta^i_{j',k'}.sync = 1$
10: **end if**
11: **end for**

maintain the WCETs between the nodes until reaching the next synchronized node whose date cannot change according to the first rule. The operation is done in the while at line 6. The shift value is computed at line 4 from the date of the node to synchronize. In order to desynchronize a node $\eta^i_{j,k}$ the operation is somewhat inverted: the successor nodes until the next synchronization are shifted towards inferior dates so that the new date of $\eta^i_{j,k}$ is:

$$\eta^i_{j,k} = s_{last}(\eta^i_{j,k}).d + \sum_{\eta^i_{j,l}=s_{last}(\eta^i_{j,k})}^{\eta^i_{j,k}-1} wcet(\eta^i_{j,l}.it, \eta^i_{j,l+1}.it)$$

It is important to note that postponing the synchronization date of a node $\eta^i_{j,k}$ to $wced(\eta^i_{j,k}.it)$ has no effect on theWCET of the task. First, if the equivalent nodes to synchronize are at the same date, then none is postponed during their synchronization so theWCET is not changed. We propose to study the other case (i.e. at least one of the equivalent nodes has a different date than the others) in the remainder of the section.

If two equivalent nodes $\eta^i_{j,k}$ and $\eta^i_{l,m}$ do not have the same date, then they have either a different number of predecessors or their predecessors are not all equivalent 2 by 2 (i.e. $\exists h < k, \neg(\eta^i_{j,h} \sim \eta^i_{l,h})$). We assume that infeasible paths are not discarded during the enumeration of the traces, so the set of instructions that can be executed just after $\eta^i_{j,k}.it$ is the same regardless of the execution context (i.e. regardless of the trace). Therefore, there exists an equivalent node $\eta^i_{l',m'}$ from a different trace than t^i_j whose successor $\eta^i_{l',m'+1}$ is equivalent to $\eta^i_{j,k+1}$.

Property 8. *Let's consider $\eta^i_{j,k}$ and $\eta^i_{l,m}$ s.t. $\eta^i_{j,k} \sim \eta^i_{l,m}$ and $\eta^i_{j,k}.d \neq \eta^i_{l,m}.d$. Then:*

$$\exists \eta^i_{l',m'}, l' \neq j, m', (\eta^i_{j,k} \sim \eta^i_{l',m'}) \wedge (\eta^i_{j,k+1} \sim \eta^i_{l',m'+1}) \tag{4.2}$$

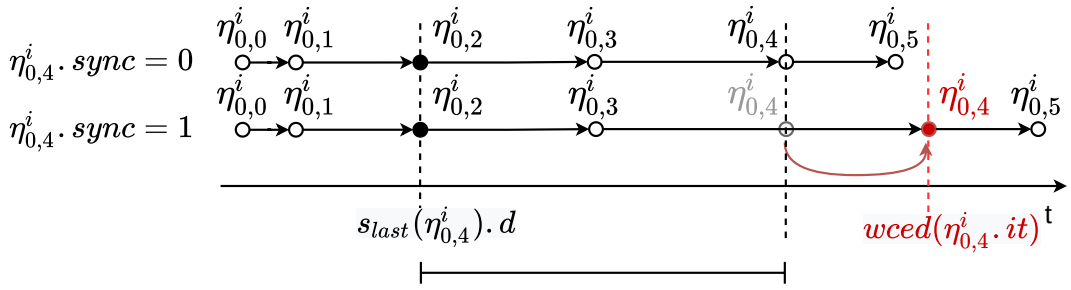


Figure 4.6: A trace with and without its node $\eta^i_{0,4}$ synchronized.

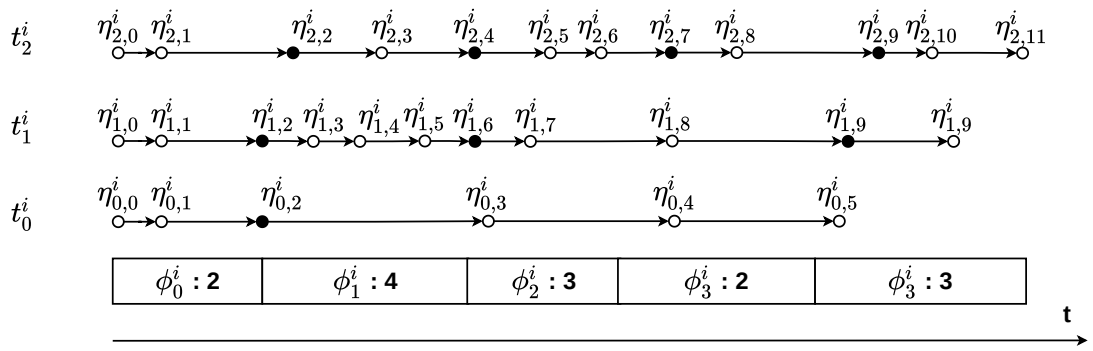


Figure 4.7: 3 traces and a profile with an optimization on the synchronizations used for t_0^i.

By consequent, if $\eta_{j,k}^i$ is postponed due to its synchronization, we are guaranteed to find an equivalent node $\eta_{l',m'}^i$ that executes at date $wced(\eta_{j,k}^i.it)$, whose successors are a series of nodes that are equivalent to the successors of $\eta_{j,k}^i$. Postponing $\eta_{j,k}^i.d$ and the dates of its successors accordingly would thus only make these two traces coincide until the end, so the WCET of the task cannot change due to the synchronization of a node.

1.2.4 Optimizing the selection of synchronizations

The systematic selection of synchronizations is beneficial during the design of a profile because it is fast and efficient to limit the access over-approximation. Nonetheless, it also selects synchronizations that have no effect on the count of accesses, while we want to limit their number. Figure 4.7 shows an example of an efficient selection of synchronizations with three different traces. Note that t_0^i has very few accesses compared to the other traces. If we applied the systematic selection method, all its nodes except $\eta_{0,1}^i$ would be synchronized while if we synchronize only $\eta_{0,2}^i$ the number of accesses in each phase does not change. For instance, t_1^i performs up to 4 accesses in ϕ_1^i which is also the number of accesses that t_0^i can perform (from $\eta_{0,2}^i$ to $\eta_{0,5}^i$), so synchronizing $\eta_{0,3}^i$ has no effect on the accesses number of ϕ_1^i.

Algorithm 5 presents an optimization that identifies such redundant synchronizations and removes them. The idea is to assess for each synchronized node if its desynchronization would increase the number of accesses in a predecessor of the phase where it is synchronized. It is not necessary to go back to the first phase of the profile but only to the phase containing the previous synchronization because this synchronization prevents the accesses from being performed in a prior phase. The function iterates backwards through the phases of $\mathbb{P}^i$ using *idxPhase*. For each phase, it retrieves the synchronized nodes within the phase using function *getSyncIn* that is presented by Algorithm 6 thereafter. Then, the for at line 5 iterates through the synchronized nodes retrieved in order to assess if they can be desynchronized. As equivalent nodes are synchronized and desynchronized together, they are processed at the same time so we prevent the algorithm from studying the same set of equivalent nodes multiple times by storing the instructions that have already been studied in the list *visited* (line 9). If a set of equivalent nodes has already been processed, the *if* at line 6 passes to the next synchronized node to study.

In the *for* from line 11, the algorithm looks at all the equivalent nodes $\eta_{l,m}^i$ under study to assess if their desynchronization on their trace t_l^i may increase the number of accesses in the current phase $\phi_{idxPhase}^i$. First, it computes the number of accesses in $\phi_{idxPhase}^i$ performed by t_l^i denoted *accs* (line 12). If $\eta_{l,m}^i$ were to be desynchronized, this would be the number of additional accesses from t_l^i that the predecessor phases finishing after $s_{last}(\eta_{l,m}^i).d$ would have to account for. Thus, the *while* at line 14 checks if the number of accesses for these predecessor phases $\phi_{idxPhaseLast}^i$ (those finishing after $s_{last}(\eta_{l,m}^i).d$) without $\eta_{l,m}^i$'s synchronization is not superior to their current number of accesses (line 17). If so, the desynchronization is cancelled by setting *noAddOverApp* to *False*

Algorithm 5 optimSyncSelection

Require: $\mathbb{T}^i$; $\mathbb{P}^i$
1: $idxPhase = \Phi^i - 1$
2: **while** $idxPhase > 0$ **do**
3: $sync\_nodes = getSyncIn(\phi^i_{idxPhase}, \mathbb{T}^i)$
4: $visited = []$
5: **for** $\eta^i_{j,k}$ **in** $sync\_nodes$ **do**
6: **if** $\eta^i_{j,k}.it \in visited$ **then**
7: **continue**
8: **end if**
9: $visited.push(\eta^i_{j,k}.it)$
10: $noAddOverApp = True$
11: **for** $\eta^i_{l,m}$ **in** $\mathbb{N}^i_{equiv}(\eta^i_{j,k}.it)$ **do**
12: $accs = \sum_{\eta^i_{l,p} \in t^i_l |_{\phi^i_{idxPhase}}} \eta^i_{l,p}.m$
13: $idxPhaseLast = idxPhase$
14: **while** $(s_{last}(\eta^i_{l,m}).d < \phi^i_{idxPhaseLast}.d) \wedge (noAddOverApp)$ **do**
15: $idxPhaseLast = idxPhaseLast - 1$
16: $currTraceAcc = \sum_{\eta^i_{l,p} \in t^i_l |_{\phi^i_{idxPhaseLast}}} \eta^i_{l,p}.m$
17: **if** $currTraceAcc + accs > \phi^i_{idxPhaselast}.m$ **then**
18: $noAddOverApp = False$
19: **end if**
20: **end while**
21: **if** $\neg noAddOverApp$ **then**
22: **break**
23: **end if**
24: **end for**
25: **if** $noAddOverApp$ **then**
26: $desynchronize(\eta^i_{j,k})$
27: **end if**
28: **end for**
29: $idxPhase = idxPhase - 1$
30: **end while**

and the algorithm passes to the next synchronized node (see the **break** at line 22). Otherwise, if *noAddOverApp* is still True after that all the equivalent nodes have been studied, then they are all desynchronized (line 26).

Algorithm 6 is used to retrieve the synchronized nodes within a given phase. It consists in iterating through the nodes composing the traces (line 4) to push the synchronized nodes whose synchronization date is within the phase into list *syncNodes* (line 5). The list *syncNodes* is returned at the end of the function.

1.3 Correction and optimization of the profile

This section describes methods that are applied after the creation of a profile. These methods enforce the correctness criteria presented in the previous chapter and some user-defined properties as for instance the minimum duration of a phase. Moreover, they favor some characteristics that increase the multi-phase model performance during the interference analysis. Initially the optimizations were developed specifically to be applied on the phases created with the KDE technique, but they can be applied to any existing profile.

As these methods change the dates of phases, synchronizations must be re-selected and the number of accesses in the phases recomputed once they have been applied.

1.3.1 Enforcing the minimum duration of phases

In order to harmonize the profiles of the tasks, it can be interesting to specify a minimum duration for the phases. We denote this parameter δ in the following. This parameter has a consequent impact on the scheduling and on the interference analysis results so it must be tuned with precaution. When the minimum duration is too high the profiles may be too close to the single-phase representation and limit the benefit of the multi-phase representation. On the contrary, a minimum duration that is too low will allow profiles with a lot a phases, on which the over-approximation of accesses is more difficult to control for two reasons: (1) it requires more synchronizations to prevent nodes from being executed in several phases, while we wish to keep the number of the synchronizations low and (2) there are more chances that some of the over-approximation cannot be controlled by the selection of synchronizations (see Section 2.3.1).

In addition, if the duration of phases in the profiles is not harmonized, there are more chances

Algorithm 6 getSyncIn

Require: ϕ_p^i ; $\mathbb{T}^i$
1: $syncNodes = []$
2: **for** $t_j^i \in \mathbb{T}^i$ **do**
3: $k = 0$
4: **while** $(k < N_j^i) \wedge (\eta_{j,k}^i.d \leq \phi_{p+1}^i.d)$ **do**
5: **if** $(\eta_{j,k}^i.sync) \wedge (\eta_{k,j}^i \in t_j^i|_{\phi_p^i})$ **then**
6: $syncNodes.push(\eta_{j,k}^i)$
7: **end if**
8: $k = k + 1$
9: **end while**
10: **end for**
11: **return** $syncNodes$

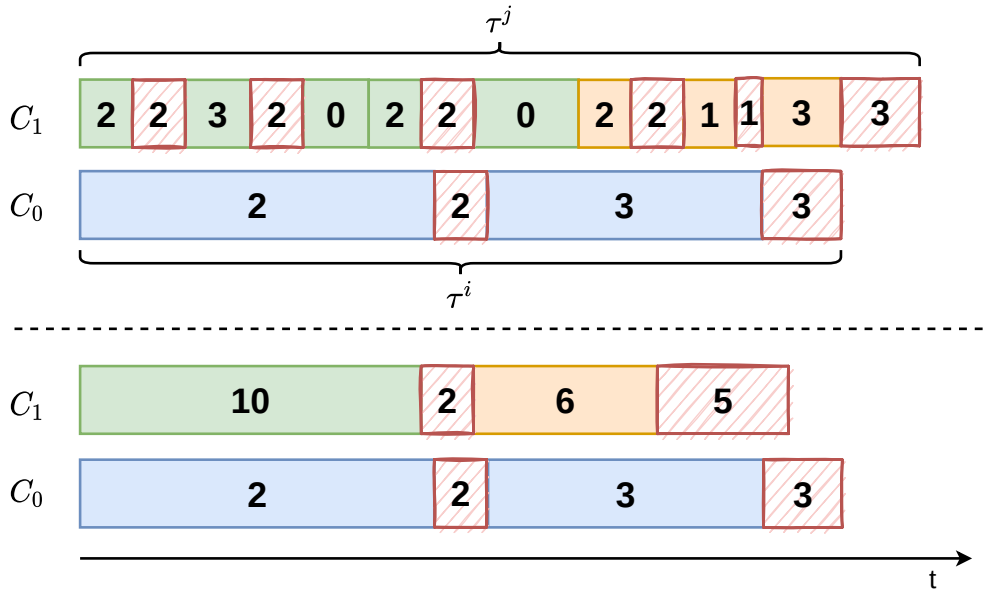


Figure 4.8: The profiles of two tasks before and after a pseudo-harmonization process.

that we find situations in which the multi-phase model accounts for more accesses than the single-phase model. Typically, the multi-phase model is less efficient when a long phase is in parallel with a lot of short phases whose total count of accesses is superior to the count of accesses of the long phase. The problem is that the interference analysis must consider that the accesses of the long phase may contend with any of the short phases in parallel. Harmonizing the duration of phases in the profiles makes this situation less probable. See as an example Figure 4.8 that depicts τ^i and τ^j respectively scheduled on core 0 and core 1, with different profiles for τ^i at the top and at the bottom. At the top, the phases of τ^i are much shorter than those of τ^j. The interference analysis must assume that the long phases of τ^j may create contentions with all the short phases of τ^j that have accesses. Therefore, the two blue phases are responsible for 12 possible contentions in τ^i while they have a total of 5 accesses. Suppose that a minimum duration has been set for the phases and that it resulted in the new profile of τ^i at the bottom: the green phases and then the orange phases are merged together. In this situation, the blue phases are responsible for only 7 possible contentions in τ^i which significantly decreases the end date of τ^i.

In order to enforce the minimum duration of phases, the correction method iterates over the phases and whenever $\phi^i_j.dur < \delta$, then the phase budget is increased by $\delta - \phi^i_j.dur$. If ϕ^i_{j+1} is completely covered by ϕ^i_j following this operation, then it is deleted. Otherwise, $\phi^i_{j+1}.d$ is postponed according to the increase of ϕ^i_j.

1.3.2 Accounting for the memory access latency in isolation

Previously, we accounted for the nodes in the phases assuming that an access is performed instantaneously at the date of its node. Indeed, nodes are represented at their worst-case date or at their synchronization date and we account for a node in a trace according to this date but without taking into account the latency of the corresponding memory accesses. Because of this latency, a node at the end of a phase can actually perform some of its accesses during the next phase. According to the configuration of the other traces on these two phases, we may have to account for this access in both of them, which increases the over-approximation of accesses in the model. The situation is illustrated in Figure 4.9 representing in blue an original profile computed using any technique and the output of the successive correction and optimization passes in green. The orange rectangles represent the worst-case memory latency for each node. With the initial profile in blue, node $\eta^i_{1,3}$ is accounted for only in ϕ^i_2 but may perform its accesses in ϕ^i_3.

It is possible to increase the number of accesses in ϕ^i_3 to account for $\eta^i_{1,3}$ but then the over-approximation of accesses increases. Instead, this correction pass ensures that a given node cannot span over several phases due to its memory accesses latency by increasing the duration of phases if necessary: if for a node $\eta^i_{j,k}$ accounted for in a phase ϕ^i_m we have $\eta^i_{j,k}.m \times l > \phi^i_m.d + \phi^i_m.dur$ with l the worst-case latency of one access, then we increase the $\phi^i_m.dur$ to cover the left hand term and correct ϕ^i_{l+1} accordingly. Back to Figure 4.9, the correction (a) increases the duration of ϕ^i_2 so $\eta^i_{1,3}$ can only perform its accesses in ϕ^i_2 in the green profile. Consequently, $\phi^i_3.d$ is postponed to $\eta^i_{j,k}.d + \eta^i_{j,k}.m \times l$.

1.3.3 Maximizing the span of empty phases

The presence of phases that do not perform accesses, called *empty phases* in the following, and specifically the amount of time they take in the task execution can increase the performance of the multi-phase model because they cannot interfere with other phases scheduled in parallel. We propose two operations to maximize the time without accesses in a profile:

1. Set the start date of phases to the date of their first node performing an access. This operation is particularly interesting if the previous phase is an empty phase because the end of the empty phase can be postponed so that its duration increases.

In Figure 4.9, on the initial blue profile, phase ϕ_1^i has no accesses so it is very interesting to schedule a maximum of phases with some accesses in parallel in order to avoid contentions. However, this phase ends before the next access so the scheduler cannot benefit from some extra time without accesses. By setting the start date of ϕ_2^i to the date of the first node already in the restriction of the phase (nodes $\eta_{2,2}^i$ and $\eta_{1,2}^i$) as depicted by situation (b), we can fully benefit from the time without accesses in ϕ_1^i on the green profile.

2. Create a new empty phase from an existing phase when there is sufficient time without accesses at its end. For each phase, if the time interval between the end of its last worst-case memory access and the end of the phase is superior to δ, then a new empty phase covering this interval is created.

   For example, ϕ_0^i finishes after the maximum accesses latency of the nodes in its restriction from $\eta_{0,1}^i$, $\eta_{1,1}^i$ and $\eta_{2,1}^i$. Therefore, the duration of $\phi_0^i.dur$ is reduced with operation (c) such that an additional phase without accesses is created at its end. This new empty phase is not visible on itself in the green profile because it is merged with ϕ_1^i.

Following the two operations, we see that $\phi_1^i.dur$ has increased so it is easier to schedule phases that perform accesses in parallel so that they are accesses are no interfered.

2 Enlarging the exploration space and handling multiple criteria with meta-heuristics

In the previous section, a heuristic produces a profile on which several optimization passes are applied to improve its efficiency regarding scheduling or interference analysis, based on preliminary experiments. The developed optimization passes show that it is complex to take into account the over-approximation, the number of synchronizations, the presence of empty phases or the packing of accesses with a single heuristic. Moreover, reducing the number of synchronizations may sometimes increase the over-approximation, so in this case, a trade-off must be found.

Therefore, we propose to employ meta-heuristics to create or reshape a multi-phase profile. Meta-heuristics are generic algorithms that can provide near-optimal solutions to optimization

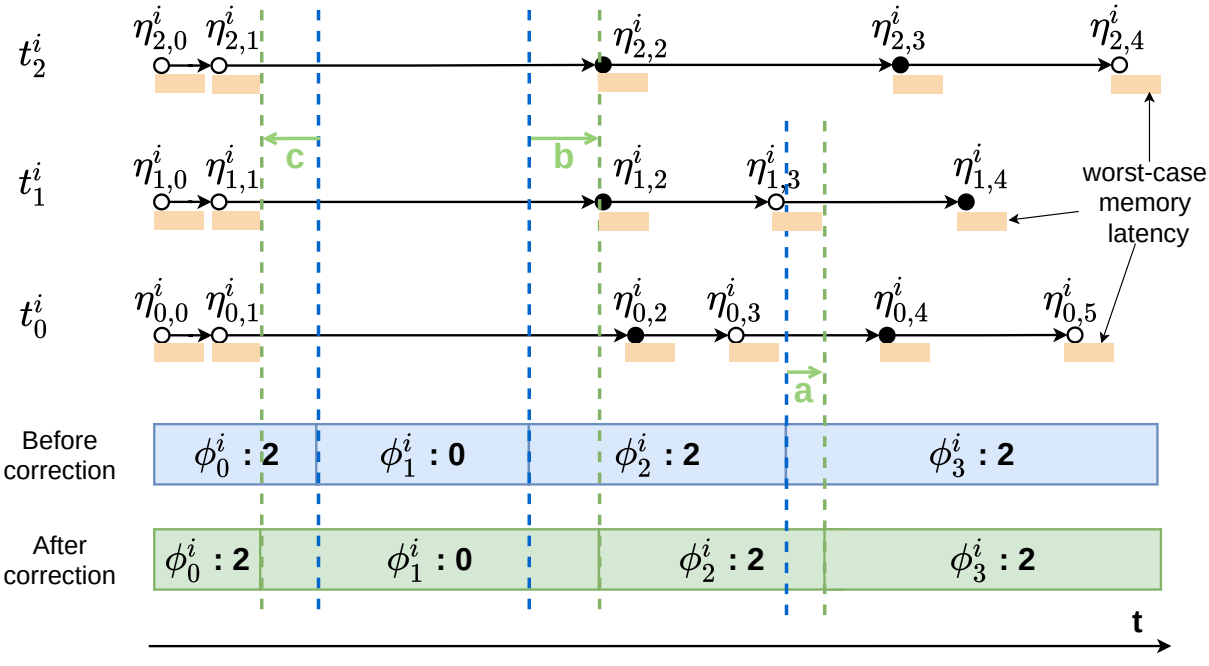


Figure 4.9: A profile before (blue) and after (green) applying corrections on the dates of phases.

problems. They are often used for complex problems when the exact solution cannot be obtained in a reasonable time or if the solution space is too large. In particular, we need a meta-heuristic that can explore a large space of solutions (because there are many ways to divide a task in phases) for a multi-objective optimization problem with many potential local optima. As discussed in Chapter 2 Section 3, there is a lot of meta-heuristics employed in research. The properties of our problem is leading to population-based methods that are more explorative [81]. In this family, the most popular algorithms are Particle Swarm Optimization (PSO) and Genetic Algorithms (GA). We implemented the latter which is very intuitive and is more adapted to problems with many local optima, partly due to the mutation of existing solutions.

2.1 General implementation of Genetic Algorithms

GAs are based on natural selection and genetic theories. The idea is to maintain a population of individual solutions, named chromosomes, over several iterations that are called generations. Each chromosome is composed of a vector of genes that encode a solution. The population evolves using mutation and crossover operations applied to the chromosomes that modify the values of genes. Chromosomes are evaluated at each generation so that they can be selected to be combined and form new chromosomes, or to evict those that represent the less interesting solutions.

We propose two techniques based on GAs: one that creates a profile and another that improves an existing profile. Firstly, Section 2.1.1 explains the advantages of GA approaches for our problem and presents the general implementation of the GA and its operators that are used in the two techniques. Then, Section 2.2 details the first technique that creates a profile from the traces of a task and Section 2.3 presents the other GA-based technique that reshapes existing profiles.

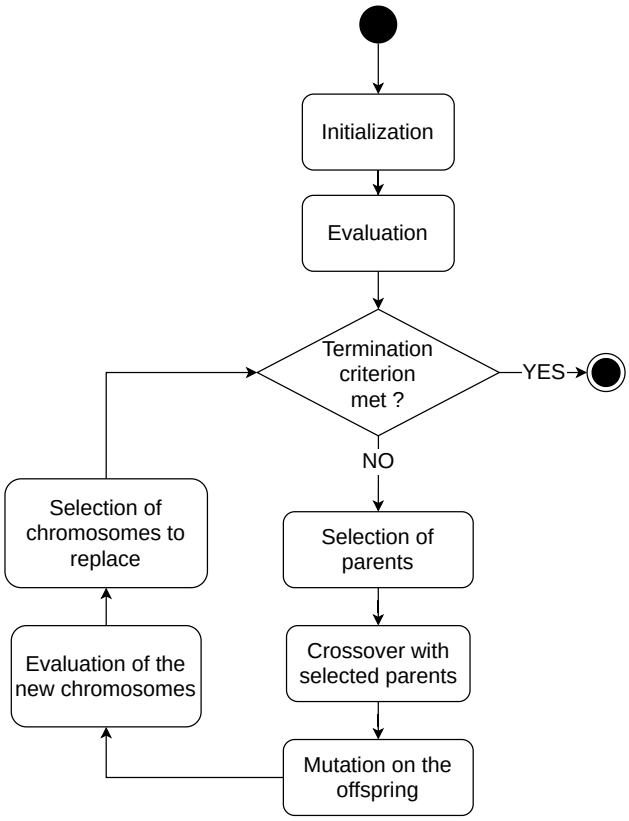


Figure 4.10: The steps of a Genetic Algorithm.

2.1.1 Introduction to Genetic Algorithms

The generic GA algorithm is represented in Figure 4.10. Firstly, the population is initialized and a first evaluation is performed to compute the fitness value of each chromosome. Then, the algorithm enters a running loop with 4 successive steps: (1) the selection evicts some chromosomes from the population and chooses others for the next step; (2) the selected chromosomes are recombined during crossover to create new chromosomes inserted in the population; (3) the mutation step modifies the values of some genes in the chromosomes and (4) the fitness value of the chromosomes is recomputed.

In the following, we denote Ψ^g the population of generation g ($g \geq 0$). Each population is composed of a set of X chromosomes χ_k^g ($0 \leq k < X$) defined by their sequence of genes $\chi_k^g = \{\gamma_{k,j}^g | 0 \leq j < \Gamma\}$.

2.1.2 Evaluation

The chromosomes are evaluated using a fitness function specific to each problem, that gives a fitness score expressing how good they are regarding the considered problem. The fitness functions used for our two GA techniques are given respectively in Sections 2.2.3 and 2.3.4.

2.1.3 Selection

The selection operator is used twice during each iteration of the GA:

1. To select the parent chromosomes before performing the crossover operation.
2. At the end of an iteration, to select the chromosomes from the current generation to keep in the new generation.

The following paragraphs describe two common selection policies for GA that we have used.

Steady-state selection. The steady-state selection method selects the best chromosomes of the population. The drawback is that these solutions may be a local maximum, so in that case, this selection method maintains the genetic algorithm in the local maximum.

Fitness proportionate selection. With the *Fitness Proportionate Selection* method, also called *Roulette Wheel Selection*, the higher the fitness of a chromosome, the higher the chance for it to be selected. This allows to further explore some poor solutions which may actually have more potential than the local best ones. The method is described by Algorithm 7. Function *compute_prob* at line 6 takes as argument a chromosome χ_k^g and returns a probability p_k such that:

$$p_k = \frac{f_k}{\sum_{t=0}^{X-1} f_t}$$

with X the size of the population and f_t the fitness of chromosome χ_t^g.

Lines 5 to 10 build the roulette wheel from the fitness values of the chromosomes in the population: the higher the fitness, the greater the surface covered by the chromosome on the wheel so the higher the probability to be selected. At line 12, the roulette wheel is set in motion to draw a random number k. Then, the chromosome covering the interval where k belongs is selected as a parent. The operation is repeated until all the necessary parents have been selected.

2.1.4 Crossover

The crossover operator consists in creating offspring chromosomes from a set of parent chromosomes selected beforehand by taking alternatively some of their genes. This allows to transfer the characteristics of the parents to the offspring. The two crossover methods that we use are presented in the next 2 sub-paragraphs:

Single-point crossover With the single-point crossover method, one crossover point is chosen randomly. The new chromosome (offspring) is then composed of the genes of the first parent until the crossover point and the remaining genes are copied from the second parent as depicted in Figure 4.11.

To present the operation formally, let χ_k^g and χ_l^g be two distinct chromosomes selected to perform a crossover, their offspring chromosome, denoted χ_{off}^{g+1} is defined as follows:

1. a random index $randIdx$ is drawn with a uniform law
2. $\forall 0 \leq j < randIdx : \gamma_{off,j}^{g+1} = \gamma_{k,j}^g$: the $randIdx$ first genes are copied from parent k to the offspring
3. $\forall randIdx \leq j < \Gamma : \gamma_{off,j}^{g+1} = \gamma_{l,j}^g$: the remaining genes are copied from the second parent

Multi-point crossover The single-point crossover can be generalized to select $k > 0$ points: the genes of the offspring take alternatively the values of their parents between the selected crossover points.

Algorithm 7 The *Fitness Proportionate Selection* algorithm

Require: Ψ^g ; $nb\_parents\_mating$
$wheel = []$
$selected\_parents = []$
$last\_prob = 0$
$j = 0$
while $j < X$ **do**
    $p_j = compute\_prob(\chi_j^g, \Psi^g)$ ▷ See formula below the algorithm
    $wheel.push(p_j + last\_prob)$
    $last\_prob = p_j$
    $j = j + 1$
end while
while $size(selected\_parents) < nb\_parents\_mating$ **do**
    $k = random(0, 1)$
    $p = 0$
    while $wheel[p] < k$ **do** ▷ Search the matching chromosome index
        $p = p + 1$
    end while
    $parent = \chi_{p-1}^g$
    $selected\_parents.push(parent)$
end while
return $selected\_parents$

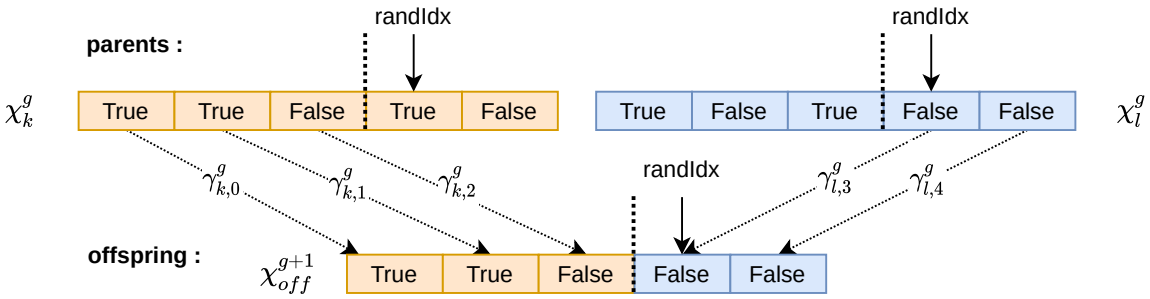


Figure 4.11: The steps of a Genetic Algorithm

2.1.5 Mutation

Mutation is applied to offspring chromosomes (produced by crossover operations) to randomly change the value of a gene and boost the exploration of the genetic algorithm. In the following, an *adaptive* strategy is used in which the chromosomes with a low fitness have more chance to see their genes mutate than the others. In practice, we separate the chromosomes in two groups, one for chromosomes with a fitness value above the average of the current generation and the others in the second group. Then, the chromosomes of the group with the highest fitness mutate with a lower probability than the chromosomes of the other group.

2.1.6 Termination

It is possible to stop the computation using a convergence criterion. For example, the GA can be stopped if:

- The highest fitness score in the population is constant (or does not vary enough) for a number of generations.
- There are some identical solutions in the population.

However, a known limitation of the GA is its tendency to converge towards local optima. Indeed, as for many meta-heuristics, a trade-off must be found between exploration, i.e. covering a maximum of the solution space, and exploitation, i.e. remaining in the neighborhood of promising solutions and find the best. Exploitation tends to overcome exploration when mutation and crossover operators do not introduce enough diversity. Therefore, relying on a convergence criterion for problems with a large solution space is not advised because there are high chances that the algorithm falls in a local optimum. Instead, we set a fixed number of iterations that is determined empirically. Another advantage is that it is easier to control the computation time.

2.2 Traces-based GA

Now that we have covered the basics of GA, we introduce our first GA for multi-phase profile creation. This GA builds a profile and selects synchronizations based on the set of memory instructions selected during the static analysis of the task, i.e. the instructions that compose the execution traces of the task.

2.2.1 Chromosome encoding

A chromosome is a list of boolean genes representing all the nodes from all the traces of the task. The nodes whose corresponding gene is True are used to create phases and select synchronizations. The solution space, i.e. the possible combinations of phases and synchronizations that can be built from the nodes in the traces, is potentially large. Therefore, 3 versions to decode the chromosomes, i.e. to convert them into profiles, are proposed with different degrees of assistance from heuristics:

- Version *all*: the phases and the synchronizations directly define the profile, a memory instruction whose gene is True is synchronized at its worst-case date and all synchronization dates are used to create phases.
- Version *allOptim*: the phases and synchronizations are created in the same way as the *all* version, but the optimization presented in Section 1.2.4 (Algorithm 5) is used to remove redundant synchronizations.
- Version *onlyPhases*: the chromosome is only used to create the phases and the synchronizations are selected using the systematic selection (Algorithm 2), then Algorithm 5 is applied to remove redundant synchronizations.

The GA takes as input the minimum duration of a phase δ, so for any decoding version, if the duration between two synchronization dates is inferior to this value, then only one phase is created instead of two. Once the phases and synchronizations have been set, the worst-case number of accesses in each phase is computed and the optimization pass is applied to create empty phases by splitting existing phases (see Section 1.3.3).

The *onlyPhases* version respects all the synchronization selection rules because it uses the systematic selection algorithm.

The two other versions use the genes values to select synchronization so they do not follow rule 3 that by itself constitutes a selection method. However, the other rules are respected:

- Rule 1: the genes corresponding to synchronized nodes selected prior to the design of the profile are initialized to True and cannot be modified so they must be synchronized in the represented profile.
- Rule 2: the genes corresponding to nodes inside loops are initialized to False and cannot be modified so they cannot be synchronized in the represented profile.
- Rules 4 and 5: the nodes are synchronized using Algorithm 4 so that equivalent nodes are synchronized together and at the same date (i.e. the worst-case date of the instruction they represent).

2.2.2 Initialization of the population

A good initial population of chromosomes can reduce the time needed to reach good solutions and it influences the quality of the final results. We need enough synchronizations to limit the access over-approximation but their number must remain limited. A trade-off can be found during the initialization by setting the probability of choosing True or False for the genes value of the initial chromosomes. We empirically set the probability of choosing True to 0.2 and of choosing False to 0.8.

2.2.3 Fitness

The fitness function relies on three criteria:

- The **over-approximation of accesses** in the profile. It is important to limit the over-approximation in order to maintain the multi-phase model performance. The fitness score associated to this criterion is computed from the access over-approximation rate Δ as defined in Property 4 of Chapter 3:
$$fit\_ovApp = \frac{1}{e^{\Delta}} \tag{4.3}$$
The range of variation of the access over-approximation rate is very large. The first population generally contains chromosomes where it is several hundreds of % but these solutions can be subsequently improved through the generations. This formula makes sure that the score remains between 0 and 1, and the exponential greatly rewards the improvements regarding this criterion.
- The **number of synchronizations** nb_sync. As said before, the number of synchronizations must be kept as low as possible. The fitness score associated to the number of synchronizations is computed using the number of memory instructions in the traces set, which is also the number of genes Γ:
$$fit\_sync = (\Gamma - nb\_sync)/\Gamma \tag{4.4}$$
Once again, this score lies between 0 and 1 but in practice it never reaches 1. As only genes can be synchronized, fit_sync is positive.

- The **proportion of time guaranteed without memory access**. The more time without accesses in the profile, the more potential to avoid contentions in the schedule. The fitness score corresponding to this criterion is computed as:

$$fit\_empty = empty\_dur/WCET(\tau^i) \tag{4.5}$$

with *empty_dur* the total duration without accesses in the profile and $WCET(\tau^i)$ the WCET of the task.

The global fitness of the chromosome is then:

$$fitness = a \times fit\_ovApp + b \times fit\_sync + c \times fit\_empty \tag{4.6}$$

with $a \geq 0$, $b \geq 0$ and $c \geq 0$ defining the weight of each criterion.

2.2.4 Balancing the three criteria

As mentioned above, the range of variation of the over-approximation is very large so the GA takes a long time before finding solutions that are able to limit this criterion compared to the others. Therefore, another 2-steps strategy is adopted. Each step is conducted for one half of the number of generations:

1. $a = 1$ and $b = c = 0$: only the over-approximation criterion is optimized. This first phase allows to stabilize the GA with a population of solutions where the over-approximation of accesses is already at a reasonable level.

2. a, b and c take the value chosen by the user: in this second phase, the two other criteria are also part of the optimization problem so that the algorithm can exploit the solutions from the first phase. As the over-approximation has already received attention, a can be set to a low value compared to b and c so that the GA focuses on the number of synchronizations and the duration without accesses.

2.2.5 Selection

The parents selection is performed using the *Fitness Proportionate Selection* method. Any chromosome can be parent, which contributes to the diversity of the population, but in the long term, only the best solutions remain. The *steady-state* method is used to select and keep the best individuals of the population at the end of each iteration.

2.3 Phases-based GA

Here, as opposed to the *traces-based GA*, the purpose of the GA is not to build a profile from scratch but rather to improve the characteristics of an existing one by merging some phases. The objective is to reduce the over-approximation of accesses or reduce the number of synchronizations that are required to implement the profile without degrading the worst-case interference scenario computed later by the interference analysis. As the over-approximation grows, the number of contentions grows artificially compared to the single-phase equivalent.

2.3.1 Merging phases to lower the over-approximation of accesses

The over-approximation of accesses can be limited by a clever selection of the synchronizations. However, other factors are at play. Figure 4.12 illustrates in red a situation where an optimal choice of synchronization (adding new synchronizations cannot reduce the count of accesses in any phase) cannot eliminate the over-approximation of accesses. The single-phase model would account for 13 accesses corresponding to the execution of trace t_0^2, but the red profile has a total of 15 accesses. This

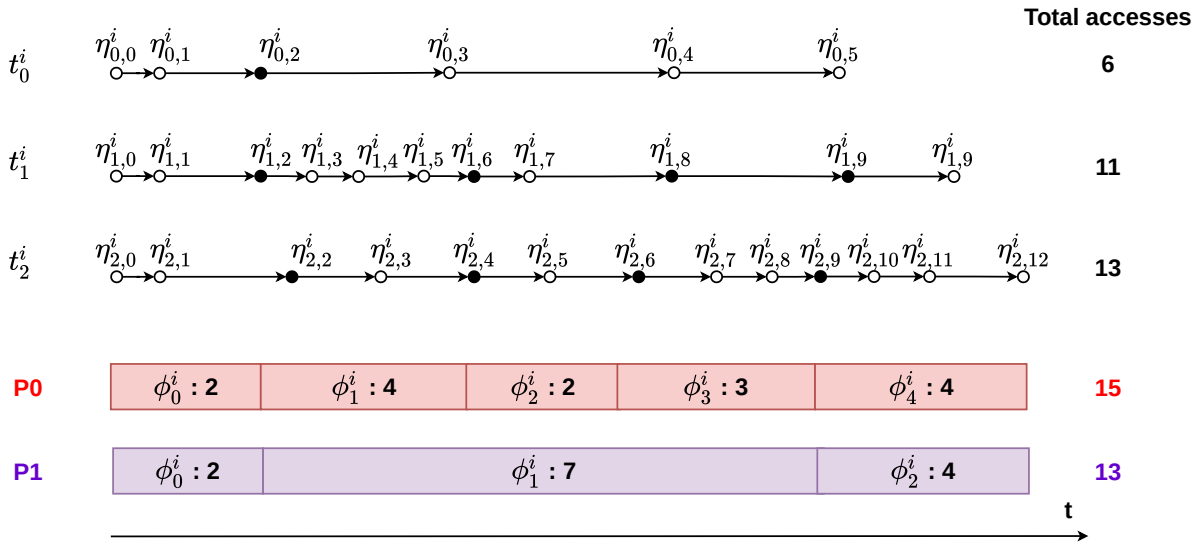


Figure 4.12: Three traces and a profile. Synchronized nodes are in black and the value in each phase gives its worst-case number of accesses.

over-approximation of accesses cannot be eliminated by selecting more synchronizations. Therefore, it comes from the way the profile is divided: we see that t_1^i is performing most of its accesses during the first phases while for t_2^i there are more at the end of the profile. It is possible to identify the sources of over-approximation by separating the traces between two sets:

1. The set of traces that perform the most accesses in the whole task, named *global max traces* that is invariant for the profile.
2. The set of traces that perform the most accesses in a particular phase, named *local max traces* which varies throughout the profile.

The number of accesses performed by the global max traces during the task execution is equal to the maximum number of accesses in the single-phase representation. Then, the over-approximation of accesses appears when at least one global max trace is not among the local max traces in a given phase (i.e. it performs less accesses than another trace in this phase). Back to Figure 4.12, we see that the global max trace is t_2^i with 13 accesses. The multi-phase profile P0 in red accounts for 15 accesses because t_1^i locally performs 2 accesses more than t_2^i in ϕ_1^i, so t_1^i is the local max trace in ϕ_1^i. A solution to eliminate the access over-approximation is to merge phases ϕ_1^i, ϕ_2^i and ϕ_3^i to produce the P1 profile in purple. Profile P1 has no access over-approximation because the global max trace (t_2^i) is always among the local max traces.

2.3.2 Initial profile

As mentioned above, this GA is not able to build a profile from scratch. Instead, it uses an initial profile and searches how to fuse the phases to reshape the profile and obtain a better solution. Since this GA can only merge phases, the initial profile has the highest number of phases that a solution can contain. Therefore, we must ensure that the initial profile has enough phases so that the GA can explore a large solution space. Moreover, the quality of the initial profile influences the quality of the GA results.

In the following, we denote $\mathbb{P}_{init}^i$, with Φ_{init}^i phases, the initial profile of task τ^i.

2.3.3 Chromosome encoding

A chromosome, or solution, $\chi_k^g = \{\gamma_{k,j}^g | 0 \leq j < \Gamma\}$ represents a possible profile of the task τ^i. Each gene $\gamma_{k,j}^g$ is a boolean that is true if the corresponding phase $\phi_j^i \in \mathbb{P}_{init}^i$ is fused with the next

Algorithm 8 Function decoding a chromosome to return its corresponding multi-phase profile

```
1:  procedure ChromToProf(χ^g_k, ℙ^i_init, 𝕋^i)
2:      ℙ^i_k = newProfile()
3:      j_init = 0                                  ▷ index of the current phase in ℙ^i_init
4:      while j_init < Φ^i_init do
5:          toMerge = [getPhase(ℙ^i_init, j_init)]
6:          while γ^g_{k,j_init} do
7:              toMerge.push(getPhase(ℙ^i_init, j_init))
8:              j_init = j_init + 1
9:          end while
10:         newPhase = merge(toMerge, 𝕋^i)
11:         addPhase(ℙ^i_k, newPhase)
12:         j_init = j_init + 1
13:     end while
14:     return ℙ^i_k
15: end procedure
```

phase ϕ^i_{j+1} in the encoded profile. Note that because there are as many genes as phases in the initial profile we have $\Gamma = \Phi^i_{init}$. Figure 4.13 represents 2 chromosomes and their corresponding profiles. For the chromosome on the left, the first three phases of the initial profile are merged and it is the two first and the two last ones for the other chromosome.

Algorithm 8 presents the $ChromToProf$ function that decodes a chromosome χ^g_k to obtain its profile $\mathbb{P}^i_k$. Firstly, the new profile $\mathbb{P}^i_k$ is initialized without phases with function $newProfile$. j_{init} is used to denote the index of the current phase in $\mathbb{P}^i_{init}$ and of its corresponding gene in χ^g_k. We use function $getPhase(\mathbb{P}^p, idx)$ to retrieve the idxth phase of $\mathbb{P}^p$. The first while at line 4 uses j_{init} to iterate through the phases of the initial profile. A list of phases to merge together named $toMerge$ is initialized with the current phase as only element. The second $while$ at line 6 detects if there is a sequence of phases to merge from the current phase (i.e. a sequence of genes whose value is True) and store this sequence in $toMerge$. Then, function $merge$ returns a phase whose date is equal to the first phase in $toMerge$ and whose duration is $\sum_{\phi^i_k \in toMerge} \phi^i_k.dur$. The number of accesses in the new phase is computed using $\mathbb{T}^i$. Indeed, a simple addition $\sum_{\phi^i_k \in toMerge} \phi^i_k.m$ is too conservative because if an access is accounted for in several of the merged phases (creating over-approximation) then it is accounted for several times in the new phase.

2.3.4 Fitness

In our problem, the fitness function must find a trade-off between the optimization of the profile and the preservation of its good properties. This trade-off is expressed using 4 distinct criteria, two of them being dedicated to the optimization and the two remaining aiming at preserving the good

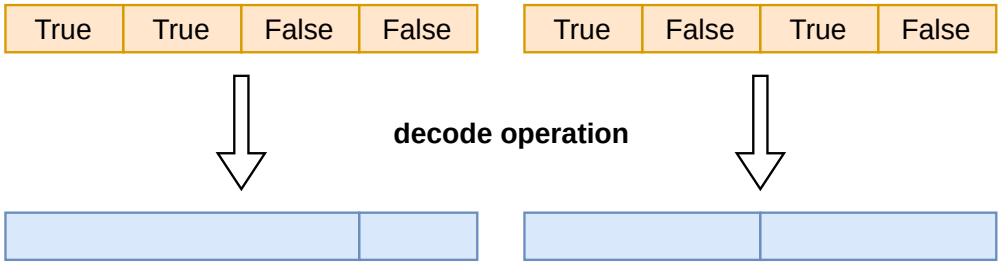


Figure 4.13: Decoding operation for two chromosomes

properties of the profile. First, the two optimization criteria are the same as for the Traces-based GA:

- **the over-approximation of accesses** as defined by Property 4: the higher the over-approximation value, the more potential contentions compared to the single-phase model.
- **the number of synchronizations used in the profile**: we want to reduce their number.

Additionally, preserving the good properties of the initial profile is challenging firstly because identifying these properties is difficult and secondly because the combination of several good properties may actually not produce good results. Nonetheless, we propose two criteria to preserve the initial profile properties:

- **The proportion of time guaranteed without access**. As opposed to the traces-based GA, this criterion cannot be improved by this GA because it works on existing phases that it can only merge so it cannot create empty phases. However, we want to avoid the elimination of the empty phases present in the initial profile.
- **The variability of phase durations**. This variability has an impact on the efficiency of a profile during the interference analysis (an investigation is conducted later in Chapter 7). To measure this variability we use the Gini index that indicates the statistical dispersion of a distribution. This index is often used to measure inequalities in economics. It equals 0 when all the values in the distribution are equal (perfect parity) and goes up to 1 when only one value is not 0 (perfect inequality). As we could not determine an ideal Gini value for a given profile during preliminary experiments, we use the Gini value of the initial profile as a reference for the resulting profile. In other words, we try to remain as close as possible to the value of the initial profile.

The fitness scores of each criterion are:

- Over-approximation: $fit\_ovApp = 1 - \Delta$ with Δ the access over-approximation rate as defined in Property 4 of Chapter 3. We do not use an exponential because, contrary to the Traces-based GA, the initial method is supposed to have controlled the value so we consider that $\Delta \leq 1$.
- Number of synchronizations: $fit\_sync = (nb\_sync\_init - nb\_sync)/nb\_sync\_init$ with nb_sync_init and nb_sync the number of synchronizations respectively in the initial and the current profile.
- Proportion of empty duration: $fit\_empty = (empty\_init - empty)/empty\_init$ with $empty\_init$ and $empty$ the proportion of duration without accesses respectively in the initial and the current profile.
- Variability of the durations: $fit\_var = 1 - |Gini\_init - Gini|$ with $Gini\_init$ and $Gini$ the Gini index value respectively in the initial and the current profile.

The overall fitness equation is given by the weighted sum of the criteria fitness scores:

$$fitness = a \times fit\_ovApp + b \times fit\_sync + c \times fit\_var + d \times fit\_empty \tag{4.7}$$

2.3.5 Tuning the weight of each criterion

Tuning the weights of the criteria allows to set the trade-off between optimization and preservation. This trade-off is determined empirically because it is specific to each tasks system (according to the characteristics of the code), and to the initial methods used. The idea is to set the highest weight to the optimization criterion that has the most improvement potential and setting the

preservation weights according to the proportion of change that is authorized to achieve the improvement. For example, if an initial profile has a high over-approximation value but also many empty phases, it is necessary to set the highest weight for a in order to lower the over-approximation level but d must also be significant so that as many empty phases as possible are kept.

2.3.6 Crossover and Selection

This GA uses the single-point crossover method. The multi-point crossover is not more efficient according to preliminary results maybe because the solution space is less important than for the Traces-based GA. Also, it uses the same selection strategy as in the Traces-based GA, i.e. the *Fitness Proportionate Selection* to select parents and the *Steady State Selection* to keep chromosomes for the next generation.

3 Conclusion

Using the formalization presented in Chapter 3, we are able to compute the worst-case number of accesses in any multi-phase profile given the traces representation of a task. This opens up a new field of possibilities to divide a task, in addition to the TIPs or StAMP techniques. However, the design of a multi-phase profile must be performed with the scheduling and interference analyses in mind so that the profile is efficient to reduce the makespan of the schedule in the presence of interference. We proposed a heuristic method based on Kernel Density Estimation (KDE) that is applied to the distribution of the worst-case dates of accesses so that we can create phases according to the intervals where there are the least or the most accesses locally. The advantage of relying on the dates of accesses is that it is easier to pack the accesses in phases and to select efficient synchronizations (i.e. common to a majority of traces) so that the over-approximation is easier to limit.

The selection of synchronizations is crucial to limit the over-approximation of accesses. A heuristic is proposed to select them in a systematic manner for each trace so that no node can span over several phases. However, although this heuristic is efficient to tackle the access over-approximation, it selects redundant synchronizations that are synchronizations whose presence does not modify the count of accesses in the phases. Therefore, we also presented an optimization that can be applied after the heuristic to eliminate such synchronizations and reduce the impact of the multi-phase model implementation in the code. This impact depends on the synchronization mechanism used. For example, taking into account the cost of retrieving the synchronization dates varies according to whether they are stored in a local or a remote memory. In the first case, as we assumed in this chapter, the dates may be loaded in a local memory or a scratchpad so that synchronizations may only add a few cycles to the execution time of the task. However, in the second case, a new cache analysis would be required to account for the additional accesses, and the number of accesses in the phases would also be modified. Some profile corrections and improvements are also proposed that can be applied to any existing profile. They correct the dates of phases so that the memory latency of accesses is taken into account, and realign the dates of phases on the date of their accesses so that the clusters of accesses are better enclosed in phases and the duration of potential empty phases is increased. These empty phases are interesting because they cannot produce contentions so they are particularly efficient when scheduled in parallel with phases that have a lot of memory accesses.

The different aspects discussed when designing or optimizing profiles are difficult to take into account at the same time. Therefore, we propose to adopt Genetic Algorithms (GA), a meta-heuristic that can explore more solutions and combine multiple criteria in its fitness function. In a first version, we propose to design a profile from the traces of a task as for the KDE-based method. This Traces-based GA must find a trade-off between the number of synchronizations, the proportion of time without accesses and the over-approximation of accesses in the profile. However,

synchronizations are not always sufficient to eliminate the over-approximation. Another source of access over-approximation is inherent to the profile itself because accesses can occur in different parts of the profile depending on the trace. For instance, if two traces each have a cluster with N accesses that are respectively the maximum count of accesses in two consecutive phases, then it is interesting to merge the two phases so that the count of accesses locally is not 2N but just N. Therefore, another GA is used to optimize a profile by fusing some phases together so that the final profile has less access over-approximation and less synchronizations, without degrading too much the properties of the initial profile. With this second GA version, one can realize that it is complex to identify good properties for a multi-phase profile. The preservation of the initial properties is ensured by a measure of the dispersion of the duration of the phases and by the proportion of time without access. However, the efficiency of these metrics to represent the good properties of a profile is debatable. In preliminary experiments, some other metrics were evaluated but it is complex to determine to what extent they influence the results of a scheduling and interference analysis.

One solution to take into account the interdependence between the shape of the profiles and the results of the interference analysis is to iterate over a cycle composed of profile design, scheduling and interference analysis steps so that the interference analysis results are used as a feedback to reshape the profiles and improve the results of the subsequent interference analysis. However, an extensive analysis of which characteristics actually influence the results of the interference analysis is required so that a precise interpretation of the feedback can be performed, to guide the next design step. Such an analysis has been attempted but the results were highly dependent on the type of tasks and the way they were scheduled. Another solution would be to employ learning techniques so that a method could determine a near-optimal shape for each task without iterating. The training would also require the identification of the parameters influencing the interference analysis results but the system could learn by itself how to interpret and tune these parameters to improve the shape of a profile.

The next chapter compares the different design techniques presented in this chapter by assessing their efficiency to reduce interference when a basic scheduling algorithm is used without optimization.

Chapter 5

Comparative study: designing a task profile

Contents

The previous chapter presented three new approaches to build a multi-phase profile. This chapter confronts these approaches to the TIPs approach [12] from the state of the art, by applying them on several case studies. Two aspects are studied: first the statistics of the profiles obtained for each task system and second the gain obtained when scheduling the task system with the multi-phase profiles compared to when using the single-phase model of tasks. The first aspect focuses mainly on the number of synchronizations to inject in the code and the resulting over-approximation, which are related to the number of phases. The second aspect is assessed by scheduling the tasks with a basic algorithm and performing an interference analysis, and then looking at the makespan of the schedule in the presence of interference.

A good design method is both easy to implement and efficient to reduce the number of contentions in the schedule. Intuitively, such a method creates enough phases to represent the execution and the possible contentions with precision but with a minimum number of synchronizations and a low access over-approximation. The solution to this trade-off is not unique because it depends on the considered task system and architecture. Therefore, we compare the design methods with two

case studies so that the reader can figure out which method is more adapted to a given configuration (architecture and task system).

The first section presents the two case studies and the remainder of the chapter is organized in two parts to separate the two aspects that we consider in our comparison.

1 Case studies

The first case study is a benchmark that is used in the WCET research community and the second is a synthetic system generated specifically for the comparison (the methodology is described thereafter).

The traces representation of a task is required to compute the worst-case number of accesses in each phase. In this thesis, we derive the traces representation of a task from a static analysis of the code using the Time Interest Points (TIPs) methodology described in [53] and [12] to build a TIPsGraph, a light version of a CFG that essentially represents the instruction performing memory accesses (see Chapter 2 section 3 for more details).

In the following, the dependencies between the tasks of T are specified using a DAG $G = (T, E)$ in which vertices are the tasks of T and each edge $e_{i,j} \in E$ between τ^i and τ^j indicates that τ^i must be completed before τ^j can start.

The two case studies are presented successively thereafter.

1.1 Rosace

Rosace is a flight controller application presented in [82]. We do not include the environment simulation tasks in the experiments because there is a disproportion between their WCET and the WCET of the control and command tasks.

The tasks have been analyzed with OTAWA [15] to extract their CFG and perform a cache analysis. We considered a target hardware architecture composed of an ARM-based multi-core processor in which each core features a L1 LRU data cache, and an instruction scratchpad which holds the totality of the code needed by the core to execute. We considered a memory latency of 50 cycles for non-cached accesses.

Rosace is a multi-periodic application, so we convert the task system into a DAG of single-period tasks over one hyperperiod following the methodology of [83]. The resulting DAG is composed of 77 tasks.

1.2 Synthetic tasks system

The static analysis of a task system requires a consequent effort to prepare the code (e.g. adding flow-fact annotations, splitting or modifying the code to make it analyzable or speed up the analysis). In addition to an existing benchmark, the evaluation of the methods has been conducted from a set of 20 synthetic tasks whose traces representation are derived from 20 synthetic TIPsGraphs. The synthetic TIPsGraph generation not only allows to skip the code analysis step, but also offers control on the output traces. We used this control to build traces that are more complex to analyze than Rosace so that it is easier to observe differences between the evaluated methods to design a profile.

The generation begins from a single initial node (the program entry point), from which one of the following structures is created according to their respective probabilities p_{seq}, p_{loop}, and p_{if}:

1. A *sequence*: a simple sequence of 5 to 10 nodes with exactly one entry and one exit point
2. A *loop*: a loop with 1 to 4 inner nodes, 3 to 6 iterations and an exit from the header or from the last inner node in the loop

3. An *if-then-else*: creates new leaf nodes, their number if chosen randomly between 2 and 3 (only one for the other structures)

The probabilities to choose the structures have been set to $p_{seq} = 0.7$, $p_{loop} = 0.2$ and $p_{if} = 0.1$ and the number of nodes to create in a structure is picked randomly, as is the local WCETs between the nodes that are chosen in the interval $[50, 300]$. The process is repeated for each new leaf node until the cumulative WCET from the entry point to the new leaf exceeds a duration that has been picked randomly in the interval $[500, 10000]$ for each task.

The DAG of the synthetic tasks has been generated by expanding vertices without successors either with *parallel sub-graphs* (as in a fork operation) with a probability of 0.7 or a *series* of new vertices with a probability of 0.3 until the number of nodes matches the number of tasks. We force the first expansion to be parallel in order to enable the use of multiple cores since the beginning and at some points, nodes have the same successor, e.g. in a join following a fork.

2 Generation of the profiles

This section explains the configuration of each method to build the profiles from traces. Each method has been applied with $\delta \in [1000, 500, 200, 100, 50]$ cycles (the minimum duration of a phase) to give different profiles from one method. The methods we used are those introduced in the previous chapter along with the TIPs method from the state-of-the-art [12], described in more details in Chapter 2 section 3.

2.1 Generation with the traces-based GA method

To set the parameters of the GA, we first used values in the same order of magnitude as previous works and then progressively tuned them during several trials:

- Number of chromosomes in the population $N_{chr} = 25$. This allows to decode and evaluate the chromosomes of a population with a reasonable computation time while representing a variety of solutions.
- Mutation probabilities $p_{low\_fitness} = 0.2$ and $p_{high\_fitness} = 0.05$. The $p_{high\_fitness}$ value of 0.05 is often used in other works without adaptive mutation strategy, but 0.2 is a high value for $p_{low\_fitness}$ because we want to quickly explore other solutions using low fitness chromosomes.
- Number of parents to keep in each generation $keep\_parents = 15$. Hence, a majority of solutions remains from one generation to another to bring stability.
- Number of parents mating at each generation $n_{mating} = 4$.

Then, in order to set the number of generations of the GA, we experimented several values and observed the variation of the criterion that can vary the most which is the access over-approximation. The tests were conducted with the *all* version that has the largest exploration space and on the Synthetic benchmark. Figure 5.1 shows the results of the experiment. The trend-line indicates that the access over-approximation stabilizes after 2000 generations. Therefore, the number of generations was set to 5000 so that the access over-approximation criterion is stabilized within the 2500 first iterations alone.

2.2 Optimization with the phases-based GA method

We apply the phases-based GA on the profiles produced using the KDE-based and the TIPs heuristics separately for $\delta = \{1000, 500, 200, 100, 50\}$ cycles. They are called respectively *tips+ga*

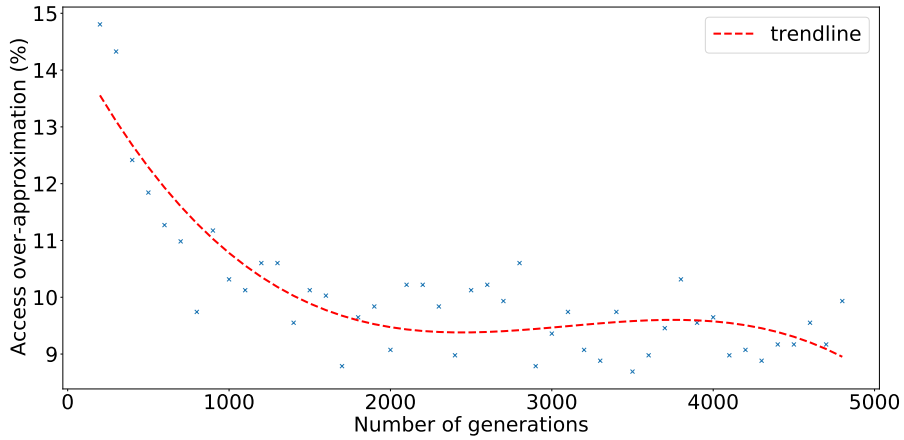


Figure 5.1: Average access over-approximation for different number of generations (traces-based GA version *all* applied on the synthetic benchmark).

and *kde+ga* in the following. The weights of the fitness function are different between Rosace and the synthetic tasks, they have been determined empirically.

For Rosace, the initial profiles have a very low over-approximation level and include many empty phases with a low δ. This is due to the fact that the traces are similar (the same instruction sequences are found in several traces with only a few differences) and, as a result of synchronizations, the worst-case access dates tend to occur at the same date, which favors the presence of empty phases. Hence, reducing the over-approximation is not a priority so we can focus on reducing the required synchronizations while keeping a maximum of the empty phases. The weights are : 1 for the over-approximation, 6 for the number of synchronizations, 1 for the variability of durations and 3 for the empty duration.

Regarding the Synthetic benchmark, the access over-approximation is more important so we do not prioritize an optimization criterion over the other. Likewise, the two preservation criteria have the same weight. However, the first goal of the GA is the optimization of the profile so we chose to set a higher weight for the optimization criteria: 2 for the over-approximation and the number of synchronizations, 1 for the variability of durations and the empty duration.

The other parameters of the GA are common for each tasks system and have been set empirically in order that the GA has enough time to explore and exploit the solutions: there are 200 generations, with 15 chromosomes per population, 10 chromosomes are transferred from a generation to another and 4 are mating during crossover.

3 Statistics of the generated profiles

In this section, we compare the statistics of the generated profiles with four metrics:

- Number of phases: we do not know how many phases a profile must have in order to outperform the single-phase model. However, we know that generally, the more phase there are, the more difficult it is to tackle the access over-approximation and the more synchronizations are required.
- Number of synchronizations: they are essential to limit the access over-approximation but we want to limit their number to reduce the impact on the code.
- Number of synchronizations per phase: it puts into perspective the two previous metrics that are correlated.

- Access over-approximation rate: we use a formula that is more adapted to a system-wise study than the formula used in Property 4:

$$nbAccMulti = \sum_{\tau^i \in T} \Big(\sum_{\phi^i_j \in \mathbb{P}^i} \phi^i_j.m \Big) \tag{5.1}$$

$$nbAccSingle = \sum_{\tau^i \in T} \Big(\max_{0 \leq j < T^i} \Big(\sum_{0 \leq k < N^i_j} \eta^i_{j,k}.m \Big) \Big) \tag{5.2}$$

$$ovApp = ((nbAccMulti - nbAccSingle)/nbAccSingle)) \times 100 \tag{5.3}$$

  With this formula, we have a better idea of how many additional accesses are present in the multi-phase system compared to the single-phase one than if we simply average the values of all tasks. For example, consider two tasks performing up to 10 and 100 accesses respectively and whose multi-phase profile accounts for 12 and 110 accesses. Then, the access over-approximation rates are respectively 20% and 10% with an average value of 15%, but with formula 5.3, the over-approximation is 10,91%. It gives less weight to the over-approximation of the task with 12 accesses as it represents a lower amount of accesses than the other task, thus the system-wise over-approximation is more representative.

The metrics are interdependent so a fair comparison of the methods requires to take all of them into account. First, they are presented individually and then a summary section synthesizes the results.

3.1 Number of phases

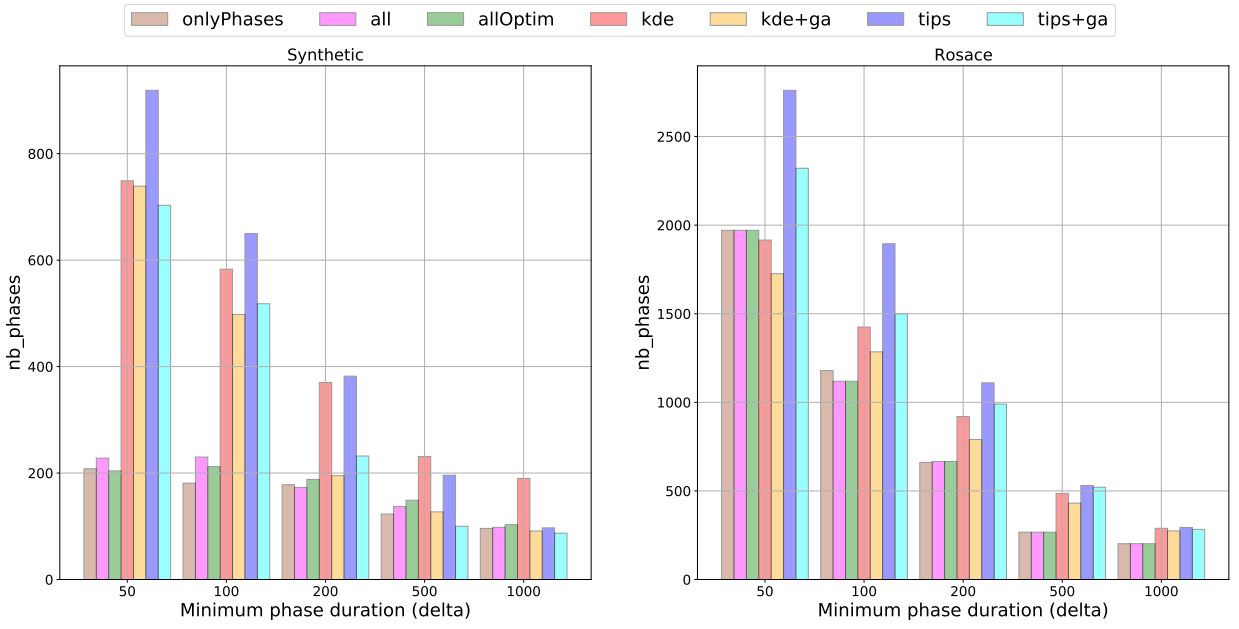


Figure 5.2: Number of phases in the profiles.

Figure 5.2 displays the number of phases in the profiles of the Synthetic benchmark and Rosace for each method. We observe that the three traces-based GA methods (see Chapter 4 Section 2.2.1 page 61) create approximately the same number of phases in both the Synthetic benchmark and Rosace profiles. For the Synthetic benchmark, the *tips* and *kde* methods always create the most phases and their number grows faster than the traces-based GA. The phases-based GA always reduces the number of phases and obtains a similar number as the traces-based GA methods when $\delta \geq 200$.

For Rosace, the *tips* creates the most phases and even when the phases-based GA is applied, *tips+ga* has more phases than *kde*. The traces-based GA versions have almost always the same number of phases, which is the lowest except when $\delta = 50$ cycles where *kde* has slightly less phases (along with *kde+ga*).

3.2 Number of synchronizations

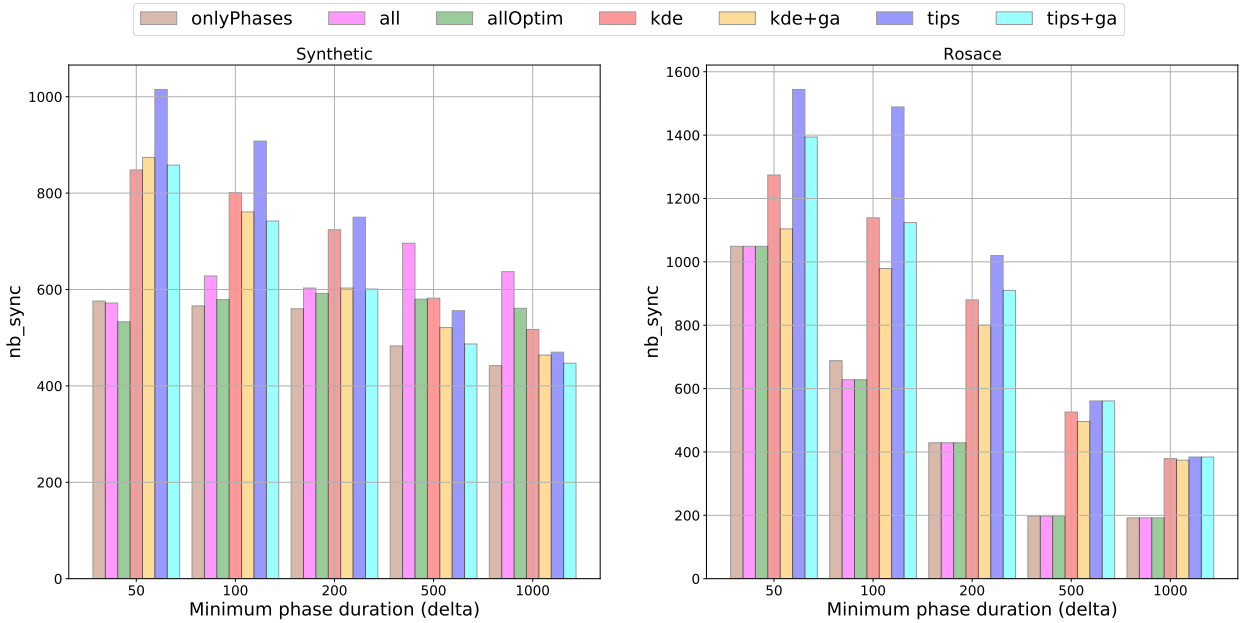


Figure 5.3: Number of synchronizations in the profiles.

The number of synchronizations is depicted in Figure 5.3. For the Synthetic benchmark, the traces-based GA versions that directly use the chromosomes to select synchronizations (*all* and *allOptim*) have more synchronizations than *onlyPhases* version when $\delta > 100$ cycles. Moreover, when $\delta > 200$ cycles, *tips* and *kde* select approximately the same number of synchronizations as the traces-based GA versions, but when $\delta \leq 200$ cycles they select more synchronizations and the difference increases as δ increases. This is due to the rapid increase of the number of phases for *tips* and *kde*. However, the gap is not as important as with the number of phases. The phases-based GA are efficient to reduce the number of synchronizations of the initial profiles: the reduction is between 4.89% (with $\delta = 1000$ cycles) and 19.87% (with $\delta = 200$ cycles) when it is applied on *tips* profiles, and between 7.92% (with $\delta = 1000$ cycles) and 32.71% (with $\delta = 200$ cycles) when it is applied on *kde* profiles.

For Rosace, as *tips* has the most phases it also requires the most synchronizations. The phases-based GA does not reduce the number of synchronizations of the *tips* profile when $\delta \geq 500$ cycles, but the reduction is of 24.51% with $\delta = 100$ cycles. When applied to *kde*, the reduction varies from 1.32% with $\delta = 1000$ cycles to 14.05% with $\delta = 100$ cycles. The traces-GA versions have the same number of synchronizations.

3.3 Number of synchronizations per phase

As we studied both the number of phases and the number of synchronizations, it is interesting to see the number of synchronizations per phase shown in Figure 5.4. First, the number of phases increases more rapidly than the number of synchronizations in the profiles for all the methods. This highlights that some synchronizations are more important than others so they are selected when

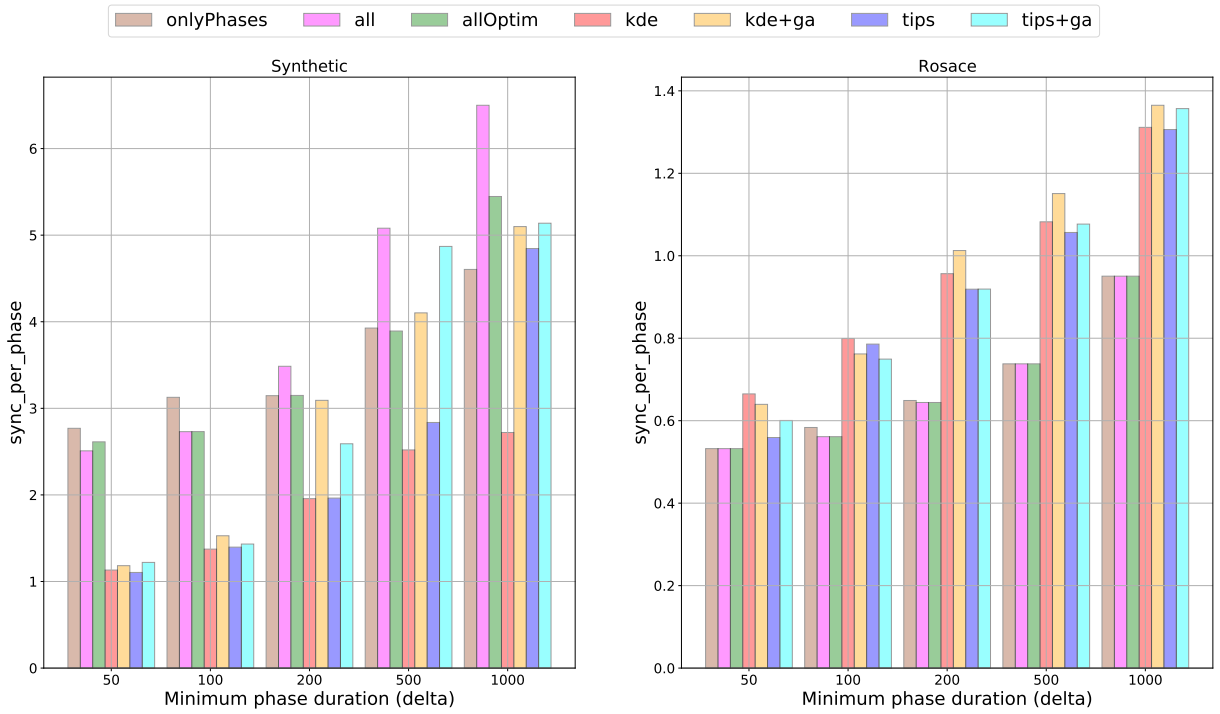


Figure 5.4: Number of synchronizations per phase in the profiles.

$\delta = 1000$ cycles and they remain efficient when δ is lower (they avoid the use of other redundant synchronizations).

With the Synthetic benchmark, *tips* and *kde* both have a better ratio than the others but this is because they create much more phases. This is highlighted by the phases-based GAs that always have a worse ratio than their initial profiles, even with less synchronizations. When comparing the traces-based GA versions, *all* is the least efficient when $\delta > 100$ cycles but then it has the same efficiency as the others. Indeed, when δ is high, many of the phases created by a chromosome are merged with others to respect the minimum duration. However, the synchronizations associated to the phases merged remain because the optimization to remove redundant synchronizations is not applied.

With Rosace, there is no difference between the traces-based GA versions which all have the lowest number of synchronizations per phase. We observe that sometimes the phases-based GA have a lower value than their initial profile so they removed more synchronizations than they merged phases.

3.4 Access over-approximation

Finally, Figure 5.5 presents the access over-approximation rate for the two case studies. For Rosace, the over-approximation is always below 3.5% and even at 0% for all the methods when $\delta \geq 500$ cycles. We see that the phases-based GA reduces the over-approximation of the initial profiles in most cases but the difference is small because the initial values are already low.

Regarding the Synthetic benchmark, except for the *onlyPhases* method that is very stable, the access over-approximation increases when δ decreases. This shows that the selection of synchronizations cannot eliminate the over-approximation alone and that it is dependent on the shape of the profile (i.e. the phases) created beforehand. *onlyPhases* maintains a same level of access over-approximation whatever δ because it uses the synchronizations selection heuristic which allows it to find a subset of solutions with a low over-approximation faster than the other versions. Then, it can focus on the other fitness criteria. On the contrary, the other versions must explore

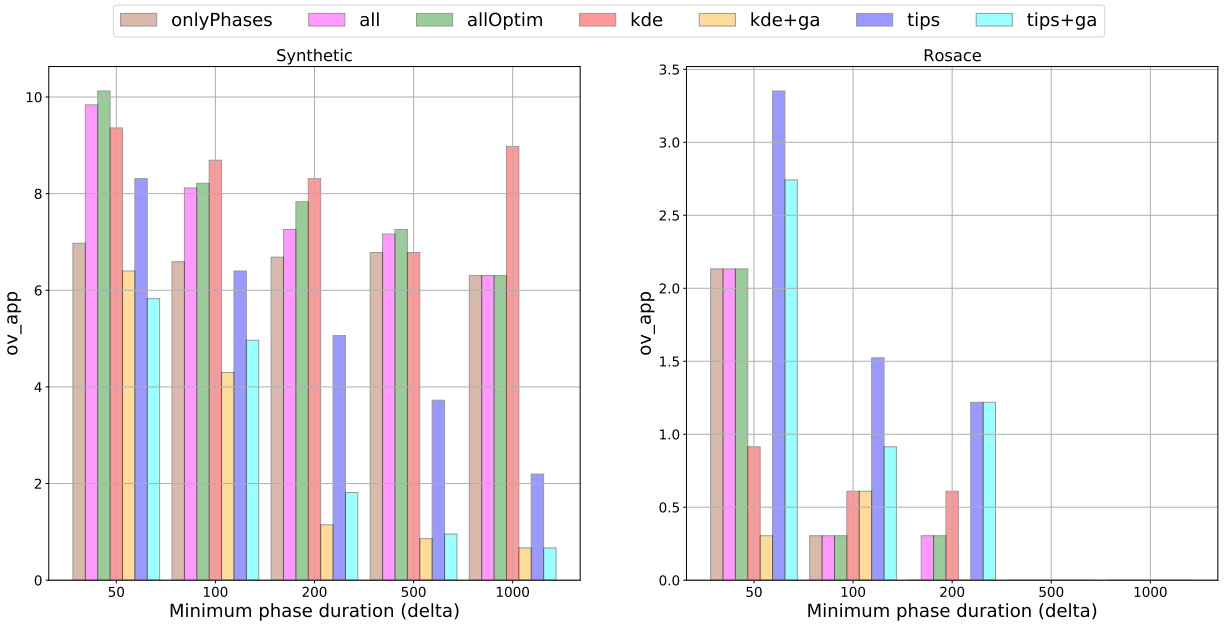


Figure 5.5: Access over-approximation rate in the profiles.

multiple solutions to select synchronizations on similar or identical profiles so they converge slower to the best trade-off. Compared to *kde*, *tips* better handles the access over-approximation when δ decreases because it already has a low value of around 2% with $\delta = 1000$. The phases-based GA is very efficient to reduce this over-approximation: the reduction is between 22.39% (with $\delta = 100$ cycles) and 74.36% (with $\delta = 500$ cycles) when it is applied on *tips* profiles, and between 31.00% (with $\delta = 50$) and 88.89% (with $\delta = 1000$ cycles, when the initial profile already has a low value) when it is applied on *kde* profiles.

3.5 Summary

Throughout the study of the four metrics, we observed that *tips* and *kde* create generally more phases than if we use one of the GA versions. As a consequence, they always require more synchronizations to tackle the access over-approximation than *onlyPhases* that uses the same algorithms to select and optimize the synchronizations. Moreover, with the Synthetic benchmark, the high number of phases also causes more access over-approximation inherent to the phases themselves, that cannot be eliminated by selecting more synchronizations. The *onlyPhases* GA version seems to have the best trade-off between the criteria because it has in average the less synchronizations and the access over-approximation is stable. In the following, we only present this version of the GA in the results.

4 Efficiency regarding the interference analysis

In this section, the profiles obtained with each method are scheduled using a simple heuristic. Then, we apply the interference analysis and compare the results with each profile generation method. We deduce which one best reduces the effect of interference in the analysis.

4.1 Scheduling and interference analysis

The scheduler selects a task from a list of ready tasks (i.e. without predecessor or for which all predecessors have already been scheduled), schedules it ASAP on the core having the minimum

makespan on its partial schedule, updates the list of ready tasks, and iterates until all tasks have been scheduled.

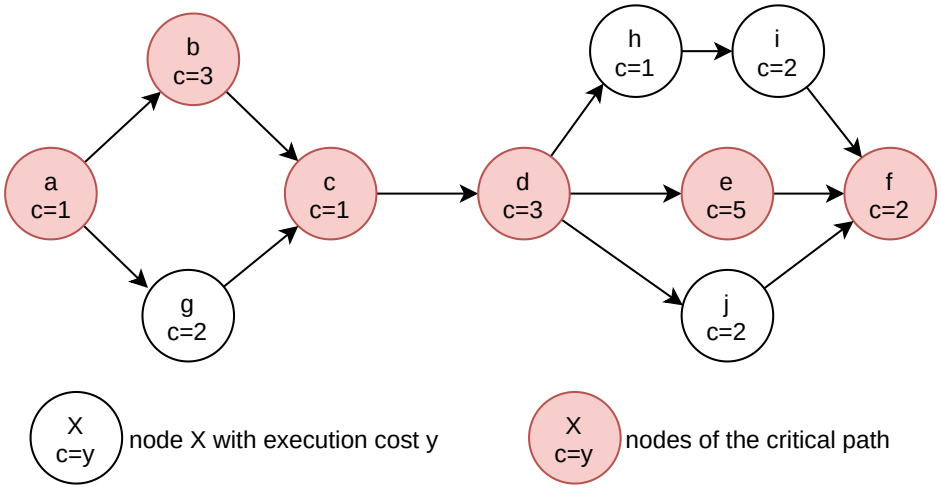


Figure 5.6: CPC priority assignment example.

Additionally, when the list of ready tasks is updated, we apply one of 3 sorting policies, thus determining priorities on the tasks:

- *releaseDate* gives the priority to the task whose predecessors finish the sooner
- *minBudget* (resp. *maxBudget*) gives the priority to the task with the minimum (resp. the maximum) budget where the budget of τ^i is $\sum_{\phi^i_j \in \mathbb{P}^i} \phi^i_j.dur$ (see Chapter 3 section 2.2)
- *CPC* (*Concurrent Provider and Consumer model*) assigns priorities to tasks following the method of [68]. In a nutshell, the following rules are applied: (1) the nodes belonging to the critical path of G have the highest priority, (2) nodes that can execute in parallel of the critical path (but can delay nodes in the critical path) are assigned a lower priority, and (3) if multiple (non-critical) parallel paths can delay the same node of the critical path, the nodes belonging to the longest path get a higher priority than the nodes of the other paths. An example is shown in Figure 5.6. The red nodes composing the critical path have the highest priority. Then, as g can delay the execution of c, it gets the second highest priority. Finally, f can be delayed by j (cost 2) and by h followed by i (cost 1+2=3), so h and i get a higher priority than j.

Once the schedule has been computed, an interference analysis is performed to account for the effects of memory contentions with the cost of a contention equal to the duration of a memory access in isolation (i.e. 50 cycles). A usual objective with DAG task systems is to minimize the makespan of the schedule. Therefore, for each test, we compute the gain between the makespan obtained with the multi-phase (*makespan_multi*) and single-phase (*makespan_single*) models in percent as follows:

$$gain = ((makespan\_single - makespan\_multi)/makespan\_single) * 100 \quad (5.4)$$

with *makespan_single* and *makespan_multi* the makespan of the schedule with the single-phase model and the multi-phase model respectively.

Each system under study is scheduled with the three different sorting policies and then we only retain the schedule that has the best gain value.

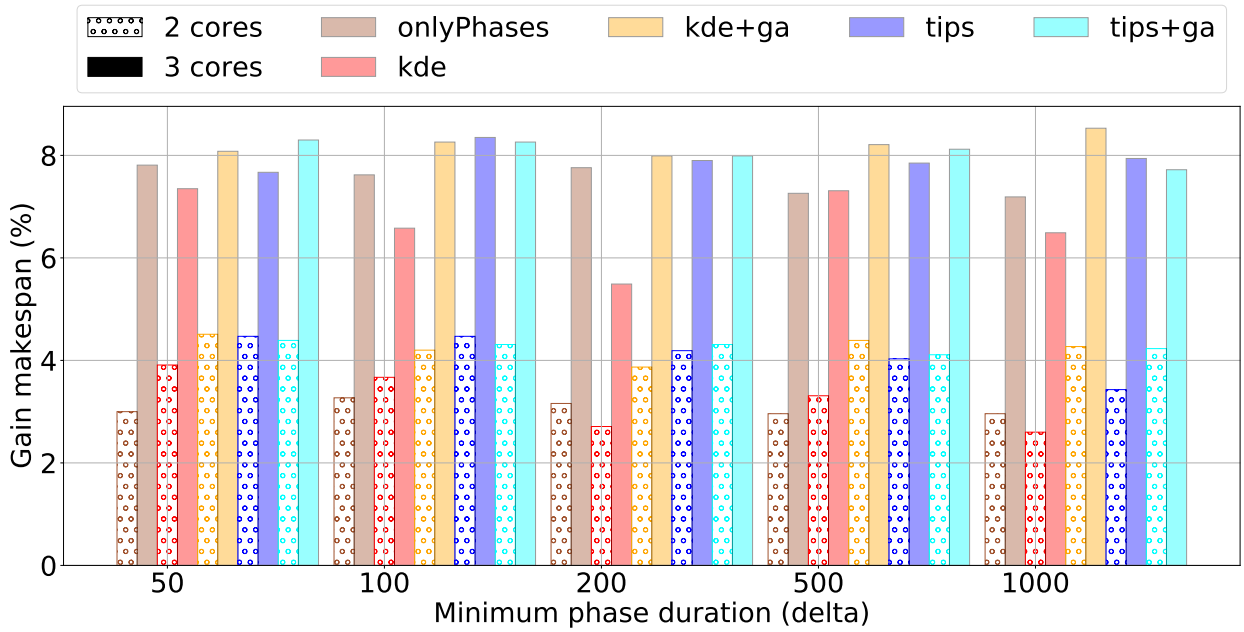


Figure 5.7: Makespan gain between the multi-phase and the single-phase representation (Synthetic).

4.2 Results

Figure 5.7 represents the gain in makespan for the profiles of the Synthetic benchmark when scheduled on 2 and 3 cores. We observe that the gain of the multi-phase model is always higher when the system is scheduled on 3 cores than on 2 cores. However, the impact of δ on the gain is not linear. For some methods such as *onlyPhases*, it does not really change the results. Therefore, in addition to Figure 5.7 we present the maximum gain for each method on 2 and 3 cores in Table 5.1.

On 2 cores, *onlyPhases* has the lowest average gain whatever the number of cores. The difference with the other methods is lower on 3 cores than on 2 cores. Regarding the phases-based GA, it systematically improves the gain when it is applied on *kde* profiles. However, applied to *tips* the resulting profile has a slightly lower gain in average than the initial one although it sometimes outperforms it (e.g. with 3 cores and $\delta = 50$ cycles). The *tips* profiles are in average the best on 2 cores, but on 3 cores it is *kde+ga* although *kde* is less efficient than *tips*.

Table 5.1: Maximum gain of the methods according to the number of cores (Synthetic).

cores (#)	onlyPhases	kde	kde + ga	tips	tips + ga
2	3.27	4.07	4.35	4.47	4.39
3	7.81	7.99	8.49	8.35	8.30

The results for Rosace are given in Figure 5.8 and we also present the maximum results per core in Table 5.2. For this case study, the gain increases when δ decreases for all the methods. Hence, in this case, decreasing δ is interesting to improve the efficiency of the multi-phase model but a trade-off must be found with the number of synchronizations to implement which also increases. As opposed to the results with the Synthetic benchmark, even if *kde+ga* improves the gain of *kde*, it never outperforms *tips* that is often the best method regardless of the value of δ. However, the gain of *tips+ga* is always less than 1 point under the gain of *tips* (maximum difference is 0.97). Additionally, the results of *onlyPhases* are close to those of *tips+ga* and it has the best gain when $\delta = 200$ cycles.

In short, in terms of makespan gain for our 2 case studies, *tips* is in average the best method and *kde* among the worst to create profiles. The phases-based GA increases the gain of *kde* profiles

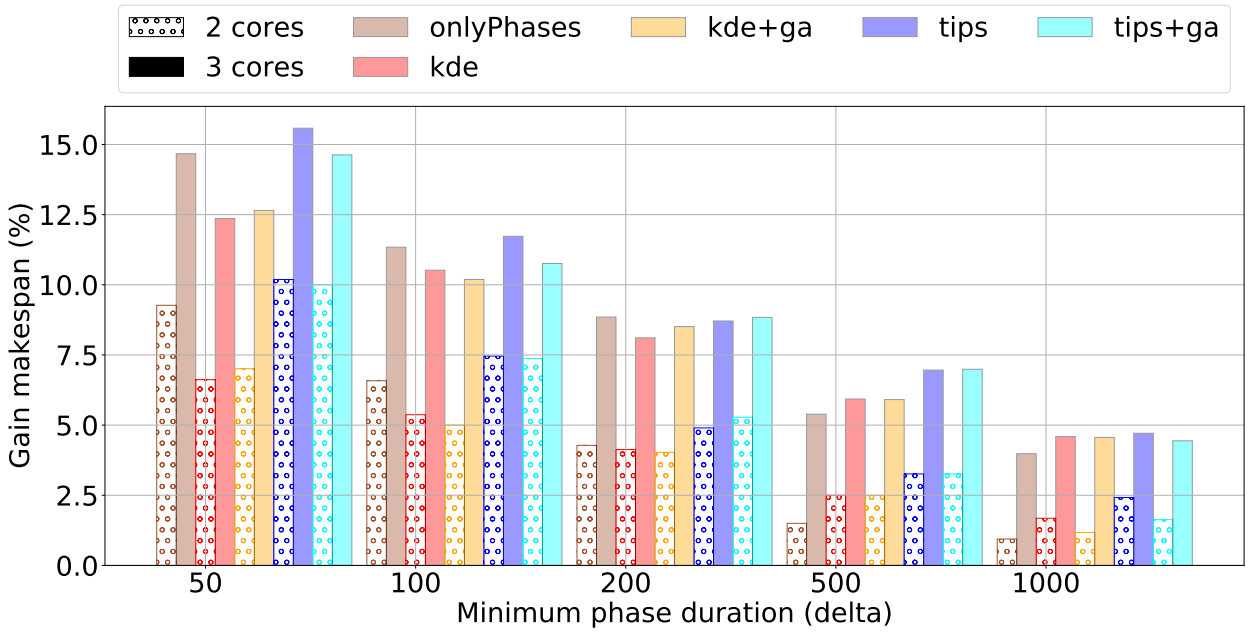


Figure 5.8: Makespan gain between the multi-phase and the single-phase representation (Rosace)

Table 5.2: Maximum gain of the methods according to the number of cores (Rosace).

cores (#)	onlyPhases	kde	kde + ga	tips	tips + ga
2	9.27	6.62	7.01	10.19	10.00
3	14.67	12.36	12.65	15.58	14.63

such that *kde+ga* sometimes even outperforms *tips*. However, applied to Rosace, the gain of the profiles is slightly decreased with a few exceptions. The traces-based GA (i.e. version *onlyPhases*) has a low gain when applied to the Synthetic benchmark, but achieves comparable results to those of *tips+ga* with Rosace.

5 Summary

The *tips* profiles are in general among the most efficient in terms of makespan gain. However, *kde* provides good initial profiles for the phases-based GA with the Synthetic benchmark because the resulting profiles can outperform *tips*. Applied to *tips* profiles, the phases-based GA produces profiles that are slightly less efficient in terms of gain than the original profiles but they still use less synchronizations. The results of *onlyPhases* on the Synthetic benchmark are similar to those of *kde* but on Rosace they are close to those of *tips* and *tips+ga* although the profiles generated by this method also use less synchronizations.

Table 5.3: Ratio gain / number of synchronizations (×100) for each method according to δ.

δ (cycles)	Rosace onlyPhases	kde	kde + ga	tips	tips + ga	Synthetic onlyPhases	kde	kde + ga	tips	tips + ga
1000	2.07	1.21	1.22	1.23	1.16	1.63	1.62	1.82	1.69	1.73
500	2.74	1.13	1.19	1.24	1.25	1.50	1.29	1.72	1.41	1.67
200	2.06	0.92	1.06	0.85	0.97	1.39	0.95	1.52	1.05	1.33
100	1.65	0.92	1.04	0.79	0.96	1.35	0.80	1.27	0.92	1.11
50	1.40	0.97	1.15	1.01	1.05	1.36	0.71	1.02	0.76	0.97

In order to better evaluate the trade-off between the number of synchronizations and the gain of each method, Table 5.3 presents the ratio between the gain and the number of synchronizations (multiplied by a factor 100). With the Synthetic benchmark, when $\delta \geq 200$ cycles the *kde+ga* method has the best ratio value and for $\delta < 200$ it is *onlyPhases*. With Rosace, *onlyPhases* is always the best method according to this metric. Moreover, its minimum value with $\delta = 50$ cycles is higher than the best value for the other methods regardless of δ. In addition, we observe that except for Rosace when $\delta = 1000$ cycles, the phases-based GA systematically improves the trade-off obtained with its initial profile. This shows that the GA methods generally find a better trade-off between the over-approximation and the number of synchronizations than the other methods.

6 Conclusion

The objective of this chapter is to compare the design methods presented in the previous chapter and the TIPs method from the state of the art. The experiments are conducted on two case studies so the results of the comparison cannot be generalized. However, we identified some trends.

In the first part of the experiments, we observed that the simple heuristic methods (i.e. *kde* and *tips*) generally create more phases than the GA approaches. Indeed, the traces-based GA creates the least phases and the phases-based GA is able to significantly reduce their number when applied on *kde* or *tips*. Due to the higher number of phases, the simple heuristics also require a higher number of synchronizations to control the access over-approximation. Thanks to the phase merges, the phases-based GA manages to reduce this over-approximation on the Synthetic benchmark. The initial profiles already had a low over-approximation for Rosace so the phases-based GA did not make a difference for this aspect.

In the second part, the profiles are used to schedule the case studies on 2 and 3 cores. The *tips* method generally generates profiles that better reduce the makespan of the system than profiles generated by other methods. We observed that the phases-based GA slightly reduces the gain of *tips* in average but improves systematically the gain of *kde*. The results of *kde+ga* were even better than *tips* for the Synthetic benchmark on 3 cores. The traces-based GA yields more gain with Rosace than with the Synthetic benchmark compared to the other methods. The results of the scheduling experiment must be interpreted in conjunction with those of the first part to evaluate how the methods handle the trade-off between the makespan gain and the number of synchronizations. The results showed that *onlyPhases* was the most efficient in general, and that the phases-based GA managed to improve this trade-off in almost all the cases.

Therefore, the GA methods are efficient to create a profile from traces and to improve existing profiles. Their main benefit is the reduction of the number of synchronizations but they also yield good gain results. These results can be further improved if we identify some other criteria that influence the gain of a multi-phase profile over the single-phase representation. Such new criteria could be easily integrated in the fitness function of the GAs while their integration in the other heuristics (e.g. *tips*, *kde*) would be more difficult.

Throughout these experiments, we used a simple scheduling algorithm that schedules the tasks ASAP according to a pre-set order which was sufficient to compare the efficiency of the design methods. The results on the 2 case studies show that the multi-phase model was better than the single-phase model to reduce the effects of the interference analysis on the makespan, regardless of the method used to generate the multi-phase profiles (the gain was always positive). We could increase this gain by designing scheduling techniques that take into account the effects of contentions and work specifically at the phase granularity level may yield better gains.

Chapter 6

Static scheduling of multi-phase tasks on multi-core platforms

Contents

In the previous chapters, we presented new techniques to derive a multi-phase profile from the binary code of a task and evaluated them. In order to evaluate their impact on the makespan of the task systems, we used a basic list scheduling algorithm. Although this simple algorithm is sufficient and fair to compare the different methods to build a profile, the multi-phase model efficiency may be increased by the use of more elaborate scheduling algorithms that take into account the effects of contentions to take decisions or benefit from the phases granularity.

In this chapter, we address the multi-core scheduling problem for multi-phase task systems. As pointed out in the related work (see Table 2.1), many papers already addressed this problem but among those who consider memory contentions, few of them use multi-phase models other than AER or PREM and tolerate, or even assume that some contentions exist. Therefore, we propose to consider the generic multi-phase model that we described in Chapter 3 and to build a static schedule with the objective to minimize its makespan in the presence of interference.

Some ILP formulations with multi-phase models have already been proposed. The formulation proposed for the TIPs model is the closest to our problem but it assumes non-periodic tasks without dependencies and each core can only host one task. We propose an ILP formulation that relies on the multi-phase model defined in Chapter 3, with dependencies between tasks. The formulation also takes into account the possible contentions between the phases scheduled in parallel.

The ILP suffers from scalability issues so we also propose some heuristics that account for the effects of interference in the schedule. These heuristics adopt different strategies: some of them explore the possible tasks ordering while others focus on the possible start dates of tasks. In addition, they either account for contentions when scheduling the tasks, i.e. on partial schedules, or only on complete schedules.

Moreover, the chapter presents an optimization to apply on existing schedules that tries reducing the amount of contentions in the model. This optimization detects phases that distribute too much

contentions and attempts to merge phases to reduce the amount of contentions, which may in turn lead to a reduction of the overall makespan.

1 Problem definition

Notation	Definition
τ^i	task i
G	DAG defining tasks' dependencies
E	set of edges (dependency relations) of G
$preds(\tau^i)$	set of predecessors of τ^i
$succs(\tau^i)$	set of successors of τ^i
$\mathbb{C}$	set of cores composing the architecture
C_k	core with index k
ω_k^i	True if τ^i is mapped to C_k
ρ_j^i	True if τ^i and τ^j are mapped to the same core
ϕ_k^i	phase k in the representation of τ^i
$\phi_k^i.d$	start date of ϕ_k^i without interference
$\phi_k^i.dur$	duration of ϕ_k^i without interference
$\phi_k^i.m$	maximum number of memory accesses performed within ϕ_k^i
$\phi_l^i.p$	timing penalty added to ϕ_l^i due to potential interference
$\phi_l^i.d^{\#}$	*post-analysis* date of ϕ_l^i, i.e. date in the presence of interference
$\chi_{i,j\_k,l}$	True if the intervals covered by ϕ_j^i and ϕ_l^k overlap
$\theta_{i,j\_k,l}$	True if ϕ_j^i starts before the end of ϕ_l^k
$\phi_{\Phi^i}^i.d^{\#}$	end date of τ^i in the presence of interference
$\phi_j^i.\gamma$	number of potential contentions suffered by ϕ_j^i
$\phi_j^i.\gamma_k$	number of potential contentions suffered by ϕ_j^i from C_k

Table 6.1: Notations

We target the following static scheduling problem: given a set of homogeneous cores connected to a shared memory through a First-In First-Out (FIFO) bus and a system composed of data-dependent tasks specified as a Directed Acyclic Graph (DAG), schedule the tasks on the cores in order to minimize the interference-aware makespan of the system. In this problem instance, we consider non-preemptive tasks only, and tasks are not partitioned to the cores prior to the scheduling phase.

The multi-core architecture and the dependencies between the tasks are defined formally as in the experiments of Chapter 5. The six first lines of Table 6.1 recall these notations and the remainder is introduced in the following. In the remainder of the document, we call *dependency-free tasks* the tasks that have no predecessors and no successors (i.e. $(preds(\tau^i) = \emptyset) \wedge (succs(\tau^i) = \emptyset)$).

Our objective is to build a schedule $\mathbb{S}$ of the tasks of T on the cores composing $\mathbb{C}$.

For each core C_k, we define the following attributes in $\mathbb{S}$:

- $\mathbb{S}(C_k)$: the schedule on C_k which is a sequence of phases, ordered by their starting date.
- $\mathbb{S}(C_k).end$: the end date of the last phase scheduled on C_k.

The makespan of the task system under schedule $\mathbb{S}$ is $makespan(\mathbb{S}) = max_{C_k \in \mathbb{C}}(\mathbb{S}(C_k).end)$.

2 ILP Formulation

We now provide an ILP formulation of the problem. In this formulation we use bold font to denote the variables of the ILP system, ILP.1 is the objective function of the ILP formulation and the other equations numbered ILP.X are the constraints.

We first introduce variable $mksp$ denoting the makespan of the task system. It appears in the objective function that minimizes the makespan:

$$minimize\ \boldsymbol{mksp} \tag{ILP.1}$$

We use $\boldsymbol{\phi^i_{\Phi^i}.d^{\#}}$ to denote the end date of τ^i, which is the end date of its last phase:

$$\boldsymbol{\phi^i_{\Phi^i}.d^{\#}} = \boldsymbol{\phi^i_{\Phi^i-1}.d^{\#}} + \phi^i_{\Phi^i-1}.dur + \boldsymbol{\phi^i_{\Phi^i-1}.p} \tag{ILP.2}$$

.

The makespan of the system is greater than the end date of all tasks:

$$\forall \tau^i, \boldsymbol{mksp} \geq \boldsymbol{\phi^i_{\Phi^i}.d^{\#}} \tag{ILP.3}$$

Moreover, each task τ^i starts after date 0 and after the end of all its predecessors:

$$\forall \tau^i, \boldsymbol{\phi^i_0.d^{\#}} \geq 0 \tag{ILP.4}$$

$$\forall \tau^k \in preds(\tau^i), \boldsymbol{\phi^i_0.d^{\#}} \geq \boldsymbol{\phi^i_{\Phi^k}.d^{\#}} \tag{ILP.5}$$

Following the definition of the start date of a phase in Chapter 3 section 3.2 (equation 3.2), we can express the date of each subsequent phase as:

$$\forall \tau^i, \forall 0 \leq j < \Phi^i - 1, \boldsymbol{\phi^i_{j+1}.d^{\#}} = \boldsymbol{\phi^i_j.d^{\#}} + \phi^i_j.dur + \boldsymbol{\phi^i_j.p} \tag{ILP.6}$$

We use boolean variable $\boldsymbol{\omega^i_k}$ to express the mapping of task τ^i: $\boldsymbol{\omega^i_k} = 1$ if and only if τ^i is mapped on C_k. Each task is mapped to a unique core so we add the constraints:

$$\forall \tau^i : \sum_{0 \leq k < N_c} \boldsymbol{\omega^i_k} = 1 \tag{ILP.7}$$

We also introduce variable $\boldsymbol{\rho^i_j}$ that is equal to 1 if and only if τ^i and τ^j are mapped to the same core:

$$\forall \tau^i, \tau^j, \boldsymbol{\rho^i_j} = \sum_{0 \leq k < N_c} \boldsymbol{\omega^i_k} \wedge \boldsymbol{\omega^j_k}$$

Because of the conjunction $\wedge$, the above equation is not linear. Therefore, we have to use a new variable $\boldsymbol{\Omega^{i,j}_k} = \boldsymbol{\omega^i_k} \wedge \boldsymbol{\omega^j_k}$ and add the following equations:

$\forall \tau^i, \tau^j, 0 \leq k < N_c$,

$$\boldsymbol{\Omega^{i,j}_k} \leq \boldsymbol{\omega^i_k} \tag{ILP.8}$$

$$\boldsymbol{\Omega^{i,j}_k} \leq \boldsymbol{\omega^j_k} \tag{ILP.9}$$

$$\boldsymbol{\Omega^{i,j}_k} + 1 \geq \boldsymbol{\omega^i_k} + \boldsymbol{\omega^j_k} \tag{ILP.10}$$

Therefore, the equation becomes:

$$\boldsymbol{\rho^i_j} = \sum_{0 \leq k < N_c} \boldsymbol{\Omega^{i,j}_k} \tag{ILP.11}$$

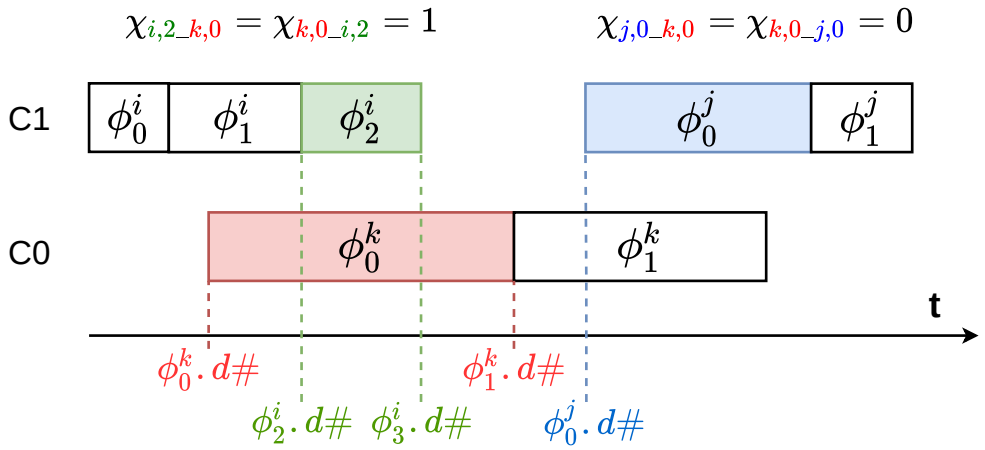


Figure 6.1: 3 tasks scheduled on 2 cores

In the following, any other conjunction will be converted to a linear form in the same way. For clarity reasons, we do not provide the details for the other linearizations of conjunctions.

Two phases may contend with each other if they are scheduled on different cores and their execution intervals overlap. We introduce the boolean variable $\boldsymbol{\chi_{i,j\_k,l}}$ that is true if the intervals covered by ϕ_j^i and ϕ_l^k overlap:

$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow \neg((\boldsymbol{\phi_{l+1}^k.d^\#} \leq \boldsymbol{\phi_j^i.d^\#}) \vee (\boldsymbol{\phi_{j+1}^i.d^\#} \leq \boldsymbol{\phi_l^k.d^\#}))$$
$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow (\boldsymbol{\phi_j^i.d^\#} < \boldsymbol{\phi_{l+1}^k.d^\#}) \wedge (\boldsymbol{\phi_l^k.d^\#} < \boldsymbol{\phi_{j+1}^i.d^\#})$$

The overlapping is illustrated by Figure 6.1. Phase ϕ_0^k overlaps with ϕ_2^i but not with ϕ_0^j so $\chi_{k,0\_i,2} = \chi_{i,2\_k,0} = 1$ and $\chi_{k,0\_j,0} = \chi_{j,0\_k,0} = 0$.

We need to decompose the equivalence relation into several constraints into the ILP system. That is why we define $\boldsymbol{\theta_{i,j\_k,l}}$ as:

$$\boldsymbol{\theta_{i,j\_k,l}} \Leftrightarrow \boldsymbol{\phi_j^i.d^\#} < \boldsymbol{\phi_{l+1}^k.d^\#}$$

so that the equivalence becomes

$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow \boldsymbol{\theta_{i,j\_k,l}} \wedge \boldsymbol{\theta_{k,l\_i,j}} \tag{ILP.12}$$

$\boldsymbol{\theta_{i,j\_k,l}}$ is defined using the big-M notation and a cancellation variable $\boldsymbol{\beta_{i,j\_k,l}}$:

$\forall \tau^i, \tau^j, 0 \leq j < \Phi^i, 0 \leq i < \Phi^k$,

$$1 + \boldsymbol{\phi_j^i.d^\#} \leq \boldsymbol{\phi_{l+1}^k.d^\#} + M(1 - \boldsymbol{\theta_{i,j\_k,l}}) \tag{ILP.13}$$
$$\boldsymbol{\phi_j^i.d^\#} \geq \boldsymbol{\phi_{l+1}^k.d^\#} - M(1 - \boldsymbol{\beta_{i,j\_k,l}}) \tag{ILP.14}$$
$$\boldsymbol{\beta_{i,j\_k,l}} + \boldsymbol{\theta_{i,j\_k,l}} = 1 \tag{ILP.15}$$

The overlapping of 2 phases is forbidden if their tasks (resp. τ^i and τ^k) are scheduled on the same core ($\boldsymbol{\rho_k^i} = 1$). Therefore:

$$\boldsymbol{\chi_{i,j\_k,l}} \leq 1 - \boldsymbol{\rho_k^i} \tag{ILP.16}$$

In order to compute the time penalty of ϕ_j^i, we multiply the number of contentions it may receive ($\boldsymbol{\phi_j^i.\gamma}$) by the cost of one penalty denoted *penalty_cost*, so we have:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi_j^i.p} = \boldsymbol{\phi_j^i.\gamma} \times penalty\_cost \tag{ILP.17}$$

$\boldsymbol{\phi^i_j.\gamma}$ is the sum of the contentions that may be caused by tasks on all the cores:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi^i_j.\gamma} = \sum_{0 \leq k < N_c} \boldsymbol{\phi^i_j.\gamma_k} \tag{ILP.18}$$

with $\boldsymbol{\phi^i_j.\gamma_k}$ the number of contentions that ϕ^i_j may experience from tasks scheduled on core k. As we consider a shared memory bus following a FIFO policy, $\boldsymbol{\phi^i_j.\gamma_k}$ is bounded by $\phi^i_j.m$:

$$\forall \tau^i, 0 \leq j < \Phi^i, 0 \leq k < N_c, \boldsymbol{\phi^i_j.\gamma_k} = \min(\phi^i_j.m, \sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^l_q.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k})) \tag{6.1}$$

the term $(\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k})$ states that ϕ^i_j receives contentions from ϕ^l_q if and only if ϕ^l_q is mapped to core k and overlaps with ϕ^i_j.

Finally, to linearize the minimum operator, we use the following equations with $\boldsymbol{\alpha^i_{j,k}} \in \{0, 1\}$ guaranteeing that one of the proposed values is taken:

$\forall \tau^i, \tau^j, 0 \leq j < \Phi^i, 0 \leq k < N_c,$

$$\boldsymbol{\phi^i_j.\gamma_k} \leq \phi^i_j.m \tag{ILP.19}$$

$$\boldsymbol{\phi^i_j.\gamma_k} \leq \sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k}) \tag{ILP.20}$$

$$\boldsymbol{\phi^i_j.\gamma_k} \geq (\sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k})) - M \times \boldsymbol{\alpha^i_{j,k}} \tag{ILP.21}$$

$$\boldsymbol{\phi^i_j.\gamma_k} \geq \phi^i_j.m - M(1 - \boldsymbol{\alpha^i_{j,k}}) \tag{ILP.22}$$

3 Heuristics

As we will show in chapter 7, the ILP resolution time does not scale up when the number of tasks or phases grows. In this section, we present several alternative scheduling heuristics.

In the following algorithms, we use function *computeContentions*($\mathbb{S}$) that computes the values of the $\phi^i_j.d^\#$ by applying formula (6.1) page 85 on each phase ϕ^i_j of $\mathbb{S}$.

3.1 Greedy policies

We present 2 scheduling policies that are implemented as part of a list scheduling algorithm: the algorithm selects a task from a list of ready tasks, schedules it following the policy, updates the list of ready tasks, and iterates until all tasks have been scheduled. In order to select one of the ready tasks to schedule, we use the same four priorities defined in Chapter 5 section 4.1: *readyDate*, *minBudget*, *maxBudget* and *CPC*.

3.1.1 As Soon As Possible scheduling (ASAP)

The As Soon As Possible (ASAP) policy takes the current partial schedule (initially empty) and builds as many schedules as there are cores in $\mathbb{C}$ by selecting a task and scheduling it as soon as possible on each of the cores. It then selects the partial schedule that has the lowest makespan and moves on to the next task. The interference analysis is performed only once all the tasks have been scheduled. Consequently, this is the simplest and the fastest algorithm of all the presented heuristics.

3.1.2 Starting Date Enumeration (SDE)

The ASAP strategy is not always the best choice to minimize the makespan in the presence of interference. For instance, Figure 6.2 shows 3 different ways to schedule a new task (the orange one) on core C_1. At the top, when scheduling the task as soon as possible, the phase with 10 accesses overlaps with 2 other phases in parallel and creates in the worst case 13 (8+5) contentions on core 0 (depicted in red). In the schedule below, we postponed the task start date to the end of the phase with 8 accesses so that the 10-accesses phase may only create 5 contentions, and this choice yields a reduction of the makespan. In the last schedule at the bottom, the task is postponed even more, to the next phase date in parallel, so that no contention can appear, yielding the smallest makespan. Following that idea, we developed the *Starting Date Enumeration (SDE)* heuristic that attempts to schedule the current task at several dates on each of the cores and performs an interference analysis for each possibility before selecting the one that minimizes the makespan.

Algorithm 9 describes SDE. It takes as inputs the current task to schedule, τ^i, and the current partial schedule $\mathbb{S}$, on which an interference analysis has been performed. The enumeration of the possible start dates for τ^i is limited to the interval $[\![minDate, maxDate]\!]$ in which minDate is the earliest possible start date of τ^i due to precedence constraints, and maxDate is the current makespan of the partial schedule. For each core C_k (line: 4), function *parallelDates* extracts the start and end dates of phases scheduled on the other cores, which fall in the $[\![minDate, maxDate]\!]$ interval. Then, τ^i is iteratively scheduled at each of these dates on C_k in $\mathbb{S}$ (line: 7), and only the result yielding the smallest makespan (after interference analysis) is kept (line: 9). In the end, τ^i is scheduled on the core and at the date that yielded the best makespan.

It is worth mentioning that SDE attempts to place the new task only based on the date of its first phase. The computational complexity is already high but it would be much higher if the principle was also extended to the other phases of the profile.

3.2 Iterative Priority Scheduling Heuristic (IPH)

The Iterative Priority Heuristic (IPH), detailed in Algorithm 10, is an adaptation of the main algorithm of [45]. This algorithm has already been successfully adapted to the AER model in [44],

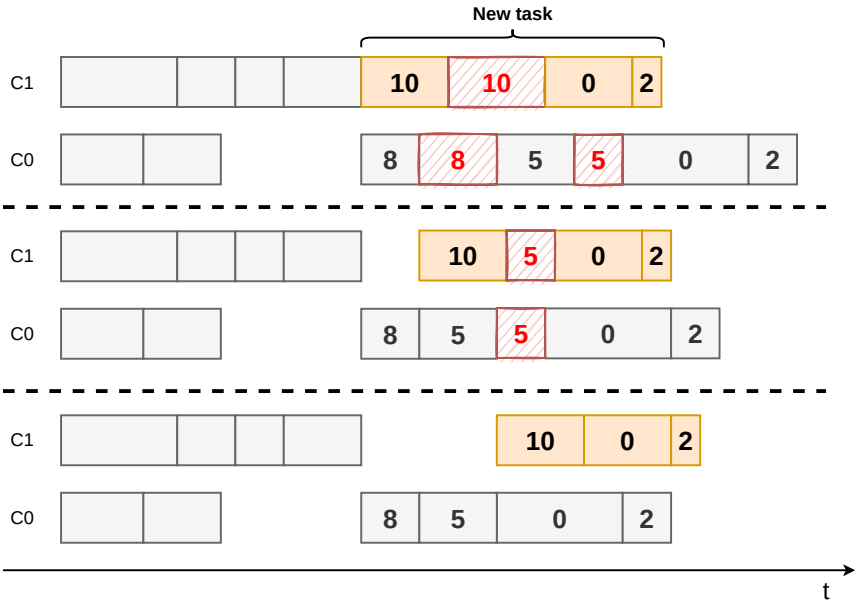


Figure 6.2: 3 different placements for a new task: the numbers within phases indicate their worst-case number of accesses and the red hatched rectangles are the additional penalty due to possible interference.

Algorithm 9 SDE

Require: τ^i ; $\mathbb{S}$
1: $minDate = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
2: $maxDate = makespan(\mathbb{S})$
3: $bestMakespan, bestSched = +\infty, \mathbb{S}$
4: **for** C_k **in** $\mathbb{C}$ **do**
5: $dates = parallelDates(\mathbb{S}, C_k, minDate, maxDate, \tau^i)$
6: **for** d **in** $dates$ **do**
7: $\mathbb{S}' = scheduleTask(\mathbb{S}, C_k, \tau^i, d)$
8: $computeContentions(\mathbb{S}')$
9: **if** $makespan(\mathbb{S}') < bestMakespan$ **then**
10: $bestMakespan = makespan(\mathbb{S}')$
11: $bestSched = \mathbb{S}'$
12: **end if**
13: **end for**
14: **end for**
15: **return** bestSched

but our task model is more generic and some assumptions made in [44] are not applicable here. As a consequence Algorithms 10 and 11 were adapted from [45] to address the multi-phase model and create IPH.

The principle of this algorithm is to test iteratively different combinations of tasks priorities, called *priority vectors*, while converging to the best makespan, given by the objective variable *Obj* until no progress is made. In the initialization, we build the initial best schedule $\mathbb{S}^{best}$ using our ASAP greedy heuristic. Then, $\mathbb{S}^{best}$ is used to build the initial target interval $[\![LB, UB]\!]$ (line 1) and *Obj* is chosen as the median value of this target interval. The initial values of the bounds do not have a huge impact on the algorithm performance because the interval is re-adjusted throughout the iterations, but setting them close to a viable objective can save a few initial iterations. In order to speed up the computations, we used and when necessary, adapted, the following optimizations that were present in the original algorithm of [45]:

1. Using a symmetric instance of the scheduling problem because scheduling backwards may open other scheduling options, that is why we distinguish the two graphs $G^{forward}$ and $G^{backwards}$ (line 5) and the priority vectors on both directions (line 34).

2. Implementing the algorithm in parallel so that several priority vectors are tested at the same time by separate threads.

3. Using a hash set to store the priority vectors that have already been tried to avoid repetitions (line: 6).

4. The priority vector is modified using information about conflicting tasks that prevent each other to be scheduled before *Obj* (line: 33).

The two first optimizations were directly implemented in our heuristic. We adapted the third optimization to exploit the fact that two different priority vectors may produce the same scheduling order because of tasks dependencies. For example, if we consider tasks A, B and C with B and C successors of A, then assigning priorities 3, 2, 1 to respectively A, B and C yields the same scheduling order (A then B then C) as when assigning priorities 2, 3, 1 because task A must be executed before B and C has a priority inferior than B. Therefore, instead of saving the priority vectors in the hash set, our algorithm computes and saves an equivalence class of the priority vectors given the dependencies of the system (i.e. the scheduling order of the tasks) (line: 11). We also adapted the fourth optimization so that, when there is no conflicting tasks, the algorithm

Algorithm 10 IPH

Require: $G = (T, E), \mathbb{C}$
1: $UB, LB, \mathbb{S}^{best} = init(G, \mathbb{C})$
2: $Obj = (LB + UB)/2$
3: $failCount = 0$
4: $G^{forward} = G$
5: $G^{backward} = reverse(G)$
6: $prioHashSet = \{\}$
7: $init\_prio = [UB - \phi_0^i.d^{\#}]_{\forall \tau^i \in T}$
8: $sQueue = [(G^{forward}, Obj, init\_prio)]$
9: **while** $(LB < UB) \wedge (sQueue \neq [])$ **do**
10: $(G^c, Obj, prio) = sQueue.pop()$
11: $hash = Hash(eq\_class(prio, G^c))$
12: **if** $hash \in prioHashSet$ **then**
13: continue
14: **end if**
15: $prioHashSet.add(hash)$
16: $\mathbb{S} = findSchedule(G^c, \mathbb{C}, Obj, prio)$
17: **if** $makespan(\mathbb{S}) < makespan(\mathbb{S}^{best})$ **then**
18: $\mathbb{S}^{best} = \mathbb{S}$
19: **if** $UB > makespan(\mathbb{S})$ **then**
20: $UB, LB = update(UB, LB, \mathbb{S})$
21: **end if**
22: $Obj^{new} = UB - 100$
23: $priority = [Obj - \phi_0^i.d^{\#}]_{\forall \tau^i \in T}$
24: **else**
25: $failCount++$
26: **if** $failCount \geq log2(|T|)$ **then**
27: $LB = LB + (UB - LB)/4$
28: $failCount = 0$
29: **end if**
30: $Obj^{new} = \lceil min(UB, 1.1 \times Obj) \rceil$
31: **end if**
32: $prio_1 = [Obj - prio[i]]_{\forall \tau^i \in T}$
33: $prio_2 = modPrio(prio, \mathbb{S})$
34: $G^{c1}, G^{c2} = switchOrder(G^c, G^{backward}, G^{forward})$
35: $sQueue.push(\{G^{c1}, Obj^{new}, prio_1\})$
36: $sQueue.push(\{G^{c2}, Obj^{new}, prio_2\})$
37: **end while**
38: **return** $\mathbb{S}^{best}$

relies on the amount of contentions to modify the priority vector. However, relying on contentions in a more systematic way did not yield any improvement of the results.

At each iteration, the algorithm calls function $findSchedule$ (described in Algorithm 11 that we detail later) to build a schedule $\mathbb{S}$ from scratch using a task system G^c, a vector $prio$ that gives priorities to the tasks, and an objective Obj for the makespan of the schedule (line: 16). Once $\mathbb{S}$ is built, the algorithm compares its makespan with the makespan of the best schedule found so far: $\mathbb{S}^{best}$. If it is inferior, schedule $\mathbb{S}$ is saved as the new $\mathbb{S}^{best}$, the UB and LB are updated (line: 20) in order to lower the makespan objective in the next iteration, and changes are made to the task priorities to reflect the order of the starting dates of tasks in $\mathbb{S}$ (lines: 19-23). If it is superior to Obj however, Obj is increased in order to give some more slack to the algorithm in the next iteration,

and LB is increased as well if the algorithm has failed enough times (lines: 25-30). The algorithm then iterates, until either LB reaches UB or it runs out of new priority vectors to test.

There are several constants impacting the computation cost of the algorithm that are defined in an empirical way:

- Line 22: Obj^{new}, the next objective is set to $UB-100$. The value must not be too ambitious to allow $findSchedule$ to find suitable schedules and the convergence towards the best priority vectors. As the minimum contention duration that we applied in our tests is 50 cycles, the number of contentions to avoid in order to improve the makespan is reasonable and 100 is also an order of magnitude below the duration of the tasks we scheduled who had a WCET superior to 1000 cycles (and sometimes superior to 20000 cycles).

- Line 26: $log2(|T|)$ bounds the number of consecutive attempts of $findSchedule$ without finding a better schedule than $\mathbb{S}^{best}$ before increasing LB. This bound must be high enough to let $findSchedule$ reach Obj but is also responsible for stopping the search when it is not possible. The number of tasks in the system impacts the size of the solution space. In our experiments (Chapter 7), we had from 4 to 329 tasks so the $log2$ allows enough attempts for small systems of a few tasks and not too much for the systems with many tasks.

- Line 30: whenever a failure occurs, the objective is increased by at least 10% of its value (bounded by the current UB). This value has been kept from the original algorithm in [45].

One important point here is that the heuristic does not test all possible combinations of task priorities: at each iteration the current priority vector is modified, and the resulting vector is used in the next iteration if it has not already been used in a prior iteration. The way the algorithm modifies the priority vector does not guarantee that all priority combinations will be explored. In fact the objective of the heuristic is precisely to converge to a solution without having to explore all the combinations.

Algorithm 11 describes the $findSchedule$ function. This function iteratively creates a schedule $\mathbb{S}$ of the tasks of G^c on $\mathbb{C}$, using tasks priorities $prios$ and an objective value Obj for the makespan of $\mathbb{S}$. A set of tasks ready to be scheduled (i.e. whose predecessors have already been scheduled) is maintained, and at each iteration, the ready task with the highest priority is selected for scheduling (line: 4). The selected task τ^i is scheduled following a given policy (in our experiments we used ASAP) and an interference analysis is performed on the resulting partial schedule $\mathbb{S}'$ (line: 6). Note that the priority vector does not define the mapping of the tasks so the scheduling policy is responsible for choosing the cores where tasks are scheduled. If $makespan(\mathbb{S}')$ falls within objective Obj, the algorithm updates the set of ready tasks and iterates with the next ready task (line: 18). If, however, the partial schedule spans more than Obj cycles, the algorithm is allowed to de-schedule some tasks that are put back in the set of ready tasks in order to make room for τ^i before Obj (line: 9). The de-scheduled tasks are the tasks that start after the end of the last predecessor of τ^i and before $Obj - WCET(\tau^i)$, as well as all their (already scheduled) successors. Tasks that start after this date and are not successors of de-scheduled tasks are not put back in the ready set, but are directly rescheduled following the ASAP policy, in respect of their potential dependencies, in order to benefit from the free intervals in the schedule left empty by the de-scheduled tasks (lines: 11-14). Task τ^i is then scheduled ASAP (line: 15). Even if objective Obj is still unmet, the algorithm then goes on to the next task to schedule, hoping that further de-schedulings in the next iterations will allow to meet the objective. The de-scheduling of tasks significantly affects the execution time of the algorithm compared to a greedy solution, and can create an infinite loop under certain circumstances. In order to prevent it, an exploration budget (defined in line: 2) guarantees that the main scheduling loop will not iterate more than a fixed number of times, even though some tasks remain to be scheduled. If the number of iterations reaches the budget, the algorithm exits the loop and falls back to a greedy strategy (line: 21) for the tasks that remain to be scheduled. Tuning the budget value thus allows to trade execution time for precision.

Algorithm 11 *findSchedule*

Require: G^c, $\mathbb{C}$, Obj, $prios$
1: $readyTasks = initRT(G^c)$
2: $budget = \alpha \times |T|$ ▷ α is tuned according to the size of the task system
3: **while** $(readyTasks \neq \emptyset) \wedge (budget > 0)$ **do**
4: $\tau^i = getNext(readyTasks, prios)$
5: $d = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
6: $\mathbb{S}' = scheduleASAP(\mathbb{S}, \mathbb{C}, \tau^i, d)$
7: $computeContentions(\mathbb{S}')$
8: **if** $makespan(\mathbb{S}') > Obj$ **then**
9: $resched, desched, \mathbb{S}_{temp} = unsched(\mathbb{S}, d, Obj, \tau^i)$
10: $readyTasks = readyTasks \cup desched$
11: **for** τ^j **in** $resched$ **do**
12: $\mathbb{S}_{temp} = scheduleASAP(\mathbb{S}_{temp}, \mathbb{C}, \tau^j, d)$
13: $budget = budget - 1$ ▷ τ^j is scheduled again
14: **end for**
15: $\mathbb{S}' = scheduleASAP(\mathbb{S}_{temp}, \mathbb{C}, \tau^i, d)$
16: $computeContentions(\mathbb{S}')$
17: **end if**
18: $updateRT(readyTasks, \tau^i)$
19: $budget = budget - 1$ ▷ accounting for τ^i
20: **end while**
21: **while** $readyTasks \neq \emptyset$ **do**
22: $\tau^i = getNext(readyTasks, prios)$
23: $d = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
24: $\mathbb{S}' = scheduleASAP(\mathbb{S}', \mathbb{C}, \tau^i, d)$
25: $updateRT(readyTasks, \tau^i)$
26: **end while**
27: $computeContentions(\mathbb{S}')$
28: **return** $\mathbb{S}'$

As for Algorithm 10, we define a constant α (line 2) that sets the number of rescheduling operations allowed to reach the objective. We set $\alpha = 3$ for tasks systems with less than 26 tasks so that up to 75 tasks can be reschedule and $\alpha = 1.2$ for the others which allows 394 rescheduling operations for the biggest task system which is already consequent.

3.3 Monte-Carlo Tree Search scheduling (MCTS)

Monte-Carlo Tree Search (MCTS) is a search method that incrementally builds a tree of partial solutions being extended by leaves until reaching a complete solution (i.e. a leaf from which no further action can be taken). Monte-Carlo random simulations are used to grow the tree in the best direction. This algorithm is often applied in games in order to select the best move against an opponent. More generally, it is applied on exploratory problems when the traditional tree search methods are not efficient. Initially the tree has a single node that corresponds to the initial state of the solution, in which no action has been taken (e.g. an empty schedule in our problem, the initial position of pieces for a board game). Then, we can add new nodes to the tree by choosing actions from the current state. The main steps of MCTS are:

1. Selection: the algorithm selects a promising path to explore further.
2. Expansion: the tree is expanded by choosing a possible action from the selected node.

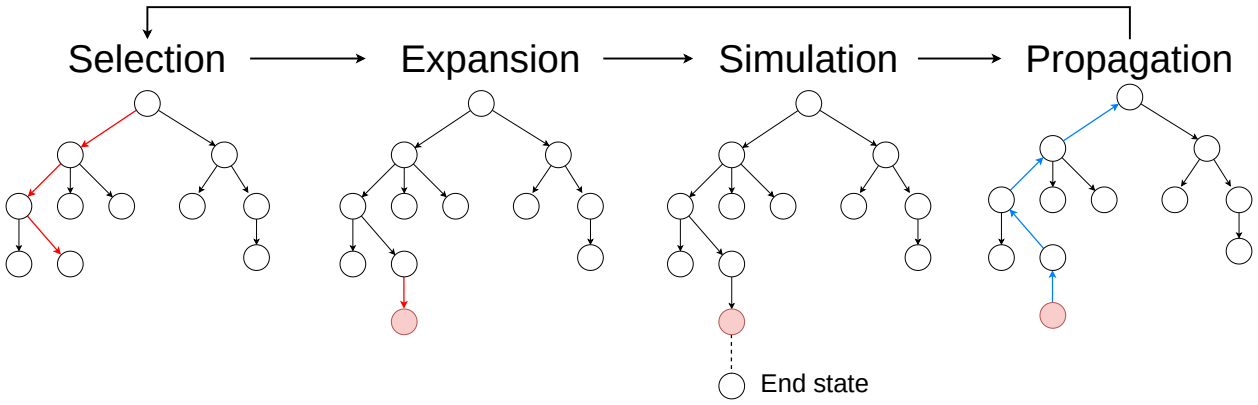


Figure 6.3: Steps of Monte-Carlo Tree Search.

3. Simulation: actions are picked randomly from the expanded node until reaching an end state where no further action can be taken. A reward is computed to assess how good the reached state is, regarding the problem under study.

4. Propagation: the reward of the simulation step is back-propagated to update the statistics of the selected nodes.

Decision problems such as MCTS must find the proper balance between exploitation and exploration:

- Exploitation consists in choosing the best action identified so far, considering that the data gathered is reliable and sufficient.

- Exploration consists in choosing an action to discover new solutions, considering that the data available is insufficient to choose the best action.

Hence, MCTS must run a sufficient number of simulations from each nodes so that their reward can be considered reliable. There are multiple methods to select a node, for instance it is possible to select the node with the highest reward, or the node with the highest number of visits and the highest reward (see the survey [84]). However, the most common selection method consists in computing the score of a node i using the Upper Confidence Bound (UCB) equation, which we use in the remainder of the thesis:

$$UCB_i = \frac{\omega_i}{n_i} + c\sqrt{\frac{\ln n_{parent}}{n_i}} \tag{23}$$

with ω_i the sum of the simulation results that have been back-propagated to i, n_i the number of simulations that passed through i, n_{parent} the number of simulations that were performed from the parent node of i and c the exploration parameter.

The value of c is dependent on the problem, it balances the weight of the exploration score compared to the exploitation score of a node. The idea is that if few simulations passed through a node compared to the others, then we have little confidence on its exploitation score and we need new simulations passing through it. The logarithm smoothes the impact of the number of simulations on the exploration score so that the exploitation score is preponderant at some point. Generally, the default value of c is $\sqrt{2}$.

In the multi-core scheduling problem, an action is a couple (i, j) where i is the index of the next task to schedule and j is the core on which the task is scheduled. The date at which the tasks are scheduled is not part of the action because it would make the exploration space too large to get a good result in a reasonable time, and the implementation is not trivial as the set of possible dates depends on the already scheduled tasks. Instead, the solution we retained is to always choose the ASAP date. An end state is reached whenever there are no more ready tasks to schedule.

According to the definition of an action, the solution space is composed of all the possible tasks orders for all the possible ways to map tasks to cores. Moreover, we want to account for the effects of interference to take decisions but performing an interference analysis is computationally expensive. In short, the solution space is potentially very large and the cost of simulations makes the exploration too long compared to the other scheduling heuristics. For this reason, we adapted the usual MCTS algorithm inspired by the implementation in [65] where MCTS is actually executed as many times as there are decisions:

1. Initially the root node represents an empty schedule, the MCTS is launched for a fixed number of iterations
2. At the end of the iterations, the child node of the current tree root with the highest UCB score is chosen as the next scheduling decision.
3. This node also becomes the new root of the tree, if there are other tasks to schedule then another MCTS is launched to choose the next decision (back to step 1)

This relocation of the root node allows to quickly cut the search space, which is an efficient way to limit the exploration effort. It is worth noticing that the scores of each node are kept from one MCTS run to another so that each run contributes to the decision of the subsequent runs. The number of iterations for each run must be set according to the size of the search space.

In addition, we propose to stop the simulations after N random decisions, so before reaching an end state, in order to reduce their computation time. The partial schedules with the lowest makespan must be the most rewarded but we must also take into account that some of them may actually have scheduled shorter tasks than others to operate a fair comparison. Hence, the makespan is weighted by the sum of the WCET of the tasks that have been scheduled. The reward of a simulation is:

$$reward = -\frac{makespan_{partial}}{\sum_{\forall \tau^i \in scheduled} weight(\tau^i)} \tag{24}$$

with $makespan_{partial}$ the makespan of the partial schedule, *scheduled* the set of scheduled tasks and $weight(\tau^i)$ a function that returns the normalized WCET of the task in the system.

The negative sign ensures that a high makespan is penalized. Only the WCET of the tasks is used to refine the estimation, while the amount of avoided interference or the remaining accesses in the tasks that are not yet scheduled could also help to increase the precision. However, it is difficult to predict which accesses in a task will create contentions when the task is not scheduled yet, even more when computing the interference at the phase level granularity. Our preliminary attempts to include an estimation of the interference remaining in the reward function produced poor results.

3.4 Merging Optimization

In certain situations, the multi-phase model may incur an overestimation of the number of contentions during the interference analysis. In the example depicted in Figure 6.4, the yellow phase may contend with the three phases in parallel. As a result, the interference analysis will count 3 contentions coming from the yellow phase for each of these phases, resulting in 9 contentions in total. In practice this is impossible, as the yellow phase only performs 3 accesses in total. In order to reduce this pessimism, we developed a phase merging algorithm that can be applied on a partial or complete schedule. This optimization detects local situations in which merging together multiple phases of a task would reduce the overestimation of the number of contentions during the interference analysis.

In practice, the optimization looks for phases ϕ_j^i (called saturated phases in the following) that create more than $(|\mathbb{C}|-1) \times \phi_j^i.m$ contentions to phases in parallel during the interference analysis.

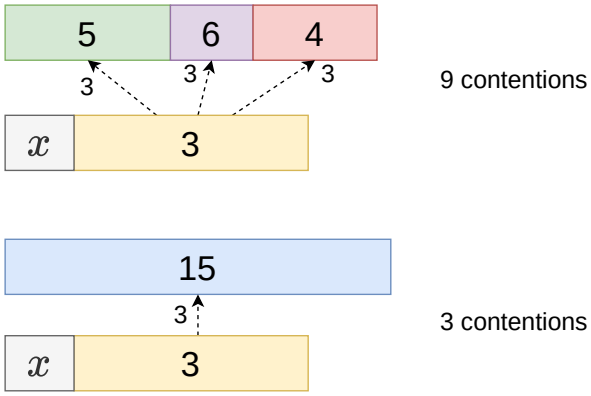


Figure 6.4: An example of local overestimation of contentions.

This formula was chosen as another trade-off between speed and precision. Once a saturated phase is discovered, the algorithm looks for phases scheduled in parallel and assesses whether or not it would be beneficial to merge them together. Indeed, the local benefits of merging phases (w.r.t. a given saturated phase) can be outweighed by the effects of the merge on adjacent tasks. This can be illustrated using Figure 6.4 :

- At the top of the figure, the maximum number of contentions each phase may suffer is :
  - $min(5, x + 3)$ for the green phase.
  - $min(6, 3) = 3$ for the purple phase.
  - $min(4, 3) = 3$ for the red phase.
  - $min(x, 5)$ for the grey phase.
  - $min(3, (5 + 6 + 4)) = 3$ for the yellow phase.

  So if the phases are not merged, the interference analysis counts $9 + min(x, 5) + min(5, x + 3)$ contentions in total for the two cores.

- At the bottom, when the phases are merged, this number is :
  - $min(15, x + 3)$ for the blue phase.
  - $min(x, 15)$ for the grey phase.
  - $min(3, 15) = 3$ for the yellow phase.

  So the interference analysis counts $min(15, x + 3) + min(x, 15) + 3$ contentions in total.

Therefore, if the value of x is strictly greater than 6, the merge is not globally beneficial. For example:

- If $x = 6$, there are $9 + min(6, 5) + min(5, 9) = 19$ contentions when the phases are not merged and $min(15, 9) + min(6, 15) + 3 = 18$ contentions otherwise, so the merge allows to reduce the over-estimation of the number of contentions.
- If $x = 7$, there are $9 + min(7, 5) + min(5, 10) = 19$ contentions when the phases are not merged and ùo, $(15, 10) + min(7, 15) + 3 = 20$ contentions otherwise, so the merge actually increases the over-estimation of the number of contentions.

Algorithm 12 describes the merging optimization. As for the SDE algorithm, computing the contentions several times is necessary to identify the saturated phases and to assess whether or not a merge is profitable. The algorithm retrieves the list of all scheduled phases and iterates over it

until a saturated phase ϕ_j^i is found. When a phase is saturated, the algorithm enters the inner while loop (line 6) to try some merges. The merges are attempted using *candidates*, the list of phases in parallel of ϕ_j^i, that is retrieved by function $getPhasesWithin$ (line 8). Then, function $getMergeablePhases$ searches for two phases of *candidates* that are in the same task, consecutive and have not been studied before (if so they are present in *alreadyAttempted*). When no such phases have been found, the inner while is exited with a break (line 11). Otherwise, the phases are added to the *alreadyAttempted* list and a new schedule $\mathbb{S}'$ is created with the two phases ϕ_l^k and ϕ_{l+1}^k merged using function *mergePhases* that also recomputes the contentions. If the makespan of $\mathbb{S}'$ is better than $\mathbb{S}$ then the merge is confirmed at line 16.

Note that the number of accesses in a phase resulting from the merging of 2 other phases is set conservatively to the sum of the accesses in the 2 merged phases. However, in some cases, it could be profitable to recompute the number of accesses of the new phases from the traces of the task because the over-approximation of accesses may be reduced. Indeed, an access accounted for in both the merged phases can happen only once in the new phase.

The ASAP-based greedy heuristic described in Section 3.1 does not compute the contentions in the system before the schedule is produced. As a result, the scheduling decisions are not impacted by potential merges, so we can apply our merging optimization only once the full schedule has been constructed. On the other hand, the SDE algorithm is interference-aware, so calling the merging optimization at each scheduling step can influence its decisions. In the remainder of the document, whenever the merging optimization is used, it is used after the scheduled is produced with the ASAP policy, and during its construction with the SDE policy. We do not apply the optimization with IPH because it does not improve the trade-off between the computation speed and its efficiency to reduce the makespan of the schedule.

Algorithm 12 *mergeOptimization*

Require: τ^i ; $\mathbb{S}$; start; end
1: $phases = getPhasesIn(\mathbb{S}, start, end)$
2: $idx = 0$
3: **while** $idx < size(phases)$ **do**
4: $\quad \phi_j^i = phases[idx]$
5: $\quad alreadyAttempted = []$
6: $\quad$ **while** $isSaturated(\mathbb{S}, \phi_j^i)$ **do**
7: $\quad\quad end = \phi_j^i.d^{\#} + \phi_j^i.dur + \phi_j^i.p$
8: $\quad\quad candidates = getPhasesWithin(\mathbb{S}, \phi_j^i.d^{\#}, end)$
9: $\quad\quad \phi_l^k, \phi_{l+1}^k = getMergeablePhases(candidates, alreadyAttempted)$
10: $\quad\quad$ **if** $\phi_l^k == null$ **then** ▷ no phases left that can be merge together in *candidates*
11: $\quad\quad\quad$ **break**
12: $\quad\quad$ **end if**
13: $\quad\quad alreadyAttempted.push((\phi_l^k, \phi_{l+1}^k))$
14: $\quad\quad \mathbb{S}' = mergePhases(\mathbb{S}, \phi_l^k, \phi_{l+1}^k)$
15: $\quad\quad$ **if** $makespan(\mathbb{S}') < makespan(\mathbb{S})$ **then**
16: $\quad\quad\quad \mathbb{S} = \mathbb{S}'$
17: $\quad\quad$ **end if**
18: $\quad$ **end while**
19: $\quad idx = idx + 1$
20: **end while**

4 Conclusion

In this chapter, we consider the problem of scheduling a system of data-dependent tasks represented with the multi-phase model and assuming an homogeneous multi-core architecture. The problem is introduced by an ILP formulation with the objective of minimizing the makespan of the schedule in the presence of interference. Then, a set of heuristics is proposed.

The scheduling heuristics and the ILP have been designed for homogeneous cores, but they can be easily adapted to heterogeneous architectures. Indeed, in such a case the process of analyzing the code of a task and building its profile must be repeated for each core kind. Then, the scheduling heuristic would only use the different profiles of each task according to the core on which it is scheduled. However, we chose to prohibit tasks preemptions because it may have a important impact on our assumptions for the model and we did not have enough time to assess this impact and then to adapt our heuristics. We may need to introduce new properties and correctness criteria or to modify the existing ones (in Chapter 3). Indeed, supporting preemption may require additional constraints on the selection of synchronizations and new rules to count the accesses in the phases.

The presented heuristics each have their own strategy to explore the scheduling possibilities. Firstly, ASAP schedules pre-ordered tasks as soon as possible in each core and retains the schedule with the minimum makespan without considering contentions. SDE also uses a pre-defined order of tasks but attempts to schedule them at several dates on the cores and perform an interference analysis on the partial schedules to compare before making its decision. It is worth noting that it does not make predictions about the impact of its choices on future decisions. The IPH heuristic adopts another strategy by exploring the set of possible tasks orderings. Each considered task order is scheduled with the ASAP heuristic and an interference analysis is used to find new orders that can improve the overall makespan. Finally, in the last heuristic, the problem is encoded into a decision tree where a decision consists in scheduling one of the ready tasks on one of the cores. Thus, the algorithm searches for the best order and mapping of the tasks using MCTS. In order to speed up the execution, the simulations used to assess the potential of each possible decision are not building complete schedules. However, the reward function is designed to offer a fair comparison of the partial schedules by including information on the tasks remaining to schedule.

Each heuristic can explore a limited set of the possibilities to schedule the system, either because it considers the possible dates of the tasks but not their order or inversely. Moreover, they differ in the way they account for contentions to take decisions: ASAP and IPH do not consider them at all (until the schedule is entirely built), SDE and MCTS compute them on partial schedules, and MCTS goes further by comparing different partial schedules, accounting for the scheduled tasks. The next chapter compares the efficiency of each heuristic to minimize the makespan of a task system in different contexts (number of cores, interference penalty...) and with a variety of profile shapes.

Chapter 7

Comparative study: static scheduling in multi-core platforms

Contents

In the previous chapter, an ILP introduced the problem of scheduling multi-phase tasks in multi-core platforms taking into account contentions. While the problem already admits a large solution space when considering single-phase tasks, the presence of phases enlarges this solution space. Indeed, there are multiple ways for two tasks to contend with each other according to the subset of their phases that interfere. In the first part of the chapter, we compare the multi-phase and single-phase models using the ILP to obtain optimal solutions. The tasks systems are generated synthetically by varying different parameters such as the shape of the profile, the access rate or the presence of empty phases. We also compute the schedules using the scheduling heuristics presented in the last chapter to observe how far they are from optimal solutions. As the ILP solving time increases quickly with the number of tasks and phases to schedule, this first part only considers small task systems.

In the second part, we extend the experiments to larger task systems, only using the scheduling heuristics. Each heuristic has its own strategy to explore the space while keeping a limited computational complexity. We compare how they perform regarding several characteristics of the task systems and try to explain their behavior. Moreover, we study the general influence of the parameters used to generate the synthetic tasks, which may guide techniques to generate multi-phase profile efficiently.

The third part of the experiments applies the heuristics on two case studies to show how they perform with realistic applications. Moreover, the multi-periodic nature of the two applications allows us to evaluate the heuristics with new metrics.

1 Synthetic tasks

The evaluation of the multi-phase model against the traditional single-phase model requires to test a large set of profiles with different characteristics and shapes that we could encounter in practice. In order to build such a sample of profiles we employ synthetic multi-phase profiles generators.

1.1 Generation of multi-phase tasks systems

Each task system is generated from some characteristics of the multi-phase profiles that we identified as having a potential a impact on the results:

- The number of tasks in the system.
- The number of phases in the profiles.
- The number of accesses per cycle.
- The temporal shape of the profile, describing how the duration of each phase is chosen.
- The accesses shape of the profile which, similarly to the temporal shape, defines how the accesses are distributed in the phases.
- The ratio between the average duration of short and long phases that indicates how many short phases may be scheduled in parallel to a long phase in average.
- The proportion of empty phases (i.e. phases without any accesses) in the profile. Indeed, the presence of phases without accesses in profiles is expected in practice e.g. due to cache effects, and is likely to benefit the multi-phase model as discussed in Chapter 4.
- The access over-approximation rate as described in Property 4 of Chapter 3. It is applied at the task-level, so we do not know which phases in a profile are responsible for the over-approximation. With this parameter, we are able to observe the influence of the precision of the multi-phase profiles on the constructed schedule.

The creation of a task system is a two step process. Firstly, the system generator creates the tasks with their own number of phases such that each task composing the system may have a different number of phases (and WCET), but in average the number of phases is equal to the input value. Then, the tasks system generator calls the profiles generator to create the phases of each task. The number of accesses and the duration of each phase are assigned by the generator using pre-defined policies that rely on random probability laws. They represent different ways of shaping a profile according to how the duration and the number of accesses are assigned to the phases. We think that these generation policies can influence the results of the scheduling and interference analyses. However, they may not allow to represent all the possible profile shapes that we may encounter in practice.

There are 2 types of temporal shapes that are defined by the probability laws used to draw the durations of the sequence of phases :

1. Normal (N): each duration is drawn from a single normal law, with a unique average value and a standard deviation defined such that the durations are around the same value but may also be quite different. Two examples of profiles generated with the Normal policy are given in Figure 7.1.
2. Bi-Normal (BN): the durations are drawn from two distinct normal laws that have a different average value such that there are short and long phases. Moreover, a long phase is systematically followed by a short phase while a short phase can be followed by another short phase. Figure 7.2 shows 2 profiles generated with this policy.

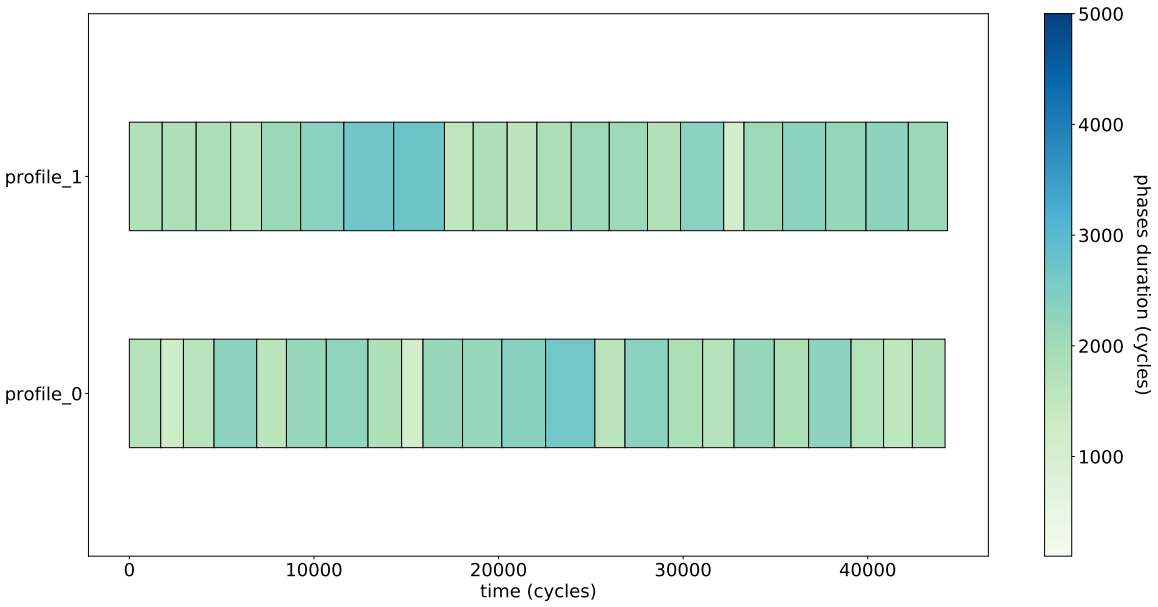


Figure 7.1: Two profiles generated with the Normal duration policy.

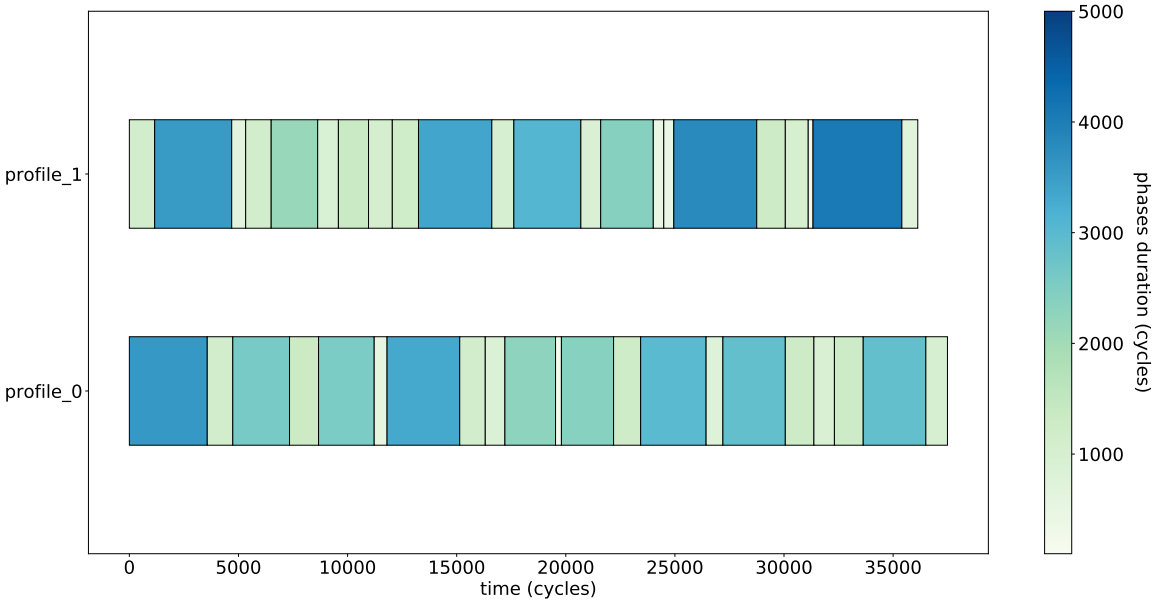


Figure 7.2: Two profiles generated with the Bi-Normal duration policy.

The Bi-Normal type may correspond to profiles that successfully packed clusters of accesses in some phases or, on the contrary, that feature intervals with few accesses. With the Normal type, such a situation may also appear since there are phases that are shorter than others. However, the difference between the durations is smaller, as if the profile was describing the task execution with less precision.

Similarly, concerning the accesses shape, we defined 3 manners to distribute the accesses into the phases corresponding to 3 different access shape policies:

1. Normal (N): the access rate of phases are drawn from a unique normal law, centered on the average access rate of the task given as input to the generator. Therefore, two phases with the same duration have approximately the same number of accesses.

2. Uniform (U): given the average access rate and the durations of the phases, the total number of accesses in the profile is computed and then the number of accesses in each phase is drawn from an uniform law such that the sum equals the total number of accesses. Hence, two phases with very different durations may have the same number of accesses.

3. Beta-Uniform (βU): same method as Uniform, but the process is separated for short and long phases such that the average access rate in short phases is equal to β times the average access rate in long phases. Note that it does not directly rule the number of accesses in the phases. For example, if $\beta = 2$, there is not necessarily twice as many accesses in total in short phases as in long phases, because long phases generally occupy a greater proportion of the WCET of the task.

In the following, the shape of a synthetic profile is defined by the combination of its temporal and its access shape type used by the generators to create it. This shape is then designated by a pair *temporal_shape_acronym*+*access_shape_acronym* (e.g. BU+U).

The generation of a profile begins by the generation of a list of durations using the temporal shape type. Then, the number of accesses in the phases is chosen according to the access rate of the task, its WCET and the access shape type. Once the duration and accesses of each phase have been chosen, a correction pass is performed to ensure that, for any phase, the sum of the duration of its accesses does not exceed its duration. Otherwise, some accesses are given to other phases until this condition is fulfilled. If the constraint is still not respected, the duration of the phase with too many accesses is increased.

The βU access shape requires to compute and affect separately the accesses of short and long phases, so the variables dedicated to computing the number of accesses are duplicated.

In order to compare the multi-phase and single-phase models, the multi-phase profiles are converted to their single-phase equivalent by summing the duration and the number of accesses in each phase. Then, the number of accesses with the single-phase model is adjusted according to the access over-approximation parameter.

1.2 Generation of tasks dependencies

In the experiments using synthetic systems, we rely on single-periodic tasks that are released synchronously at date 0 among all cores. The task dependencies are defined by a DAG. Therefore, for each test, we generated a DAG with the same method as for the experiments of Chapter 5.

Table 7.1: Description of the tests input parameters.

Parameter	Description
Number of cores	the number of cores available in the architecture.
Access cost	cost of an access to the main memory without interference in cycles.
Penalty factor	the multiplier applied to the access cost to compute the interference time penalty.
Temporal shape	the policy to generate the durations of the phases.
Access shape	the policy to generate the number of accesses into the phases.
Over-approximation	the proportion of additional accesses into the multi-phase representation compared to the single-phase one in %.
Number of tasks	the number of tasks to schedule.
Number of phases	the average number of phases per task.
Access rate	the average number of accesses performed within a interval of 10 000 cycles.
β	the ratio between the average number of accesses in short and long phases (only used for the βU accesses policy).
Empty phases	the proportion of empty phases, i.e. phases that do not perform any accesses, composing each profile in average (%).
(p_{ser}, p_{par})	the couples of probabilities used to generate tasks dependencies (DAG), $(0, 0)$ means no dependencies.

1.3 Tests parameters and metrics

1.3.1 Parameters

The parameters defining each test and their description are listed in Table 7.1. For each experiment, the same table will be presented with the pool of values of the parameters that were used to build the task systems. The values of parameters that characterize a profile have been chosen after analyzing the profiles obtained with real applications such as those studied in the case studies experiment in Section 2.

We use the penalty factor parameter to tune the cost of contentions as a multiple of the cost of an access in isolation. Indeed, the memory latency of an access in the presence of interference can be several times the cost of an access in isolation due to indirect effects, e.g. in the pipeline [85]. Setting the penalty factor to 1 is therefore the most optimistic assumption for an architecture, which usually is an unfavorable assumption for our experiments. Hence, we conduct our experiments with a penalty factor of 1 and 3, which correspond respectively to an optimistic and a more realistic assumption.

In all the tests, the BN temporal shape has been obtained with short phases being in average 3 times shorter than long phases. Preliminary experiments demonstrated that increasing this factor (up to 6) did not influence the results, so we focused on other parameters.

Some combinations of parameters are impossible, such as for the β parameter that can only be specified when using the βU policy (but still measurable for any profile generated with the BN policy), or the ratio between long and short durations that is not available with the N temporal shape. Each feasible combination of the parameters is tested 5 times to obtain the complete set of tests.

1.3.2 Tests execution and metrics

The execution of a test consists in three steps. First, we generate the set of multi-phase profiles corresponding to the input parameters. Second, we schedule the multi-phase tasks and their equivalent single-phase counterpart. Then, we perform an interference analysis using function *computeContentions* (introduced in Chapter 6 Section 3) on both schedules.

The objective of the scheduler in each test case is to minimize the makespan of the schedule as defined by the objective equation of the ILP formulation. The metrics used in this section are:

- Makespan: the makespan of the schedule in the presence of interference, i.e. the date from which all the tasks of the system have been executed.
- Contentions: the worst-case number of contentions in the whole schedule, computed by the interference analysis.

For each metric m presented, we define the notion of *gain* comparing the m value in a given schedule to the m value in a baseline schedule:

$$gain = (m\_value\_baseline - m\_value\_schedule)/m\_value\_baseline.$$

Moreover, we call a positive test with respect to a metric m and a baseline a test for which $gain \geq 0$.

1.4 Comparison with optimal multi-phase

In this section, we use the ILP formulation of Chapter 6 Section 2 to schedule synthetic task systems on 2 or 4 cores with both the multi-phase and single-phase models. Thanks to the ILP we obtain optimal schedules that are used to show the potential of the multi-phase model to reduce the over-estimation of the makespan after the interference analysis. We also compare the optimal schedules with those obtained by applying the scheduling heuristics defined in the previous chapter.

As our problem does not scale well using the ILP, we generated a sample with small task systems and with a few phases per task. The number of phase per task is constant for this sample (it is not only an average value among the tasks) due to the small size of the systems generated. In half of the tests, the tasks have no dependencies so the ILP has full freedom to place the tasks at its advantage. The other parameters and their values are given in Table 7.2.

Table 7.2: Tests input parameters.

Parameter	Description
Nb cores	{2, 4}
Access cost	{50} cycles
Penalty factor	{1, 3}
Temporal shape	{N, BN}
Access shape	{N, U, βU}
Over-approximation	{0} %
Nb tasks	{4, 5, 6}
Nb phases	{4, 5, 6}
Access rate	{25, 50, 75} accesses per 10k cycles
Ratio long / short	{1, 3}
β	{1.0, 1.5, 2.0}
Empty phases	{0, 20} %
(p_{ser}, p_{par})	{(0, 0), (0.3, 0.7)}

We used the academic version of Gurobi 9.5.1 [86] to solve the ILP formula. The computation of contentions greatly degrades the solving time of the ILP, so we set a timeout to 6 hours. Consequently, 4521 systems have been successfully scheduled with the multi-phase model which represents 76% of the tests attempted. With 2 cores, the proportion of timeouts is 11% but it goes up to 43% with 4 cores.

1.4.1 ILP multi-phase vs single-phase

The distribution of the gain values obtained by the ILP multi-phase compared to the ILP single-phase is represented in Figure 7.3. The extreme values are not represented for readability reasons: the gain varies from -66.49% to 69.38% and the average value is 9.42%. Moreover, 96.19% of the results of the multi-phase ILP were positive, i.e. at least as good as the single-phase ILP. As these results have been obtained with only small instances of the problem, we cannot draw general conclusions. However, we note that the gap between the two models tends to increase with the number of cores since the experiments with 2 cores have an average gain of 8.88% while it is 10.85% for the tests with 4 cores.

1.4.2 ILP multi-phase vs heuristics

Table 7.3 shows the average gain and the proportion of results where each heuristic applied with the multi-phase model was at least as good as the ILP solving respectively the multi-phase and the single-phase model. All the heuristics are in average less than 10% worse than the optimal multi-phase result and IPH is only 3.71% worse on 2 cores and 5.02% worse on 4 cores. When the merging optimization was used, ASAP and SDE were even able to beat the multi-phase ILP (in respectively 2.5% and 3.3% of the tests) because of the new profiles generated by the optimization. We do not propose a version of the ILP with possible merges because this would considerably increase its complexity and solving time. We also see that our heuristics applied on multi-phase profiles are at least as good as the optimal solution for the equivalent 1-phase profiles in more than 63% of the experiments (up to 90.51% for IPH with 2 cores). Finally, IPH finds the optimal multi-phase schedule in 7.31% and 1.29% of these (simple) experiments respectively on 2 et 4 cores. A

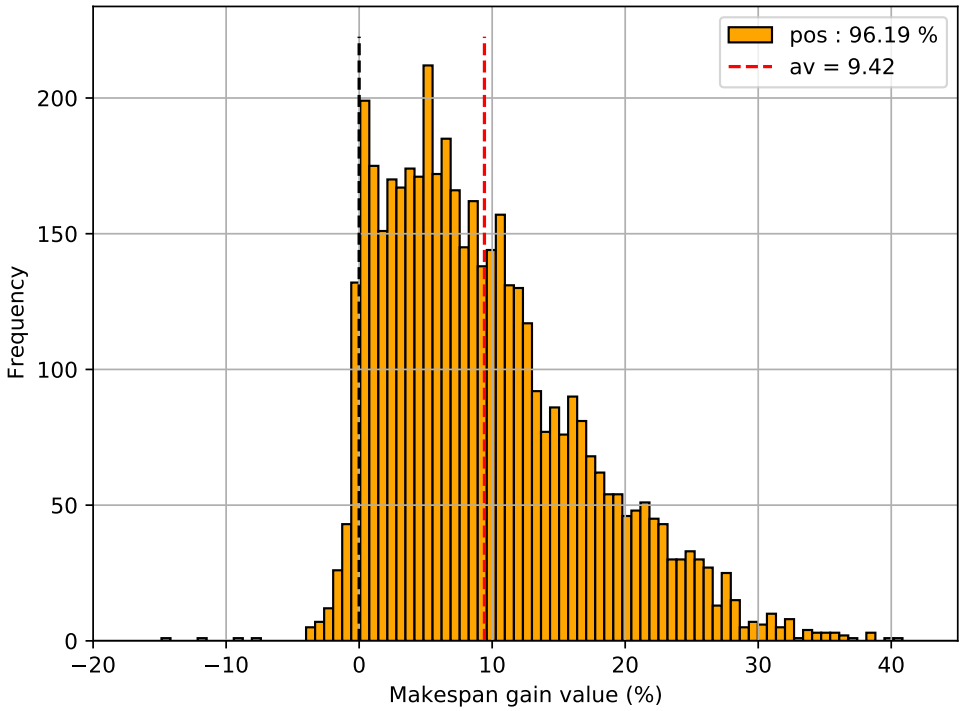


Figure 7.3: Makespan gain of multi-phase ILP vs single-phase ILP. in %

Table 7.3: Share of positive results (gain makespan ≥ 0) and average makespan gain value for all heuristics with multi-phase compared to ILP with multi-phase or single-phase

		Gain vs ILP multi		Gain vs ILP single	
cores	heuristic	share pos. (%)	av. gain (%)	share pos. (%)	av. gain (%)
2	IPH	7.31	-3.71	90.51	5.17
	SDE	2.70	-5.69	73.77	3.20
	SDE + merge	6.34	-5.05	77.47	3.83
	ASAP	2.15	-7.77	63.77	1.11
	ASAP + merge	5.15	-6.76	70.50	2.12
	MCTS	5.12	-4.12	88.84	4.78
4	IPH	1.29	-5.02	88.06	5.82
	SDE	1.37	-5.77	81.29	5.07
	SDE + merge	3.06	-5.43	83.87	5.42
	ASAP	0.56	-9.35	64.84	1.50
	ASAP + merge	1.69	-8.42	70.40	2.42
	MCTS	1.58	-5.58	86.69	5.27

hierarchy between the heuristics is emerging but the following tests on a larger and more complex experimental sample will allow to discuss this hierarchy in more details.

1.5 Heuristics evaluation

In this experiment, we use the scheduling heuristics on synthetic tasks systems with more tasks and phases than what was permitted by the ILP. The parameters of the experiments are given in Table 7.4: there are 20 or 25 tasks per test and the tasks have either 15 or 20 phases in average. In total, 29,500 tasks systems have been generated and used in this sample. The tasks are scheduled

Table 7.4: Tests input parameters.

Parameter	Description
Nb cores	{2, 4}
Access cost	{50} cycles
Penalty factor	{1, 3}
Temporal shape	{N, BN}
Access shape	{N, U, βU}
Over-approximation	{0, 5, 10, 15, 20, 25, 30} %
Nb tasks	{20, 25}
Nb phases	{15, 20}
access rate	{25, 50, 75}
β	{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0}
Empty phases	{0, 20} %
(p_{ser}, p_{par})	{(0.3, 0.7)}

with ASAP, SDE+merge, IPH and MCTS using the multi-phase model and then using the single-phase model equivalent (taking into account the over-approximation) with ASAP and IPH. The merge optimization is only applied to SDE because it has the greatest effect on this heuristic, but it is too costly in terms of computation time to apply to all the heuristics.

The objective is to compare the efficiency of the heuristics according to different parameters, and also to see if one profile shape is better than another according to our tests.

First, the presentation of the results is focused on the differences between the profiles shapes (duration and accesses) in the next Section 1.5.1. The other aspects are presented without distinguishing the shapes in the subsequent sections.

1.5.1 Shapes evaluation

In this section, the tests are used to understand how the shape of a profile can influence the results of the interference analysis. This shape is controlled by the policy used to generate the durations and accesses in each synthetic tasks system, and each combination of the two policies offers a unique way to characterize a profile. Tables 7.5 and 7.6 show respectively the average proportion of positive tests and the average gain value in terms of makespan for all the possible pairs of *duration+access* shapes.

Note that the results of all the scheduling heuristics are taken into account here but the results and their interpretation also hold when they are taken separately.

Table 7.5: Share of positive tests regarding the makespan for the different profile shapes compared to single-phase IPH

cores	max ov-app	share of positive tests ($gain \geq 0$)				
		Dur. BN			Dur. N	
(#)	(%)	Acc. N	Acc. U	Acc. βU	Acc. N	Acc. U
2	0	63.31	83.19	82.32	66.38	82.17
	5	54.13	81.32	79.77	59.53	79.83
	10	45.37	79.03	77.58	55.23	77.43
4	0	73.89	88.46	88.06	78.61	89.72
	5	60.37	87.53	85.17	69.10	86.18
	10	47.96	83.46	81.72	61.53	83.10

Sample BN temporal type When using the BN temporal policy, profiles are composed of long phases followed by groups of short phases (see the two examples in Figure 7.2). With this shape, short phases are easily scheduled with only one phase in parallel for each other core.

Table 7.6: Average makespan gain value for the different profile shapes compared to single-phase IPH

cores	max ov-app	average gain (%)				
		Dur. BN			Dur. N	
(#)	(%)	Acc. N	Acc. U	Acc. βU	Acc. N	Acc. U
2	0	0.45	4.03	3.69	1.73	3.92
	5	-0.02	3.59	3.23	1.19	3.47
	10	-0.50	3.18	2.85	0.78	3.05
4	0	0.82	6.15	5.32	2.97	5.49
	5	0.12	5.34	4.50	2.10	4.62
	10	-0.67	4.55	3.78	1.41	3.92

From Tables 7.5 and 7.6, we observe that the U accesses type is the more adapted to the BN temporal type. Indeed, with the N access type, short phases generally have less accesses than the long ones, so even if they have only one contender per core in parallel (often a long phase), all their accesses may create contentions. Long phases have chances to create less contentions than their number of accesses when they are scheduled in parallel with a task that has a lower access rate.

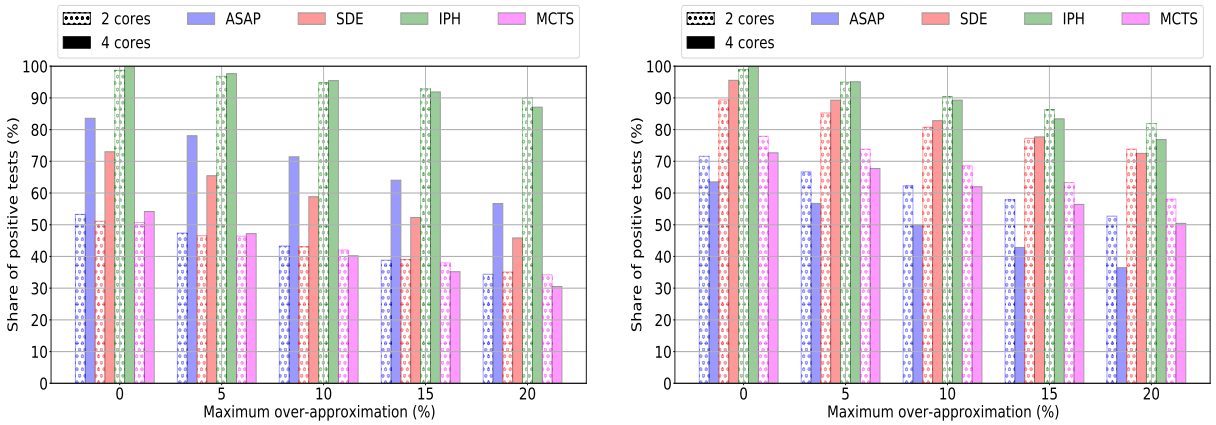
On the contrary, with the U access type, phases are often scheduled in parallel with other phases that have less accesses whatever if they are short or long (even if the access rates of the tasks scheduled in parallel are similar). Therefore, the possibilities to avoid contentions are more frequent which explains the difference in the results that we obtained.

To sum up, our tests suggest that a profile with an alternation between long and groups of short phases is not efficient when the short phases have less accesses than the long ones (as with the N Access Policy). Indeed, in this case, most of the short phases create contentions with all their accesses because the phases in parallel have approximately the same number of accesses in general. Therefore, a profile design method that creates a profile that resembles the BN temporal shape must ensure that some short phases have more accesses than long ones to "save" some accesses from suffering contentions. Using the βU temporal type, Section 1.5.2 studies how the gain evolves when the access rate of short phases is higher than that of long phases.

Sample N temporal type The N temporal type creates phases without considering short or long durations (see Figure 7.1). Although a difference between the durations may appear locally, there is less chances that a phase is in parallel with only one other phase which entirely covers it.

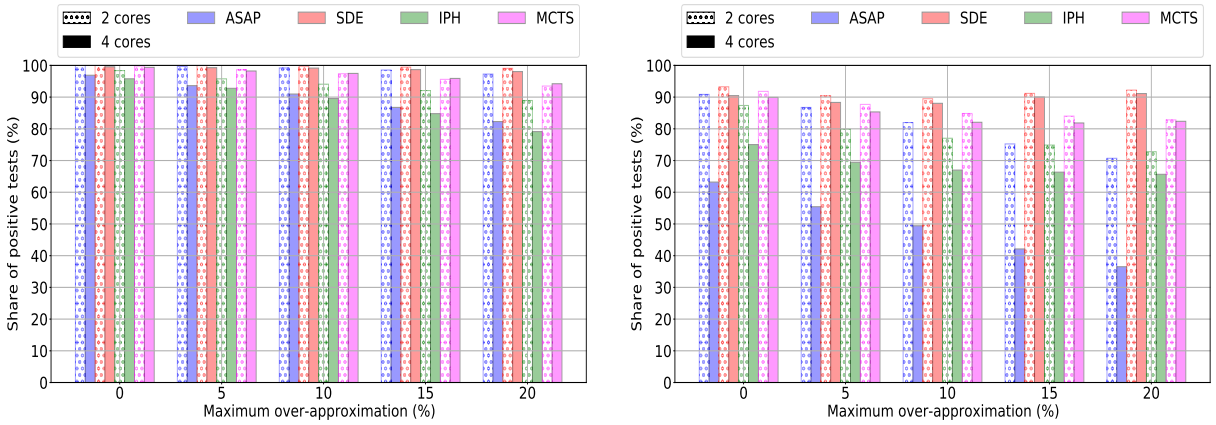
As for the BN temporal type, the results of Tables 7.5 and 7.6 indicate that the U access type outperforms the N one. In the case of the N access policy, the phases have a similar number of accesses because their duration is also similar. Therefore, the phases generally create as many contentions as their number of accesses so the results of the interference analysis are close to those obtained with the single-phase model. A method that creates phases with similar durations should then ensure that there are enough differences between the number of accesses in the phases to create profitable situations for the multi-phase model.

Conclusion From our investigation on the shapes we can deduce that, regardless of the variability of the phases duration (i.e. the temporal shape), a method should design the phases so that they have a very different number of accesses. Indeed, it favors situations where the phases with the highest number of accesses cannot create contentions with all their accesses while these contentions would exist using the single-phase model (depending on the access over-approximation). We can also observe in Tables 7.5 and 7.6 that the results of BN+U and N+U are close to each other, even if the first slightly outperforms the second in most of the cases.



(a) penalty = 50 cycles (b) penalty = 150 cycles

Figure 7.4: Share of positive results in terms of makespan according to the access over-approximation for 2 interference penalty values compared to single-phase IPH.



(a) penalty = 50 cycles (b) penalty = 150 cycles

Figure 7.5: Share of positive results in terms of contentions according to the access over-approximation for 2 interference penalty values compared to single-phase IPH.

1.5.2 Influence of the other parameters on the efficiency of the multi-phase model

Influence of the target architecture: interference penalty and number of cores The effects of interference can be increased by varying the interference penalty and the number of cores. The latter impacts the number of possible concurrent memory accesses in the interconnect. When an additional core is added, a memory access can be delayed by an additional interference penalty.

Regarding the makespan gain (Figure 7.4), we observe that the hierarchy between the schedulers changes according to the interference penalty. In Figure 7.4a with an interference penalty of 50 cycles, the order of efficiency is IPH, ASAP, SDE and MCTS even if with 2 cores, SDE slightly outperforms ASAP when the access over-approximation rate is superior to 10%. However, when the interference penalty is set to 150 cycles (i.e. the most realistic assumption), SDE and MCTS are better than ASAP. We observe that SDE is even at the same level as IPH with a low access over-approximation.

The gain in contentions depicted by Figure 7.5 shows that IPH is the heuristic with the least gain in contentions, except with 4 cores and a penalty of 150 cycles where this is ASAP. The best heuristic for this metric is SDE, followed by MCTS. These results demonstrate that the gain in contentions is not directly correlated to the makespan gain, particularly when the effects of

interference are limited. For SDE, the interference analysis is performed each time a task is added so it is actually based on the new contentions that the task may create on the schedule. The impact on the makespan for the whole schedule is not known or considered. Therefore, the tasks that have been postponed to reduce the number of contentions locally may create too much slack time and indirectly degrade the overall makespan.

The average makespan gain is represented by Figures 7.6 and 7.7 respectively against single-phase ASAP and IPH. We can make the same observations as with the share of positive results: SDE and MCTS are the least efficient heuristics when the penalty is 50 cycles but their gain is improving when the penalty is increased. With a 150 cycles penalty, the gain of the multi-phase

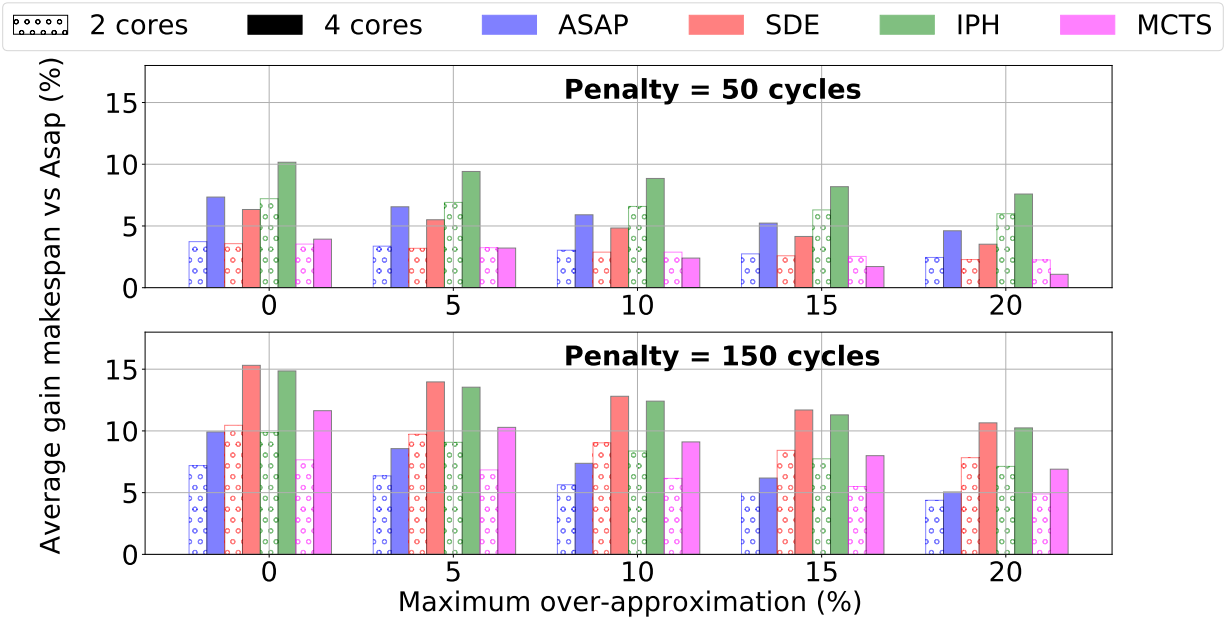


Figure 7.6: Average makespan gain vs ASAP single-phase according to the access over-approximation.

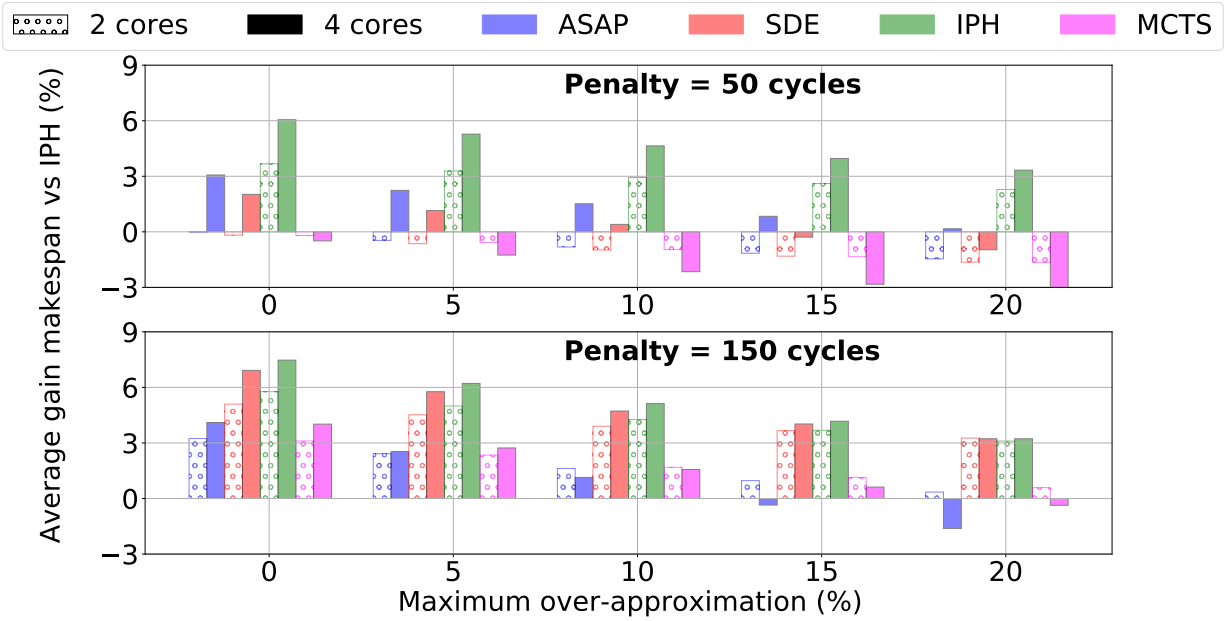


Figure 7.7: Average makespan gain vs IPH single-phase according to the access over-approximation.

reaches 15.86% using IPH against single-phase ASAP, while the maximum is 7.47% against single-phase IPH.

In a nutshell, IPH is the most adapted to reduce the makespan of the task systems while SDE is the most efficient to reduce contentions. The reason is that SDE tends to take short-term decisions that mainly reduce the contentions. However, it is sometimes better to accept more contentions locally to reduce the makespan of the entire system. When the effects of contentions are more important (i.e. a higher number of cores or a greater interference penalty), avoiding contentions is more correlated to reducing the makespan of the schedule so SDE becomes more efficient to reduce the makespan.

Influence of the tasks system characteristics

Table 7.7: Share of positive results and average gain value considering the makespan for all heuristics with multi-phase compared to ASAP single-phase according to the access rate.

acc. rate (# per 10k cycles)	share positive results (%)				average gain (%)			
	ASAP	SDE	IPH	MCTS	ASAP	SDE	IPH	MCTS
25	84.32	71.38	93.76	62.81	2.97	2.38	6.16	1.24
50	80.00	81.89	88.93	67.63	3.26	4.77	6.03	2.28
75	71.46	88.97	92.60	74.33	2.41	8.37	7.58	3.86

Influence of the access rate Table 7.7 compares the results of the schedulers according to the access rate of the tasks systems (whatever the over-approximation levels). The results show that SDE is particularly efficient to manage task systems with memory intensive tasks. MCTS also yields better results when the access rate increases. For the two other heuristics the results are not as clear, the share of positive results decreases when the access rate increases for ASAP but not necessarily for IPH. There is no tendency either regarding the average gain for these two heuristics.

These results recall those of the gain in terms of contentions depicted in Figure 7.5 where we already observed that SDE and MCTS were the most capable of reducing contentions in a schedule. When the access rate increases, the gains in makespan and in contentions are more correlated because the timing penalties due to contentions in the whole schedule can represent a higher proportion of the overall cores occupancy after the interference analysis.

Table 7.8: Share of positive results and average gain value considering the makespan for all heuristics with multi-phase compared to ASAP single-phase according to the proportion of empty phases.

empty phases (%)	share positive results (%)				average gain (%)			
	ASAP	SDE	IPH	MCTS	ASAP	SDE	IPH	MCTS
0	71.87	74.05	88.80	61.85	1.93	3.78	5.41	1.44
20	85.32	87.44	94.72	74.66	3.83	6.57	7.77	3.48

Influence of the empty phases Table 7.8 compares the results of the schedulers according to the presence of empty phases (20%) or not. IPH is already achieving almost 90% of positive tests without empty phases so their presence increases the rate by 6 points. For the other heuristics, the increase is between 13 and 14 points approximately. The average gain also increases significantly in the presence of empty phases, and SDE has the greatest improvement.

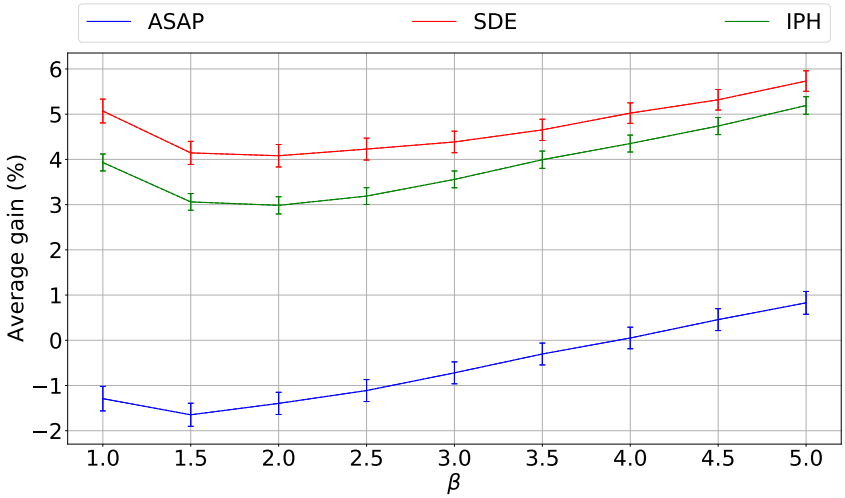


Figure 7.8: Makespan gain vs IPH single-phase (%) given the input β value (penalty = 150 cycles).

Influence of β Figure 7.8 shows the average gain of the schedulers compared to the IPH single-phase schedule according to the β value (only for the BN+β-U tests with a penalty of 150 cycles). We also plotted for each average gain the error bar with a 95% confidence interval.

We observe an important decrease from $\beta = 1.0$ to $\beta = 1.5$ and then the average gain increases with β from $\beta = 2.0$ with ASAP and from $\beta = 2.5$ for SDE and IPH. In order to explain these results, we must keep in mind that β is applied to the access rate but not directly to the number of accesses in short and long phases, and that the overall duration of short phases represents a minor proportion of the WCET of the generated tasks. Therefore, when $\beta = 1.0$, long phases have much more accesses than short phases and it is frequent that the accesses of these long phases are not all creating contentions. However, when β increases a bit, short phases have a total number of accesses still inferior but closer to the number of accesses in long phases, so there is less chances that long phases cannot create contentions with all their accesses. Once the accesses in short phases is close to the accesses in long phases, the short phases can in turn create less contentions than their number of accesses so the results are improving.

In the BN+U tests, where β is not considered to choose the number of accesses in the phases, the β value measured was varying between 1.47 and 7.92 with an average value of 2.86.

1.5.3 Computation time

Table 7.9: Average computation time for the different heuristics

tasks (#)	phases per task (#)	Average computation time (s) ASAP	SDE	IPH	MCTS
20	15	< 1	55	334	359
	20	< 1	112	499	533
25	15	< 1	88	596	688
	20	< 1	172	911	945
	all tests	< 1	105	574	620

Table 7.9 gives the average computation time of the heuristics according to the number of tasks in the system and the average number of phases. When comparing the difference between ASAP and the others, we see that performing interference analyses is very expensive. Moreover, we see that, as opposed to MCTS and IPH, SDE is more sensitive to the number of phases than the

number of tasks because the number of interference analyses that it performs to schedule each task depends on the number of phases in parallel. On the contrary, MCTS and IPH explore tasks orderings or mappings so the number of interference analyses to perform depends on the number of tasks.

2 Case studies : Rosace and Papabench

This section proposes to apply the scheduling heuristics on two case studies: Rosace [82], the multi-periodic flight controller already used in Chapter 5, and Papabench [87] that is derived from an open-source UAV control application.

2.1 Tasks profiles

The first activity consisted in producing multi-phase profiles. As for the experiments of Chapter 5, for the static analysis of the code we considered a target hardware architecture composed of an ARM-based multi-core processor (2, 3 or 4 cores) in which each core features a L1 LRU data cache, and an instruction scratchpad which holds the totality of the code needed by the core to execute. The memory latency is still 50 cycles for non-cached accesses.

The profiles and the task system of Rosace are the same as those used in Chapter 5. The design of profiles for Papabench was more difficult because some of its tasks have many traces due to consecutive conditional structures with several cases. In order to reduce the complexity of the analyses, some of the original tasks have been split to ease the profiles generation, so Papabench (resp. Rosace) is composed of 329 tasks (resp. 77 tasks). The two tasks with the most traces for Papabench have respectively 62502 and 153013 traces which makes the computation of the worst-case number of accesses in their phases and the selection of synchronizations very expensive computationally. Therefore, the GA techniques presented in Chapter 4 were too costly for the two tasks with the most traces even with $\delta = 1000$ cycles and we do not present results with these profiles in this section. The results are firstly presented with the TIPs profiles that yielded the best results but we also propose some results with KDE (Table 7.13) to provide some comparison points.

2.2 DAG scheduling

In the first part of the experiments, both benchmarks have been converted from multi-periodic task systems to DAGs of single-period tasks, following the methodology of [83] as in chapter 5: we consider each job for one hyperperiod of the system as a separate task, but we do not use release dates and consider only the activation rhythms.

2.2.1 Results

The two benchmarks have been scheduled with IPH, SDE (+ merge) and ASAP on 2, 3 and 4 cores. MCTS has not been used due to the high number of tasks to schedule in both the case

Table 7.10: Statistics of Papabench profiles according to the employed design method.

δ	KDE sync (#)	KDE phases (#)	KDE ov-app (%)	TIPs sync (#)	TIPs phases (#)	TIPs ov-app (%)
1000	6 225	3 323	10.47	5 622	2 755	4.01
500	8 460	5 573	15.10	12 937	4 788	4.73
200	13 101	10 607	31.60	24 891	9 009	7.45

Table 7.11: Results of heuristics to schedule Papabench tasks with TIPs profile.

			penalty = 50 cycles			**penalty = 150 cycles**		
	nb cores	δ	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP
ASAP	2	1000	**7.17**	-0.64	8.32	9.15	3.37	17.58
		500	8.31	0.58	18.37	10.90	5.23	25.72
		200	9.07	1.41	24.71	**11.93**	6.33	28.96
	3	1000	4.81	1.18	14.18	6.13	1.60	7.32
		500	6.37	2.88	22.35	7.94	3.49	17.75
		200	6.96	3.41	23.68	9.00	4.61	13.18
	4	1000	**4.66**	1.25	19.85	**5.13**	1.41	13.96
		500	6.24	2.88	25.62	7.97	4.36	18.16
		200	6.84	3.51	27.43	9.01	5.44	19.61
SDE + merge	2	1000	-2.36	-10.98	47.13	9.94	4.21	57.06
		500	**0.96**	-7.38	54.73	**13.61**	8.11	64.70
		200	0.75	-7.61	40.59	12.85	7.31	62.83
	3	1000	**-5.05**	-9.06	30.50	**7.44**	2.98	59.58
		500	-1.39	-5.26	37.88	12.03	7.79	64.45
		200	-1.02	-4.86	38.46	9.88	5.53	68.35
IPH	2	1000	12.86	5.52	14.25	13.02	7.49	25.28
		500	14.18	6.95	25.85	**16.31**	10.99	35.08
		200	**14.97**	7.81	31.29	16.12	10.79	38.56
	3	1000	8.88	5.56	30.97	**10.96**	6.67	38.59
		500	9.98	7.36	39.56	14.91	10.80	37.37
		200	9.59	6.18	35.28	14.51	10.38	49.37
	4	1000	**7.80**	4.28	42.53	11.62	8.15	42.51
		500	9.42	6.26	44.13	13.83	10.44	47.20
		200	9.63	6.65	45.61	14.52	11.16	47.74

studies that is too important. Indeed, the algorithm could not enumerate and store all the possible branches of the tree to explore without crashing due to an insufficient RAM capacity. Only the best ready tasks sorting policy in terms of makespan gain is listed for SDE and ASAP for readability. We scheduled the 1-phase model with both ASAP and IPH as IPH tends to perform better than ASAP. Moreover, as for the previous experiments, we used both a 50 cycles and 150 cycles interference penalty to represent respectively the most optimistic scenario and a more realistic one.

The results for the TIPs profiles are presented in Table 7.11 for Papabench and in Table 7.12 for Rosace. A first observation is that the multi-phase model globally yields better results than the 1-phase model, with a makespan gain up to 16.31% for Papabench (IPH on 2 cores with $\delta = 500$ cycles and a penalty of 150 cycles) and 24.00% for Rosace (SDE on 4 cores with $\delta = 200$ cycles and a penalty of 150 cycles).

For Papabench, IPH always performs the best improvements from 7% to 16% compared to the 1-phase ASAP and between 4% and 11% compared to the 1-phase IPH. When the penalty is 50 cycles, SDE is the worst heuristic and its makespan is often higher than if the tasks are represented with the single-phase model (i.e. $gain < 0$). However, with a 150 cycles penalty per contention, SDE is more efficient than ASAP with a gain ranging from nearly 7% to 13% for the makespan. For Rosace, SDE is more efficient because the gain is always positive and often close to ASAP with 50 cycles of penalty, and it is even the best heuristic when the penalty is 150 cycles. Decreasing the value of δ generally yields better results when using ASAP or SDE but this is not systematic. For example, with Papabench the results with $\delta = 500$ cycles is the best for IPH on 3 cores. This may be partially due to the significant gap of the access over-approximation rate between $\delta = 500$ and $\delta = 200$ cycles which rises from 4.87% to 7.45% according to Table 7.10.

The two tables also display the gain in terms of contentions. For Papabench (resp. Rosace), this gain ranges from 6.92% to 64.36% (resp. 2.42% to 55.80%) compared to 1-phase ASAP scheduling.

Table 7.12: Results of heuristics to schedule Rosace tasks with TIPs profile.

			penalty = 50 cycles			**penalty = 150 cycles**		
	nb cores	δ	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP
ASAP	2	1000	**2.42**	0.76	9.03	**2.31**	1.28	3.80
		500	3.26	1.10	13.82	5.86	4.87	11.83
		200	4.90	2.77	22.35	9.15	8.19	18.93
	3	1000	4.71	-0.04	2.42	5.01	2.83	3.98
		500	6.96	2.32	9.79	6.48	4.33	7.12
		200	8.71	4.16	14.28	8.65	6.56	9.88
	4	1000	11.18	3.17	5.94	13.23	0.83	9.70
		500	13.65	5.86	11.59	14.89	2.73	13.22
		200	**15.78**	8.18	17.15	**16.80**	4.92	16.76
SDE + merge	2	1000	**0.90**	-1.32	15.82	10.75	9.81	39.66
		500	3.78	1.63	31.20	13.28	12.36	55.20
		200	2.85	0.67	37.11	17.04	16.17	50.69
	3	1000	1.11	-3.82	10.79	**9.39**	7.31	51.99
		500	5.17	0.44	23.82	12.34	10.32	44.12
		200	7.64	3.03	26.23	17.55	15.65	44.12
	4	1000	7.37	-0.99	14.58	20.36	8.98	55.80
		500	13.87	6.09	20.88	20.54	9.19	35.48
		200	**15.54**	7.92	26.31	**24.00**	13.14	40.11
IPH	2	1000	**4.26**	2.12	7.10	4.87	3.87	10.21
		500	5.70	3.59	14.32	9.20	8.24	21.86
		200	7.22	5.14	21.48	11.79	10.86	25.28
	3	1000	6.64	1.98	3.71	**5.19**	3.01	34.67
		500	9.05	4.52	10.54	8.95	6.86	24.95
		200	10.64	6.18	17.09	11.32	9.28	27.49
	4	1000	14.68	6.98	0.27	15.56	3.50	23.46
		500	17.35	9.90	7.27	17.82	6.08	45.18
		200	**19.04**	11.73	9.90	**19.69**	8.21	48.58

This means that on top of reducing the makespan of the computed schedules, our heuristics, coupled with the multi-phase model, are able to significantly improve the timing predictability of the scheduled applications because there is less variability in the number of contentions that may occur in the system (i.e. the maximum interference scenario is closer to the average case scenario). SDE is the best heuristic to reduce contentions, even when it obtains negative makespan gains, which is coherent with what we observed with the synthetic systems. As a recall, we observed that SDE tends to postpone too much the tasks based on local data, which is detrimental for the overall makespan when the cost of contentions is not high enough.

Table 7.13 shows the results of Rosace and Papabench for the KDE profiles, only with a penalty of 150 cycles. As expected following the observation of the comparative study of Chapter 5, the results are not as good as for the TIPs. Still, the gain is important in some cases: it ranges from 2.42% to 14.14% for Rosace and from 4.31% to 15.84% for Papabench.

With $\delta = 1000$ on 2 cores, the time required to schedule Papabench (resp. Rosace) with ASAP was 1 minute (resp. less than 1 second) while this was nearly 8 hours when applying SDE (resp. 43 seconds) and 6 hours (resp. 3 minutes) to run IPH with up to 31 threads (resp. 19) computing a solution at a time.

Table 7.13: Results of heuristics to schedule Rosace and Papabench tasks with KDE profile with a penalty equal to 150 cycles.

			Rosace			**Papabench**		
	nb cores	δ	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP	gain makespan (%) vs ASAP	vs IPH	gain contentions (%) vs ASAP
ASAP	2	1000	**2.73**	1.71	2.93	7.39	2.04	4.81
		500	5.44	4.44	9.34	8.39	3.80	10.04
		200	**6.89**	5.91	14.94	**10.58**	6.28	18.21
	3	1000	4.14	1.94	1.69	**4.31**	-0.31	9.84
		500	6.26	4.11	6.65	5.34	0.77	11.72
		200	5.28	3.10	3.33	7.60	3.14	15.81
SDE + merge	2	1000	**8.49**	7.52	38.36	8.92	3.19	54.91
		500	10.49	9.54	45.76	10.14	4.43	56.38
		200	**14.14**	13.23	46.51	**10.52**	4.83	52.95
	3	1000	9.20	7.11	43.43	**6.62**	2.11	63.14
		500	11.23	9.19	35.86	8.10	3.67	63.34
		200	13.18	11.19	47.91	8.51	4.10	59.09
IPH	2	1000	**3.52**	2.50	18.93	11.35	5.71	31.62
		500	8.12	7.15	13.20	12.90	7.36	28.17
		200	**9.36**	8.40	17.25	**15.84**	10.48	41.27
	3	1000	3.79	1.58	9.22	**10.42**	6.09	46.48
		500	7.05	4.92	20.46	10.71	6.40	30.28
		200	7.55	5.43	33.29	13.54	9.37	51.27

Table 7.14: Makespan gain of Papabench (TIPs profile) with IPH according to the timeout value on 2 cores.

δ	timeout(hours)	gain vs IPH(%)	gain vs ASAP (%)
500	1	0.58	8.31
	2	1.30	8.97
	3	3.80	11.28
	4	3.80	11.28
	5	6.95	14.18
1000	1	3.50	11.41
	2	4.72	12.53
	3	5.08	12.86
	4	5.08	12.86
	5	5.08	12.86

However, as IPH is an iterative heuristic, it is able to find the best result or at least a satisfying result within the early iterations. Table 7.14 shows the gain of Papabench with IPH for different δ and timeout values on 2 cores. We can see that the time to converge to the best solution increases when δ decreases. This is because there are more phases on which the contentions must be computed so the interference analyses take longer to execute (Papabench grows from 2755 to 9009 phases according to Table 7.10). With $\delta = 1000$, the gain quickly reaches the best value displayed in Table 7.11 and it is already close to this value with 2 hours, whereas time has a great impact on the results for $\delta = 500$. Therefore, setting a timeout for IPH can significantly degrade its efficiency especially when the the task set is large because there are more solutions to explore. This issue is mitigated by the fact that we target static scheduling techniques, which are performed

before the system is put in service so their execution time usually is not an issue. However, under certain industrial constraints, computing (or recomputing) a static schedule must be done in a limited period of time (e.g. 24 hours to find a bug, correct it, recompile, reschedule and re-verify the system before it is put in production again). In these particular contexts, a trade-off must be found between the duration of the scheduling algorithm and the quality of the resulting schedule.

2.3 Multi-periodic scheduling

For this second part of the case study, we consider the multi-periodic nature of the tasks (with release dates and deadlines). In this context, the makespan of the system is no longer a relevant metric. However, the gain in terms of contentions, whose effect is sometimes inhibited in the first part of the experiments, may in the contrary exhibit interesting properties of the multi-phase model. Indeed, the main metric employed to evaluate scheduling methods for periodic systems is the schedulability: the system is deemed schedulable if all the tasks end before their deadline which is a requirement for HRT. In the context of multi-core platforms, interference can compromise the schedulability of a periodic system so the reduction of the number of contentions by the multi-phase model may also improve the schedulability of the system.

We only conducted the tests using ASAP and SDE because as discussed previously, IPH is a very slow heuristic while these tests require to compute many schedules (with different processor frequencies). MCTS is not used for the same reason as in the previous part.

2.3.1 Metrics

We propose to use two new metrics in order to assess the performance of the multi-phase model over the single-phase model regarding schedulability:

- the minimum processor frequency from which the system is schedulable called *min frequency*: this metric must be minimized to show that even a processor with a low frequency can run the task system safely.
- the minimum difference between a task end date and its deadline called *min slack to deadline*: on the contrary, the higher this value, the greater the margin before falling into an unsafe schedule.

Table 7.15: Schedulability analysis of Rosace and Papabench for the TIPs profile with an interference penalty of 50 cycles.

			Papabench				**Rosace**			
	nb	δ	min freq (kHz)		min slack to deadline		min freq (kHz)		min slack to deadline	
	cores		single	multi	single	multi	single	multi	single	multi
ASAP	2	1000	8 600	8 200	5 992	7 042	2 100	2 100	860	910
		500		8 150		8 442		2 050		1 610
		200		8 150		7 942		2 000		2 010
	3	1000	8 300	7 950	4 722	15 530	1 800	1 700	888	2 283
		500		7 900		17 316		1 700		2 633
		200		7 900		15 666		1 650		3 583
SDE	2	1000	8 600	8 500	5 992	3 040	2 100	2 050	860	1 497
		500		8 100		13 685		2 050		1 835
		200		8 500		12 125		2 000		2 799
	3	1000	8 300	7 600	4 722	19 313	1 800	1 650	888	2 884
		500		8 000		7 698		1 650		2 984
		200		7 800		19 967		1 600		4 270

Table 7.16: Schedulability analysis of Rosace and Papabench for the KDE profile with an interference penalty of 50 cycles

			Papabench				**Rosace**			
	nb	δ	min freq (kHz)		min slack to deadline		min freq (kHz)		min slack to deadline	
	cores		single	multi	single	multi	single	multi	single	multi
ASAP	2	1000	8 600	8 300	5 992	8 989	2 100	2 050	860	1 610
		500		8 200		10 239		2 050		1 410
		200		8 300		9 739		2 050		1 910
	3	1000	8 300	8 100	4 722	13 055	1 800	1 700	888	2 433
		500		8 000		15 563		1 700		2 483
		200		8 000		15 413		1 700		2 733
SDE	2	1000	8 600	8 300	5 992	9 974	2 100	2 050	860	1 111
		500		8 800		-3 841		2 050		1 766
		200		8 800		-2 944		2 000		2 186
	3	1000	8 300	7 700	4 722	17 336	1 800	1 750	888	1 810
		500		7 800		14 464		1 650		2 999
		200		7 900		1 326		1 600		3 840

These two metrics indicate how effective a method is to provide a schedule with sufficient safety margins, by reducing the overestimation inherent to static timing analyses. The two metrics also provide an indication on the potential energy savings that a method can provide. Indeed, decreasing the operating frequency allows to save energy by reducing the dynamic power consumption of the whole platform.

2.3.2 Results

Tables 7.15, 7.16 and 7.17 show the results of the schedulability analysis of Rosace and Papabench when the interference penalty is respectively once the cost of a memory access in isolation for the two first tables (i.e. the most optimistic assumption for an architecture) and three times this cost for the third (i.e. a more realistic assumption). The minimum slack to deadline is always computed at the minimum frequency obtained with the single-phase model, so a negative value in the multi-phase column means that the multi-phase model was not schedulable at this frequency.

According to Table 7.15, for Papabench on 2 cores ASAP is in average better than SDE regarding the minimum frequency although the best gain for ASAP is 5.23% while it is 5.81% for SDE. However, SDE outperforms ASAP in the minimum slack to deadline observed for $\delta \leq 500$, with a maximum gain of 128% (compared to 41% for ASAP). In the other cases (Rosace included), SDE is generally both better than ASAP regarding the minimum frequency and the minimum slack to deadline with a gain for the min slack to deadline up to 323% for Papabench (on 3 cores and with $\delta = 200$ cycles). Indeed, as a recall, SDE tends to focus on local situations on the schedule which was detrimental when using DAG systems because the local decisions were not beneficial to reduce the overall makespan in particular when the cost of contentions was low. However, with multi-periodic systems, jobs must meet their own deadlines, which can be perceived as local problems where SDE is the most efficient.

Table 7.16 shows the results when the systems are scheduled with KDE profiles instead of TIPs. We also observe the superiority of ASAP to schedule Papabench on 2 cores over SDE, and SDE is even worse than the single-phase ASAP schedule when $\delta < 1000$ cycles. The minimum slack to deadline is then negative for these two cases because one job misses its deadline. However, we also see that SDE is better than ASAP on 3 cores. For Rosace, the results between the two heuristic are similar regarding the minimum frequency with 2 cores: the reduction is only 2.38% and up to 4.76% with SDE when $\delta = 200$ cycles. However, the minimum slack to deadline is better with SDE

Table 7.17: Schedulability analysis of Rosace and Papabench for the TIPs profile with an interference penalty of 150 cycles

			Papabench				**Rosace**			
	nb	δ	min freq (kHz)		min slack to deadline		min freq (kHz)		min slack to deadline	
	cores		single	multi	single	multi	single	multi	single	multi
ASAP	2	1000	11 100	10 650	163	13 492	2 900	2 800	660	1 560
		500		10 450		16 342		2 750		3 360
		200		10 450		16 942		2 750		3 660
	3	1000	11 900	11 500	2 422	22 080	3 050	2 750	888	6 083
		500		7 900		26 716		2 750		6 355
		200		7 900		19 816		2 700		6 505
SDE	2	1000	11 100	10 300	163	25 070	2 900	2 600	660	5 505
		500		10 400		17 754		2 550		6 055
		200		10 300		29 743		2 500		7 054
	3	1000	11 900	9 900	2 422	51 691	3 050	2 650	888	6 296
		500		9 900		51 268		2 550		8 830
		200		10 100		10 655		2 450		8 763

that achieves a gain up to 332% while it is 208% for ASAP (on 3 cores with $\delta = 200$ cycles for both of them).

As observed with the DAG scheduling results, decreasing δ tends to increase the gain of the multi-phase model over the single-phase model, but the correlation seems less important. Indeed, any job may make the system not schedulable so a single task profile that is less efficient at a lower δ value can degrade the gain for all the system. For example, Papabench with the KDE profiles has a lower schedulability than its single-phase counter part when using SDE and with $\delta < 1000$ cycles because of one task misses its deadline while the deadline was respected with $\delta = 1000$ cycles.

Table 7.17 shows the results when the interference penalty is 150 cycles, so three times the cost of an access in isolation. The minimum slack to deadline gain of the multi-phase has greatly increased. For example, the maximum gain for this metric is 18 147% for Papabench on 2 cores with the SDE heuristic ($\delta = 200$) because of the very low value obtained with the single-phase model (163 cycles). Another observation is that, except regarding the minimum frequency for Papabench on 3 cores where ASAP reduces the minimum frequency of 34%, SDE always performs better than ASAP with this penalty value for the two metrics under study. For Rosace, the maximum frequency reduction of SDE is 14% on 2 cores and 20% on 3 cores, and the minimum slack to deadline gain is up to 894% also with SDE (3 cores and $\delta = 200$).

2.3.3 Summary

The results show for these two benchmarks that the multi-phase model is in most of the cases more suited than the single-phase model to schedule a multi-periodic system. Indeed, the minimum processor frequency at which the multi-phase representation is schedulable is generally lower and it offers higher safety margins (i.e. a higher minimum slack to deadline). The difference is more important when the interference penalty is increased.

Moreover, SDE can offer a consequent safety margin because it is good at optimizing local situations as making jobs respecting their deadline.

3 Conclusion

The ILP formulation has severe scalability issues because it accounts for the effects of interference at the phase level. However, it shows the potential of the multi-phase model that is better than the single-phase model in more than 95% of the performed tests. Moreover, the comparison of optimal schedules with the multi-phase or the single-phase model shows that for small systems the makespan gain is near 10% when using the muti-phase model. The heuristics aldo produce better schedules than the optimal single-phase one in average. In around 90% of the tests, IPH obtains at least the same makespan and its average gain is superior to 5%.

Another study on bigger systems showed that the results of SDE and MCTS are dependent on the correlation between the reduction of the number of contentions and of the makespan. Indeed, the two heuristics are the most effective techniques to reduce contentions (in this order) because they take scheduling decisions based on interference analyses performed on partial schedules. Hence, SDE is almost as efficient as IPH to reduce the makespan with a high timing penalty. However, when the effects of contentions are limited by the number of cores and the interference penalty value, the correlation between the gains in makespan and in contentions is weak so SDE and MCTS are the least efficient to reduce the makespan. The overall superiority of IPH seems to indicate that exploring the space of possible tasks ordering is more efficient in terms of makespan reduction than exploring the space of possible start dates. However, as SDE is closer to IPH with a more realistic timing penalty (i.e. superior to the memory latency in isolation), exploring the set of dates is also efficient.

Moreover, the experiments with synthetic systems allows to test the efficiency of different profile shapes. The results indicate that regardless of the variation of the durations of the phases, the profiles where the number of accesses varies greatly from one phase to another yield the best results. Indeed, it favors situations where one phase cannot create contentions with all its accesses. For the same reason, the presence of empty phases in the profiles improves the makespan gain.

Finally, the application of the heuristics on two realistic case studies shows that the multi-phase model can yield substantial gains in makespan and contentions (up to 24% and 64% respectively) compared to the single-phase model. In addition, by applying ASAP and SDE on multi-periodic versions of the case studies, we observe that the multi-phase model can also offer more margins between the end of tasks and their deadlines.

SDE would benefit from an assessment of the impact of its scheduling choices not only based on the tasks that are already scheduled but also on those which are to be scheduled later in order to find a better trade-off between contentions reduction and makespan gain. However, the impact is highly dependent on the results of an interference analysis which is difficult to estimate without knowing the schedule, in particular at the phase granularity which increases the set of possible interference scenario for each task.

In order to address the same issue, we attempted preliminary experiments with MCTS scheduling heuristic where we added simple information about the accesses that each non-scheduled task can perform in the reward function in addition to their duration. They have not been further explored because at the end the results of the simulation step of MCTS were too approximate so the scores of the nodes were not reflecting well their quality. In [65], MCTS is combined with deep reinforcement learning to choose better nodes and predict the results of the simulations but the contentions are not taken into account. Integrating the effects of contentions in the prediction could require to encode the information about their duration coupled with their number of accesses, and to link the information on the phases to their respective tasks. The machine learning approach is interesting but requires experience in this domain to design and train the model.

Chapter 8

General conclusion

Timing analyses are a necessary step for hard real-time systems because they must satisfy strict timing constraints. In particular, the timing analyses are used to derive the Worst-Case Execution Time (WCET) of the tasks, a safe upper bound on their execution time. At the same time, the technological improvements targeting the execution platforms are mainly designed to improve the average-case performance, which deteriorates their predictability. Indeed, these optimizations increase the variability of the execution time and makes the WCET computation more challenging. This thesis addresses the problem of timing prediction for multi-core execution platforms. In particular, we are interested in the prediction of the contentions that can occur in the interconnect connecting the cores to the main memory. Indeed, the interference analysis that computes the worst-case interference scenario in a schedule tends to over-estimate the maximum number of contentions that can occur. This over-estimation is detrimental to the schedulability of real-time systems and to the sizing of execution platforms that can support the estimated worst-case execution scenario.

1 Contributions

Figure 8.1 shows the main steps to pass from the binary code of a task to the code implementing its multi-phase profile and the contributions of the thesis to each step. These contributions are described in the next sections.

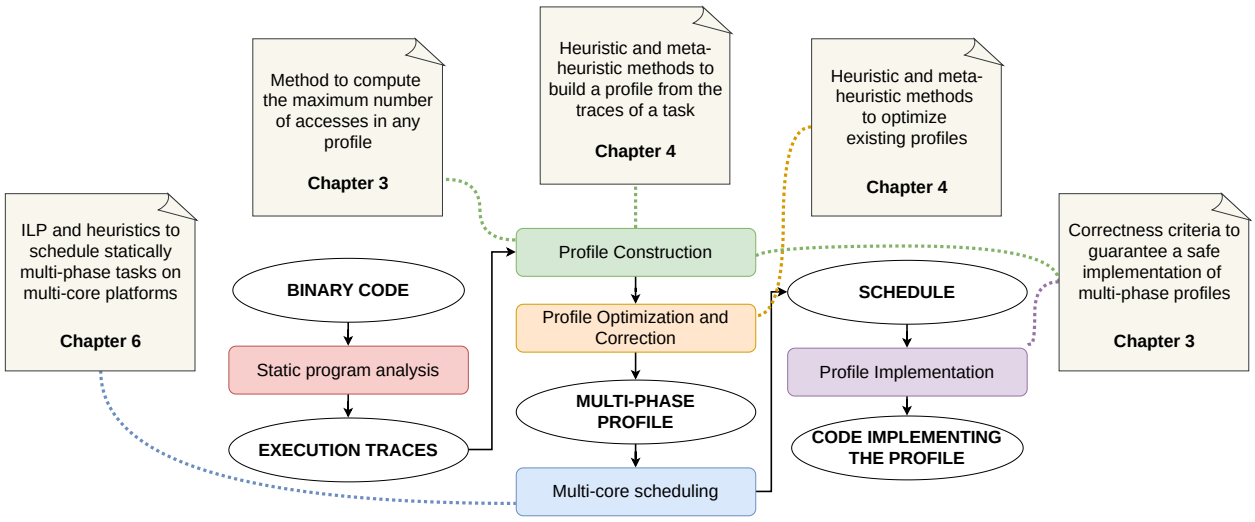


Figure 8.1: Contributions of the thesis.

1.1 Correction and safety of the multi-phase model

The contributions of the thesis rely on the multi-phase representation of tasks during the interference analysis. This model was introduced either with strong constraints on the model (AER [11] and PREM [10]) or coupled with a particular method to obtain the phases (TIPs [12] and StaMP [13]). Instead, we use a generic formalization of the multi-phase model and present a method to account for the worst-case number of accesses in each phase of any profile given the traces representation of the task. Three correctness criteria guarantee that this accounting is safe and that the implementation of the multi-phase model in the code is correct with respect to a given schedule. This implementation is ensured by the injection of synchronizations in the code to bound the possible execution dates of instructions and to ensure that their accesses can only be performed in certain phases. The implementation of the synchronization mechanism is not addressed in the thesis, but the correctness criteria are generic so they are compatible with many possibilities.

1.2 Building multi-phase profiles from the execution traces of a task

Chapter 4 presents new methods to build multi-phase profiles. Firstly, a method tries to cluster the accesses into phases using Kernel Density Estimation (KDE). This method is extended with additional optimization heuristics to apply in order to improve the precision of the profile and favor characteristics that reduce the over-estimation of the interference analysis (e.g. the presence of phases without accesses and a low access over-approximation). This set of heuristics applied to build a multi-phase profile shows that it is a multi-objective optimization problem with a large solution space. Hence, we introduced another building method based on Genetic Algorithms (GA). With this meta-heuristic, it is possible to explore more efficiently the space of solutions and to evaluate these solutions with multiple criteria at the same time using a fitness function. In our implementation, the fitness of a profile depends on the access over-approximation, the number of synchronizations used and the time guaranteed without accesses in the profile. The trade-off between these criteria is set by tuning their coefficient in the fitness function and can be different for each system. In addition, a distinct GA-based method is proposed to optimize existing profiles. This GA uses the same criteria but with an additional criterion measuring the variability of the phases duration such that, coupled with the proportion of time without accesses, the good properties of the initial profile are preserved.

These new building methods are compared with the TIPs method in Chapter 5 on two case studies. According to the makespan of the obtained schedules (with a simple scheduling heuristic), the profiles generated with the TIPs method generally yield the best makespan gain for the analyzed case studies. The results cannot be generalized as our two case studies are not representative. However, we identified strong trends regarding theGA-based methods. First, building a profile with the traces-basedGA requires less synchronizations than the other methods, so its ratio between the makespan gain and the number of synchronizations to inject in the code is better. Moreover, the GA applied on existing profiles was able to reduce the number of synchronizations required in most cases and sometimes even outperformed the makespan gain of the initial profile. Therefore, the use of meta-heuristic techniques such as a GA seems effective to take into account several criteria from the construction of a profile, but also to optimize existing ones.

1.3 Scheduling multi-phase tasks on multi-core platforms

Chapter 6 addresses the problem of statically scheduling multi-phase tasks in multi-core platforms without preemption. The problem is introduced formally by an ILP formulation that takes into account the effects of contentions and tasks precedence constraints. The phase level granularity offers a lot more possibilities than when using the traditional single-phase representation. Therefore, the chapter continues by proposing heuristics that have different strategies to efficiently explore the possible schedules. The ASAP heuristic is the most naive, it simply schedules each task as soon as possible on the core with the lowest end date and contentions are only computed on

the final schedule (i.e. they are not used to make the scheduling decisions). On the contrary, SDE tests different start dates for the tasks on each core and computes the contentions of the partial schedules under test before making its decision. IPH explores the set of possible scheduling orders of the tasks and builds a complete schedule using each order explored and the ASAP policy. Then, it performs an interference analysis to retain only the order that yields the best makespan. Finally, we propose to use Monte-Carlo Tree Search (MCTS) over the set of possible tasks orderings and core mappings, with tasks also scheduled as soon as possible. The scheduling decision is based on a partial interference analysis combined with information on the tasks remaining to schedule.

The last chapter begins by assessing the efficiency of the multi-phase model to reduce the over-estimation of the interference analysis using the ILP to obtain optimal schedules for both the multi-phase representation and its single-phase equivalent. The tests show that the multi-phase model is more efficient in more than 95% of the cases with an average gain near 10% regarding the makespan. However, these tests are limited to small tasks systems because of the limitations inherent to the ILP. Indeed, it has a low scalability due to the phase level granularity and the accounting of contentions so the conclusion cannot be generalized. Still, the heuristics are also all beating the single-phase optimal schedules in average, and particularly IPH whose average gain is between 5 and 6%.

Another experiment with larger tasks systems allows to compare the efficiency of different profile shapes. The results show that, regardless of the variability of the phases duration, the results are better when the number of accesses in the phases varies. The cause is that if the number of accesses is approximately the same for two overlapping time intervals from different cores, then most of the accesses can create contentions. The variability of accesses allows that some phases with many accesses cannot create contentions with all their accesses. The comparison of the results for the heuristics showed that IPH was always superior to the others. Moreover, the efficiency of SDE and, in a lesser degree, of MCTS to reduce the makespan depends largely on the impact of the contentions. Indeed, the two methods are the most efficient to reduce contentions in all the cases but this reduction is not transposed to the same extent for the makespan gain when the interference penalty is low. However, with a higher interference penalty, the makespan gain of SDE was similar to IPH. Finally, the heuristics are applied on two realistic case studies. The results confirm the efficiency of the multi-phase model compared to the single-phase model to reduce the over-estimation of the interference analysis both with a DAG and a multi-periodic system. Indeed, the makespan gain reaches 16% for one of the case study and 24% for the other. Moreover, we observe that the gain increases with the number of cores in the architecture considered and when the interference penalty is more than the cost of an access in isolation, which is a more realistic assumption.

2 Perspectives

As discussed in the conclusion of Chapter 3, the implementation of synchronizations has not been addressed during the thesis although the cost of their injection in the code must be taken into account to ensure that the model can be enforced during the execution of the task. Numerous techniques can be considered, from the simple injection of busy-wait loops in the code to a global scheduler. The correctness criteria defined are generic so they do not exclude any possibility, and the rules used to select synchronizations in Chapter 4 are independent of the execution context (trace being executed, number of iterations of a loop), which is a restrictive way to select synchronizations that also complies with context-aware mechanisms.

Regarding the construction of the multi-phase profiles, we looked for good characteristics to enhance the efficiency of the model for the interference analysis with the definition of the fitness functions for the two GA in Chapter 4, and then with the comparison of the results for different profile shapes in Chapter 7. However, the list of good characteristics that we identified is not extensive and we lack metrics to measure them with more precision. One can identify the influence

of another characteristic easily by adding it as a new criterion of the GA.

The experiments of Chapter 7 showed that SDE and MCTS, which both work on partial schedules with contentions, lack a component to assess the impact of their choices on future decisions. Such a component must take into account the dependencies between the tasks or their order to be able to anticipate which tasks may be scheduled in parallel in the future. The main difficulty is certainly to predict the potential contentions at the phase granularity level. Indeed, a slight difference between the predicted and the actual number of contentions occurring for a phase may completely change the sets of phases contending with each other and result in a poor prediction. Moreover, our scheduling heuristics are never exploring the possible tasks orderings and the possible dates of tasks at the same time because taking one or the other already requires a consequent computation time due to the computation of contentions. However, the two exploration strategies have proven their efficiency so it would be worth combining them. One possibility is to apply SDE instead of ASAP when scheduling with a given tasks order for IPH. This solution has been attempted but the computation time was indeed too high to apply it in all the test cases (including the two case studies). Similarly, it is possible to extend the exploration space of MCTS by including different start dates for the tasks with additional decision branches in the tree. Such solution has also been tried but we encountered problems related to the computational complexity.

The thesis work can also be extended to take into account multiple sources of contentions at the same time. In this case, the interference analysis must compute the possible contentions and their cost for each interference source separately and the overall interference penalty can be obtained by summing the penalties computed for each source.

In addition, we restrained the scheduling problem to architectures with homogeneous cores without task preemption. As mentioned in the conclusion of Chapter 6, this first restriction can easily be lifted by considering different profiles to represent a given task for each core type. It also opens the way to new optimizations that could take into account the trade-off between energy and performance, in addition to the worst-case interference schedule. We also previously discussed about preemption which we have not considered to develop our heuristics. Indeed, preemptions may affect our assumptions and properties regarding the definition of equivalent nodes, the method to conservatively count the accesses and the synchronization of nodes but we did not have time to assess their impact and to adapt our heuristics accordingly.

Bibliography

[1] J. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[2] 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," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, 2008.

[3] A. Roy, J. Xu, and M. H. Chowdhury, "Multi-core processors: A new way forward and challenges," in *Proceedings of the International Conference on Microelectronics, ICM*, pp. 454–457, 2008.

[4] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, pp. 67–77, may 2011.

[5] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. De Dinechin, "The shift to multicores in real-time and safety-critical systems," in *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015*, pp. 220–229, Institute of Electrical and Electronics Engineers Inc., nov 2015.

[6] O. Sander, F. Bapp, L. Dieudonne, T. Sandmann, and J. Becker, "The promised future of multi-core processors in avionics systems," *CEAS Aeronautical Journal*, vol. 8, pp. 143–155, mar 2017.

[7] G. Macher, A. Höller, E. Armengaud, and C. Kreiner, "Automotive embedded software: Migration challenges to multi-core computing platforms," in *Proceeding - 2015 IEEE International Conference on Industrial Informatics, INDIN 2015*, pp. 1386–1393, Institute of Electrical and Electronics Engineers Inc., sep 2015.

[8] Certification Authorities Software Team (CAST), "Postion paper cast-32a, multi-core processors," 2016.

[9] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010* (G. D. Micheli, B. M. Al-Hashimi, W. Müller, and E. Macii, eds.), pp. 741–746, IEEE Computer Society, 2010.

[10] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pp. 269–279, IEEE Computer Society, 2011.

[11] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *ERTS'14*, 2014.

[12] T. Carle and H. Cassé, "Static extraction of memory access profiles for multi-core interference analysis of real-time tasks," in *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings* (C. Hochberger, L. Bauer, and T. Pionteck, eds.), vol. 12800 of *Lecture Notes in Computer Science*, pp. 19–34, Springer, 2021.

[13] T. Degioanni and I. Puaut, "StAMP: Static Analysis of Memory Access Profiles for Real-Time Tasks," in *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)* (C. Ballabriga, ed.), vol. 103 of *Open Access Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 1:1–1:13, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[14] M. Schuh, C. Maiza, J. Goossens, P. Raymond, and B. D. de Dinechin, "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory," in *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pp. 283–295, IEEE, 2020.

[15] 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. Min, R. Pettit, P. Puschner, and T. Ungerer, eds.), vol. LNCS-6399 of *Software Technologies for Embedded and Ubiquitous Systems*, (Waidhofen/Ybbs, Austria), pp. 35–46, Springer, Oct. 2010.

[16] J. Gustafsson and A. Ermedahl, "Automatic derivation of path and loop annotations in object-oriented real-time programs," in *Proceedings of 5th International Workshop on Parallel and Distributed Real-Time Systems and 3rd Workshop on Object-Oriented Real-Time Systems*, pp. 257–262, 1997.

[17] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis," in *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)* (C. Rochange, ed.), vol. 6 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.

[18] S. Blazy, A. Maroneze, and D. Pichardie, "Formal verification of loop bound estimation for wcet analysis," in *Revised Selected Papers of the 5th International Conference on Verified Software: Theories, Tools, Experiments - Volume 8164*, VSTTE 2013, (Berlin, Heidelberg), p. 281–303, Springer-Verlag, 2013.

[19] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A Survey on Static Cache Analysis for Real-Time Systems," *Leibniz Transactions on Embedded Systems*, vol. 3, pp. 05:1–05:48, jun 2016.

[20] A. Colin and I. Puaut, "Worst case execution time analysis for a processor withbranch prediction," *Real-Time Syst.*, vol. 18, p. 249–274, may 2000.

[21] A. Colin and G. Bernat, "Scope-tree: a program representation for symbolic worst-case execution time analysis," *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pp. 50–59, 2002.

[22] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Journal of Systems Architecture*, vol. 46, no. 4, pp. 339–355, 2000.

[23] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 1, pp. 53–70, 1999.

[24] Y. T. S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *ACM SIGPLAN Notices*, vol. 30, pp. 88–98, nov 1995.

[25] H. Theiling, "Ilp-based interprocedural path analysis," in *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, (Berlin, Heidelberg), p. 349–363, Springer-Verlag, 2002.

[26] Federal Aviation Administration (FAA), "Assurance of multicore processors in airborne systems," 2016.

[27] 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, no. 3, pp. 56:1–56:38, 2019.

[28] T. Mitra, J. Teich, and L. Thiele, "Time-Critical Systems Design: A Survey," *IEEE Design and Test*, vol. 35, pp. 8–26, apr 2018.

[29] A. Löfwenmark and S. Nadjm-Tehrani, "Understanding shared memory bank access interference in multi-core avionics," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, vol. 55, pp. 12.1–12.11, Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, dec 2016.

[30] S. Altmeyer, R. I. Davis, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015* (J. Forget, ed.), pp. 129–138, ACM, 2015.

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

[32] 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 *ACM International Conference Proceeding Series*, vol. 19-21-Octo, pp. 67–76, Association for Computing Machinery, oct 2016.

[33] A. Schranzhofer, J. J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in *Real-Time Technology and Applications - Proceedings*, pp. 215–224, 2010.

[34] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Static analysis of multi-core TDMA resource arbitration delays," *Real-Time Systems*, vol. 50, pp. 185–229, aug 2014.

[35] H. Rihani, M. Moy, C. Maiza, and S. Altmeyer, "WCET analysis in shared resources real-time systems with TDMA buses," *ACM International Conference Proceeding Series*, vol. 04-06-Nove, no. 5, pp. 183–192, 2015.

[36] D. Dasari and V. Nelis, "An analysis of the impact of bus contention on the WCET in multicores," in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications, HPCC-2012 - 9th IEEE International Conference on Embedded Software and Systems, ICESS-2012*, pp. 1450–1457, 2012.

[37] J. Arora, C. Maia, S. A. Rashid, G. Nelissen, and E. Tovar, "Bus-contention aware WCRT analysis for the 3-phase task model considering a work-conserving bus arbitration scheme," *Journal of Systems Architecture*, vol. 122, p. 102345, jan 2022.

[38] M. Jacobs, S. Hahn, and S. Hack, "WCET analysis for multi-core processors with shared buses and event-driven bus arbitration," in *ACM International Conference Proceeding Series*, 2015.

[39] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time Technology and Applications - Proceedings*, pp. 55–64, 2013.

[40] M. Behnam, R. Inam, T. Nolte, and M. Sjödin, "Multi-core composability in the face of memory-bus contention," *ACM SIGBED Review*, vol. 10, pp. 35–42, oct 2013.

[41] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, "Memory-aware scheduling of multicore task sets for real-time systems," in *Proceedings - 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2012 - 2nd Workshop on Cyber-Physical Systems, Networks, and Applications, CPSNA*, pp. 300–309, 2012.

[42] S. Wasly and R. Pellizzoni, "Hiding Memory Latency Using Fixed Priority Scheduling," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[43] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2015-May, pp. 285–296, Institute of Electrical and Electronics Engineers Inc., may 2015.

[44] J. Matějka, B. Forsberg, M. Sojka, P. Šůcha, L. Benini, A. Marongiu, and Z. Hanzálek, "Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution," *Parallel Computing*, vol. 85, pp. 27–44, jul 2019.

[45] Z. Hanzálek and P. Šůcha, "Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources," *Annals of Operations Research*, vol. 248, pp. 209–237, jan 2017.

[46] B. Rouxel, S. Derrien, and I. Puaut, "Tightening contention delays while scheduling parallel applications on multi-core architectures," *ACM Transactions on Embedded Computing Systems*, vol. 16, pp. 1–20, 2017.

[47] M. D. de Dinechin, M. Schuh, M. Moy, and C. Maiza, "Scaling up the memory interference analysis for hard real-time many-core systems," in *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pp. 330–333, IEEE, 2020.

[48] R. Mancuso, R. Dudko, and M. Caccamo, "Light-prem: Automated software refactoring for predictable execution on COTS embedded systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, pp. 1–10, IEEE Computer Society, 2014.

[49] 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.

[50] F. Fort and J. Forget, "Code generation for multi-phase tasks on a multi-core distributed memory platform," in *25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2019, Hangzhou, China, August 18-21, 2019*, pp. 1–6, IEEE, 2019.

[51] M. R. Soliman and R. Pellizzoni, "Prem-based optimal task segmentation under fixed priority scheduling," in *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany* (S. Quinton, ed.), vol. 133 of *LIPIcs*, pp. 4:1–4:23, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[52] B. Forsberg, M. Solieri, M. Bertogna, L. Benini, and A. Marongiu, "The predictable execution model in practice: Compiling real applications for COTS hardware," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, pp. 47:1–47:25, 2021.

[53] T. Carle and H. Cassé, "Reducing timing interferences in real-time applications running on multicore architectures," in *18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018, July 3, 2018, Barcelona, Spain* (F. Brandner, ed.), vol. 63 of *OASIcs*, pp. 3:1–3:12, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[54] A. E. Ezugwu, A. K. Shukla, R. Nath, A. A. Akinyelu, J. O. Agushaka, H. Chiroma, and P. K. Muhuri, "Metaheuristics: a comprehensive overview and classification along with bibliometric analysis," *Artificial Intelligence Review*, vol. 54, pp. 4237–4316, mar 2021.

[55] M. Grajcar, "Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pp. 280–285, Institute of Electrical and Electronics Engineers (IEEE), jan 2003.

[56] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Information Sciences*, vol. 270, pp. 255–287, 2014.

[57] J. Oh and C. Wu, "Genetic-algorithm-based real-time task scheduling with multiple goals," *Journal of Systems and Software*, vol. 71, no. 3, pp. 245–258, 2004.

[58] X. X. Xu, X. M. Hu, W. N. Chen, and Y. Li, "Set-based particle swarm optimization for mapping and scheduling tasks on heterogeneous embedded systems," in *Proceedings of the 8th International Conference on Advanced Computational Intelligence, ICACI 2016*, pp. 318–325, 2016.

[59] G. Wang, W. Gong, B. Derenzi, and R. Kastner, "Exploring time/resource trade-offs by solving dual scheduling problems with the ant colony optimization," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, sep 2007.

[60] J. Rosén, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proceedings - Real-Time Systems Symposium*, pp. 49–60, 2007.

[61] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, pp. 681–715, nov 2012.

[62] M. Becker, D. Dasari, B. Nikolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 14–24, 2016.

[63] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global Real-Time Memory-Centric Scheduling for Multicore Systems," *IEEE Transactions on Computers*, vol. 65, pp. 2739–2751, sep 2016.

[64] K. Liu, Z. Wu, Q. Wu, and Y. Cheng, "Smart dag task scheduling with efficient pruning-based mcts method," in *Proceedings - 2019 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking, ISPA/BDCloud/SustainCom/SocialCom 2019*, pp. 348–355, Institute of Electrical and Electronics Engineers Inc., dec 2019.

[65] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," in *Proceedings - International Conference on Distributed Computing Systems*, vol. 2019-July, pp. 2037–2046, Institute of Electrical and Electronics Engineers Inc., jul 2019.

[66] X. Palomo, E. Mezzetti, J. Abella, R. J. Bril, and F. J. Cazorla, "Accurate ILP-Based contention modeling on statically scheduled multicore systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2019-April, pp. 15–28, Institute of Electrical and Electronics Engineers Inc., apr 2019.

[67] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2020-April, pp. 239–252, Institute of Electrical and Electronics Engineers Inc., apr 2020.

[68] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency," in *Proceedings - Real-Time Systems Symposium*, vol. 2020-Decem, 2020.

[69] J. M. Aceituno, A. Guasque, P. Balbastre, J. Simo, and A. Crespo, "Hardware resources contention-aware scheduling of hard real-time multiprocessor systems," *Journal of Systems Architecture*, vol. 118, p. 102223, 2021.

[70] H. Lee, S. Cho, Y. Jang, J. Lee, and H. Woo, "A Global DAG Task Scheduler Using Deep Reinforcement Learning and Graph Convolution Network," *IEEE Access*, vol. 9, pp. 158548–158561, 2021.

[71] J. Arora, C. Maia, S. A. Rashid, G. Nelissen, and E. Tovar, "Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling," in *RTNS'2021: 29th International Conference on Real-Time Networks and Systems, Nantes, France, April 7-9, 2021* (A. Queudet, I. Bate, and G. Lipari, eds.), pp. 123–133, ACM, 2021.

[72] I. Senoussaoui, H.-E. Zahaf, G. Lipari, and K. M. Benhaoua, "Contention-free scheduling of PREM tasks on partitioned multicore platforms," *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, sep 2022.

[73] Z. Lin, C. Li, L. Tian, and B. Zhang, "A scheduling algorithm based on reinforcement learning for heterogeneous environments," *Applied Soft Computing*, vol. 130, p. 109707, nov 2022.

[74] S. Pagani, P. D. Manoj, A. Jantsch, and J. Henkel, "Machine Learning for Power, Energy, and Thermal Management on Multicore Processors: A Survey," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 101–116, jan 2020.

[75] Q. Wu, Z. Wu, Y. Zhuang, and Y. Cheng, "Adaptive DAG tasks scheduling with deep reinforcement learning," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11335 LNCS, pp. 477–490, Springer Verlag, 2018.

[76] Y. Zhou, X. Li, J. Luo, M. Yuan, J. Zeng, and J. Yao, "Learning to Optimize DAG Scheduling in Heterogeneous Environment," in *Proceedings - IEEE International Conference on Mobile Data Management*, vol. 2022-June, pp. 137–146, Institute of Electrical and Electronics Engineers Inc., 2022.

[77] A. Yano and T. Azumi, "CQGA-HEFT: Q-learning-based DAG Scheduling Algorithm Using Genetic Algorithm in Clustered Many-core Platform," *Journal of Information Processing*, vol. 30, pp. 659–668, sep 2022.

[78] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016* (A. Plantec, F. Singhoff, S. Faucou, and L. M. Pinho, eds.), pp. 299–308, ACM, 2016.

[79] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, vol. 26. Chapman and Hall, 1986.

[80] P. Hall and J. S. Marron, "On the Amount of Noise Inherent in Bandwidth Selection for a Kernel Density Estimator," *The Annals of Statistics*, vol. 15, no. 1, 1987.

[81] K. Hussain, M. N. Mohd Salleh, S. Cheng, and Y. Shi, "Metaheuristic research: a comprehensive survey," *Artificial Intelligence Review*, vol. 52, pp. 2191–2233, jan 2019.

[82] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The ROSACE case study: From simulink specification to multi/many-core execution," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pp. 309–318, IEEE Computer Society, 2014.

[83] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens, "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation," *Leibniz Trans. Embed. Syst.*, vol. 2, no. 2, pp. 01:1–01:30, 2015.

[84] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," mar 2012.

[85] S. Hahn and J. Reineke, "Design and analysis of SIC: a provably timing-predictable pipelined processor core," *Real Time Syst.*, vol. 56, no. 2, pp. 207–245, 2020.

[86] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022.

[87] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "PapaBench: a Free Real-Time Benchmark," *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, 2006.

Chapter 9

Synthèse en français

1 Introduction

L'analyse pire cas de temps d'exécution, ou *Worst-Case Execution Time (WCET)* [2], est nécessaire pour un système temps-réel critique (e.g. aéronautique, spatial, médical...) afin de prouver que, quelque soit le scénario d'exécution, le système ne pourra pas être mis en défaut et causer des accidents graves. Alors que l'architecture des processeurs évolue constamment pour améliorer leur performance, l'analyse WCET doit s'adapter à ces évolutions pour fournir des résultats sûrs et le plus précis possible.

En particulier, le remplacement progressif des architectures mono-cœur par les architectures multi-cœur [3, 4] est une problématique majeure pour les systèmes critiques puisque ces dernières permettent un bond technologique mais sont aussi beaucoup moins prédictibles que les mono-cœur. En effet, dans les architectures multi-cœur, les cœurs partagent généralement des ressources (mémoires, bus, périphériques) et peuvent potentiellement accéder à ces ressources au même moment, créant des interférences. Cela complique davantage la vérification des systèmes critiques. Pour cette raison, les principales autorités de certification ont publié des documents traitant spécifiquement des architectures multi-cœur pour faire part de leurs exigences [8].

La thèse s'intéresse à cette problématique de vérification temporelle de programmes temps-réel sur des architectures multi-cœur. En particulier, elle traite des interférences nées d'accès concurrents à une même ressource. L'objectif est de réduire la sur-estimation des interférences qui peuvent avoir lieu pour améliorer la précision de la borne supérieure d'exécution en présence d'interférences.

2 Temps d'exécution pire-cas

2.1 Estimer le WCET d'une tâche

Le temps d'exécution d'une tâche varie en fonction des données d'entrée de la tâche et de la plate-forme sur laquelle elle s'exécute. Ainsi, en mesurant le temps d'exécution de la tâche pour toutes les entrées possibles et tous les états initiaux de la plate-forme d'exécution, on peut déterminer le WCET.

En pratique, c'est souvent impossible parce qu'il est compliqué d'identifier et de reproduire exactement toutes ces exécution possibles. Pour y remédier, les techniques d'analyse WCET dynamiques exécutent le programme (sur cible ou avec un simulateur) pour un sous-ensemble des temps d'exécution possibles puis estiment le WCET en utilisant des méthodes probabilistes. Cependant, il n'y pas de garantie que le WCET estimé n'est pas sous-évalué puisqu'on ne sait pas si le chemin d'exécution pire cas a été mesuré ou pris en compte.

La thèse repose sur des méthodes d'analyse statiques qui, contrairement aux méthodes dynamiques, ne nécessitent pas d'exécution des tâches. Elles reposent sur une analyse du code et

des composants micro-architecturaux de la cible et permet de couvrir tous les chemins d'exécution possibles afin d'obtenir une borne d'exécution qui ne peut pas sous-estimer le WCET réel.

2.2 Analyse WCET statique

On peut découper une analyse WCET statique en mono-cœur en 3 grandes étapes:

1. L'analyse du flot de contrôle: elle se concentre sur le code pour extraire les informations nécessaires à l'analyse sur les chemins d'exécution possibles.

2. L'analyse du comportement du processeur: cette analyse détermine le temps d'exécution des portions de code qui composent les chemins d'exécution possibles en analysant les états des composants du processeur et leur évolution.

3. Le calcul du WCET: cette étape combine les informations sur les chemins d'exécution et le temps d'exécution des instructions sur ces chemins pour trouver le chemin d'exécution au pire cas.

Pour effectuer ces étapes sans exécuter le code, les analyses statiques utilisent des modèles qui représentent le code et les composants de l'architecture. L'utilisation de ces modèles nécessite un arbitrage entre la quantité information qu'ils capturent et leur précision afin que l'analyse ne soit pas trop complexe à réaliser. Pour cette raison, l'analyse utilise des hypothèses conservatrices sur l'état des composants et les chemins d'exécution afin d'abstraire certaines informations tout en s'assurant de couvrir tous les cas d'exécution. Cela peut entraîner une sur-estimation importante du WCET.

2.3 Adoption des architectures multi-cœur

Les processeurs mono-cœur sont progressivement remplacés par des multi-cœur dans la plupart des applications industrielles, y compris pour les systèmes critiques [5, 6, 7]. En effet, ces architectures sont plus performantes (parallélisme), elles ont aussi une meilleure efficacité énergétique et elles permettent par exemple d'utiliser différents types de cœurs pour exécuter un les tâches d'un système.

Cependant, elles introduisent de nouvelles problématiques du point de vue des analyses temporelles qui sont bloquantes pour leur utilisation dans les systèmes critiques. Dans cette thèse, nous nous intéressons aux interférences inter-cœurs qui surviennent suite à des accès concurrents aux ressources partagées. En effet, lorsque plusieurs cœurs demandent simultanément l'accès à une ressource partagée (e.g. bus mémoire, cache, mémoire principale), un seul est servi et tous les autres doivent attendre. Ce temps d'attente n'est pas pris en compte dans les analyses WCET classiques conduites en mono-cœur. Pourtant, il est loin d'être négligeable puisque certaines recherches ont mesuré des temps d'exécution presque trois fois plus élevés avec que sans ces interférences.

2.4 Prise en compte des interférences dans l'analyse temporelle

L'une des approches utilisées pour prendre en compte ces interférences est l'isolation des cœurs. En effet, en s'assurant que les cœurs ne peuvent pas accéder aux mêmes ressources en même temps, on est certain qu'il ne peut pas y avoir d'interférences. Cela peut se faire en choisissant des politiques d'arbitrage comme Time Division Multiple Access (TDMA) [33, 34, 35] ou Round Robin (RR) [36, 37] pour lesquelles chaque cœur a périodiquement une fenêtre temporelle pour accéder individuellement à la ressource partagée. L'inconvénient est que l'analyse WCET va considérer que pour chaque accès, le cœur doit attendre une période entière avant de pouvoir être servi. Il en résulte une importante sous-utilisation des ressources. Pour y remédier, des travaux ajoutent des éléments de contexte afin de ne pas avoir toujours à considérer le même pire cas quelle que soit la situation [36, 9].

Toujours dans ce but d'isolation, de nouveaux modèles de tâches ont été proposés. Le modèle PRedictable Execution Model (PREM) [10] décompose l'exécution des tâches en deux phases: l'une effectue tous les transferts mémoires entre des mémoires locales à chaque cœur et celles partagées, et la suivante exécute la tâche en ne se servant que des données et instructions chargées par la première phase (sans effectuer d'accès supplémentaire à la mémoire partagée). Ce modèle a été repris en ajoutant une troisième phase à la fin de la tâche qui, comme la première, peut effectuer des accès (modèle Acquisition Execution Restitution AER [11]) et ainsi écrire en mémoire les résultats de l'exécution. En ordonnançant les tâches de façon à ce qu'aucune phase mémoire ne s'exécute en même temps qu'une autre, ou en utilisant un bus TDMA ou RR, on peut garantir un système sans interférences.

L'inconvénient est que le code est très contraint par ce modèle et qu'il faut charger un grand nombre de données localement pour être sûr que toutes les données nécessaires à l'exécution de la tâche sont disponibles avant sa phase d'exécution (i.e. le temps de chargement et l'espace de stockage nécessaire sont importants). De plus, garantir un système sans interférences n'est pas forcément la meilleure façon de réduire le temps de réponse du système. Des travaux proposent donc de considérer les cas où les phases mémoires peuvent interférer entre elles tout et proposent d'effectuer des analyses d'interférence pour tenir compte des délais supplémentaires dus aux interférences dans le pire cas [14, 46].

En s'affranchissant des contraintes de PREM et AER (nombre et type de phases), les méthodes Time Interest Points (TIPs) [53, 12] et Static Analysis of Memory Access Profiles (StAMP) [13] proposent deux techniques pour construire des représentations multi-phase directement à partir du code binaire des tâches. Ces deux méthodes requièrent un décompte sûr du nombre maximum d'accès pouvant être effectués dans chaque phase avant d'effectuer l'analyse d'interférences.

La thèse s'inscrit dans la continuité de ces méthodes moins contraignantes pour représenter et ordonnancer les tâches d'un système.

3 Modèle multi-phase: définition formelle et critères de correction

3.1 Définition formelle du modèle: phases et traces

Le Chapitre 3 présente une définition formelle générique du modèle multi-phase. À la représentation sous forme de phases s'ajoute une représentation des traces d'exécution de la tâche qui permet de vérifier si les phases représentent correctement le modèle en mettant en relation ces deux représentations.

Les traces sont des suites de nœuds représentant chacun une instruction pouvant effectuer un accès mémoire. Les transitions entre les nœuds expriment le flot de contrôle et ont une durée correspondant au WCET local entre les nœuds source et destination. Cette représentation nous fournit une borne supérieure de la date d'exécution des instructions représentées par les nœuds et on peut donc estimer sur quelle(s) phase(s) ils peuvent potentiellement s'exécuter puis compter le nombre maximum d'accès dans chaque phase.

3.2 Synchronisations

Avec le modèle multi-phase, lorsque l'intervalle temporel dans lequel un accès peut être effectué couvre plusieurs phases alors on doit compter cet accès dans toutes les phases couvertes. Un tel accès est donc compté plusieurs fois dans la tâche, ce qui n'est pas possible en utilisant le modèle mono-phase. On appelle sur-approximation des accès la différence entre le nombre total d'accès dans une représentation multi-phase et son équivalent mono-phase.

Alors que nous avons une borne supérieure de la date d'exécution des nœuds, le modèle considère par défaut que la borne inférieure est le début de la tâche. Cela dégrade grandement la sur-approximation des accès car un nœud doit alors être compté dans toutes les phases précédant sa date d'exécution pire cas. Pour obtenir une borne inférieure plus précise et réduire l'intervalle

d'exécution possible des instructions, on peut calculer leur BCET mais les outils d'analyse à notre disposition, centrés sur le WCET, ne permettent pas de le calculer directement. Ainsi, nous utilisons des synchronisations sur certains nœuds qui les empêchent de s'exécuter avant la date de synchronisation choisie. Ces synchronisations doivent être injectées dans le code sur les instructions représentées par les nœuds synchronisés. Elles permettent de mieux contrôler les intervalles d'exécution des instructions effectuant des accès et donc la sur-approximation du modèle multi-phase.

3.3 Critères de correction

En détaillant 3 critères de correction, nous montrons comment compter de façon sûre les accès dans les phases et comment ajuster les dates de synchronisation afin que le décompte des accès avant d'appliquer l'analyse d'interférence reste valable également une fois qu'elle a été effectuée.

En effet, pour prendre en compte l'effet des interférences, la durée des phases est augmentée par la pénalité d'interférence qui est proportionnelle au nombre d'interférences qu'elles peuvent subir selon l'analyse d'interférences. Suite à ce changement, les intervalles couvrant les dates possibles d'exécution des instructions recouvrent potentiellement d'autres phases que celles identifiées en isolation, ce qui invaliderait le décompte des accès dans les phases et l'analyse d'interférence elle-même.

En agissant sur la date des synchronisations, les critères de correction garantissent que l'analyse d'interférence est valide en tenant compte des pénalités d'interférence à appliquer. Comme ces trois critères sont indépendants de la méthode choisie pour construire les phases et du mécanisme de synchronisation implémenté, ils peuvent être appliqués à tout profil multi-phase.

4 Du code binaire à la représentation multi-phase d'une tâche

4.1 Heuristique simple de construction d'un profil

La définition des critères de correction nous a permis de définir de nouvelles méthodes pour créer et optimiser des profils multi-phase. Nous avons d'abord développé une heuristique pour créer un profil à partir des traces d'une tâche, basée sur l'estimation de densité par noyau, ou KDE en anglais. Le principe est de construire une distribution des dates d'accès pire cas de la tâche en utilisant les traces, puis d'extraire la fonction continue représentant cette distribution en utilisant KDE afin de créer des phases entre les pics et les creux de la fonction. Il en résulte un profil dont la date des phases est directement liée à celle des accès, si possible regroupés dans certaines phases.

4.2 Sélection des synchronisations et optimisation du profil

Nous proposons par la suite une méthode pour sélectionner efficacement des synchronisations en considérant un mécanisme de synchronisation indépendant du contexte d'exécution pour le rendre le plus général possible. Cette méthode est complétée par une heuristique qui optimise la sélection en identifiant et supprimant des synchronisations "redondantes" (i.e. dont la présence n'améliore pas le compte des accès dans les phases). En complément, nous avons implémenté des heuristiques pour corriger et améliorer l'efficacité du profil construit en favorisant de bonnes caractéristiques pour l'analyse d'interférences par l'harmonisation de la durée des phases et l'augmentation du temps passé sans accès.

4.3 Utilisation d'algorithmes génétiques

Cette accumulation d'heuristiques pour façonner un bon profil multi-phase montre que le problème admet un grand nombre de solutions et qu'on peut s'intéresser à de multiples critères. C'est pourquoi nous nous sommes aussi tournés vers les méta-heuristiques. Cette famille d'algorithmes est utilisée pour résoudre des problèmes d'optimisation complexes car elles peuvent donner une

bonne approximation de la meilleure solution. En particulier, nous avons utilisé des Algorithmes Génétiques (GA) qui sont basés sur les mécanismes de sélection naturelle. Ils sont faciles à implémenter et permettent de traiter des problèmes multi-critères avec un grand espace de solutions.

4.3.1 Créer un profil avec un GA

Le premier algorithme génétique implémenté permet de créer un profil multi-phase à partir des traces d'exécution d'un programme. Pour construire les phases, le GA utilise les dates pire cas des instructions effectuant des accès. Cependant, nous proposons 3 versions de l'algorithme pour sélectionner les synchronisations. La première version se base directement sur les nœuds dont la date pire cas sert à créer des phases. Pour les deux autres versions, c'est l'heuristique de sélection des synchronisations développée précédemment qui est appliquée mais seule l'une de ces deux autres version applique également l'optimisation retirant les synchronisations redondantes. Les critères d'évaluation des solutions sont: la sur-approximation du nombre d'accès et le nombre de synchronisations qui doivent être minimisés, ainsi que la proportion de temps sans accès de la tâche qui doit être maximisée. De plus, nous utilisons une durée de phase minimale qui permet d'harmoniser la durée des phases.

4.3.2 Optimiser un profil avec un GA

Le deuxième algorithme génétique est appliqué sur des profils existants et essaie de fusionner des phases consécutives afin d'améliorer leur potentiel. Ces fusions permettent en effet de réduire mécaniquement la sur-approximation des accès et le nombre de synchronisations nécessaires, qui sont évalués pour chaque solution. Cependant, il est important de garder les bonnes caractéristiques du profil initial. C'est pourquoi la proportion de temps sans accès est toujours évaluée ainsi que la variabilité des durées de phase, qui permet de s'assurer que la structure du profil initial ne varie pas trop.

5 Étude comparative: concevoir un profil de tâche

5.1 Comparaison des caractéristiques des profils

Les méthodes de conception de profil présentées sont comparées dans le Chapitre 5 avec la méthode TIPs de l'état de l'art. L'étude est menée en utilisant deux cas d'étude: un cas issu de l'état de l'art et un autre synthétique. D'après nos résultats, les GAs créent des profils composés de beaucoup moins de phases que les autres méthodes et qui, par conséquent, nécessitent moins de synchronisations. Le GA d'optimisation permet lui aussi de réduire le nombre de phases et de synchronisations dans les profils initiaux.

5.2 Comparaison des gains après ordonnancement

Nous avons aussi généré des profils pour les cas d'étude, et ordonnancé ces systèmes de tâches pour évaluer les gains obtenus sur le temps de réponse du système en intégrant les interférences. La méthode TIPs de l'état de l'art est généralement la meilleure de ce point de vue alors que notre heuristique basée sur KDE est plutôt moins bonne que les autres. Le GA d'optimisation permet d'améliorer les résultats des profils issus de notre heuristique et dégrade légèrement ceux de la méthode TIPs. Le GA créant lui-même un profil obtient de bons résultats avec le cas d'étude réel mais est plutôt comme notre heuristique simple avec le cas synthétique.

5.3 Équilibre entre effort d'implémentation et gain après ordonnancement

En résumé, la méthode TIPs génère les profils ayant le plus de potentiel pour réduire le temps de réponse au pire cas. Cependant, les profils qu'elle génère utilisent beaucoup plus de synchronisations

que les méthodes à base d'algorithmes génétiques. Ainsi, les méthodes intégrant un GA soit pour créer ou optimiser un profil offrent le meilleur compromis entre le gain en temps de réponse et le nombre de synchronisations à injecter dans le code pour nos deux cas d'étude.

6 Ordonnancement statique de tâches multi-phase sur des plate-formes multi-cœur

6.1 Définition du problème (ILP)

Pour finir, dans le Chapitre 6 on s'intéresse à l'ordonnancement statique de tâches décrites par des profils multi-phase (après obtention des profils, par n'importe laquelle des méthodes). Nous proposons d'abord une formulation Integer Linear Programming (ILP) du problème en prenant en compte des dépendances entre tâches et les interférences pouvant avoir lieu entre les phases. L'objectif est de réduire le temps total d'exécution du système.

6.2 Développement d'heuristiques d'ordonnancement

Ensuite, on présente des heuristiques pour pouvoir traiter des systèmes de tâches réalistes que l'ILP ne pourrait pas résoudre en un temps acceptable. La première heuristique, nommée As Soon As Possible (ASAP), ne se soucie pas des interférences : chaque tâche est ordonnancée au plus tôt sur le cœur qui donne le meilleur temps de réponse, dans une logique gloutonne.

Les deux autres heuristiques au contraire intègrent le calcul des interférences pour trouver le meilleur ordonnancement, mais leur stratégie diffère l'une de l'autre. Starting Date Enumeration (SDE) tente d'ordonnancer les tâches tour à tour (selon un ordre donné) sur tous les cœurs disponibles avec différentes dates de début puis effectue une analyse d'interférence sur les ordonnancements partiels étudiés avant de retenir celui qui donne le meilleur temps de réponse. Quant à elle, Iterative Priority Heuristic (IPH) explore les ordres dans lesquels les tâches sont sélectionnées et placées par l'ordonnancement. L'algorithme tente de converger vers le meilleur ordre en se servant des informations sur les ordres déjà essayées et en calculant les interférences sur les ordonnancement complets.

Enfin, nous tentons d'utiliser une autre méta-heuristique utilisant une méta-heuristique appelée Monte-Carlo Tree Search (MCTS). Elle explore les ordres et allocations aux cœurs possibles des tâches à l'aide d'un arbre et effectue les analyses d'interférences sur des ordonnancements partiels comme pour SDE. Toutefois, ses calculs incluent le coût des tâches restant à ordonnancer pour comparer d'une façon plus juste les différents états représentant des ordonnancement partiels lors de chaque décision.

6.3 Optimisation d'un ordonnancement multi-phase

Nous présentons aussi une heuristique d'optimisation qui peut s'appliquer à un ordonnancement partiel ou complet existant pour améliorer le temps de réponse. Cette heuristique fusionne des phases lorsque le modèle multi-phase introduit une forte sur-estimation locale des interférences.

7 Étude comparative: ordonnancement statique sur des plate-formes multi-cœur

7.1 Comparaison avec l'ILP

Le dernier chapitre (Chapitre 7) compare les résultats des heuristiques d'ordonnancement. La première partie des expérimentations est effectuée uniquement sur des profils synthétiques générés de façon à couvrir un large échantillon de profils possibles. On étudie d'abord de petits systèmes de tâches afin de pouvoir y appliquer l'ILP et obtenir une solution optimale. Cette expérience indique

que le profil multi-phase est plus efficace que le mono-phase (lui aussi ordonnancé avec l'ILP) dans plus de 95% des cas et avec en moyenne un temps de réponse diminué de 9.42%. De plus, les ordonnancements générés par les heuristiques sont eux aussi meilleurs que l'optimal en mono-phase (dans près de 90% des cas pour IPH et plus de 75% des cas pour SDE). Ces résultats ne sont pas généralisables à cause de la taille des systèmes étudiés, limitée par le temps de résolution de l'ILP.

7.2 Comparaison avec de plus grands systèmes

Les tests suivants sont effectués sur de plus grands systèmes. Les résultats montrent qu'IPH est la meilleure heuristique pour réduire le temps de réponse alors que SDE et MCTS sont les meilleures pour réduire le nombre d'interférences. On remarque que SDE et MCTS sont peu efficaces pour réduire le temps de réponse lorsque le coût d'une interférence est faible (scénario optimiste), mais SDE est au même niveau qu'IPH lorsque nous augmentons le coût des interférences (scénario plus réaliste). Nous étudions aussi l'influence des paramètres de génération sur l'efficacité du modèle multi-phase à partir de nos résultats.

7.3 Cas d'étude

Ensuite, les heuristiques sont appliquées sur deux cas d'étude issus de l'état de l'art. Les résultats confirment nos observations sur les tâches synthétiques. Le gain en temps de réponse atteint 24% lorsque le coût de pénalité est important et avec 4 cœurs tandis que la réduction des interférences monte jusqu'à presque 65%. L'étude de cas se poursuit en utilisant des systèmes multi-périodiques. L'objectif n'est plus de réduire le temps d'exécution total du système mais d'améliorer son ordonnançabilité (i.e. réduire la fréquence processeur à partir de laquelle chaque tâche respecte ses contraintes temporelles). Dans cette nouvelle expérience, nous montrons encore une fois l'efficacité du modèle multi-phase qui améliore dans presque tous les cas l'ordonnançabilité en utilisant ASAP et SDE.

8 Conclusion et perspectives

La thèse propose d'améliorer la prise en compte des interférences inter-cœurs dans les analyses temporelles statiques qui apparaissent en multi-cœurs. En particulier, les interférences étudiées sont celles apparaissant lorsque plusieurs cœurs tentent d'accéder en même temps à une mémoire partagée.

Les contributions sont toutes basées sur un modèle multi-phase générique des tâches qui permet une représentation plus précise des moments où sont effectués les accès dans une tâche. En résumé, ces contributions sont:

- Formalisation de critères de correction garantissant que le décompte des accès dans les phases ainsi que l'implémentation d'un ordonnancement donné sont sûrs par rapport à une sélection de synchronisations.
- Développement de méthodes de conception et d'optimisation de profils multi-phase, ainsi que de sélection de synchronisations suivant un mécanisme de synchronisation générique (indépendant du contexte d'exécution).
- Écriture d'une formulation ILP du problème d'ordonnancement multi-cœur avec des dépendances de tâches et la prise en compte des interférences entre les phases.
- Développement d'heuristiques d'ordonnancement multi-cœur adaptées au modèle multi-phase et prenant en compte les interférences.

Le travail pourrait être étendu par la suite en suivant les pistes suivantes:

- Développement complet d'une méthode de synchronisation respectant les propriétés introduites dans la thèse.
- Prise en compte d'autres types d'interférences (e.g. périphériques I/O, éviction de données dans les caches partagés...).
- Extension de l'étude sur l'influence des caractéristiques des profils sur l'efficacité du modèle multi-cœur pour mieux guider les techniques de conception de profils.
- Ordonnancement avec une architecture multi-cœur hétérogène.
- Sortir de la vision court-termiste des heuristiques d'ordonnancement travaillant sur des ordonnancements partiels (SDE et MCTS).
- Évaluer l'impact de la présence de préemptions sur les propriétés et hypothèses présentées dans la thèse.