Ordonnancement temps-réel conscient des caches dans des architectures multi-cœurs : algorithmes et réalisation

Viet Anh Nguyen

ANNÉE (2018)

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale MATHSTIC

présentée par

Viet Anh NGUYEN

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Alétoires

Cache-conscious Off-Line Real-Time Scheduling for Multi-Core Platforms: Algorithms and Implementation

Thèse soutenue à Rennes
le 22 Février 2018

devant le jury composé de :

Steven DERRIEN
Professeur, Professeur, Université de Rennes 1 / *Président*
Christine ROCHANGE
Professeur, Université Paul Sabatier Toulouse / *Rapporteur*
Mathieu JAN
Ingénieur-chercheur, CEA / *Rapporteur*
Damien HARDY
Maître de conférences, Université de Rennes 1 / *Examinateur*
Frédéric PÉTROT
Professeur, ENSIMAG Grenoble / *Examinateur*
Isabelle PUAUT
Professeur, Université de Rennes 1 / *Directeur de thèse*

Acknowledgments

This thesis becomes a reality with the support of many individuals. I would like to express my sincere thanks to all of them.

First of all, I would like to express my gratitude to my supervisors, Professor Isabelle PUAUT and Associate Professor Damien HARDY of University of Rennes 1, for their unwavering support and guidance.

I also would like to thank my committee members, whose gave me fruitful comments to enhance the quality of the thesis.

In addition, I appreciate the supports of all permanent staffs and my colleagues in PACAP team. Your help have made my job very much easier and more enjoyable.

And finally, I would like to thank my family. You have always encouraged and believed in me. You have helped me to focus on what has been a hugely rewarding.

Contents

Résumé de la thèse en langue Fran caise

Les systèmes temps-réel, dont le respect du temps de réponse est tou aussi important que la correction fonctionnelle, se sont maintenant répandus dans notre vie quotidienne. En particulier, les applications temps-réel peuvent être trouvées dans les voitures, les avions, les centrales nucléaires. De plus, selon [1] le marché des systèmes embarqués va certainement être témoin d'une forte demande dans les années à venir. Le marché des systèmes embarqués était évalué à 84,55 milliards de dollars US en 2016 et il est prévu qu'il grossisse à un taux de croissance annuel recomposé de 4.05% entre les années 2017 et 2023.

Avec la demande toujours plus grandissante pour des applications sûres mais intensives en terme de calcul, l'utilisation d'architectures mono-cœurs n'est plus un choix judicieux pour le déploiement de systèmes temps-réel, et ce dû aux limites technologiques de ce type d'architecture (par exemple, limite de puissance énergétique [2]). Afin de dépasser cette limitation, les géants de la fabrication de puces ont créés de nouveaux processeurs, appelés *processeurs multi-cœurs*, dans lesquels plusieurs cœurs sont intégrés sur la même puce. Les processeurs multi-cœurs se sont montrés plus efficaces en terme d'énergie avec un ratio coût/performance bien meilleur que leur ancêtre mono-cœur [3], en effet ils améliorent les performances des applications par exploitation du parallélisme de niveau threads. Des exemples d'architectures multi-cœurs incluent le Kalray MPPA-256 [4], le Tilera Tile CPUs [5], ou encore le Xeon Phi de chez Intel [6].

La migration des systèmes temps-réel vers une utilisation des processeurs multi-cœurs remplit les attentes de performance des applications gourmandes en ressources, mais lève de nombreux problèmes de prévisibilité temporelle. Dû aux effets matériels des processeurs multi-cœurs, garantir les contraintes temporelles des applications critiques et parallèles est un vrai challenge.

L'un des challenges les plus importants est d'estimer, avec précision, le Pire Temps d'Exécution (PTE) du code s'exécutant sur le multi-cœur. Il existe de nombreuses méthodes pour estimer le PTE sur processeur mono-cœur [7]. Ces techniques prennent en compte à la fois les chemins d'exécution du programme et la micro-architecture du cœur. Étendre ces méthodes aux architectures multi-cœurs est difficile, et ce dû aux ressources matérielles, tel que les caches ou les bus, qui sont partagées entre les cœurs, rendant ainsi l'estimation du PTE des tâches dépendant de l'exécution des autres tâches s'exécutant sur les autres cœurs [8, 9]. De plus, sur les architectures avec des caches locaux, l'estimation du PTE des tâches dépend du contenu du cache au démarrage de la tâche, ce qui dépend de la stratégie d'ordonnancement d'exécution des tâches. Le pire temps d'exécution d'une tâche n'est donc plus unique. Il dépend du contexte d'exécution de la tâche (les tâches s'exécutant avant, s'exécutant en concurrence), ce contexte est défini par la stratégie d'ordonnancement et de placement. Dans les faits, il est possible de considérer une estimation pire cas indépendante du contexte, mais la valeur résultante serait trop pessimiste.

De manière symétrique, l'estimation du PTE d'une tâche est nécessaire pour déterminer le placement et l'ordonnancement d'une tâche. Par conséquent, l'ordonnancement et l'estimation du PTE considérant des processeurs multi-cœurs sont des problèmes interdépendants, référés à une situation de poule et d'œuf. À cause de cette interdépendance, nous pensons que des stratégies d'ordonnancement prenant en compte l'entièreté du matériel multi-cœur doivent être définies. Prendre en compte les PTEs dépendants du contexte aide

à l'amélioration de la qualité des ordonnancements, c'est à dire à la réduction de la longueur de ces derniers. Les travaux de thèse présentés dans ce document considèrent la variation du PTE des tâches dû aux effets des caches locaux.

Présentation des objectifs et solutions

Algorithme d'ordonnancement conscient du cache. Nous proposons deux techniques d'ordonnancement pour des architectures multi-cœurs équipées de caches locaux, celles-ci incluent une méthode optimale utilisant une formulation de Programmation Linéaire en Nombre Entier (PLNE), et une méthode heuristique basée sur de l'ordonnancement par liste.

Ces deux techniques ordonnancent une seule application parallèle modélisée par un graphe de tâches, et génèrent un ordonnancement statique partitionné et non-préemptif. Dû à l'effet des caches locaux, chaque tâche τ_j n'est pas caractérisée par une seule valeur de PTE mais plutôt par un ensemble de valeurs de PTE. Le PTE le plus pessimiste d'une tâche, noté PTE_{τ_j}, est observé lorsqu'il n'y a pas de réutilisation de contenu chargé dans le cache par la tâche s'exécutant immédiatement avant τ_j. Un ensemble de valeurs de PTE noté $PTE_{\tau_i \rightarrow \tau_j}$ représente les PTEs d'une tâche τ_j lorsque τ_j réutilise des informations de τ_i, chargées aussi bien dans le cache d'instructions que de données par la tâche τ_i s'exécutant immédiatement avant τ_j sur le même cœur. L'objectif de ces deux techniques est de générer un ordonnancement dont la longeur est aussi courte que possible.

Les évaluations expérimentales sur les cas de tests de la suite de tests StreamIt [10] montrent des réductions significatives sur la longeur des ordonnancements générés par les techniques conscientes du cache comparées à leurs équivalentes ignorant les cache privés. La réduction de la taille de l'ordonnancement observée sur des applications de streaming est de 11% en moyenne avec la méthode optimale et de 9% en moyenne avec l'heuristique. De plus, l'heuristique proposée montre un bon compromis entre longueur des ordonnancements produits et efficacité de leur génération. Dans les faits, la méthode d'ordonnancement par heuristique génère des résultats très rapidement, i.e. 1 second est nécessaire pour générer l'ordonnancement d'un graphe de tâches complexe contenant 548 tâches sur un processeur de 16 cœurs. La différence entre la taille des ordonnancements générés par l'heuristique et la méthode optimale est faible, i.e., 0.7% en moyenne.

Implémentation de méthodes d'ordonnancement conscientes du cache. Nous avons réalisé l'implémentation d'un ordonnancement dirigé par le temps et conscient du cache pour le Kalray MPPA-256, un processeur multi-cœur en grappe. Pour autant que nous le sachions, nous sommes les premiers à créer et implémenter un tel ordonnanceur pour cette machine. Pour l'implémentation, nous avons premièrement identifié les challenges qui surviennent avec ce type d'implémentation, ce qui inclue:

- la pollution du cache et les délais de démarrage d'une tâche dû à l'exécution de l'ordonnanceur ;
- la contention sur le bus partagé ;

- l'absence de cohérence des caches de données.

Ces facteurs expérimentaux ont amené une augmentation du temps d'exécution des tâches, donc changeant le temps de fin des tâches entrainant, ainsi, des modifications du temps de début des tâches suivantes afin de garantir les relations de précédence entre tâches. Á partir de ce constat, nous proposons une formulation PLNE modifiant des ordonnancements conscients du cache en les adaptant aux facteurs expérimentaux identifiés, tout en garantissant la satisfaction des précédences entre tâches et une minimisation de la longueur de l'ordonnancement. De plus, nous proposons une stratégie pour générer le code d'une application prévue pour la machine cible d'après son ordonnancement.

La validation expérimentale des benchmarks avec la suite StreamIT montre la correction fonctionnelle et temporelle de notre implémentation. De plus, nous montrons qu'il est très rapide de trouver des *ordonnancements conscients du cache et adaptables* avec notre formulation PLNE. Enfin, nous quantifions l'impact des facteurs expérimentaux sur la durée de ces ordonnancements, et nous avons pu observé le facteur le plus impactant la longeur de l'ordonnancement: la contention.

Contributions

Les contributions principales des travaux présentés dans cette thèse sont les suivants:

- Nous défendons et validons expérimentalement l'importance d'adresser les effets des caches privés sur les PTEs des tâches lors de l'ordonnancement sur des architectures multi-coœur.
- Nous proposons une méthode d'ordonnancement basée sur PLNE pour statiquement trouver un ordonnancement partitionné et non-préemptif d'une application parallèle modélisée par un graphe dirigé et acyclique. Afin de réduire les temps de génération des ordonnancements, nous proposons également une technique heuristique basée sur de l'ordonnancement par liste.
- Nous fournissons les résultats expérimentaux montrant, entre autres, que les ordonnanceurs proposés génèrent des ordonnancements plus courts que leurs équivalents ignarant les caches locaux.
- Nous identifions les challenges réels qui surviennent lors de l'implémentation d'ordonnancements conscients du cache et dirigés par le temps sur la machine Kalray MPPA-256, et proposons nos stratégies pour dépasser ces challenges.
- Nous explorons l'impact des différents facteurs expérimentaux sur les ordonnancements conscients du cache.

Organisation

Cette thèse, rédigé en langue anglaise, est divisée en 5 chapitres.

- Dans le chapitre 1, nous introduisons les concepts de ces travaux de thèse. Premièrement, nous posons le problème qui a motivé ces travaux. Puis nous présentons les objectifs, ainsi qu'une vue d'ensemble des solutions proposées pour tacler les problèmes identifiés en amont. En second lieu, nous résumons brièvement les contributions de cette thèse.

- Dans le chapitre 2 (chapitre introductif), nous présentons les connaissances prérequises ainsi que les travaux apparentés de la communauté des systèmes temps-réel multi-cœurs. Nous commencons par introduire quelques concepts généraux impliquant les systèmes temps-réel, l'analyse PTE, et l'ordonnancement de tâches. Nous décrivons aussi brièvement la plateforme multi-cœur, les propriétés désirées de prédictabilité dans les architectures multi-cœurs, et les caractéristiques des mémoires caches. Ensuite nous classifions et décrivons brièvement les travaux principaux de la communauté temps-réel sur l'analyse PTE et l'ordonnancement de tâches pour les plateformes multi-cœurs, ce qui nous permet d'identifier les aspects peu étudiés, spécifiquement le problème d'ordonnancement considérant les effets des caches privés.

- Dans le chapitre 3, nous décrivons la génération d'ordonnancement conscient des cache privés. Deux méthodes d'ordonnancement sont présentées pour générer des ordonnancements statiques et non-préemptifs d'une application parallèle, incluant une méthode optimale et une heuristique. Nous présenterons aussi une évaluation de la performance des méthodes d'ordonnancement proposées en termes de qualité des ordonnancements et de rapidité de génération.

- Dans le chapitre 4, nous présentons l'implémentation d'ordonnancements conscients du cache et dirigés par le temps pour les machines Kalray MPPA-256. Nous commencons par décrire l'architecture de la plateforme, suivi de notre implémentation de l'ordonnancement. Ensuite nous identifions les défis réels qui surviennent lors du déploiement des ordonnancements sus-mentionnés, et présentons nos stratégies pour dépasser les difficultés identifiées. Enfin une évaluation expérimentale valide la correction fonctionnelle et temporelle de notre implémentation.

- Dans le chapitre 5, nous concluons ces travaux de thèse et proposons des perspectives de travaux futurs.

Chapter 1

Introduction

1 Hard real-time multi-core systems: timing predictability challenges

Real-time embedded systems, i.e., those for which timing requirements prevail over performance requirements, are now widespread in our everyday lives. In particular, real-time applications can be found in personal cars, airplanes, space ships, nuclear plants. Additionally, according to [1] the embedded systems market is likely to witness a high growth in the coming years. The embedded systems market was valued at USD 84.55 Billion in 2016 and is expected to grow at a Compound Annual Growth Rate (CAGR) of 4.05% between 2017 and 2023.

With the ever-increasing demand for safer but more compute-intensive applications, single-core architectures are no longer suitable choices for deploying real-time systems due to the technological limits of the architectures (typically referred to as power-wall [2]). In order to overcome the issue, the leading chip manufacturers have been offering new computing platforms, called *multi-core platforms*, in which multiple cores are integrated within a single chip. Multi-core platforms have been shown to have more energy-efficient and better performance-per-cost ratio than their single-core counterpart [3], as they improve the application performance by exploiting thread-level parallelism. Examples of multi-core architectures include the Kalray MPPA-256 [4], Tilera Tile CPUs [5], and Intel Xeon Phi [6].

Migrating real-time systems to multi-core platforms ensures the satisfaction of compute-intensive applications' performance, but raises the issue of the timing predictability of the systems. Due to the effects of multi-core hardware (i.e., for example, local caches, shared resources between cores), guaranteeing the real-time constraints of safety-critical parallel applications on multi-core platforms is challenging.

One important challenge is to precisely estimate the Worst-Case Execution Time (WCET) of codes executing on multi-cores. Many WCET estimation methods have been designed in the past for single-core architectures [7]. Such techniques take into account both the program paths and the core micro-architecture. Extending them to multi-core architectures

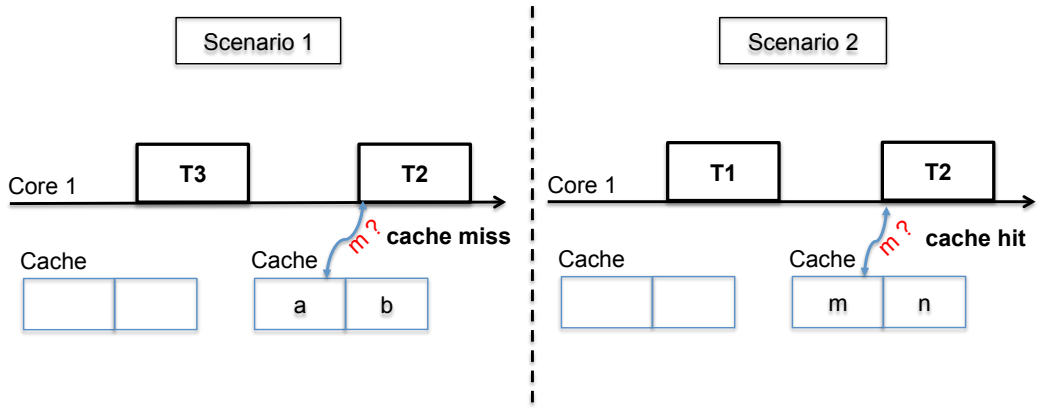


Figure 1.1: The influence of scheduling strategies on the WCET of tasks

is challenging, because some hardware resources, such as caches or buses are shared between cores, which makes the WCET of a task dependent on the tasks executing on the other cores [8, 9]. Additionally, on architectures with local caches, the WCET of a task depends on the cache contents when the task starts executing, which depends on the scheduling strategy. The WCET of one task is thus no longer unique. It depends on the execution context of the task (tasks executed before it, concurrent tasks), which is being defined by the scheduling strategy. In fact, ones could consider a context-independent WCET for a task, but the value would be too pessimistic.

Motivating example. Let us consider an overly simplified system made of three tasks, named T1, T2, and T3, executing on a dual-core processor, for which each core is equipped with an unified private cache containing two lines. T1 and T2 access to the same memory block, named m, whereas T3 is code and data independent with both T1 and T2. We consider two execution scenarios to reveal the effect of scheduling strategies on tasks' WCET. In the first scenario, T2 is placed on the same core as T3 and executes right after T3 (as illustrated in the left side of Figure 1.1). In the second scenario, T2 is placed on the same core as T1 and executes right after T1 (as illustrated in the right side of Figure 1.1). We assume that the contents of the cache at the beginning are empty. In the first scenario, the request of T2 for the memory block m is a cache miss since at the requesting time the memory block has not been loaded in the cache yet. As a consequence, T2 has to wait until the memory block m is loaded from the main memory to the cache. In contrast, in the second scenario, the request of T2 for the memory block m is a cache hit since the memory block is already stored in the cache during the execution of T1. Since the access to the cache is much faster than the access to the main memory, the worst-case execution time of T2 in the second scenario is lower than its worst-case execution time in the first scenario.

This example shows the influence of the mapping and the execution order of tasks on their worst-case execution time due to the effect of private caches. Systematically, the worst-case execution time of tasks are needed for scheduling strategies to determine the mapping and

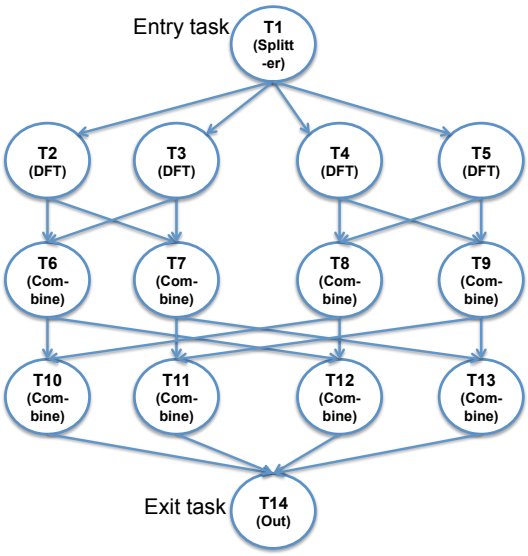


Figure 1.2: Task graph of a parallel version of a 8-input Fast Fourier Transform (FFT) application [11]

the scheduling of the tasks. Therefore, task scheduling and WCET estimation for multi-core platforms are inter-dependent problems, referred to as a chicken and egg situation.

Thesis claims. Due to the aforementioned interdependent problems, we believe that scheduling strategies that are aware of the multi-core hardware have to be defined. Taking into account context-sensitive WCETs for a task rather than a single context-independent WCET helps to enhance the quality of schedules, i.e., reduce schedules length (or makespan). In the scope of this PhD work[1], we consider the variation of the WCET of a task due to the effect of private caches.

2 Objectives and solution overview

2.1 Cache-conscious scheduling algorithms

Our first objective in the thesis is to propose cache-conscious scheduling strategies, that take benefit of cache reuse between tasks. Each task has distinct WCET values depending on which other task has been executed before it on the same core (WCETs are context-sensitive). The proposed scheduling strategies map tasks to cores and schedules tasks on cores; the goal is to account for cache reuse to obtain the shortest schedules. We focus on a single parallel application, modeled as a task graph, in which nodes represent tasks

[1]This PhD work is a part of PIA project CAPACITES (Calcul Paralléle pour Applications Critiques en Temps et Sûreté.), reference P3425-146781

and edges represent dependence relations between them. Additionally, in cache-conscious schedules generation, we solely consider the effect of private caches, while other hardware related factors are left for implementation stage.

We propose two different methods to determine a static partitioned non-preemptive schedule aiming at minimizing the schedule length, for a parallel application by taking into account the variation of tasks' WCETs due to reuse of code and data between tasks. The first method is based on an Integer Linear Programming (ILP) formulation and produces optimal schedules under the considered hypotheses. The second method is a heuristic method, which is based on list scheduling. The proposed heuristic scheduling approach produces schedules very fast, and the length of heuristic schedules are close to the optimal ones.

To further motivate our research, let us consider an example, a 8-input Fast Fourier Transform application [11]. Its task graph is shown in Figure 3.2. For instance, T2 and T3 feature code reuse since they call the same function, and T2 and T6 feature data reuse since the output of T2 is the input of T6. On that example, we observe a reduction of the WCETs of tasks of 10.7% on average when considering the cache affinity between pairs of tasks that may execute consecutively on the same core. The schedule length for that parallel application was reduced by 8% by using the Integer Linear Programming (ILP) technique presented in Chapter 3 as compared to its cache-agnostic equivalent.

Noted that the definition of the WCET of a task in this PhD work is different with the one defined in the literature. In the literature, the WCET of a task is defined as upperbound of the execution times of the task when executing in isolation. Our definition about the WCET of a task is upperbound of the execution times of the task when considering the contents of the cache at the beginning of the execution of the task.

2.2 Implementation of cache-conscious schedules

Once the benefit of considering the effect of private caches on task scheduling is shown, our successive target is to implement time-driven cache-conscious schedules on a real multi-core hardware, that is the Kalray MPPA-256 machine [4]. In the implementation stage, we first identify the practical challenges arising when deploying those schedules on the machine, such as shared bus contention, the effect of time-driven scheduler itself, the absence of hardware-implemented data cache coherence. Those practical effects require modification of the cache-conscious schedules. We thus propose an ILP formulation to adapt the cache-conscious schedules to the identified practical factors, such that the precedence relations of tasks are still satisfied, and the schedules length of the adapted schedules are minimized. Additionally, we propose a strategy for generating the code of applications to be executed on the machine according to the adapted cache-conscious schedules.

3 Contributions

The main contributions of the PhD work are as follows:

- We argue and experimentally validate the importance of addressing the effect of private caches on tasks' WCETs in scheduling.

- We propose an ILP-based scheduling method and a heuristic scheduling method to statically find a partitioned non-preemptive schedule of a parallel application modeled as a directed acyclic graph.
- We provide experimental results showing, among others, that the proposed scheduling techniques result in shorter schedules than their cache-agnostic equivalent.
- We identify the practical challenges arising when implementing time-driven cache-conscious schedules on the Kalray MPPA-256 machine, and propose our strategies for overcoming the identified challenges.
- We investigate the impact of different practical factors on cache-conscious schedules.

4 Organization

The rest of this thesis is divided into three main chapters. In Chapter 2, we present fundamental background and literature review about real-time multi-core systems. We begin by introducing some general concepts dealing with real-time systems, WCET analysis, and task scheduling. We also briefly describe multi-core platforms, the desired properties of time-predictable multi-core architectures, and the characteristics of cache memories. We then classify and briefly describe the main works from the real-time literature related to WCET analysis and task scheduling for multi-core platforms. This allows us to identify some aspects that have not been much studied, specifically the scheduling problem that takes into account the effect of private caches.

Once those fundamental knowledge are introduced, we present the different contributions of the PhD work. In Chapter 3, we describe cache-conscious schedules generation. Two scheduling methods are presented for generating static partitioned non-preemptive schedules of parallel applications, including an optimal method and a heuristic one.

The implementation of time-driven cache-conscious schedules on the Kalray MPPA-256 machine is presented in Chapter 4. We first describe the architecture of the machine, and our time-driven scheduler implementation. We then identify the practical challenges arising when deploying the time-driven cache-conscious schedules on the machine, and present our strategies for overcoming the identified issues.

Finally, in Chapter 5, we conclude this PhD work and propose some perspectives for future work.

Chapter 2

Real-time systems and multi-core platforms: background and state-of-the-art

In this chapter, we present fundamental background and the literature review about real-time multi-core systems. We begin by introducing general concepts dealing with real-time systems, worst-case execution time (WCET) analysis, and task scheduling. We also briefly describe multi-core platforms, the desired properties of time-predictable multi-core architectures, and the characteristics of cache memories. We then classify and briefly describe the main works from the real-time literature related to WCET analysis and task scheduling for multi-core platforms.

2.1 Background

2.1.1 Real-time systems

Real-time systems are defined as systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [12].

Real-time systems are subjected to timing constraints. Depending on the criticality of those timing constraints, real-time systems can be classified in three categories [13]:

- *Soft real-time systems*: missing deadlines does not cause the systems to fail, but it may cause performance degradation. Examples of soft real-time systems include Internet Protocol (IP) communication, video streaming.
- *Hard real-time systems*: missing deadlines may cause catastrophic consequences on these systems. Examples of hard real-time systems include engine control unit, cruise control system in automotive systems, flight control systems, and chemical/nuclear plant control systems.

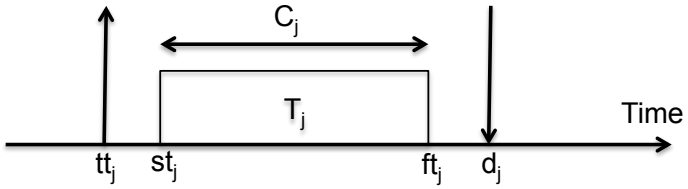


Figure 2.1: Typical parameters of real-time tasks

In order to support hard real-time applications, the following properties are necessary when designing real-time systems [13].

- *Timeliness*. Real-time systems have to produce the correct results in a timely manner. In other words, both functional correctness and timing correctness have to be satisfied.
- *Predictability*. The upperbounds of the execution times of tasks must be able to be estimated offline, and the satisfaction of timing constraints of tasks must be able to be checked before the system starts executing.
- *Efficiency*. Most real-time systems are embedded into small devices with hard constraints in term of space, weight, energy, memory, and computational power. Therefore, managing available resources efficiently is an essential feature of real-time systems.
- *Fault tolerance*. Real-time systems should not be damaged by single hardware and software failures.

In general, a real-time task can be characterized by the following parameters (as illustrated in Figure 2.1)

- *Arrival time (tt_j)*. The time at which the task is ready to execute,
- *Start time (st_j)*. The actual time at which the task starts executing,
- *Computation time (C_j)*. The upperbound of the execution times of the task when executing in isolation, referred to as worst-case execution time (WCET).
- *Finish time (ft_j)*. The time at which the task finishes its execution,
- *Deadline (d_j)*. The time before which the task should be completed.

2.1.2 Worst-case execution time estimation

The worst-case execution time (WCET) of a task is the maximum of its execution times under any input configurations and any initial hardware states [7].

The WCET of tasks are required to validate timing constraints. Therefore, WCET estimation is an essential step in the development and the validation process for hard real-time systems.

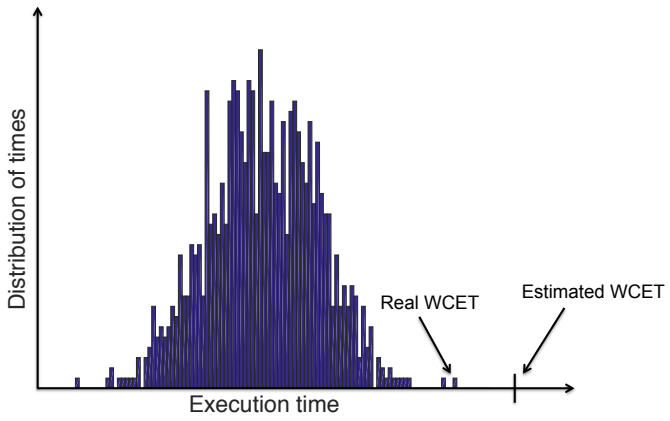


Figure 2.2: The variation of execution times of a task depending on the input data or different behavior of environment

As shown in Figure 2.2, a task may have variation of execution times depending on different input data or different initial hardware states. It is in general impossible to determine the exact WCET for a task (real WCET in the figure) since it is too time consuming to explore exhaustively all possible executions. Therefore, WCET analysis techniques, which produce estimated WCET for tasks (estimated WCET in the figure), are in need.

The estimated WCET is defined as the upperbound of estimated execution times of a task when executing in isolation. For simplicity, we use the term "WCET" as "estimated WCET" hereafter. In [7], Wilhelm et al. give an overview of the WCET analysis techniques and the available WCET analysis tools. According to this survey, WCET analysis techniques can be classified into the following categories:

- *Static WCET analysis methods*: the methods derive the WCET of tasks without running them on real hardwares. The approaches combine static analysis techniques with the abstract models of hardware architectures to produce WCET.
- *Measurement-based methods*: the methods measure the actual execution times of tasks when executing them on a real processor or a simulator of the processor under representative test conditions. Then the WCET of tasks are estimated based on the measured values according to a specific formulation. For example, for end-to-end measurements, the WCET of a task is equal to the maximum of its actual execution times.

Typically, static WCET analysis methods require three steps:

- *Flow analysis.* That step builds the control flow graph of the task under analysis from its executable code, and determines the loop bounds, as well as feasible execution paths through the task [14, 15].
- *Low-level analysis.* In that step, the worst-case execution costs of basic blocks are computed with taking into account the trace of the target hardware (e.g., caches,

pipeline, etc.). An example of the techniques that analyze the trace of hardware components is abstract interpretation [16].

- *WCET computation.* In the step, the WCET of the task is derived by combining the worst-case execution costs of basic blocks estimated in the second step with the loop bounds and the feasible execution paths found in the first step. The popular method used to find the longest execution path and its execution time is the Implicit Path Enumeration Technique (IPET) [17]. IPET considers all execution paths implicitly by using integer linear programming.

Static WCET analysis techniques vs. measurement-based methods. There are two main criteria for evaluating the quality of estimated results produced by WCET analysis techniques. The first criterion is *safety*, i.e., whether the upperbound of the execution times is produced or not. The second criterion is *precision*, i.e., how close of the produced upperbound to the exact value. In other words, the estimated WCETs have not to be *underestimated*, but should not be *overestimated* too much.

The static WCET analysis methods cover all feasible execution paths and consider all possible context dependencies of processor behavior. Therefore, the methods guarantee that the estimated WCETs are safe (assuming the hardware model is safe). For this safety, the methods have to model processors behavior. Since the abstraction models lose information, especially for complex hardwares architecture, so that the estimated WCETs are possibly imprecise. Several commercial and academic tools that use static analysis techniques to compute WCETs are available, such as aiT [18], Bound-T [19], OTAWA [20] or Heptane [21].

On the other hand, measurement-based methods use measurements instead of modeling processor behavior. The changes in execution times of tasks due to hardware context-dependent may be missed if the measurements are not performed in the worst-case initial state of hardwares. Additionally, in general it is too time consuming to derive input data that lead to the worst-case execution path. As a consequence, the estimated WCETs resulted by measurement-based methods are not guaranteed to be safe. The advantage of the methods is that it is simple to apply them to a new hardware architecture since the abstract model of the hardware architecture is not required. Rapitime distributed by Rapita Ltd [22] is one of the commercial tools that perform measurements in the process of estimating WCETs.

2.1.3 Real-time task scheduling

For single-core platforms, task scheduling is a process that determines when and in what order tasks should execute. For multi-core platforms, along with the aforementioned scheduling problem for every core, task scheduling also attempts to solve a mapping problem, which determines on which cores tasks should execute.

Task models and the objective of task scheduling. In the literature, there are many task models have been used in real-time task scheduling. In this review, instead of listing all task models, we present two particular classifications of them according to dependencies between tasks, and the frequency of execution of tasks.

- Independent task models and dependent task models. In the independent task models, there is no dependencies between tasks, whereas, in the dependent task models, there are precedence relations between pairs of tasks. Directed Acyclic Graph (DAG) [23] is one of representative graphs used for dependent task models.

- Single execution task models and recurrent execution task models. In single execution task models, tasks have one and only one execution instance, whereas, in recurrent execution task models, tasks compose of an infinite sequence of jobs (as described in the survey [24]). Examples of recurrent execution task models are periodic task models, in which the distance of arrival times of jobs belong to the same task are fixed, and sporadic task models, in which the distance of arrival times of these jobs are varied.

  When applying to single execution task models, the objective of task scheduling methods is to minimize schedules length (or makespan), whereas, when applying to recurrent execution task models, the objective of task scheduling methods is to ensure that all tasks meet their deadline.

Classification of task scheduling algorithms. The task scheduling algorithms for single-core platforms can be classified as:

- *preemptive scheduling*: a task can be preempted by a higher priority task during its execution.

- *non-preemptive scheduling*: once a task starts executing, it will execute until completion without being preempted.

In preemptive scheduling algorithms, a task which requires "urgent" service (i.e., task whose deadline is reached soon) is allowed to preempt the current execution task for possessing computing resources of the core. Preemptions introduce runtime overheads due to preemption itself, cache misses, and prefetch mechanisms. Furthermore, due to the preemption, hardwares' state (e.g., caches, pipeline, etc.) are hard to manage. As a consequence, the preemption costs are difficult to predict [25, 26]. For overcoming this issue, many approaches have been proposed (read [27] for a survey). Some approaches have been proposed to limit the number of preemptions for each task, thus reducing the preemption costs. Besides, some approaches explicitly introduce preemption points in programs for improving the predictability of the preemption costs.

Regarding the allocation of tasks on multi-core platforms, task scheduling algorithms can be classified as:

- *global scheduling*: a task is allowed to execute on different cores.

- *partitioned scheduling*: a task is allocated to one and only one core.

In global scheduling, the migration of tasks leads to the change of hardware state (i.e., cache contents), which causes migration overheads are hard to predict. Semi-partitioned scheduling techniques [28] which make most tasks non-migrating, have been proposed to overcome this issue.

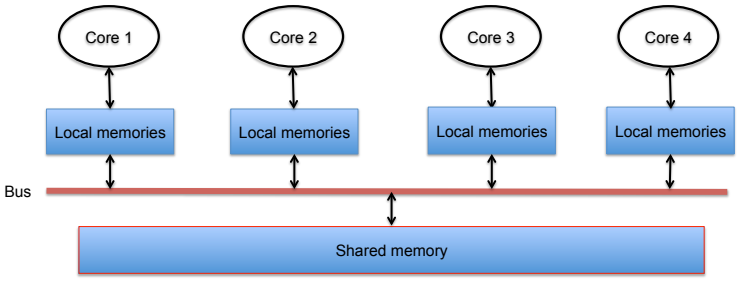


Figure 2.3: An example of multi-core architecture

Additionally, task scheduling algorithms can be classified based on the relative time (to the execution of applications) at which the scheduling decisions are made.

- *online scheduling*: the allocation and the schedule of tasks are determined at run time [29].
- *offline scheduling*: the allocation and the schedule of tasks are determined before the execution of applications [23].

Furthermore, according to the way used of activating the execution of tasks, task scheduling algorithms can be classified as:

- *event-driven scheduling*: the execution of tasks are triggered at the emergence of a specific event.
- *time-driven scheduling*: the execution of tasks are triggered at predefined instants of time.

2.1.4 Time-predictable multi-core hardware

Multi-core architectures. According to the Moore's Law [30] the number of transistors on a chip doubles once in every 18 months to meet growing demands on computing power. Studies in [2] have shown that integrating more and more transistors on a single core to meet the demands comes along with major bottlenecks such as issues on power consumption, heat dissipation, chip fabrication costs, as well as faulty hardware. Those issues impair the reliability and the life expectancy of chips, and require more budgets for cooling systems. The consideration on those issues is the key driving force behind the development of multi-core architectures.

Typically, a multi-core architecture is a single chip which contains several cores on the chip (as illustrated in Figure 2.3). As compared to single-core architectures in which tasks running at the same time compete against each other for chip's computing resources, in multi-core architectures those tasks can be assigned to different cores to run in parallel, thus boosting the performance of systems [3]. Additionally, with the capability of executing tasks in parallel, cores inside multi-core architectures are not necessarily clocked at a high

frequency. That property makes multi-core architectures more energy efficient than single-core architectures [31].

According to the demand of applications, multi-core architectures can be implemented in different ways. Multi-core architectures can be implemented as a group of homogeneous cores or as a group of heterogeneous cores or as a combination of both. In homogeneous multi-core architectures, all cores are identical, whereas heterogeneous multi-core architectures consist of dedicated application specific cores. Each core is equipped with local memories (e.g., caches, scratchpad memories (SPM)). Tasks executing on different core can share data either through a shared memory or with the support of cache coherence protocol (if have) [32].

Major challenges faced by multi-core architectures. Although multi-core architectures offer many advantages, the technology faces a number of major challenges. First of all, in order to exploit the computing power of multi-core architectures, applications are required to be written in a way that exposes parallelism. Therefore, it requires to redesign a huge amount of legacy programs developed for single-core architectures. For addressing this issue, much effort have been spent on developing compilers to automatically generate parallel tasks for a given application [33]. Additionally, many parallel programming interfaces have been proposed, such as OpenMP [34], Pthreads [35].

The most important challenge arising when embedding multi-core architectures into real-time systems is shared resources contentions [8, 9]. Cores inside a multi-core architecture share hardware resources, such as bus, memories. Due to hardware sharing, the execution of tasks are delayed (when they access shared resources currently used by another core). Precisely estimating the delays offline is hard since the delays depend on many factors, such as the concurrent tasks involve to the contentions, the actual time at which the contentions occur, and the resources arbitration policy. Many research have been spent on limiting the issue, including the research in task timing analysis, in task scheduling, as well as in time-predictable multi-core architectures. The brief overview of the research in task timing analysis, and task scheduling will be given in Section 2.2.1, and Section 2.2.2, respectively. In the rest of the section, we describe the main characteristics of time-predictable multi-core architectures.

Time-predictable multi-core architectures. The main objective in designing time-predictable multi-core architectures is to eliminate (or mitigate) properties that make WCET analysis difficult, and potentially excessively pessimistic. The desired characteristics of time-predictable multi-core architectures are summarized as follows:

- Timing compositionality. The architectures with that property allow the result of worst-case timing analysis of each hardware component can be simply added together to determine the worst-case execution time of whole systems [36]. An example of multi-core platforms having that property is Kalray MPPA-256 [4].

- Spatial and temporal shared resource isolation. Isolating shared resources helps to mitigate the amount of possible interferences at the hardware level. This will isolate spatial and temporal of concurrent tasks running on different cores. Therefore, WCET

analysis for a task can be carried out independently. Such architecture is proposed in [37], which uses time-division multiple access(TDMA)-based bus sharing policy. In that architecture, the bus schedule contains slots of a certain size, each with start time, that are dedicated to cores. Therefore, any memory accesses issued by a core in its bus slots is guaranteed to be contention free. However, dedicating bus slots to cores impairs average performance of multi-core systems. The reason is that memory accesses of a core must to wait until its dedicated bus slots even though there is no requests from other cores.

- Local memories accesses are predictable. Such architectures with that property use software-controlled memory, i.e., scratchpad. Scratchpad is a fast on-chip memory and it is explicitly controlled by the user or managed by the system software, e.g., a compiler. Therefore, each memory access to scratchpad becomes predictable. T-Crest [38] is an example of multi-core platforms equipped with scratchpad memories.

- Controlling the time at instruction set architecture (ISA) level. PRET (precision time machine) [39] has been developed with this goal in mind. In that architecture, ISA is extended with time constraints. Therefore, the bounds (i.e., lower bound and upper bound) of the execution time of a code block are specified, which can be used to improve the estimation of shared resources contentions.

Since most designs for multi-core architectures are not dedicated to real-time systems, and there is a trade-off between the performance of whole systems and the timing predictability needed to be considered, so that it still requires a lot of work in the future to produce efficient time-predictable multi-core architectures.

2.1.5 Cache memories

Main principles. As demonstrated in [40], there is a big gap between the processor and memory speeds. In order to improve the performance of whole systems, bridging that gap is in need. Cache memories have been proposed with that goal in mind. As depicted in Figure 2.4, a cache is a memory located between the processor registers and the main memory. A cache is smaller than a main memory, and the access time to the cache is much faster than the main memory. The function of a cache is to store the memory blocks loaded from the main memory. Therefore, the later accesses to those blocks by the core will be served directly by the cache. When the core finds a memory block it needs in the cache, the access to the memory block is a cache hit, otherwise, the access to the memory block is a cache miss. A cache miss has a much higher cost (in term of access time, and power consumption) than a cache hit [41], because missing blocks have to be loaded into the cache from the main memory.

The effectiveness of caches is based on the principle of locality of references, explained as follows:

- *spatial locality*: there is a high probability that a reference will be requested if a reference close to it has been requested recently, i.e., instructions are often requested sequentially.

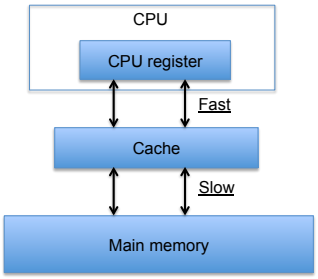


Figure 2.4: The location of cache

- *temporal locality*: already-requested references are more likely to be requested again in the near future, i.e., instructions in a loop.

Caches can be categorized depending on the type of information they store:

- *Instruction caches*: store only program instructions;
- *Data caches*: store only program data;
- *Unified caches*: store both program instructions and program data.

According to [42], smaller caches have lower access latencies, whereas larger caches have higher ones. However, smaller caches are more costly than higher ones. For achieving fast memory accesses, and at the same time providing large memory space at the low cost, memory hierarchies have been introduced. As shown in Figure 2.5, the memory hierarchy is made of several cache levels. Caches which are closer to the CPU are smaller, faster, and more costly than ones which are further to the CPU.

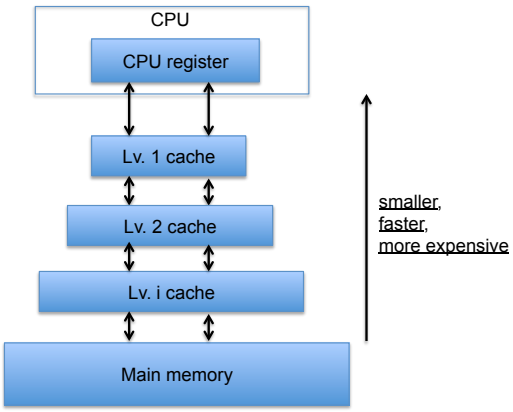


Figure 2.5: An example of memory hierarchy

Cache organization. A cache is divided into equal-sized cache lines. Each cache line can store one memory block loaded from the main memory, which contains a set of data (for the data cache) or several instructions (for the instruction cache) in order to benefit from spatial locality. Since the number of cache lines is lower than the number of main memory blocks, blocks stored in the cache will be evicted and replaced by new ones loaded from the main memory.

Different strategies (which are described in [43]) have been proposed for determining to which cache line a given memory block will be mapped:

- *Direct mapping*: a memory block is mapped into only one possible cache line. The mapping is determined based on the address of the memory block.
- *Fully-associative mapping*: a memory block is mapped to any cache line.
- *Set-associative mapping*: a cache is divided into sets of equal number of cache lines, and a memory block is mapped to any cache line in a particular set. The address of the memory block is used to determine to which set the memory block is mapped.

For fully-associative mapping and set-associative mapping, when the set to which a new memory block is mapped is full, it is essential to have a *replacement policy* to decide which cached items should be evicted to make room for the new one. For direct mapping, there is only one possible location to which a memory block is mapped, so that it does not require a replacement policy.

Panda et al. [44] give a survey of replacement policies, including optimal and sub-optimal algorithms. The optimal algorithm is the one that can look into the future and determine what should be placed in the cache according to the limit of the cache's capacity and organization [45]. However, in case of dynamic systems in which memory references are only revealed at run time, the optimal strategies can not be applied. For those systems, there is no optimal replacement policy (according to [46]). Therefore, a number of sub-optimal replacement policies have been proposed:

- *Least Recently Used (LRU)*: the memory block which has been least recently used is discarded.
- *First-In First-Out (FIFO)*: the memory block which has been in the cache longest is discarded.
- *Least Frequently Used (LFU)*: the memory block which has experienced the fewest references is discarded.

The study in [47] showed that LRU is the best policy among these replacement policies in term of cache-state predictability.

Timing anomalies. The presence of caches substantially improves the performance of programs, but makes the timing behavior of the processors harder to predict. Lundqvist et al. [48] first defined the notion of the timing anomaly for out-of-order processors, stating that a cache hit which is locally faster (than a cache miss) leads to an increase of the execution time of the whole program.

2.2 Timing analysis for multi-core systems: state-of-the-art

2.2.1 WCET analysis techniques for multi-core architectures

In static WCET analysis for single core platforms, the task under analysis is assumed to be not influenced by any external events, i.e., things related to the execution of another task or hardware devices, such as memory refreshes. However, this assumption needed to be reconsidered when applying to multi-core platforms since resources sharing introduces interferences on the execution of tasks [8, 9]. The interferences include data corruption, i.e., data stored in the shared cache, which is currently used by a task is invalidated by concurrents tasks, as well as conflicts on shared bus, i.e., several tasks have accesses to shared memories at the same time.

Due to the effect of shared resources contentions, WCETs produced by measurement-based techniques for multi-core platforms are possibly unsafe. The reason is that it is too time consuming to exhaustively capture all possible shared resources contentions at run time of tasks. In the section, we focus on static WCET analysis techniques, which is the most suitable when considering hard real-time tasks.

Many approaches have been proposed to address the resources sharing issues. According to the survey in [49], those approaches can be classified into two categories:

- The first category includes approaches that take all possibles interferences into account when computing the WCET of a task. Most approaches in that category require the knowledge of all tasks that may execute concurrently to the task under analysis. Some solutions have been proposed to analyze all possible contentions on shared caches by concurrent tasks [50, 51, 52, 53]. Those approaches first separately perform cache analysis (for both private caches and shared caches) for every task (i.e., by ignoring interferences). The analyzed results for the shared cache are then modified with considering interferences (caused by concurrent tasks). Additionally, some solutions have been proposed for modeling the interaction of shared caches and shared bus with other basic microarchitectural components, such as pipeline and branch predictor [54]. Furthermore, event-bus arbitration and more complex processor pipeline have been analyzed in [55, 56].

- The second category includes approaches that aim at controlling contentions to ease the WCET estimation. Most approaches in that category analyze the timing behavior of tasks independently with the support of either predictable hardware models or predictable execution models, which ensure the temporal and spatial isolation between running tasks. Shared cache thrashing can be avoided by using cache partitioning techniques [57, 58]. Additionally, the accesses of tasks to the shared memory can be isolated through either bank privatization [59, 60] or memory bandwidth reservation [61, 62, 63, 64, 65]. Furthermore, PRedictable Execution Model (PREM) [66] has been proposed to make the execution of tasks more predictable. In PREM, the code and data of a task are prefetched in the local memories (i.e., private locked cache or scratchpad memory) of the core to which the task is assigned, thus preventing those data from being evicted by the execution of concurrent tasks.

Along with WCET analysis approaches that address the resources sharing issues, there are many approaches focusing on communication/synchronization between tasks executing on different cores [67, 68, 69, 70]. In [67], they provide formulas that combine the WCETs of code snippets and the worst-case stall time (WCST) due to synchronization to obtain the WCET of a parallel application. In [69], they introduce new locking primitives to reduce the WCST, thus improving the estimated WCET of whole applications. In contrast to those approaches in which WCET analysis of different code snippets are performed separately, the approach proposed in [70] integrates the timing analysis of code regions running on the same core at once. By doing that, the hardware states (i.e., cache state) between the execution of different code regions on the same core are captured accurately, which improve the estimated WCET of the application.

2.2.2 Task scheduling algorithms for multi-core architectures

Much research effort has been spent on scheduling for multi-core platforms. Research on real-time scheduling for independent tasks is surveyed in [7]. This survey gives a taxonomy of multi-core scheduling strategies: global vs. partitioned vs. semi-partitioned, preemptive vs. non preemptive, time-driven vs. event-driven. Besides, there are many studies on real-time scheduling for dependent tasks. A seminal study in this class of work is presented in [71]. In this study, tasks are scheduled with respect to their precedence relations (i.e., tasks are only ready to be executed after the termination of their predecessors). Along with precedence relations between tasks, the scheduling approach presented in [72] also considers exclusion relations between tasks (i.e., at a time only one task has accesses to shared resources). The scheduling problem addressed in these works is partitioned non-preemptive scheduling. As noted in [73], finding optimal solution for partitioned non-preemptive scheduling problems is NP-hard. In order to overcome the issue, a lot heuristic scheduling approaches are proposed (please refer to [23] for the survey of these approaches). Additionally, there are scheduling approaches that simultaneously schedule tasks and messages exchanged between them. Techniques proposed in [74, 75] consider core-to-core communication, while techniques proposed in [76, 77, 78] consider communication through Network-On-Chip(NoC). Furthermore, many scheduling methods that address the effect of shared resources contentions have been proposed. The main concept of the approaches are presented below.

Taking into consideration shared resources contentions in task scheduling. Most scheduling approaches in that category integrate interference delays in task scheduling problem. The goal is to minimize shared resources interference so as to minimize the schedules length. The underlying concept of these scheduling approaches is based on the fact that WCET estimation and tasks scheduling are interdependent problems, i.e., scheduling algorithms require WCETs of tasks as prior knowledge, whereas, the WCETs of tasks are varied depending on their mappings and their execution order. Such scheduling approaches in the category have been proposed in [79, 80, 81]. In [79, 80] they focus on modeling shared cache contention, whereas in [81] they focus on modeling shared bus contention. Also attempting to reduce shared bus contention, but in [82], Martinez et al. approaches in a different way. Given the fixed mapping and the execution order of tasks, they introduce slack time between the execution of pairs of tasks consecutively assigned to the same core to limit the

contention between concurrent tasks. In that way, the contentions that existed in existing schedules are reduced.

Jointly considering task and memory scheduling. The main goal of scheduling approaches in that category is to achieve temporal isolation between running tasks. Many scheduling approaches adopt the PREM model [66], in which the execution of task is divided into a memory phase and an execution phase. Since the execution phase of tasks are ensured to be free from contention, scheduling approaches only have to pay attention to mitigate/advoid memory access delays in the memory phases of tasks. The study in [83] uses Time Division Multiple Access (TDMA) bus schedule policy to ensure that the memory phases of concurrent tasks are free from contention. The studies in [84, 85] attempt to hide memory access latencies of memory phases of tasks by overlapping the execution phase of a task with the memory phases of other tasks.

Jointly performing task scheduling and memory allocation. The main goal of scheduling approaches in that category is to achieve spatial isolation between running tasks. The scheduling approaches proposed in [86, 87] take the benefit of banks privatization of shared memory thanks to the support of the Kalray MPPA-256 machine [4]. In [86], the private code and data of tasks which are assigned to different cores are allocated to different banks, only shared data between tasks are mapped to the same banks. Therefore, tasks have private accesses to their dedicated memory banks during their execution. Scheduling strategies are designed to ensure that there is no contention between the reading/writing process of concurrent tasks to the shared banks. In [87], the workloads of tasks are allocated with respect to shared memory capacity. Scheduling strategies are designed to ensure that the contentions between concurrent tasks are minimized. Regarding the architectures with shared caches, cache partitioning techniques (i.e., isolate cache space for cores/tasks) are combined with scheduling techniques to minimize/advoid inter-core interference. For example, in [88, 89], cache partitioning and task scheduling are jointly performed such that at any time the cache space of any two running tasks are non-overlapped.

2.3 Summary and thesis context

In this chapter, we introduced basic notions related to real-time systems, WCET analysis, task scheduling, multi-core architectures, and cache memories. Then we briefly describe existing strategies in WCET analysis, and in task scheduling for multi-core platforms.

Due to the effects of multi-core hardware, WCET estimation and task scheduling are interdependent problems. In this PhD work, we address that problem by taking into consideration the effect of local caches on tasks' WCETs in task scheduling. We perform scheduling for a single parallel application, which is made of (single execution) dependent tasks, on a timing compositionality multi-core platforms, in which cores are homogeneous, and every core is equipped with local caches. Additionally, according to the discussion in Section 2.1, it is hard to manage run time overheads in preemptive, and global scheduling. Therefore, we focus on non-preemptive, and partitioned scheduling. Schedules are generated offline, and the execution of tasks in the schedules are triggered at predefined instants of time.

Chapter 3

Cache-conscious scheduling: algorithms

This chapter presents cache-conscious scheduling methods for multi-core architectures, in which every core is equipped with local instruction and data caches[1]. Two scheduling methods are presented for generating static partitioned non-preemptive schedules of parallel applications, including an optimal method and a heuristic one. The optimal method is based on an Integer Linear Programming (ILP) formulation, which produces optimal schedules whose length is minimized. The heuristic method is based on list scheduling, which produces schedules very fast, and the schedules length are close to the optimal ones. Both the scheduling methods take into account the effect of private caches on tasks' WCETs, i.e., the WCET of a task varies depending on the amount of cache reuse according to the execution order of the task. Note that this chapter focuses on schedules generation for an abstraction of multi-core architectures except caches. Practical challenges arising when implementing cache-conscious schedules on a real multi-core hardware will be addressed in Chapter 4.

The organization of the chapter is as follows. Section 3.1 describes our system model and formulates the scheduling problem. Section 3.2 introduces the proposed ILP formulation and the proposed heuristic scheduling method. We experimentally evaluate our proposed scheduling methods in Section 3.3. Section 3.4 surveys related work. Finally, we summarize the contents of the chapter in Section 3.5.

3.1 System model and problem formulation

3.1.1 Hardware model

The class of architectures addressed in this work is the class of identical multi-core architectures, in which each core is equipped with a private instruction cache and a private data

[1]This work was published in the proceeding of Euromicro Conference on Real-Time Systems (ECRTS) 2017 [90].

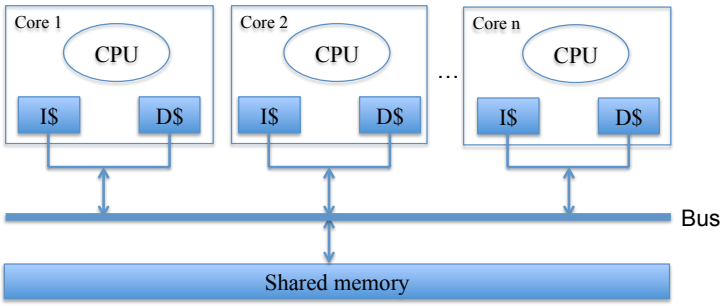


Figure 3.1: Considered multi-core architecture

cache, as illustrated in Figure 3.1. Tasks executing on different cores communicate through a shared memory.

3.1.2 Task and execution model

Each application is modeled as a Directed Acyclic Graph (DAG) [91], as illustrated in Figure 3.2. A node in the DAG represents a task, denoted τ_i. An edge in the DAG represents a precedence relation between the source and target tasks, as well as possibly a transfer of information between them. A task can start executing only when all its direct predecessors have finished their execution, and after all data transmitted from its direct predecessors are available. A task with no direct predecessor is an *entry* task, whereas a task with no direct successor is an *exit* task. Without loss of generality it is assumed that there is a single entry task and a single exit task per application.

The structure of the DAG is static, with no conditional execution of nodes. The volume of data transmitted along edges (possibly null) is known offline. Each task in the DAG is assigned a distinct integer identifier.

A communication for a given edge is implemented using transfers to and from a dedicated buffer located in shared memory. The worst-case cost for writing data to and reading data from the buffer is integrated in the WCETs of the sending and the receiving tasks.

Due to the effect of caches, each task τ_j is not characterized by a single WCET value but instead by a set of WCET values. The most pessimistic WCET value for a task, noted $WCET_{\tau_j}$, is observed when there is no reuse of cache contents loaded by a task executed immediately before τ_j. A set of WCET values noted $WCET_{\tau_i \rightarrow \tau_j}$ represent the WCETs of task τ_j when τ_j reuses some information, loaded in the instruction and/or data cache by a task τ_i that is executed immediately before τ_j on the same core. Noted that the definition of the WCET of a task in this PhD work is different with the one defined in the literature. In the literature, the WCET of a task is defined as upperbound of the execution times of the task when executing in isolation. Our definition about the WCET of a task is upperbound of the execution times of the task when considering the contents of the cache at the beginning of the execution of the task.

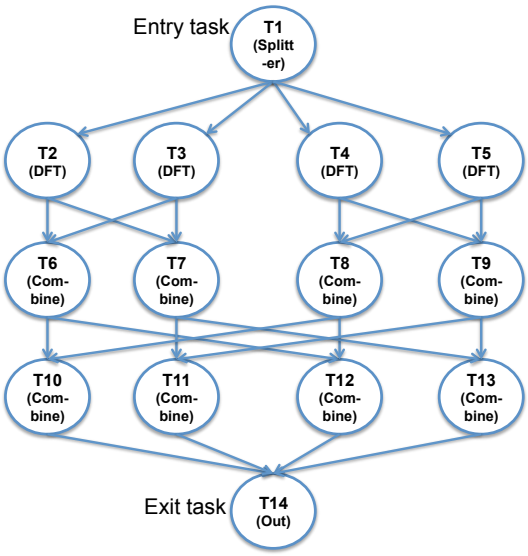


Figure 3.2: Task graph of a parallel version of a 8-input Fast Fourier Transform (FFT) application [11]

3.1.3 Assumptions

As far as schedules generation is concerned, we assume that

- Clocks of all cores are synchronized.
- In this study of benefit from taking into consideration cache reuse between tasks in task scheduling, we ignore shared resources contentions. Therefore, we assume that the architecture is free from contention to access shared resources (shared bus, shared memory). Noted that, shared resources contentions will be considered in the implementation stage (to be presented in Chapter 4).
- The cost for triggering tasks at specific instants of times is null.
- The cost for ensuring the consistency of communicated data stored in the shared memory is null.

Those assumptions relax cache-conscious schedules generation from constraints of any hardware platforms. In Chapter 4 we will address practical issues arising when implementing time-driven cache-conscious schedules on a real multi-core hardware.

3.1.4 Scheduling problem statement

Our proposed scheduling methods take as inputs the number of cores of the architecture and the DAG of a single parallel application decorated with WCET information for each task,

and produce an offline time-driven partitioned non-preemptive schedule of the application. More precisely, the produced schedule for each core determines the start and finish times of all tasks assigned to the core. The objective of the mapping and the scheduling decisions is to have the schedule length (also called makespan in the literature) as small as possible.

3.2 Cache-conscious task scheduling methods

For solving the scheduling problem presented in Section 3.1.4, we propose two methods:

- An ILP formulation that allows to reach the optimal solution, i.e. the one that minimizes the application schedule length) (see Section 3.2.1);

- A heuristic method, based on list scheduling, that allows to find a valid schedule very fast and generally very close to the optimal one (see Section 3.2.2).

The notations used in the description of the scheduling methods are summarized in Table 3.1. The first block defines frequently used notations to manage the task graph. The second block defines integer constants, using upper case letters, used throughout the chapter. Finally, the third block defines the variables, using lower case letters, used in the ILP formulation.

Symbol	**Description**	**Data type**
τ	The set of tasks of the parallel application	set
$dPred(\tau_j)$	The set of direct predecessors of τ_j	set
$dSucc(\tau_j)$	The set of direct successors of τ_j	set
$nPred(\tau_j)$	The set of tasks that are neither direct nor indirect predecessors of τ_j (τ_j excluded)	set
$nSucc(\tau_j)$	The set of tasks that are neither direct nor indirect successors of τ_j (τ_j excluded)	set
K	The number of cores of the processor	integer
$WCET_{\tau_j}$	The worst-case execution time of τ_j when not reusing cache contents	integer
$WCET_{\tau_i \rightarrow \tau_j}$	The worst-case execution time of τ_j when executing right after τ_i	integer
sl	The length of the generated schedule	integer
$wcet_{\tau_j}$	The worst-case execution time of τ_j	integer
st_{τ_j}	The start time of τ_j	integer
ft_{τ_j}	The finish time of τ_j	integer
f_{τ_j}	Indicates if τ_j is the first task running on a core or not	binary
$o_{\tau_i \rightarrow \tau_j}$	Indicates if τ_j is a co-located task of τ_i and executes right after τ_i or not	binary

Table 3.1: Notations used in the proposed scheduling methods

3.2.1 Cache-conscious ILP formulation

In this section, we present the cache-conscious ILP formulation for solving the considered scheduling problem, noted CILP (for Cache-conscious ILP) hereafter.

Since the considered hardware model is homogeneous multi-core platforms, the *exact* core onto which a task is mapped does not matter. As illustrated in Figure 3.3, because core 1 core 2 are identical, the mapping of tasks on core 1 and those on core 2 can be swapped without changing the length of the schedule. Based on that observation, CILP focuses on finding the sets of co-located tasks with their running order, as well as the start time and the finish time of tasks. The assignment of sets of co-located tasks to cores is then straightforward (i.e. one set per core).

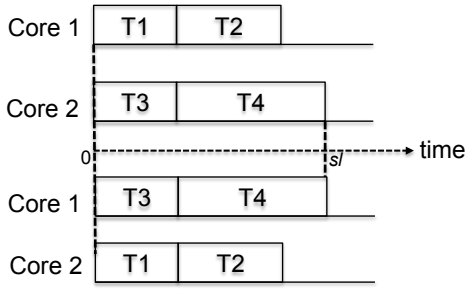


Figure 3.3: An example of the swapping of tasks' allocation

The objective function of CILP is to minimize the schedule length sl of the parallel application, which is expressed as follows:

$$\text{minimize} \quad sl \tag{3.1}$$

Since the schedule length for the parallel application has to be larger than or equal to the finish time ft_{τ_j} of any task τ_j, the following constraint is introduced:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &sl \geq ft_{\tau_j} \end{aligned} \tag{3.2}$$

The finish time ft_{τ_j} of a task τ_j is equal to the sum of its start time st_{τ_j} and its worst case execution time $wcet_{\tau_j}$:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &ft_{\tau_j} = st_{\tau_j} + wcet_{\tau_j} \end{aligned} \tag{3.3}$$

In the above equation, variable $wcet_{\tau_j}$ is introduced to model the variations of tasks' WCETs due to the effect of private caches and is computed as follows:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &wcet_{\tau_j} = f_{\tau_j} * WCET_{\tau_j} + \sum_{\tau_i \in \ nSucc(\tau_j)} o_{\tau_i \to \tau_j} * WCET_{\tau_i \to \tau_j} \end{aligned} \tag{3.4}$$

The left part corresponds to the case where task τ_j is the first task running on a core ($f_{\tau_j} = 1$). The sum in the right part corresponds to the case where the task τ_j is scheduled just after another co-located task τ_i ($o_{\tau_i \to \tau_j} = 1$). As shown later, only one binary variable among f_{τ_j} and variables $o_{\tau_i \to \tau_j}$ will be set by the ILP solver, thus assigning one and only one of the WCET values to τ_j depending on which other task is executed before it.

Constraints on the start time of tasks. A task can be executed only when all of its direct predecessors have finished their execution. In other words, its start time has to be larger than or equal to the finish times of all its direct predecessors.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in dPred(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} \ if \ dPred(\tau_j) \neq \emptyset \\ &st_{\tau_j} \geq 0 \ otherwise \end{aligned} \tag{3.5}$$

In the above equation, when the task has no predecessor, its start time has to be larger than or equal to zero.

Furthermore, in case there is a co-located task τ_i scheduled right before τ_j, τ_j cannot start before the end of τ_i. In other words, the start time of τ_j has to be larger than or equal to the finish time of τ_i. Note that τ_j can be scheduled only after a task τ_i that is neither its direct nor indirect successor.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq o_{\tau_i \to \tau_j} * ft_{\tau_i} \end{aligned} \tag{3.6}$$

For linearizing equation (3.6), we use the classical big-M notation which is expressed as:

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} + (o_{\tau_i \to \tau_j} - 1) * M \end{aligned} \tag{3.7}$$

where M, is a constant[2] higher than any possible ft_{τ_j}.

Constraints on the execution order of tasks. A task has at most one co-located task scheduled right after it, which is expressed as follows:

$$\begin{aligned} &\forall \tau_j \in \tau, \ if \ nPred(\tau_j) \neq \emptyset \\ &\sum_{\tau_i \in nPred(\tau_j)} o_{\tau_j \to \tau_i} \leq 1 \end{aligned} \tag{3.8}$$

Note that task τ_j can be only scheduled before task τ_i which is neither its direct nor indirect predecessor.

Furthermore, a task has one co-located task scheduled right before it which is neither its direct nor indirect successor or it is the first scheduled task, thus:

[2]For the experiments, M is the sum of all tasks' WCETs when not reusing cache contents, to ensure that M is higher than the finish time of any task.

$$\forall \tau_j \in \tau,$$
$$\sum_{\tau_i \in nSucc(\tau_j)} o_{\tau_i \to \tau_j} + f_{\tau_j} = 1 \tag{3.9}$$

Finally, since the number of cores is K, the number of tasks that can be the first to be scheduled on cores is at most K:

$$\sum_{\tau_j \in \tau} f_{\tau_j} \leq K \tag{3.10}$$

The result of the mapping/scheduling problem after being solved by an ILP solver is then defined by two sets of variables. Task mapping is defined by variables f_{τ_j} and $o_{\tau_i \to \tau_j}$ that altogether define the set of co-located tasks and their execution order. The static schedule on every core is defined by variables st_{τ_j} and ft_{τ_j}, that define the start and finish time of the tasks assigned to that core.

3.2.2 Cache-conscious list scheduling method (CLS)

Finding an optimal schedule for a partitioned non-preemptive scheduling problem is NP-hard [73]. Therefore, we developed a heuristic scheduling method that efficiently produces schedules that are close to the optimal ones. The proposed heuristic method is based on list scheduling (see [23] for a survey of list scheduling methods).

The proposed heuristic method (CLS, for Cache-conscious List Scheduling) first constructs a list of tasks to be scheduled. Then, the list of tasks is scanned sequentially, and each task is scheduled without backtracking. When scheduling a task, all cores are considered for hosting the task and a schedule that respects precedence constraints is constructed for each. The core which allows the earliest finish time of the task is selected and the corresponding schedule is kept.

The ordering of the tasks in the list has to follow topological ordering such that precedence constraints are respected. Here, we select a topological order that also takes into account the WCETs of tasks. Since a task may have different WCETs according to the other task executed before it, we associate to each task a *weight* that approximates its WCET variation. The weight of a task τ_j, noted tw_{τ_j}, is defined as follows:

$$tw_{\tau_j} = \frac{1}{K} * min_{\tau_i \in nSucc(\tau_j)}(WCET_{\tau_i \to \tau_j}) + (1 - \frac{1}{K}) * WCET_{\tau_j} \tag{3.11}$$

This formula integrates the likeliness that the WCET of task τ_j is reduced, which decreases when the number of cores increases. Different definitions of tasks weights were tested to take into account the WCET variation of tasks. As it will be shown in Section 3.3.2 there is no major difference in schedules length when using different tasks weights definitions so that we selected this simple definition.

Given tasks' weights, the order of tasks in the list is determined based on two classical metrics, both respecting topological order. The first metric is called in the following *bottom level*. It defines for task τ_j the longest path from τ_j to the exit task (τ_j included), cumulating tasks' weights along the path:

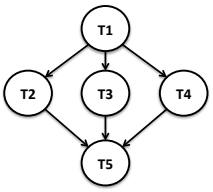
$$bottom\_level_{exit} = tw_{exit}$$
$$bottom\_level_{\tau_j} = max(bottom\_level_{\tau_i} + tw_{\tau_j}), \forall \tau_i \in dSucc(\tau_j) \quad (3.12)$$

The second metric is called *top level*. Symmetrically, it defines for task τ_j the longest path from the entry task to τ_j (τ_j excluded):

$$top\_level_{entry} = 0$$
$$top\_level_{\tau_j} = max(top\_level_{\tau_i} + tw_{\tau_i}), \forall \tau_i \in dPred(\tau_j) \quad (3.13)$$

As it will be shown in Section 3.3.2 none of the two metrics was shown to outperform the other for all task graphs, we thus kept both variations. In the following:

- CLS_BL refers to a sorting of tasks according to their bottom levels; in case of equality, their top levels is used to break ties; if a tie still exists, the task identifier is used to sort tasks.
- CLS_TL refers to a sorting of tasks according to their top levels, with bottom level and task identifier as tie breaking rules.
- CLS refers to the method, among CLS_BL and CLS_TL, resulting in the shortest schedule length for a given task graph.



τ_j \ τ_i	T1	T2	T3	T4	T5	Weight	Bottom Level
T1	10	-	-	-	-	10	42.5
T2	15	20	10	20	-	15	27.5
T3	20	15	25	25	-	20	32.5
T4	15	20	20	20	-	17.5	30
T5	15	10	10	10	15	12.5	12.5

Figure 3.4: Illustrative example for CLS_BL

We illustrate the execution of CLS_BL on a very simple task graph whose characteristics are given in Figure 3.4. The left part of Figure 3.4 gives the task graph. The right part gives the WCETs of tasks when not reusing cache contents ($WCET_{\tau_j}$, diagonal of the left part of the table) and values of $WCET_{\tau_i \rightarrow \tau_j}$; the next two columns give the weights and the bottom levels of tasks.

The execution of CLS_BL is illustrated in Figure 3.5 for a dual-core architecture. First, the tasks are ordered in a list according to their bottom levels. Then, the task at the head of the list (T1) is scheduled and T1 is removed from the list. Here, T1 is assigned to the first core. Next task in the list (here, T3) is then scheduled and removed from the list; T3 is mapped to the first core, which meets the precedence constraint between T1 and T3 and minimizes the finish time of T3 (date 30 on core 1 as opposed to date 35 on core 2). This process is repeated until the list is empty.

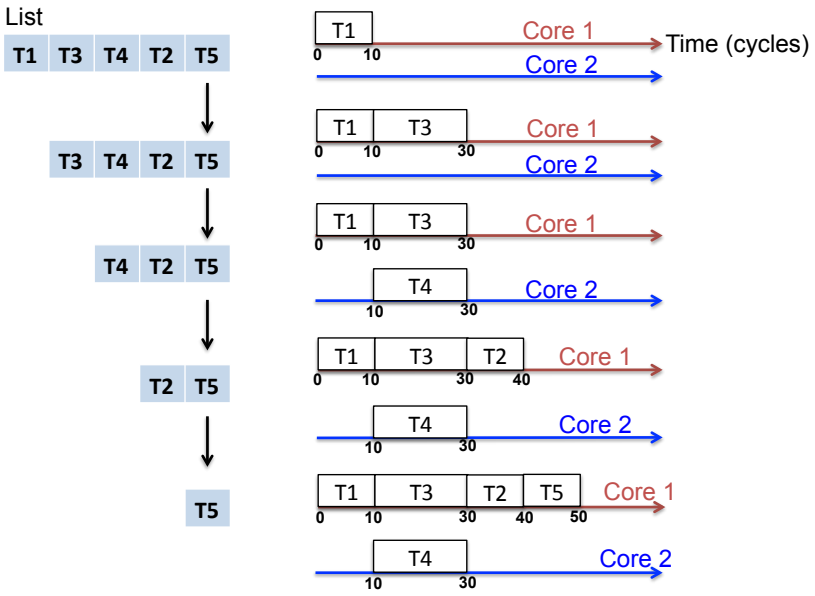


Figure 3.5: Illustration of CLS_BL

3.3 Experimental evaluation

In this section we evaluate the quality of generated schedules and the required time for generating them for the two proposed cache-conscious scheduling methods. We also evaluate the impact of several parameters, such as the number of cores, on the generated schedules. Note that experiments are performed with ignoring implementation details, such as hardwares sharing, cache pollution, the absence of hardware-implemented data cache coherence, the effect of a time-driven scheduler on tasks' execution. Those implementation issues will be identified in Chapter 4.

The organization of the section is as follows. Experimental conditions are described in Section 3.3.1. Experimental results are then detailed in Section 3.3.2.

3.3.1 Experimental conditions

Benchmarks. In our experiments, we use 26 benchmarks of the StreamIt benchmark suite [10]. StreamIt is a programming environment that facilitate the programming of streaming applications. We use that infrastructure to generate a sequential code of each benchmark in C++ and to get a representation of its Synchronous Data Flow graph (SDF). In the SDF graph, nodes represent filters or split-join structure and edges represent data dependencies between nodes. Each filter in the SDF consumes and produces a known amount of data. Each filter then has to be executed a certain number of times to balance the amount of data produced and consumed.

Each benchmark in the StreamIt benchmark suite starts with an initialization of the data

Benchmark (directed acyclic graph)	**No. of tasks**	**No. of Edges**	**Maximum graph width**	**Average graph width**	**Graph Depth**
AudioBeam	20	33	15	3.3	6
Autocor	12	18	8	2.4	5
Beamformer	42	50	16	4.2	10
BitonicSort	50	66	4	2.1	24
Cfar	67	129	64	16.8	4
ChannelVocoder	264	512	201	33	8
Cholesky	95	148	11	2.3	41
ComparisonCounting	37	67	32	6.2	6
DCT	13	15	3	1.3	10
DCT_2D	10	11	2	1.3	8
DCT_2D_reference_fine	148	280	64	18.5	8
Des	247	468	48	9.9	25
FFT_coarse	192	254	64	12.8	15
FFT_fine_2	115	150	16	3.7	31
FFT_medium	131	204	16	4.7	28
FilterBank	34	45	8	2.4	14
FmRadio	67	85	20	5.6	12
IDCT	16	19	3	1.3	12
IDCT_2D	10	11	2	1.3	8
IDCT_2D_reference_fine	548	1072	256	68.5	8
Lattice	45	53	2	1.3	36
MergeSort	31	37	8	2.6	12
Oversampler	36	61	16	3.6	10
RateConverter	6	6	2	1.2	5
VectorAdd	5	4	2	1.3	4
Vocoder	71	94	7	2.2	32

Table 3.2: Summary of the characteristics of StreamIt benchmarks in our case studies.

for the initial execution of filters, followed by a steady state where the execution of filters is repeated. In our experiments, we focus on the execution of one iteration of the steady state. To obtain a directed acyclic graph corresponding to our task model, we transformed manually the sequential code generated by streamIt compiler to expose parallelism. In fact, each execution of a filter is considered as a task. The number of tasks which are cloned from a filter equals to the number of times the filter has to be executed. Those replicated tasks share the same code, but operate on different data. We validated our transformations by systematically comparing the outputs of the sequential and parallel versions.

The characteristics of the obtained task graphs are summarized in Table 3.2. In the table, the maximum width of a task graph is defined as the maximum number of tasks with the same rank[3]. The maximum width defines the maximum parallelism in the benchmark.

[3]The rank of a task is defined as the longest path in term of the number of nodes to reach that task from the entry task.

Benchmark	Code size (Bytes)		Communicated data (Bytes)
	Entire application	μ / σ of tasks	μ
AudioBeam	38076	1458 / 1897	6
Autocor	12348	1014 / 538	66
Beamformer	333424	1879 / 718	10
BitonicSort	57952	1154 / 503	9
Cfar	181808	1906 / 5513	6
ChannelVocoder	302012	881 / 159	6
Cholesky	87336	916 / 667	22
ComparisonCounting	33564	893 / 840	20
DCT	23180	1188 / 831	8
DCT_2D	17248	1704 / 1101	9
DCT_2D_reference_fine	120392	724 / 145	12
Des	212808	783 / 185	12
FFT_coarse	418576	2161 / 467	52
FFT_fine2	122428	1060 / 574	9
FFT_medium	178660	1358 / 408	27
FilterBank	101096	834 / 192	4
FmRadio	374812	1072 / 679	4
IDCT	24336	1507 / 1239	7
IDCT_2D	17608	1740 / 1063	9
IDCT_2D_reference_fine	452924	802 / 154	7
Lattice	37812	817 / 274	5
MergeSort	34208	1088 / 366	16
Oversampler	56824	777 / 115	4
RateConverter	12348	683 / 247	11
VectorAdd	3080	593 / 148	4
Vocoder	125272	1064 / 1319	6

Table 3.3: The size of code and communicated data for each benchmark (average μ and standard deviation σ).

The average width is an average of the number of tasks for all ranks. The average width defines the average parallelism of the application. The higher the average width, the better the potential to benefit from a high number of cores. The depth of a task graph is defined as the longest path from the entry task to the exit task.

Additional information on the benchmarks is reported in Table 3.3. Reported information is the code size for the entire application, the average and standard deviation of code size per task, and the average amount of data communicated between tasks.

Hardware and WCET estimation. The target architecture used for the experiments is the Kalray MPPA-256 machine [4], more precisely its first generation, named *Andey*. The

Benchmark	**WCET in cycles (w/o cache reuse) (μ/σ)**	**Weighted average WCET reduction**
AudioBeam	1479.0 / 2869.6	13.3
Autocor	3163.0 / 1855.1	5.5
Beamformer	4896.9 / 2950.2	4.5
BitonicSort	678.0 / 391.6	22.8
Cfar	2767.0 / 11612.7	13.0
ChannelVocoder	8084.5 / 26265.9	3.8
Cholesky	1512.5 / 3152.3	10.7
ComparisonCounting	1249.6 / 1477.5	14.4
DCT	718.3 / 685.0	19.1
DCT_2D	812.7 / 741.4	18.6
DCT_2D_reference_fine	1072.6 / 1519.2	17.1
Des	893.2 / 1236.2	23.4
FFT_coarse	3465.9 / 3062.3	9.8
FFT_fine_2	745.5 / 469.6	19.5
FFT_medium	1470.7 / 1456.3	11.6
FilterBank	3634.0 / 3701.0	4.6
FmRadio	2802.5 / 2652.1	5.5
IDCT	687.7 / 632.9	21.2
IDCT_2D	805.6 / 743.5	18.7
IDCT_2D_reference_fine	1538.5 / 3864.9	14.9
Lattice	515.6 / 381.8	28.6
MergeSort	1010.4 / 662.1	17.4
Oversampler	4195.3 / 684.5	6.5
RateConverter	19779.0 / 34471.5	0.9
VectorAdd	923.8 / 979.6	20.1
Vocoder	804.1 / 1227.8	15.8

Table 3.4: Tasks' WCETs (average μ / standard deviation σ) without cache reuse and weighted average WCET reduction

Kalray MPPA-256 is a many-core platform with 256 compute cores organized in 16 compute clusters of 16 cores each. Each compute cluster has 2MB of shared memory. Each compute core is equipped with an instruction cache and a data cache of 8KB each, both set-associative with a Least Recently Used (LRU) replacement policy. An access to the shared memory, in case no contention occurs takes 9 cycles with 8 bytes fetched on each consecutive cycle [86].

Many techniques exist for WCET estimation [7] and could be used in our study to estimate WCETs and gains resulting from cache reuse. Since WCET estimation is not at the core of our scheduling methods, WCET values were obtained using measurements on the platform. Measurements were performed on one compute cluster, with no activity on the other cores, providing fixed inputs for each task. The execution time of a task is retrieved using the machine's 64-bit timestamp counter counting cycles from boot time [4]. The effect

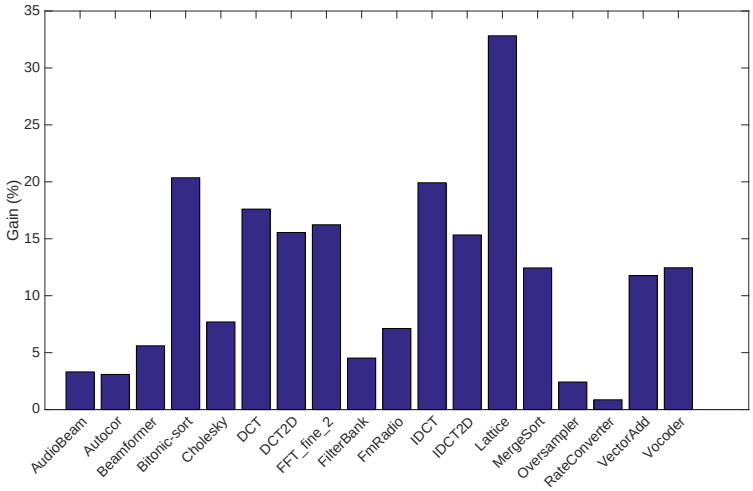


Figure 3.6: Gain of CILP as compared to NCILP ($gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100$) on a 16 cores system

of reading the timestamp counter on the execution time of a task turned out to be negligible as compared to the execution time of the task. We further observed that thanks to the determinism of the architecture, when running a task several times, in the same execution context, the execution time is constant (the same behavior was observed in [92]). For each task, we record its execution time when not reusing cache contents, as well as when executed after any possible other task.

Table 3.4 summarizes the statistical numbers of obtained execution times. This table shows the average and standard deviation of tasks' WCET when having no cache reuse. It also shows the weighted average WCET reduction for each benchmark, computed as follows. For each task τ_j we calculate its average WCET reduction in percent:

$$r_{\tau_j} = 100 * \frac{\sum_{\tau_i \in nSucc(\tau_j)} \frac{WCET_{\tau_j} - WCET_{\tau_i \to \tau_j}}{WCET_{\tau_j}}}{|nSucc(\tau_j)|} \tag{3.14}$$

Since tasks with low WCET (when having no cache reuse) tend to have high WCET reductions (when having cache reuse) although they have low impact on schedule length, we weighted each value by its WCET (when having no cache reuse), yielding to the following definition of weighted average reduction:

$$wr = \frac{\sum_{\tau_j \in \tau} (r_{\tau_j} * WCET_{\tau_j})}{\sum_{\tau_j \in \tau} WCET_{\tau_j}} \tag{3.15}$$

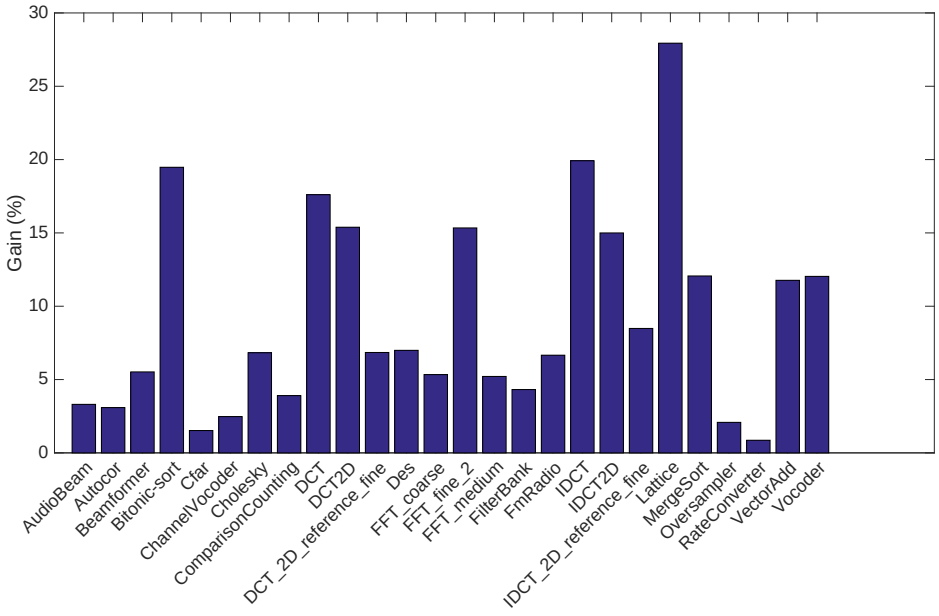


Figure 3.7: Gain of CLS as compared to NCLS ($gain = \frac{sl_{NCLS} - sl_{CLS}}{sl_{NCLS}} * 100$) on a 16 cores system

Experimental environment. We use *Gurobi optimizer* version 6.5 [93] for solving our proposed ILP formulation. The solving time of the solver is limited to 20 hours. The ILP solver and heuristic scheduling algorithms are executed on 3.6 GHz Intel Core i7 CPU with 16GB of RAM.

3.3.2 Experimental results

Benefits of cache-conscious scheduling. In this sub-section, we show that cache-conscious scheduling, should it be implemented using an ILP formulation (CILP) or a heuristic method (CLS), yields to shorter schedules than equivalent cache-agnostic methods. This is shown by comparing how much is gained by CILP as compared to NCILP, the same ILP formulation as CILP except that cache effect is not taken into account (variable $wcet_{\tau_j}$ is systematically set to the cache-agnostic WCET, $WCET_{\tau_j}$). The gain is evaluated by the following equation, in which *sl* stands for the schedule length:

$$gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100. \tag{3.16}$$

Benchmarks	sl_CILP	sl_CLS	time_CILP (s)	time_CLS (s)	gap (%)
AudioBeam	20746^{o}	20746	< 1	< 1	**0.00**
AutoCor	17455^{o}	17455	< 1	< 1	**0.00**
Beamformer	29778^{o}	29803	2	< 1	**0.08**
BitonicSort	15445^{o}	15616	78	< 1	**1.11**
Cfar	120370^{f}	120476	72000	< 1	
ChannelVocoder	x	302933	72000	< 1	
Cholesky	113474^{o}	114539	< 1	< 1	**0.94**
ComparisonCounting	19618^{f}	19640	72000	< 1	
DCT	6613^{o}	6613	< 1	< 1	**0.00**
DCT2D	5856^{o}	5867	< 1	< 1	**0.19**
DCT_2D_reference_fine	33337^{f}	32572	72000	< 1	
Des	100632^{f}	98596	72000	< 1	
FFT_coarse	x	134873	72000	< 1	
FFT_fine_2	30007^{o}	30326	66984	< 1	**1.06**
FFT_medium	89782^{f}	87144	72000	< 1	
FilterBank	47083^{o}	47185	15	< 1	**0.22**
FmRadio	29969^{o}	30125	4376	< 1	**0.52**
IDCT	7268^{o}	7268	< 1	< 1	**0.00**
IDCT2D	5803^{o}	5826	< 1	< 1	**0.40**
IDCT_2D_reference_fine	x	101970	72000	1	
Lattice	13253^{o}	14217	< 1	< 1	**7.27**
MergeSort	14501^{o}	14563	1	< 1	**0.43**
Oversampler	39143^{o}	39279	8	< 1	**0.35**
RateConverter	117278^{o}	117278	< 1	< 1	**0.00**
VectorAdd	3704^{o}	3704	< 1	< 1	**0.00**
Vocoder	32759^{o}	32916	9	< 1	**0.48**
Average					**0.72**

- x: no solution is found in 20 hours
- f: feasible solution is found
- o: optimal solution is found

Table 3.5: Comparison of CILP and CLS (schedule length and run time of schedule generation)

The gain is also evaluated using a similar formula for the heuristic method CLS (shorter schedule results for CLS_BL and CLS_TL) as compared to its cache-agnostic equivalent NCLS.

Results are reported on Figures 3.6 and 3.7 for a 16 cores architecture. In Figure 3.6, only results for the benchmarks for which the optimal solution was found in a time budget of 20 hours are depicted. These figures show that both CILP and CLS reduce the length of schedules, and this for all benchmarks. The gain is 11% on average for CILP and 9% on average for CLS. The higher reductions are obtained for the benchmarks with the higher weighted WCET reduction as defined in Table 3.4.

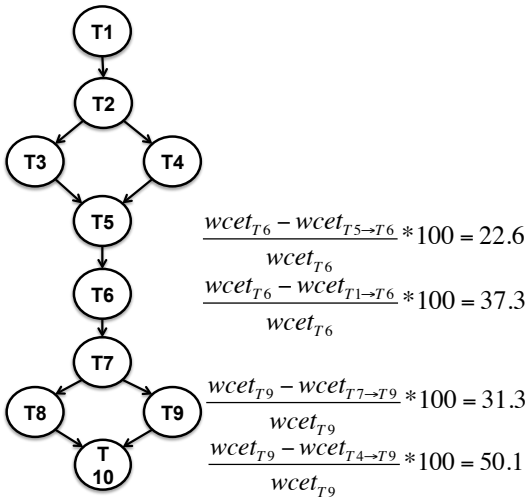


Figure 3.8: The reuse pattern found in the *Lattice* benchmark

Comparison of optimal (CILP) and heuristic (CLS) scheduling techniques. In this sub-section, we compare CILP and CLS according to two metrics: the quality of the generated schedules, estimated through their lengths (the shorter the better) and the time required to generate the schedules. All results are obtained on a 16 cores system.

Table 3.5 gives the lengths of generated schedules (sl_{CILP} and sl_{CLS}), the run time of schedule generation and the gap (in percent) between the schedule lengths, computed by the following formula:

$$gap = \frac{sl_{CLS} - sl_{CILP}}{sl_{CILP}} * 100. \tag{3.17}$$

The shorter the gap, the closer CLS is from CILP. The gap between CLS and CILP is given only when CILP finds the optimal solution in a time budget of 20 hours.

The table shows that CLS offers a good trade-off between the efficiency and the quality of its generated schedules. CLS generates schedules very fast as compared to CILP (i.e., about 1 second for the biggest task graph *IDCT_2D_reference_fine* which contains 548 tasks). When scheduling big task graphs, such as *IDCT_2D_reference_fine*, *DES*, and *ChannelVocoder*, CILP is unable to find the optimal solution in 20 hours. When CILP finds the optimal solution, the gap between CILP and CLS is very small (0.7% on average).

The highest gap (7.3%) is observed for the *Lattice* benchmark. It can be explained that the WCETs of tasks in the *Lattice* benchmark are small and the benchmark contains a reuse pattern (illustrated in Figure 3.8) where reuse is higher between indirect predecessors than between direct predecessors. For example, the reduction of the WCET of T6 when executing directly after T1 (37.3%) is higher than when executing directly after T5 (22.6%). Similarly, the reduction of the WCET of T9 when executing directly after T4 (50.1%) is higher than when executing directly after T7 (31.3%). For such an application, the static sorting of CLS never places indirect precedence-related tasks (for which the higher reuse

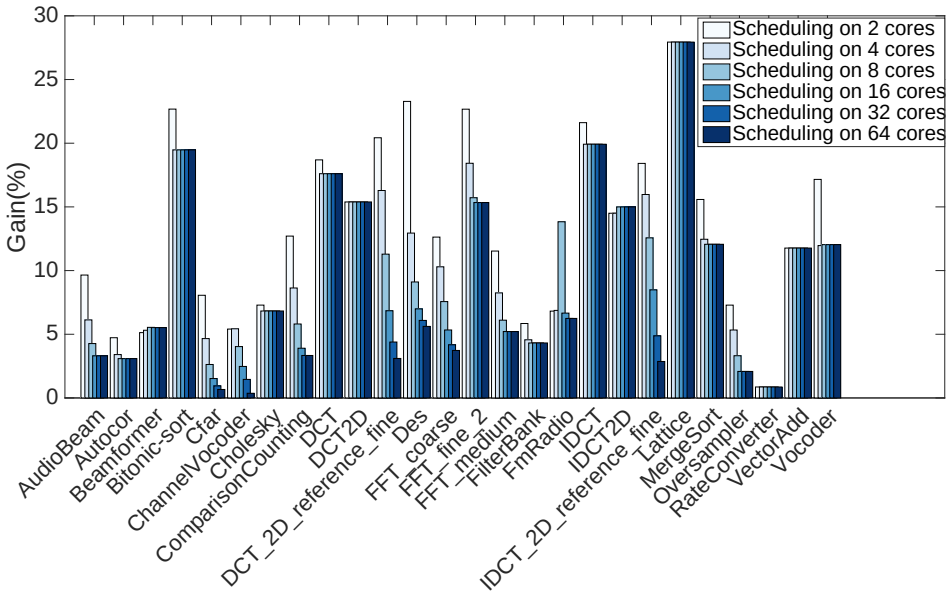


Figure 3.9: Impact of the number of cores on the gain of CLS against NCLS

occurs) contiguously in the list, and then does not fully exploit the cache reuse present in the application.

Impact of the number of cores on the gain of CLS against NCLS. In this subsection, we evaluate the gain in term of schedule length of CLS against its cache-agnostic equivalent when varying the number of cores. The results are depicted in Figure 3.9 for a number of cores from 2 to 64.

In the figure, we can observe that whatever the number of cores, CLS always outperforms NCLS, meaning that our proposed method is always able to take advantage of the WCET reduction due to cache reuse to reduce the schedule length. Another observation is that the gain decreases when the number of cores increases, up to a given number of cores. This behavior is explained by the fact that when increasing the number of cores, the tasks are spread among cores which provides less opportunity to exploit cache reuse since exploiting the parallelism of the application is more profitable. However, even in that situation, the reduction of the schedule length achieved by CLS against NCLS is most of the time significant.

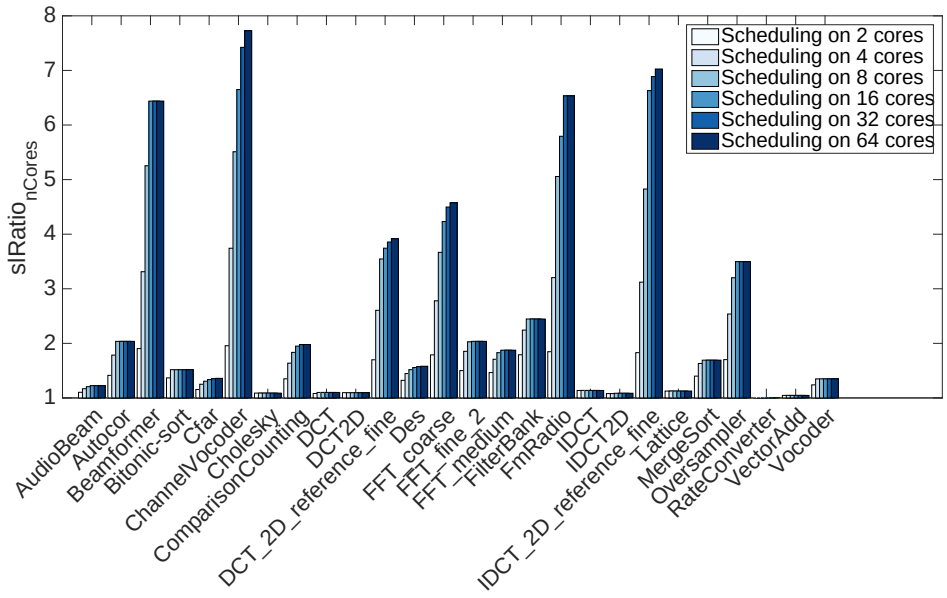


Figure 3.10: Impact of the number of cores on schedule length (CLS method)

Impact of the number of cores on schedule length. In this sub-section, we study the impact of the number of cores on schedule length for the CLS scheduling technique. This is expressed by depicting the ratio of the schedule length on one core sl_{1Cores} to the schedule length on n cores sl_{nCores}: $slRatio_{nCores} = \frac{sl_{1Core}}{sl_{nCores}}$. Results are given in Figure 3.10 for a number of cores $n = 2, 4, 8, 16, 32$ and 64. The higher the ratio, the better CLS is able to exploit the multi-core architecture for a given benchmark.

The figure shows that for all benchmarks the ratio increases up to a certain number of cores and then reaches a plateau. The plateau is reached when the benchmark does not have sufficient parallelism to be exploited by the scheduling algorithm, which is correlated to the width of its task graph as presented in Table 3.2.

It can be noticed that for some benchmarks (*ChannelVocoder*, *DCT_2D_reference_fine*, *FFT_coarse* and *IDCT_2D_reference_fine*) the plateau is never reached because these benchmark have too much parallelism for the number of cores. Even if the average width is below 64, we observe for these benchmarks that the maximal width is above 64 and up to 256 for *IDCT_2D_reference_fine* which explains why the plateau is not reached for these benchmarks.

An exception is observed for *RateConverter* where there is absolutely no improvement.

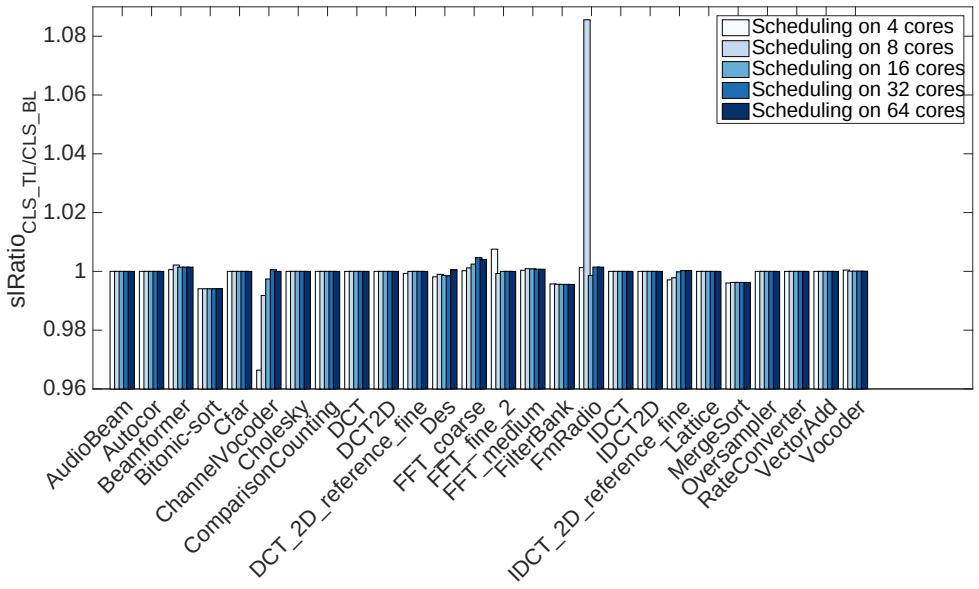


Figure 3.11: Comparison of schedule lengths for CLS_TL and CLS_BL

The graph of this benchmark is an almost linear chain of tasks with only a pair of tasks that may execute in parallel. However, there are cache reuse between these two tasks and thus the best schedule, whatever the number of available cores, is obtained when assigning all tasks to the same core.

Finally, for most benchmarks, the ratio does not increase linearly with a slope of 1. The first reason is that the WCETs of tasks are not identical. The second reason is that task graphs contain precedence relations so that whatever number of cores on which task graphs are scheduled joint tasks are remained on critical paths of generated schedules.

Comparison of schedule lengths for CLS_TL and CLS_BL. In this sub-section, we study the impact of the sorting technique of the list scheduling technique on the quality of schedules. For each benchmark, Figure 3.11 depicts the ratio of the length of the schedules generated by CLS_TL to that of CLS_BL as $slRatio_{CLS\_TL/CLS\_BL} = \frac{sl_{CLS\_TL}}{sl_{CLS\_BL}}$. A ratio of 1 indicates that the two techniques generate schedules with identical length. Results are given for different numbers of cores (4, 8, 16, 32 and 64).

The figure shows that there is no method which dominates the other for all benchmarks. Furthermore, the lengths of schedules generated by CLS_TL and CLS_BL are most of the

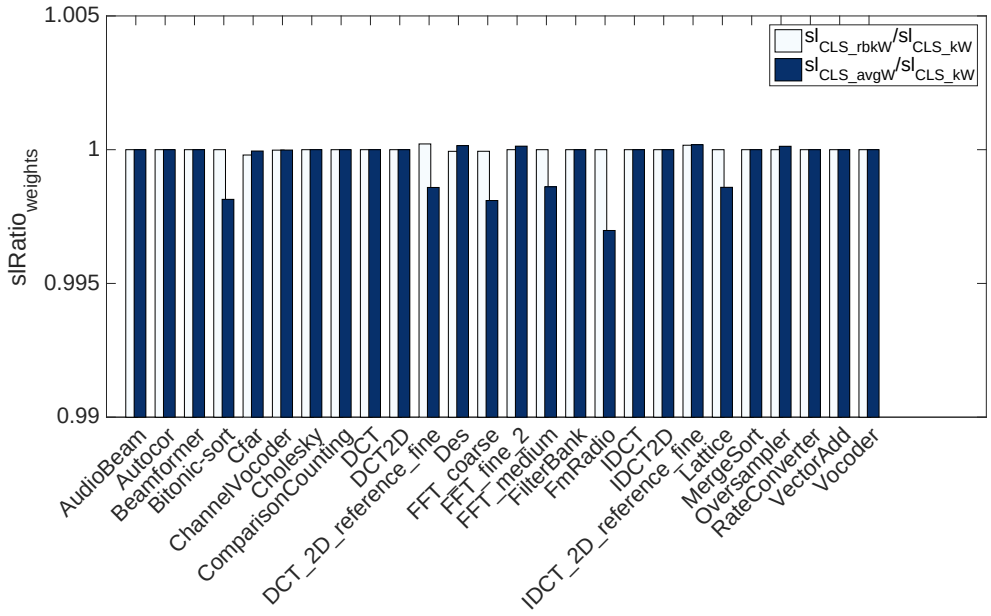


Figure 3.12: Comparison of schedules lengths for CLS using different tasks weight functions in the case that tasks are sorted in the list according their top levels

time very close to each other.

There is a significant difference between CLS_TL and CLS_BL only in two cases, *ChannelVocoder* on 4 cores and *FmRadio* on 8 cores. The distances between the lengths of the schedules generated by CLS_TL and CLS_BL in these cases are then 3% and 8% respectively. It shows that in some special cases, the change in the order of tasks in the list significantly affects the mapping of tasks, hence the quality of generated schedules. Since both CS_TL and CLS_BL generate schedules very fast, we have throughout this chapter always used both and selected the best result obtained.

Comparison of schedule lengths for CLS using different tasks weight functions. In this sub-section, we study the impact of using different tasks weight functions on the length of generated schedules. In Section 3.2.2 we already defined a tasks weight function as presented in equation (3.11). For the study, two other tasks weight functions are defined as follows.

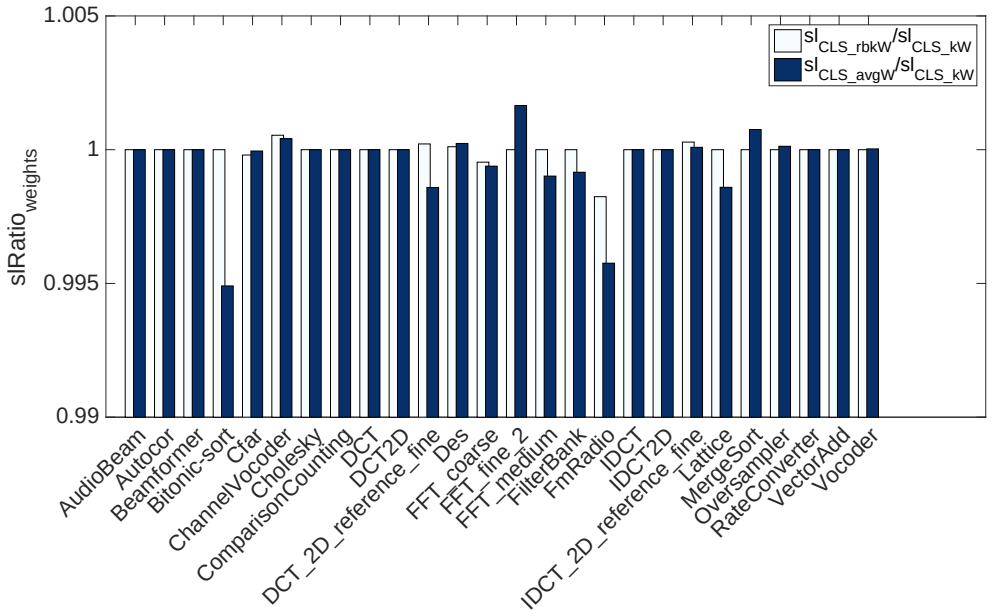


Figure 3.13: Comparison of schedules lengths for CLS using different tasks weight functions in the case that tasks are sorted in the list according their bottom levels

The first additional definition of the weight of τ_j is represented as:

$$tw_{\tau_j} = \frac{1}{K * (|nSucc(\tau_j)| + 1)} * min_{\tau_i \in nSucc(\tau_j)}(WCET_{\tau_i \to \tau_j}) + (1 - \frac{1}{K * (|nSucc(\tau_j)| + 1)}) * WCET_{\tau_j} \tag{3.18}$$

In this formula, $|nSucc(\tau_j)|$ is the number of tasks that are neither direct nor indirect successors of τ_j (i.e., tasks that may precede τ_j on the same core). The formula integrates the likeliness that the WCET of τ_j decreases when τ_j has more chances to be assigned on the same core and right after the task producing more contents that τ_j can reuse, which is likely to be achieved when the number of cores as well as the number of tasks that may precede τ_j on the same core increases.

Alternatively, the weight of τ_j is computed as the average of its possible WCETs:

$$tw_{\tau_j} = \frac{\sum_{i \in nSucc(\tau_j)} WCET_{\tau_i \to \tau_j} + WCET_{\tau_j}}{|nSucc(\tau_j)| + 1}, \tag{3.19}$$

For the comparison, we name the heuristic scheduling method (CLS) using the tasks

Benchmark	No. of tasks	No. of possible pairs	Profiling time (s)
AudioBeam	20	295	5
AutoCor	12	94	5
Beamformer	42	1326	7
BitonicSort	50	1341	7
Cfar	67	4227	11
ChannelVocoder	264	57481	170
Cholesky	95	7108	18
ComparisonCounting	37	1162	7
DCT	13	83	5
DCT_2D	10	47	5
DCT_2D_reference_fine	148	15414	49
Des	247	38185	135
FFT_coarse	192	34428	97
FFT_fine_2	115	7799	23
FFT_medium	131	10043	37
FilterBank	34	774	6
FmRadio	67	3841	11
IDCT	16	126	5
IDCT_2D	10	47	5
IDCT_2D_reference_fine	548	219238	625
Lattice	45	999	7
MergeSort	31	688	6
Oversampler	36	785	6
RateConverter	6	16	5
VectorAdd	5	11	5
Vocoder	71	2961	11

Table 3.6: Cost of estimating cache reuse

weight function as defined in equation (3.11) , equation (3.18), and equation (3.19) as CLS_kW, CLS_rbkW, and CLS_avgW, respectively.

For each benchmark, we compute the normalization of schedules length generated by CLS_rbkW and CLS_avgW to that generated by CLS_kW, i.e., $slRatio_{weights} = \frac{sl_{CLS\_x}}{sl_{CLS\_kW}}$ in which $x = \{rbW, avgW\}$. Figure 3.12 depicts the normalization when tasks are sorted in the list according to their top levels, whereas Figure 3.13 depicts the normalization when tasks are sorted in the list according to their bottom levels. Results are given for a 16 cores architecture.

From the figures, we observe that there is no tasks weight function that dominates the other for all benchmarks. Furthermore, using different tasks weight functions always generate schedules whose lengths are very close to each other. Therefore, throughout this chapter we select the simple definition as presented in equation 3.11 for defining tasks weight.

Cost of estimating cache reuse. The information given in Table 3.6 allows to evaluate the cost of estimating cache reuse (estimation of values of $WCET_{\tau_i \rightarrow \tau_j}$) for the StreamIt benchmarks. The table reports for each benchmark its number of tasks, the number of task pairs that may be executed one after the other due to precedence constraints, and the time taken to evaluate all WCET values using measurements. The number of task pairs to be considered depends on the structure of the task graph. The worse observed profiling time is 10 minutes for the most complex benchmark structure *IDCT_2D_reference_fine* (i.e., which contains 548 tasks).

3.4 Related work

Schedulability analysis techniques rely on the knowledge of the Worst-Case Execution Times of tasks. Originally designed for single-core processors, static WCET estimation techniques were extended recently to cope with multi-core architectures. Most research have focused on modeling shared resources (e.g., shared caches, shared bus, shared memory) in order to capture interferences between tasks which execute concurrently on different cores [65, 50, 64, 94, 51, 95]. Most extensions of WCET estimation techniques for multi-cores produce a WCET for a single task in the presence of concurrent executions on the other cores. By construction, those extensions do not account for cache reuse between tasks as our scheduling techniques do. The scheduling techniques we propose have to rely on WCET estimation techniques to estimate the effect of local caches on tasks' WCETs.

Some WCET estimation techniques pay attention to the effect of private caches on WCETs. In [96], when analyzing the timing behavior of a task, Nemer et al. take into account the set of memory blocks that has been stored in the instruction cache (by the execution of previous tasks on the same core) at the beginning of its execution. Similarly, Potop-Butucaru and Puaut [70], assuming task mapping on cores known, jointly perform cache analysis and timing analysis of parallel applications. These two WCET estimation techniques assume task mapping on core and task schedule on each core known. In this thesis, in contrast, task mapping and scheduling are selected to take benefit of cache reuse to have the shortest possible schedule length.

Much research effort has been spent on scheduling for multi-core platforms. Research on real-time scheduling for independent tasks is surveyed in [24]. This survey gives a taxonomy of multi-core scheduling strategies: global vs. partitioned vs. semi-partitioned, preemptive vs. non preemptive, time-driven vs. event-driven. The scheduling techniques we propose in this thesis generate offline time-driven non-preemptive schedules. Most of the scheduling strategies surveyed in [24] are unaware of the hardware effects and consider a fixed upper bound on tasks' execution times. In contrast, the scheduling techniques we propose in this thesis address the effect of private caches on tasks' WCETs. Our work integrates this effect in the scheduling and mapping problem by considering multiple WCETs for each task depending on their execution contexts (i.e. cache contents at the beginning of their execution).

Some scheduling techniques that are aware of hardware effects were proposed in the past. They include techniques that simultaneously schedule tasks and the messages exchanged between them [76, 77, 97, 78]; such techniques take into consideration the Network-On-Chip

(NoC) topology in the scheduling process. Some other techniques aim at scheduling tasks in a way that minimizes the contentions when accessing shared resources (e.g., shared bus, shared caches) [80, 88, 79]. Besides, some approaches [83, 86, 98, 66, 99] schedule tasks according to execution models that guarantee temporal isolation between co-running tasks. In that way, scheduled tasks are guaranteed to be free from the contentions when accessing shared resources. In [100], Suhendra et al. consider data reuse between tasks to perform task scheduling for multi-core systems equipped with scratchpad memory (SPM); in their work, the most frequently accessed data are allocated in SPM to reduce the accesses latency to an off-chip memory. Our scheduling solutions in this thesis differ from the above-mentioned previous works because we pay attention to the effect of private caches on tasks' WCETs. In our proposed scheduling methods, tasks are scheduled to get benefit from the effect of private caches.

Related studies also address the effect of private caches when scheduling tasks on multi-core architectures [101, 102, 103]. However, they are based on global and preemptive scheduling techniques, in which the cost of cache reload after being preempted or migrated has to be accounted for. Compared to these works, our technique is partitioned and non preemptive. We believe such a scheduling method allows to have better control on cache reuse during scheduling. Furthermore, [102] and [103] focus on single core architectures while our work target multi-core architectures.

3.5 Summary

In this chapter, we presented the concept of cache-conscious scheduling methods (i.e., an optimal method and a heuristic method) for multi-core platforms equipped with local instruction and data caches. The static time-driven partitioned non-preemptive schedules of parallel applications modeled as directed acyclic graphs are generated. Experimental results showed that cache-conscious schedules produced by our proposed cache-conscious scheduling methods have smaller length as compared to schedules produced by the cache-agnostic equivalent scheduling methods. It shows the benefit of considering the effects of private caches on tasks' WCETs in tasks mapping and tasks scheduling. Furthermore, the proposed heuristic scheduling method shows a good trade-off between the efficiency and the quality of generated schedules, i.e., the heuristic schedules are generated very fast, and the length of those schedules are close to the length of the optimal ones. In next chapter, we will study the implementation of static time-driven partitioned non-preemptive cache-conscious schedules on a real multi-core platform.

Chapter 4

Cache-conscious scheduling: implementation

In Chapter 3, we presented cache-conscious scheduling algorithms for generating time-driven partitioned non-preemptive schedules of parallel applications for a homogeneous multi-core platform model, in which a core is equipped with private caches. In this chapter, we identify practical challenges arising when implementing those schedules on a real multi-core hardware − the Kalray MPPA-256 machine −, and propose our strategies for tackling the identified challenges.

The organization of the chapter is as follows. Section 4.1 gives an overview of the architecture of Kalray MPPA-256 machine. Section 4.2 presents our time-driven scheduler implementation. The practical issues arising when implementing time-driven cache-conscious schedules on Kalray MPPA-256 machine are identified in Section 4.3. Section 4.4 and Section 4.5 present our strategies to overcome the identified issues. An experimental evaluation is given in Section 4.6, followed by the description of related works presented in Section 4.7. Finally, the content of the chapter is summarized in Section 4.8.

4.1 Architecture of the Kalray MPPA-256 machine

Our target architecture is the Kalray MPPA-256 Andey [4] which is a clustered many-core platform organized as depicted in Figure 4.1. The platform contains 288 cores which are organized into 16 compute clusters and 4 I/O clusters. Those clusters are interconnected with a dual 2D-torus Network on Chip (NoC). In the study, we implement cache-conscious schedules on a compute cluster of the machine for which the overview is given as follows.

A Kalray MPPA-256 compute cluster (as depicted in the right side of the Figure 4.1) contains 17 identical VLIW (Very Long Instruction Word) cores. The first 16 cores, referred to as processing elements (PEs), are dedicated to general-purpose computations, whereas the 17th core, referred to as resource manager (RM), manages processor resources for the entire cluster. Additionally, Kalray MPPA-256 compute cluster contains a Debug Support Unit (DSU), a NoC Rx interface for receiving data, and a NoC Tx interface for transmitting

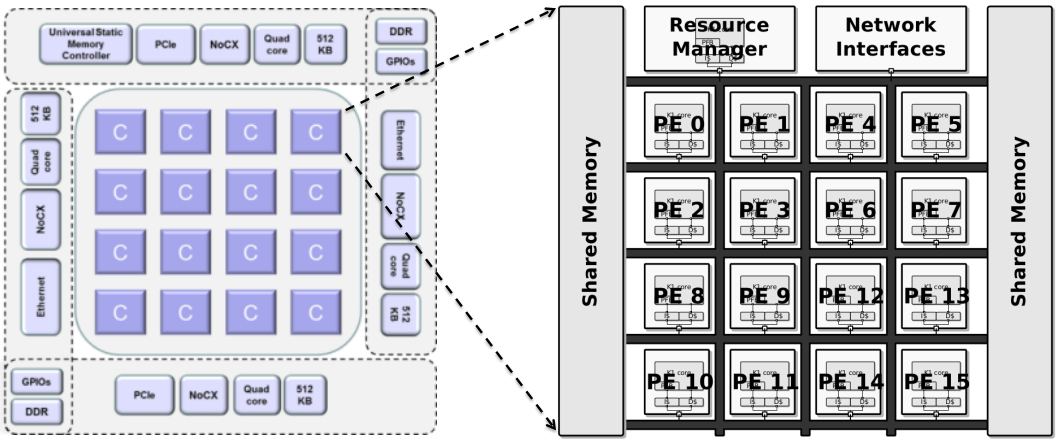


Figure 4.1: Overview of Kalray MPPA-256 [4]

data (supported by DMA − Directed Memory Access − engine).

As announced in [4], every core in Kalray MPPA-256 machine is fully timing compositional [104]. Each core is equipped with a private instruction cache and a private data cache of 8 KB each. Both are two-way associative with a Least Recently Used (LRU) replacement policy. The size of an instruction cache line is 64 bytes, while the size of a data cache line is 32 bytes. The default write policy of the data cache is *write-through*. Data being flushed out from the data cache is not immediately committed to shared memory. The flushed data is temporally held in a write buffer which is 8-way fully associative and each way contains 8 bytes. Note that there is no hardware-implemented data cache coherence between cores, therefore data coherency between cores has to be controlled by software.

The compute cluster shared memory (SMEM) comprises 16 independent memory banks that are arranged in two sets of 8 banks, named the left side and the right side. The size of each bank is 128 KB, for a total memory capacity of 2MB per cluster. The default mode of memory address mapping of the SMEM is configured as *interleaved*. As illustrated in Figure 4.2, in the mode the sequential addresses move from one bank to another every 64 bytes. The configuration is useful for high-performance and parallel applications since memory references tend to spread evenly across the memory banks so the overall memory throughput is maximized [4].

Each memory bank is associated with a dedicated requests arbiter (i.e., for 16 memory banks there are 16 requests arbiters in total) that serves 12 bus masters: the D-NoC Rx interface, the D-NoC Tx interface, the DSU, the RM core, and 8 Processing Elements (PEs) pairs. Each bus master has private paths connected to the 16 memory bank arbiters. The connections between memory bus masters are replicated in order to provide independent accesses to the left and right sides of the shared memory. The arbitration of memory requests to SMEM's banks is performed in 3 stages, as depicted in Figure 4.3. The first two stages use round-robin (RR) arbitration scheme. The first level arbitrates between memory requests from the instruction cache (IC) and the data cache (DC) of each PE in a PEs pair.

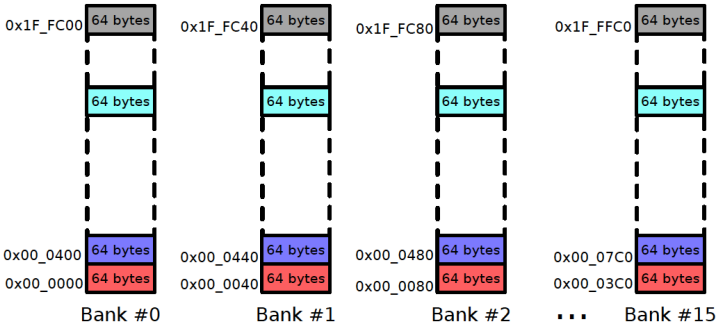


Figure 4.2: SMEM interleaved address mapping [4]

At the second stage, the requests issued from each PEs pair compete against those issued from other PEs pairs, the D-NoC Tx, the DSU, and the RM. Finally, at the third stage, the requests compete against those coming from D-NoC Rx under static-priority arbitration, where the requests from D-NoC Rx always have higher priority.

Regarding time management, a compute cluster provides a global cycle counter (located in the DSU) for timing synchronization between cores in the compute cluster. The cycle counter starts counting when the compute cluster is booted, and its timing information is stored at a specific address in the SMEM. Besides, Kalray MPPA-256 machine supports two performance counters per PE: one is used for counting the number of instruction cache

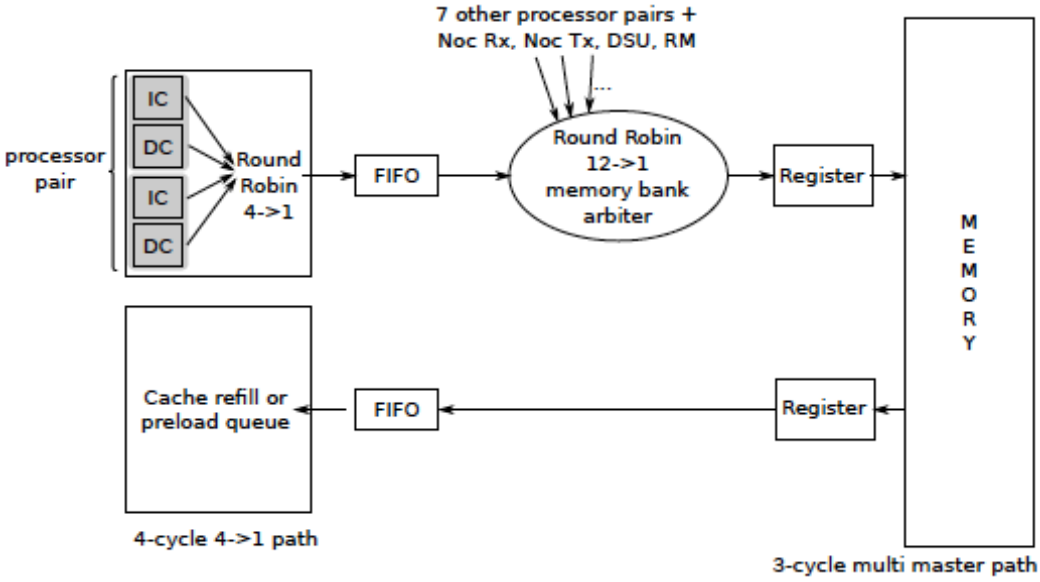


Figure 4.3: SMEM memory request flow [4]

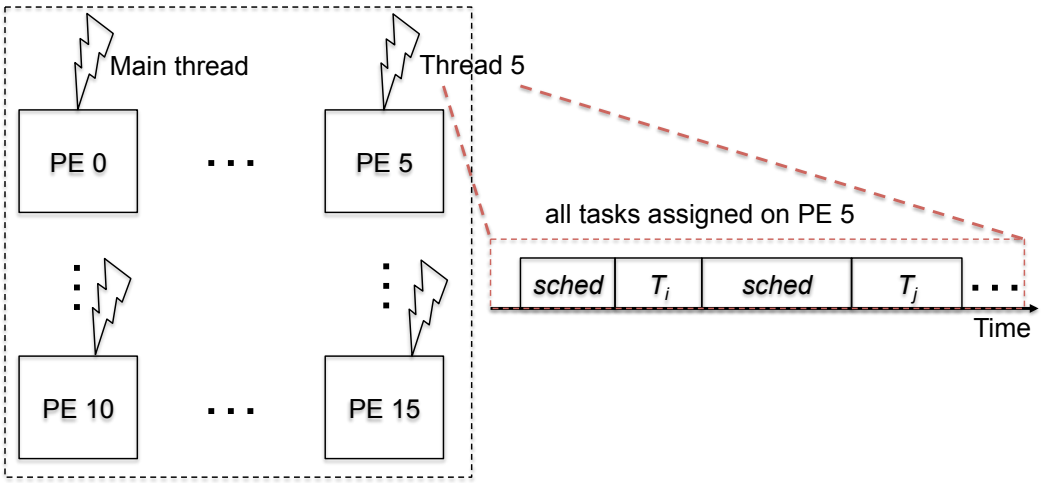


Figure 4.4: Structure of our proposed time-driven scheduler

misses, while the other one is used for counting the number of data cache misses. Hereafter, we use the term core and PE interchangeably.

4.2 General structure of our proposed time-driven scheduler

In the section we present the structure of our time-driven scheduler, and explain how to trigger the execution of a task at a specific instant of time on a specific core.

As illustrated in Figure 4.4, a *main* thread is executed on the first PE, whose main role is to initialize the data of the application, and to create other threads, as well as to synchronize them. Each thread is in charge of executing all tasks mapped on each PE. All threads are synchronized to run at the same time. Every task is preceded by a scheduling time-checking function, named *sched*, whose main role is to trigger their execution at a specific instant of time.

The *sched* function is simply a loop that repeatedly gets the timestamp of the global cycle counter in the compute cluster until the retrieved value is equal to or larger than the input of the function, which is the instant of time for triggering the execution of a task. The code of the *sched* function is given in Listing 4.1.

4.3 Practical challenges

This section identifies the practical challenges arising when implementing static time-driven cache-conscious schedules on a Kalray MPPA-256 compute cluster.

```
1 void sched(uint64_t triggerTime){
2   uint64_t curTimeStamp = 0;
3   do
4   {
5    // get the timing information from the global cycle counter
6    curTimeStamp = __k1_read_dsu_timestamp();
7     // check the criterion for exiting from the loop
8   } while(curTimeStamp < triggerTime);
9 }
```

Listing 4.1: The code of the *sched* function

4.3.1 Cache pollution caused by the scheduler

According to our time-driven scheduler implementation (presented in Section 4.2) the execution of a pair of tasks consecutively assigned to the same core is interleaved with the execution of the scheduling time-checking function (*sched*). As a result, the cache (i.e., the instruction cache and the data cache) in between the execution of the pair of tasks are polluted by the execution of the *sched* function. That may attenuate the benefit of cache reuse in term of tasks' WCET reduction. Therefore, the context-sensitive WCET of every task (due to cache reuse) has to be estimated in the presence of the *sched* function.

4.3.2 Shared bus contention

In a Kalray MPPA-256 compute cluster, concurrent requests issued from different cores to the same memory bank(s) compete against each other since each memory bank is equipped with only one requests arbiter. Therefore, the overhead induced on tasks by shared bus contention has to be taken into account. Note that according to our time-driven scheduler implementation, a PE is consecutively occupied by the executions of either the *sched* function or tasks mapped on the PE. Therefore, it may happen that memory requests of a task compete against those of both concurrent tasks *and* the *sched* function.

4.3.3 Delay to the start time of tasks because of the execution of the scheduling time-checking function

In our time-driven scheduler implementation, a task only starts executing when the scheduling time-checking function (*sched*) which triggers the execution of the task terminates. As illustrated in Figure 4.5, in the worst-case, the execution of the task is postponed by the amount of time that the *sched* function spends in its last iteration. There is a gap between the trigger time of a task (i.e., the instant of time given to the *sched* function) and the actual start time of the task (i.e., the instant of time at which the task effectively starts executing). The delay to the start time of a task leads to the change of its finish time. Therefore, the trigger time of every task has to be updated in order to satisfy their precedence relation(s).

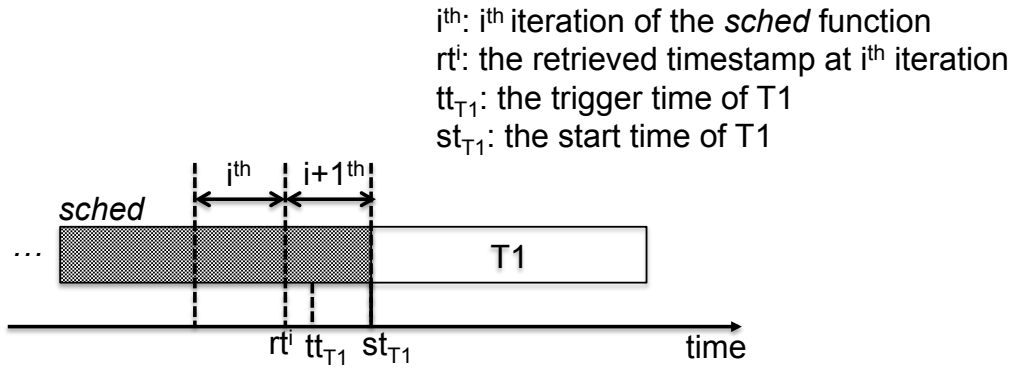


Figure 4.5: Illustrative example of the delay to the start time of a task caused by the execution of the *sched* function

4.3.4 Absence of hardware-implemented data cache coherence

The Kalray MPPA-256 machine does not support hardware-implemented cache coherence between cores. In a compute cluster, tasks executing on different cores communicate through the SMEM. However, data being written from the data cache to the SMEM is hold on the write buffer, but is not committed immediately to the SMEM. That property may cause the communications between pairs of tasks executing on different cores fail. In fact, if a task is assigned to a different core with its predecessor and starts executing right after the termination of its predecessor, the task may operate on obsolete data. The reason is that the freshest data (which the task intends to receive from its predecessor) has not been committed to the SMEM yet. Therefore, in order to guarantee that the communication between a pair of tasks executing on different cores is performed correctly, all memory stores of the sending task must be committed to the SMEM before its termination. This can be done by inserting particular instructions at the end of the execution of the task, which forces data hold on the write buffer to be committed to the shared memory. We name the instructions as *write buffer flushing instructions* hereafter (i.e., the instructions will be presented in Section 4.5).

Additionally, due to the lack of hardware-implemented cache coherence, data miss-alignment is another factor that can cause the failure of the communication between a pair of tasks executing on different cores. That effect is illustrated in the following example.

Let us consider three tasks T1, T2, and T3 executed on two cores in a compute cluster. The mapping and the scheduling of these tasks are depicted in the left side of Figure 4.6. T1 sends data to T3 and the data is stored in a dedicated buffer, named B_1_3, while T2 accesses data stored in another dedicated buffer, named B_2. We assume that T2 finishes its execution sooner than T1, and B_2 and B_1_3 are consecutively allocated in the SMEM and share the same data cache line (as illustrated in the right side of Figure 4.6). During the execution of T2, the data cache line will be loaded from the SMEM to the data cache. As a result, when T2 completes, a piece of data of B_1_3 has been stored in the data cache of *core 2*. Therefore, when T3 starts executing on *core 2* and is looking for the data sent from T1 (stored in B_1_3), it will find some in the data cache of the core (that have been

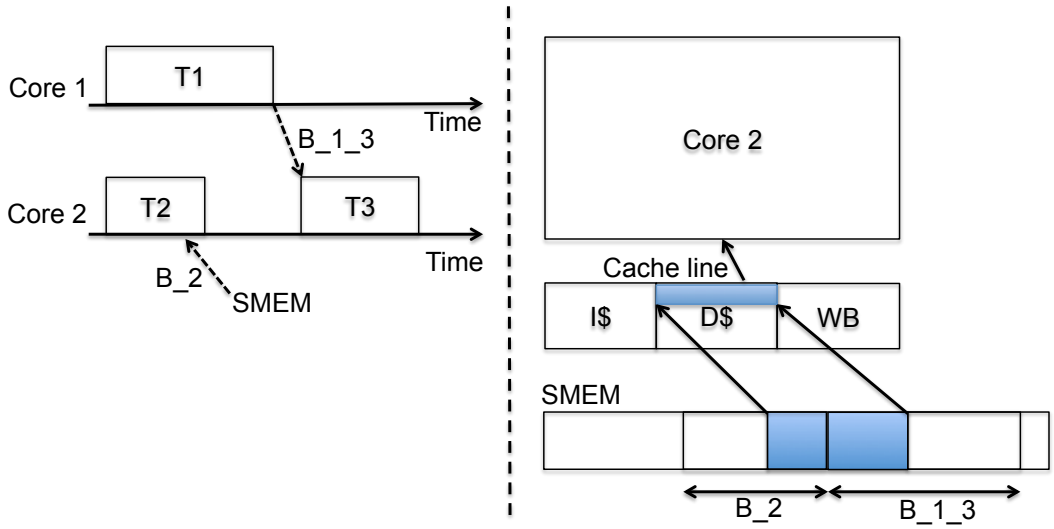


Figure 4.6: Illustrative example of the effect of data miss-alignment

loaded by T2). Since T2 finishes sooner than T1, those data may be obsolete. Operating on the obsolete data leads T3 to produce incorrect outputs. In order to overcome the issue, communication buffers are aligned on data cache line boundaries, as detailed in Section 4.5.

4.4 Adaptation of time-driven cache-conscious schedules to the practical issues

This section presents our strategy to adapt static time-driven cache-conscious schedules to the practical issues presented in Section 4.3. Let us name the stage in which cache-conscious schedules are generated with completely ignoring all the practical issues as *basic stage*, and the stage in which those schedules are adapted to the practical issues as *adapted stage*. In the *basic stage*, the cache-conscious scheduling methods presented in Chapter 3 are used for producing cache-conscious schedules, and those schedules are named *basic cache-conscious schedules*. The schedules produced in *the adapted stage* are named *adapted cache-conscious schedules*. Those schedules will be implemented on a Kalray MPPA-256 compute cluster.

Figure 4.7 shows the relation between the *basic stage* and the *adapted stage*, as well as the common and the different aspects between *a basic cache-conscious schedule* and the adapted one, the *adapted cache-conscious schedule*. The mapping and the execution order of every task in the *adapted cache-conscious schedule* are kept the same as those in the *basic cache-conscious schedule*.Figure 4.8 zooms in the different aspects between the *basic cache-conscious schedule* and the *adapted cache-conscious schedule*. While in the *basic cache-conscious schedule* the effects of all the practical issues are ignored, in the *adapted cache-conscious schedule* the overheads caused by those issues are considered in the execution of tasks. The overheads include:

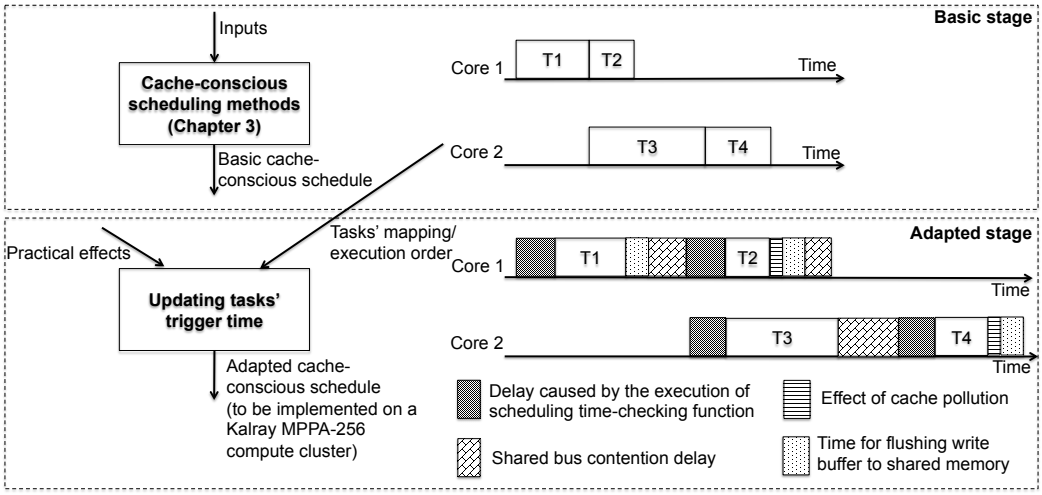


Figure 4.7: Two stages in producing static time-driven cache-conscious schedules to be implemented on a Kalray MPPA-256 compute cluster

- The overhead due to cache pollution by the execution of the scheduling time-checking function (*sched*).
- The delay to the start time of tasks caused by the execution of the *sched* function.
- The cost for flushing the write buffer to the shared memory. Note that the cost is only appended to the exit task (i.e., to make sure the outputs of the application are committed to the shared memory) and tasks which have to send data to any task assigned to a different core with them.
- The delay due to shared bus contention.

Due to the practical factors, the trigger time of every task has to be updated in order to satisfy their precedence relation.

The rest of the section is organized as follows. Section 4.4.1 presents data structures of the scheduler, including the task graph of an application and the metadata attached to tasks. Section 4.4.2 presents the restrictions of the execution of an application on a Kalray MPPA-256 compute cluster. Finally, Section 4.4.3 details the adaptation of *basic cache-conscious schedules* to the practical issues.

4.4.1 Data structures

An application is presented as a DAG (whose structure is previously presented in Section 3.1) decorated with the metadata of tasks (as depicted in Figure 4.9), which includes:

- The mapping and the execution order of tasks. Those information are retrieved from *basic cache-conscious schedules*. For example, C(T2) retrieves the core to which T2

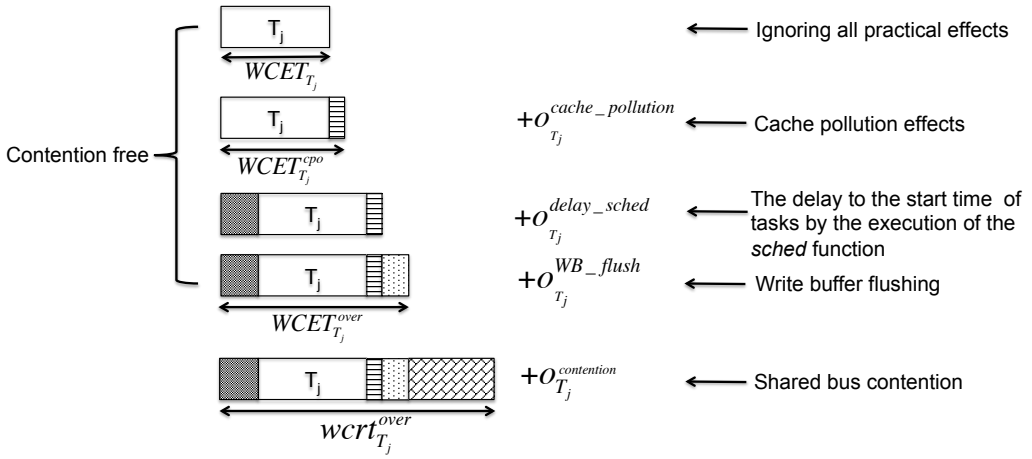


Figure 4.8: The difference in the execution of a task in a *basic cache-conscious schedule* and an *adapted cache-conscious schedule*

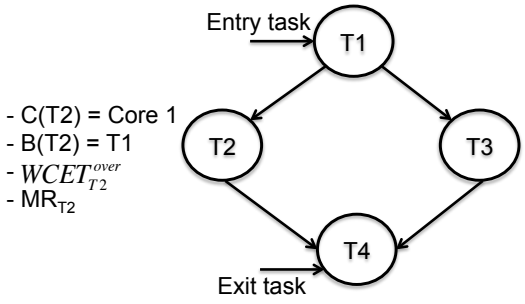


Figure 4.9: Data structures used in the *adapted stage*

is assigned, and B(T2) retrieves the task executing on the same core and right before T2, i.e., as illustrated in Figure 4.7, C(T2) = core 1, and B(T2) = T1.

- The worst-case execution time of tasks accounted for all overheads caused by the aforementioned practical issues except shared bus contention delay, noted as $WCET^{over}_{\tau_j}$ for τ_j. To be safe, $WCET^{over}_{\tau_j}$ is the sum of the following factors:
  - the context-sensitive worst-case execution time of τ_j, which takes into account the effect of cache pollution (denoted as $WCET^{cpo}_{\tau_j}$),
  - the worst-case delay to the start time of τ_j caused by the execution of the *sched* function (denoted as $o^{delay\_sched}_{\tau_j}$),
  - the upperbound of the cost of committing outputs of τ_j to the shared memory at the end of its execution (denoted as $o^{WB\_flush}_{\tau_j}$, if required).

Since the execution order of tasks are known (i.e., retrieved from the *basic cache-conscious schedule*) so $WCET_{\tau_j}^{cpo}$ is predetermined. Additionally, $o_{\tau_j}^{delay\_sched}$ is equal to the worst-case execution time of one iteration of the *sched* function (denoted as $WCET_{sched}$), and $o_{\tau_j}^{WB\_flush}$ is equal to the uppbound of the cost of flushing the write buffer to the shared memory (denoted as U_{WB_flush}). Those factors are constants. Therefore, $WCET_{\tau_j}^{over}$ is predetermined. The estimation of $WCET_{\tau_j}^{cpo}$, $WCET_{sched}$, and U_{WB_flush} will be detailed in Section 4.6.1.

- The number of memory requests during the execution of tasks, noted as MR_{τ_j} for τ_j. To be safe, MR_{τ_j} is the sum of the following factors:
  - the number of cache misses (i.e., includes data cache misses and instruction cache misses) of τ_j, which takes into account the effect of cache pollution,
  - the maximum number of memory requests of one iteration of the *sched* function,
  - the maximum number of memory accesses for flushing the write buffer to the shared memory after the execution of τ_j (if required).

  Similar to $WCET_{\tau_j}^{over}$, MR_{τ_j} is predetermined. The estimation of the factors which are subsumed into MR_{τ_j} will be detailed in Section 4.6.1.

4.4.2 Restrictions of the execution of an application

In order to implement our static time-driven cache-conscious schedules on a Kalray MPPA-256 compute cluster, the following restrictions of the execution of an application are imposed:

- We consider an application whose code and data fit into the SMEM of the compute cluster. Therefore, the Resource Manager (RM) loads the application entirely onto a compute cluster before the application starts, and later does not interfere with the execution of the application.
- An application is executed in isolation on a compute cluster, such that there is no accesses from the NoC during its execution. Therefore, there is no contention between operations on the NoC and the execution of the application.
- An application is not executed in debug mode such that the operations on Debug Support Unit (DSU) do not contend with the execution of the application. As a consequence, the interference on shared bus occurs only between processing elements (PEs). That means the arbitration of memory requests to the SMEM's banks can be simplified from three stages (as described in Section 4.1) to one stage, which are regulated by a round-robin policy.

4.4.3 Adapting basic cache-conscious schedules to the practical effects

Given the mapping and the execution order of tasks in a *basic cache-conscious schedule* we formulate an ILP formulation to compute the trigger time of tasks with respect to the source

Symbol	Description	Data type
τ	The set of tasks of the parallel application	set
$dPred(\tau_j)$	The set of direct predecessors of τ_j	set
$allPred(\tau_j)$	The set of direct and indirect predecessors of τ_j	set
$allSucc(\tau_j)$	The set of direct and indirect successors of τ_j	set
$B(\tau_j)$	A co-located task executing right before τ_j	set
$C(\tau_j)$	A core to which τ_j is assigned	set
c	The set of cores to which tasks are assigned	set
$LT(c_j)$	The last running task on core c_j	set
$WCET_{\tau_j}^{over}$	The worst-case execution time of τ_j accounted for all overheads caused by the practical issues except shared bus contention delay	integer
MR_{τ_j}	The total number of memory requests during the execution of τ_j	integer
$DMEM$	The upperbound of memory access latency when contention free	integer
sl	The length of the adapted cache-conscious schedule	integer
tt_{τ_j}	The trigger time of τ_j	integer
ft_{τ_j}	The finish time of τ_j	integer
$o_{\tau_j}^{contention}$	The overhead induced on the execution of τ_j by shared bus contention	integer
$wcrt_{\tau_j}^{over}$	The worst-case response time of τ_j	integer
$\delta_{\tau_j}^{c_j}$	The maximum number of memory accesses issued from core c_j that interfere with the execution of τ_j	integer
δ_{τ_j}	The maximum number of memory accesses that interfere with the execution of τ_j	integer
$intf_{\tau_j}^{c_j}$	Indicates if the memory accesses from core c_j interfere with the execution of τ_j or not	binary

Table 4.1: Notations used in the ILP formulation in the *adapted stage*

of overheads listed in Section 4.3, such that their precedence relation are satisfied and the length of the *adapted cache-conscious schedule* is minimized.

Contention model. According to the structure of our time-driven scheduler (presented in Section 4.2), there is activity on all cores (either the execution of the *sched* function or the execution of tasks). Therefore, in order to ensure that all possible contentions are captured, we consider that a task always interfere with operations on a core except when the task is triggered after the termination of the last task running on the core. Furthermore, when contention occurs, we consider that all memory requests of the task are delayed.

For example, according to the mapping and the schedule of tasks as depicted in Figure 4.10, T1 and T2 are triggered before the execution of T3 and T4 (i.e., their direct successors), respectively. Furthermore, before the execution of T3 (or T4) there is activity on core 2 which is the execution of *sched* function (or T3). Therefore, it is safe to consider that all memory requests of both T1 and T2 are delayed.

The objective of our contention model is not only to have the safe bound of interference

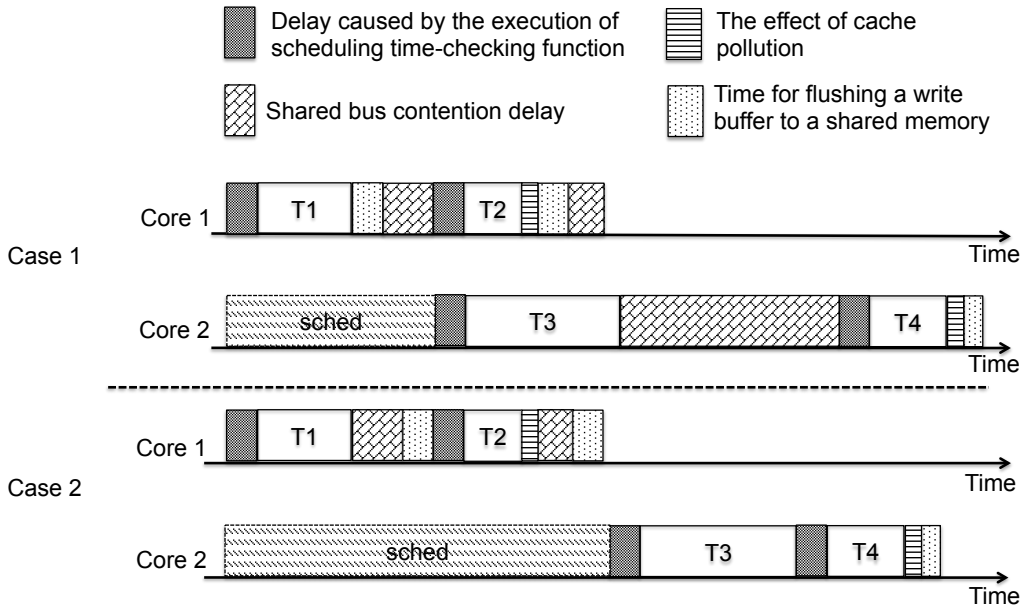


Figure 4.10: The illustrative example of assigning the trigger time of tasks

delay but also to reduce the burden of an ILP solver. According to the contention model, for minimizing schedules length, an ILP solver only has to focus on finding the trigger time of the tasks at the nearly end of schedules (for example, T3 in Figure 4.10) to reduce interference delay induced on the tasks.

As illustrated in Figure 4.10, in *case 1* the trigger time of T3 can be assigned to the finish time of T1, or in *case 2*, the trigger time of T3 can be shifted to the finish time of T2. In *case 1*, all memory requests of T3 are delayed (i.e., since the trigger time of T3 is smaller than the finish time of T2). In *case 2*, the execution of T3 is contention free. Note that the shifting has to be examined such that the length of the adapted cache-conscious schedule is minimized.

Notations used in the ILP formulation The notations used in the description of our ILP formulation are summarized in Table 4.1. The content of the table is organized as follows.

- The first block defines notations to manage the task graph of an application attached with the information derived from the *basic cache-conscious schedule* of the application. In this block, notations denoted as τ, $dPred(\tau_j)$, $allPred(\tau_j)$, and $allSucc(\tau_j)$ represent the structure of the task graph of an application. Besides, other notations represent the information derived from the *basic cache-conscious schedule*, such as $B(\tau_j)$ retrieves the task executing on the same core and right before τ_j, $C(\tau_j)$ retrieves the core to which τ_j is assigned, $LT(c_j)$ retrieves the last task running on core

c_j, and c represents the set of cores to which tasks are assigned.

- The second block defines predetermined parameters using upper case letters. The notation denoted as $WCET^{over}_{\tau_j}$ represents the worst-case execution time of τ_j accounted for all overheads caused by the practical issues except shared bus contention delay, whereas the notation denoted as MR_{τ_j} represents the number of memory requests issued in the execution of τ_j. In addition, the notation denoted as $DMEM$ stands for the upperbound of memory access latency when contention free. The estimation of those parameters is detailed in Section 4.6.1.

- The third block defines variables using lower case letters. The notations, denoted as sl, tt_{τ_j}, and ft_{τ_j}, stand for the schedule length of an *adapted cache-conscious schedule*, the trigger time of τ_j, and the finish time of τ_j, respectively. The notation, denoted as $wcrt^{over}_{\tau_j}$, stands for the worst-case response time of τ_j (as illustrated in Figure 4.8), which is the worst-case execution time of τ_j accounted for shared bus contention delay (and the other overheads). Besides, the other notations, denoted as $o^{contention}_{\tau_j}$, $\delta^{PE_y}_{\tau_j}$, δ_{τ_j}, and $intf^{c_j}_{\tau_j}$, aim to manage the contention with the execution of τ_j.

ILP formulation for adapting *basic cache-conscious schedules* to the practical effects. We name the ILP formulation as ACILP (for adapting cache-conscious schedules) hereafter. The objective function of ACILP is to minimize the schedule length, noted as sl, of a parallel application which is expressed as follows:

$$\text{minimize} \quad sl \tag{4.1}$$

Since the schedule length of a parallel application has to be larger than or equal to the finish time of any task τ_j, noted as ft_{τ_j}, the following constraint is introduced:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &\quad sl \geq ft_{\tau_j} \end{aligned} \tag{4.2}$$

The finish time of τ_j, denoted as ft_{τ_j}, is the sum of its trigger time, denoted as tt_{τ_j}, and its worst-case response time, denoted as $wcrt^{over}_{\tau_j}$.

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &ft_{\tau_j} = tt_{\tau_j} + wcrt^{over}_{\tau_j} \end{aligned} \tag{4.3}$$

In the rest of the ILP formulation, we first present constraints for computing the trigger time of tasks, then we present constraints for computing the worst-case response time of tasks.

Constraints on the trigger time of tasks. If τ_j has any direct predecessors (noted as $\tau_i \in dPred(\tau_j)$) or if τ_j has a co-located task running right before (noted as $\tau_i \in B(\tau_j)$), the task can be executed only when those tasks have finished their execution. Therefore, in order to ensure the precedence relations of a task are satisfied, the trigger time of the task

has to be larger than or equal to the finish time of all its direct predecessors and the finish time of co-located task executing right before the task (if existed).

$$\begin{aligned} \forall \tau_j \in \tau, dPred(\tau_j) \neq \emptyset \vee B(\tau_j) \neq \emptyset, \\ \forall \tau_i \in dPred(\tau_j) \vee \tau_i = B(\tau_j), \\ tt_{\tau_j} \geq ft_{\tau_i} \end{aligned} \tag{4.4}$$

When a task has no predecessor and it is the first task running on a core, the trigger time of the task has to be larger than or equal to zero.

$$\begin{aligned} \forall \tau_j \in \tau, dPred(\tau_j) = \emptyset \wedge B(\tau_j) = \emptyset, \\ tt_{\tau_j} \geq 0. \end{aligned} \tag{4.5}$$

Constraints on the worst-case response time of tasks. Since every core in Kalray MPPA-256 machine is fully timing-compositional, it is safe to compute the worst-case response time of τ_j, denoted as $wcrt^{over}_{\tau_j}$, as the sum of its worst-case execution time, denoted as $WCET^{over}_{\tau_j}$, and the shared bus contention delay induced on its execution, denoted as $o^{contention}_{\tau_j}$.

$$\begin{aligned} \forall \tau_j \in \tau, \\ wcrt^{over}_{\tau_j} = WCET^{over}_{\tau_j} + o^{contention}_{\tau_j} \end{aligned} \tag{4.6}$$

Let us denote δ_{τ_j} the maximum number of memory requests that delay the execution of τ_j, and $DMEM$ the upperbound of memory access latency in case contention free, the shared bus contention delay induced on the execution of τ_j, denoted as $o^{contention}_{\tau_j}$, is computed as:

$$o^{contention}_{\tau_j} = \delta_{\tau_j} * DMEM, \tag{4.7}$$

Let us denote $\delta^{c_j}_{\tau_j}$ the maximum number of memory requests issued from core $c_j \neq C(\tau_j)$ that interfere with the execution of τ_j. Considering all contentions with the execution of τ_j from all cores to which tasks are assigned ($c_j \in c \wedge c_j \neq C(\tau_j)$), δ_{τ_j} is computed as follows:

$$\delta_{\tau_j} = \sum_{c_j \in c \wedge c_j \neq C(\tau_j)} \delta^{c_j}_{\tau_j} \tag{4.8}$$

In order to compute $\delta^{c_j}_{\tau_j}$, we have to determine whether the memory requests of τ_j compete against those issued from c_j or not (via a binary variable, denoted as $intf^{c_j}_{\tau_j}$). According to our contention model, if there is no contention between operations on c_j and the execution of τ_j, then $intf^{c_j}_{\tau_j} = 0$, and $\delta^{c_j}_{\tau_j} = 0$; otherwise, $intf^{c_j}_{\tau_j} = 1$, and $\delta^{c_j}_{\tau_j}$ is equal to the number of memory requests of τ_j, denoted as MR_{τ_j}. The computation of $\delta^{c_j}_{\tau_j}$ is formulated as follows:

$$\delta^{c_j}_{\tau_j} = intf^{c_j}_{\tau_j} * MR_{\tau_j} \tag{4.9}$$

The value of $intf^{c_j}_{\tau_j}$ depends on the relation between the trigger time of τ_j and the finish time of the last task running on core c_j, retrieved by $LT(c_j)$. If τ_j and $LT(c_j)$ have a precedence relation, we can predetermine the value of $intf^{c_j}_{\tau_j}$. In fact:

- if τ_j is either a direct predecessor or an indirect predecessor of $LT(c_j)$, τ_j has to start executing before the execution of $LT(c_j)$. In this case, the execution of τ_j is considered to be contended with operations on c_j, that means $intf_{\tau_j}^{c_j} = 1$,
- if τ_j is either a direct successor or an indirect successor of $LT(c_j)$, τ_j has to start executing after the termination of LT_{c_j}. Therefore, the execution of τ_j does not contend with any operations on c_j, that means $intf_{\tau_j}^{c_j} = 0$.

If τ_j and $LT(c_j)$ do not have any precedence relations, the determination of $intf_{\tau_j}^{c_y}$ is presented as follows.

If the trigger time of τ_j is larger than or equal to the finish time of $LT(c_j)$, then $intf_{\tau_j}^{c_j} = 0$; otherwise $intf_{\tau_j}^{c_j} = 1$.

$$intf_{\tau_j}^{c_j} = \begin{cases} 0 & \text{if} \quad tt_{\tau_j} \geq ft_{LT(c_j)} \\ 1 & \text{if} \quad tt_{\tau_j} < ft_{LT(c_j)} \end{cases} \tag{4.10}$$

The above constraint is formulated by using classical big-M notation as:

$$\begin{aligned} &\forall \tau_j \in \tau, \forall c_j \in c \wedge c_j \neq C(\tau_j), \tau_j \notin allPred(LT(c_j)) \wedge \tau_j \notin allSucc(LT(c_j)) \\ &\quad ft_{LT(c_j)} - tt_{\tau_j} \geq 1 - M * (1 - intf_{\tau_j}^{c_j}) \\ &\quad ft_{LT(c_j)} - tt_{\tau_j} \leq M * intf_{\tau_j}^{c_j} \end{aligned} \tag{4.11}$$

where M, is a constant[1] higher than any possible $ft_{LT(c_j)}$.

4.5 Code generator

This section presents our strategy for generating the code of an application to be executed on a Kalray MPPA-256 compute cluster according to its *adapted cache-conscious schedule*.

We create a script which consists of *sed* commands to automatically generate the code of the application with respect to the structure of our time-driven scheduler implementation (presented in Section 4.2). The inputs of the script contain the *adapted cache-conscious schedule* of the application and a template (illustrated in Listing 4.2) including:

- the code of every task in the application,
- the declaration of every communication buffer,
- the initialization of the data of the application,
- guided comments that tell the script where to insert code.

In order to ensure the produced code to execute correctly on a Kalray MPPA-256 compute cluster, the issue caused by the absence of hardware-implemented data cache coherence (as explained in Section 4.3.4) has to be avoided. For that, the code generator obeys the following rules:

[1]For the experiments, M is the sum of the worst-case response time of all tasks when considering the worst-case shared bus contention delay, i.e., the total number of memory requests that interfere with the execution of τ_j is equal to $MR_{\tau_j} * (|c| - 1)$, which $|c|$ is the number of cores to which tasks are assigned, to ensure M is higher than the finish time of any task.

```
1
2  // Declaration of global buffers
3  int b_1_2[10];
4  int b_1_3[10];
5  int b_2_4[10];
6  int b_3_4[10];
7  int output = 0;
8
9  // Declaration of tasks in the application
10 void task1();
11 void task2();
12 void task3();
13 void task4();
14
15 // Code of tasks in the application
16 void task1(){
17         for (int i = 0; i <= 9; i++)
18         {
19                 b_1_2[i] = i;
20                 b_1_3[i] = i;
21         }
22 }
23 void task2(){
24         for (int i = 0; i <= 9 ; i++)
25         {
26                 b_2_4[i] = b_1_2[i] * b_1_2[i];
27         }
28 }
29 void task3(){
30         for (int i = 0; i <= 9 ; i++)
31         {
32                 b_3_4[i] = 2 * b_1_3[i];
33         }
34 }
35 void task4(){
36         for (int i = 0; i <= 9 ; i++)
37         {
38                 output += b_2_4[i] * b_3_4[i];
39         }
40 }
41
42 //guided comment
43 /* insert Sched function */
44
45 //guided comment
46 /* insert code for each thread */
47
48 int main()
49 {
50         // the initialization of the data of the application
51                         ...
52         // guided comment
53         /* insert the declaration of local variables and initialize their value */
54
55         // guided comment
56         /* insert code to join all threads*/
57
58 }
```

Listing 4.2: The template of the application

- Write buffer flushing instructions are appended to tasks which have any direct successors assigned to a different core with them, as well as to the exit task in order to ensure that the outputs of those tasks are committed to the shared memory before their termination. Note that those tasks are predetermined based on the mapping information provided in the *basic cache-conscious schedule* and the DAG of the application. The write buffer flushing instructions are __builtin_k1_wpurge() that requests the write buffer flush to the shared memory, followed by __builtin_k1_fence() that waits for all data to be committed to the shared memory. These instructions are built-in functions supported by the Kalray MPPA-256 machine.

- Each pair of communicating tasks has a dedicated buffer for storing their transferred data, and those buffers are declared globally. Each global buffer is aligned on data cache line boundaries (i.e., 32 bytes). That can be done by appending an alignment attribute (i.e., __attribute__ ((aligned (32)))) to the declaration of each global buffer.

In the rest of the section, we illustrate an example of code generation. The *adapted cache-conscious schedule* for two cores of the application whose DAG was presented in Figure 4.9 is depicted in Figure 4.11. For simplicity, in the figure, we only denote the trigger time and the finish time of tasks. Listing 4.2 presents the template used as an input of the code generator, and Listing 4.3 shows the produced code of the application. Two threads (named *Thread1* and *Thread2*) are created in the main function using the Pthreads library interface. *Thread1* is assigned to the *core 1* and is in charge of executing tasks that are assigned to that core, while *Thread2* is assigned to the *core 2* and is in charge of executing tasks that are assigned to that core. Since the communication between pairs of tasks such as (T1, T3) and (T2, T4) are performed on different cores, the write buffer flushing instructions are appended to both T1 and T2. The write buffer flushing instructions are also appended to the exit task, T4, to ensure the outputs of the application are stored in the shared memory before its termination. Every communication buffer is aligned on data cache line boundaries. Inside both *Thread1* and *Thread2* every task is preceded by the *sched* function. The input of the *sched* function is the trigger time of the following task[2].

```
1  // align global buffers on data cache line boundaries, 32 bytes
2  int b_1_2[10] __attribute__ ((aligned (32)));
3  int b_1_3[10] __attribute__ ((aligned (32)));
4  int b_2_4[10] __attribute__ ((aligned (32)));
5  int b_3_4[10] __attribute__ ((aligned (32)));
6  int output = 0;
7
8  // Declaration and code of tasks in the application
9          ...
10
11 /* insert Sched function */
12
```

[2]In the execution, since the global cycle counter of a compute cluster starts counting when the compute cluster is booted, so that the timestamp which is the input of the *sched* function is the sum of the trigger time of a task retrieved from an adapted cache-conscious schedule, and the worst-case duration for informing all cores the start point of the timeline, as well as the duration for synchronizing all threads.

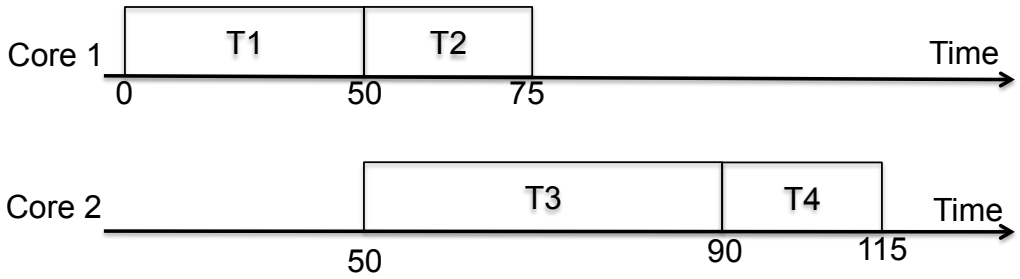


Figure 4.11: The mapping and the schedule of all tasks on two cores of the application whose DAG was depicted in Figure 4.9

```
13 void sched(uint64_t triggerTime){
14         uint64\_t curTimeStamp = 0;
15         do
16         {
17                 // get the timing information from the global cycle counter
18                 curTimeStamp = __k1_read_dsu_timestamp();
19                 // check the criterion for exiting from the loop
20         } while(curTimeStamp < triggerTime);
21 }
22
23 /* insert code for each thread */
24  void* Thread1(void *args) // contains tasks assigned to the first core
25 {
26         // thread synchronization
27                 ...
28         sched(0);
29         task1();
30         __builtin_k1_wpurge();
31         __builtin_k1_fence(); // make sure b_1_3 flushed to a SMEM
32
33         sched(50);
34         task2();
35         __builtin_k1_wpurge();
36         __builtin_k1_fence(); // make sure b_2_4 flushed to a SMEM
37 }
38
39 void* Thread2(void *args) // contains tasks assigned to the second core
40 {
41         // thread synchronization
42                 ...
43
44         sched(50);
45         task3();
46
47         sched(90);
```

```
48 ||          task4();
49 ||          __builtin_k1_wpurge();
50 ||          __builtin_k1_fence();// make sure the output of the application are committed to a SMEM
51 || }
52 |
53 | int main()
54 | {
55 |          // the initialization of data of the application
56 |                                  ...
57 |
58 |          /* insert the declaration of local variables and initialize their value */
59 ||          pthread_t thread_t_1, thread_t_2;
60 ||          pthread_attr_t attrT1, attrT2;
61 ||          const unsigned int PE1 = 0x0002;
62 ||          const unsigned int PE2 = 0x0004;
63 ||          pthread_attr_setaffinity_np(&attrT1,sizeof(unsigned int), &PE1);
64 ||          pthread_attr_setaffinity_np(&attrT2,sizeof(unsigned int), &PE2);
65 ||          pthread_create(&thread_t_1, &attrT1, Thread1,NULL); // assigned Thread1 to PE1
66 ||          pthread_create(&thread_t_2, &attrT2, Thread2,NULL); // assigned Thread2 to PE2
67 ||
68 ||          /* insert code to join all threads*/
69 ||          pthread_join(thread_t_1, NULL);
70 ||          pthread_join(thread_t_2, NULL);
71 || }
```

Listing 4.3: Example of the code of an application to be executed on a Kalray MPPA-256 compute cluster

4.6 Experimental evaluation

In this section, we first validate the functional and temporal correctness of applications when executing on a Kalray MPPA-256 compute cluster. We then investigate the overhead induced on adapted cache-conscious schedules by different practical issues listed in Section 4.3. Finally, we evaluate the performance of our proposed ILP formulation (ACILP) in both terms of the quality of adapted cache-conscious schedules and the time required for generating the schedules. Experimental conditions are described in Section 4.6.1. Experimental results are then detailed in Section 4.6.2.

4.6.1 Experimental conditions

4.6.1.1 Benchmarks

In the experiment we use four benchmarks [3] from the StreamIt benchmark suite [10], named AudioBeam, AutoCor, FmRadio, and MergeSort. Table 4.2 shows the code size of the benchmarks, whereas Table 4.3 shows the characteristics of the task graph of the benchmarks. For the detail of those tables, please refer to Section 3.3.1.

4.6.1.2 Constants estimation

Upperbound of memory access latency when contention free ($DMEM$). Memory accesses can be categorized in four types:

- Loading an instruction cache line (i.e., 64 bytes) from the shared memory to the instruction cache.
- Loading a data cache line (i.e., 32 bytes) from the shared memory to the data cache.
- Retrieving the timestamp (i.e., 8 bytes) of the global cycle counter from the shared memory.
- Committing data stored in the write buffer (i.e. 8-way fully-associative, each way contains 8 bytes) to the shared memory.

Among the memory requests listed above, the penalty caused by an instruction cache miss is the most expensive one. Therefore, the upperbound of memory access latency when contention free, noted as DMEM, is equal to the cost for loading an instruction cache line from the shared memory to the instruction cache. According to [86], an access to the shared memory, in case no contention occurs takes 9 cycles with 8 bytes fetched on each consecutive cycle, therefore, $DMEM$ is equal to 17 cycles.

[3] With those benchmarks, we do not have to modify the code of tasks to have a communication buffer per pair of communicating tasks. Having the constraint with the other benchmarks requires costly code modification process.

Benchmark	Code size (Bytes)		Communicated data (Bytes)
	Entire application	μ / σ of tasks	μ
AudioBeam	38076	1458 / 1897	6
Autocor	12348	1014 / 538	66
FmRadio	374812	1072 / 679	4
MergeSort	34208	1088 / 366	16

Table 4.2: The size of code and communicated data for each benchmark (average μ and standard deviation σ).

Benchmark	No. of tasks	No. of Edges	Maximum graph width	Average graph width	Graph Depth
AudioBeam	20	33	15	3.3	6
Autocor	12	18	8	2.4	5
FmRadio	67	85	20	5.6	12
MergeSort	31	37	8	2.6	12

Table 4.3: Summary of the characteristics of the benchmarks in our case studies.

Upperbound of the cost of flushing the write buffer to the shared memory when contention free (U_{WB_flush}). The write buffer per PE is 8-way fully associative and each way contains 8 bytes. Since the memory access granularity in a compute cluster is 8 bytes, the number of memory accesses for flushing the full write buffer to the shared memory is 8. It takes 10 cycles for each access that commits 8 bytes from the write buffer to the shared memory. Therefore, the upperbound of the cost for flushing the write buffer to the shared memory when contention free is 80 cycles.

4.6.1.3 WCET and number of cache misses estimations when contention free

For tasks. The WCET and the number of cache misses of every task are estimated according to their execution order given in *basic cache-conscious schedules*. If τ_j is the first task running on a core, its WCET and its number of cache misses are estimated when τ_j executes in isolation. If τ_j executes on the same core and right after τ_i, its WCET (denoted as $WCET^{cpo}_{\tau_j}$) and its number of cache misses are estimated with respect to cache pollution issue (as explained in Section 4.3.1). In this case, the WCET of τ_j and its number of cache misses are estimated when τ_j executes in the following order: $\tau_i \rightarrow sched(0) \rightarrow \tau_j$. The input of the *sched* function is zero, which means that the function only runs one iteration. The reason is that during the execution of the *sched* function the contents of caches do not change after the first iteration of the function.

For one iteration of the scheduling time-checking function (*sched*). We pass zero to the input of the *sched* function and execute the function in isolation in order to estimate the WCET and the number of cache misses in one iteration of the function.

All the above estimations are performed on a core of a Kalray MPPA-256 compute cluster, while the other cores are left idle. Besides, at the begin of each measurement we invalidate both the instruction cache and the data cache of the core. Similar to Chapter 3, we measure the worsts-case execution time of every task (and one iteration of the *sched* function) by using the global cycle counter in a compute cluster[4]. Additionally, we use two performance counters (one for counting the number of instruction cache misses, and one for counting for the number of data cache misses) to measure the number of cache misses of every task and one iteration the *sched* function as well.

[4]Please refer to Section 3.3.1 for more details.

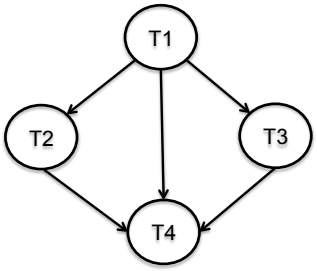


Figure 4.12: The schedule graph constructed based on the scheduling information in the adapted cache-conscious schedule as depicted in Figure 4.11

The WCET of one iteration of the *sched* function (denoted as $WCET_{sched}$) is 258 cycles, and the total number of cache misses of one iteration of the function is 10. Including one memory access for reading the timestamp of the global cycle counter, the upperbound of the number of memory requests of one iteration of the *sched* function is 11.

4.6.1.4 Experimental environment

In the *basic stage*, we use the heuristic scheduling technique proposed in Section 3.2.2 for generating basic cache-conscious schedules since the proposed heuristic method produces schedules very fast and the length of those schedules are close to the optimal ones. In the *adapted stage*, we use Gurobi optimizer version 6.5 [93] for solving ACILP formulation. The solving time of the solver is limited to one hour. The ILP solver is executed on 3.6 GHz Intel Core i7 CPU with 16GB of RAM.

4.6.2 Experimental results

4.6.2.1 Validation of the functional correctness and the timing correctness of benchmarks when executing on a Kalray MPPA-256 compute cluster

Functional correctness. For validating the functional correctness, we compare the outputs of one iteration of the sequential version of all benchmarks in the study, i.e., all tasks are executed on one core, with those of their parallel version, i.e., tasks are executed according to their mapping and their scheduling information given in adapted cache-conscious schedules. We observed that both the versions produce the same outputs.

Temporal correctness. For validating the temporal correctness, we record the actual start time and the finish time of every task when executing on a Kalray MPPA-256 compute cluster and check whether their precedence relation are satisfied or not. The results showed that the precedence constraints between tasks are not violated.

4.6.2.2 Quantification of the impact of different practical issues on adapted cache-conscious schedules

The impact of different practical issues listed in Section 4.3 on an adapted cache-conscious schedule are reflected by the portion of their overheads in the schedule length. However, it is impossible to compute the fractions because the schedule length is not a linear combination of the execution time of a set of tasks. Therefore, for reasonable quantification we find a critical path of the schedule graph (to be described later) and compute the portion of overheads caused by different practical issues on this path. The details of the quantification are given as follows.

- We first construct a schedule graph based on the schedule information of tasks in the adapted cache-conscious schedule (as illustrated in Figure 4.12). In the schedule graph, each node represents a task. Two nodes are connected by an edge if the finish time of the task represented by a source node is smaller than or equal to the trigger time of the task represented by a sink node. The weight of a node is the duration of the execution of a task represented by the node, while the weight of every edge is zero.

- We then find a critical path of the schedule graph by using implicit-path enumeration technique (IPET) [17]. The set contains tasks which are lied on the critical path is denoted as τ^{cp}. The length of the schedule graph, denoted as sl^{sg}, is computed as the sum of the duration of the execution of every task in τ^{cp}.

- Finally, we compute the fraction of overall overhead induced on the execution of every task in τ^{cp} by each practical issue to the length of the schedule graph.

The classification of the overhead induced on execution of a task by practical issues listed in Section 4.3 was shown in Figure 4.8.

- **The increase in the execution time of τ_j caused by cache pollution, denoted as $o_{\tau_j}^{cache\_pollution}$** , which is computed by subtracting the worst-case execution time of τ_j when ignoring all practical issues except cache pollution ($WCET_{\tau_j}^{cpo}$) from the worst-case execution time of τ_j when ignoring all practical issues ($WCET_{\tau_j}$[5]).

$$o_{\tau_j}^{cache\_pollution} = WCET_{\tau_j}^{cpo} - WCET_{\tau_j} \tag{4.12}$$

  Note that if τ_j is the first task running on a core, τ_j will not benefit from cache reuse, so that the execution time of τ_j is not affected by cache pollution. That means $WCET_{\tau_j} = WCET_{\tau_j}^{cpo}$, which results in $o_{\tau_j}^{cache\_pollution} = 0$.

- **The delay to the start time of τ_j caused by the execution of the *sched* function, denoted as $o_{\tau_j}^{delay\_sched}$** , which is equal to the worst-case execution time of one iteration of the *sched* function ($WCET_{delay\_sched}$). As mentioned in Section 4.6.1.3, $WCET_{delay\_sched} = 258$

[5]Note that the notation $WCET_{\tau_j}$ has different meaning with the one used in Chapter 3. Here, $WCET_{\tau_j}$ is predetermined according to its execution order retrieved from the *basic cache-conscious schedule*.

- **The cost for committing outputs of τ_j at the end of its execution, denoted as $o_{\tau_j}^{WB\_flush}$ (if required)** , which is equal to the upperbound of the cost for fulling the write buffer to the shared memory (U_{WB_flush}). As mentioned in Section 4.6.1.2, $U_{WB\_flush} = 80$
- **The delay to all memory requests of τ_j by shared bus contention, denoted as** $o_{\tau_j}^{contention}$, which is retrieved from the solution file of Gurobi optimizer after solving ACILP.

The fraction of the overall overhead induced on the execution of every task in τ_{cp} by cache pollution to the length of the schedule graph, denoted as $oo^{cache\_pollution}$, is computed as:

$$oo^{cache\_pollution} = \frac{\sum_{\tau_j \in \tau^{cp}} o_{\tau_j}^{cache\_pollution}}{sl^{sg}} \tag{4.13}$$

The fraction of the overall overhead induced on the execution of every task in τ_{cp} by the other practical issues to the length of the schedule graph (as well as the corresponding notation) are done in the same way. Furthermore, the fraction of the effective execution of tasks (i.e., ignoring all practical issues) to the length of the schedule graph is computed as $\frac{\sum_{\tau_j \in \tau^{cp}} WCET_{\tau_j}}{sl^{sg}}$.

Figure 4.13 shows the fraction of the overhead caused by different practical issues to the length of schedule graphs for all benchmarks in the study when scheduled on 2, 4, 8, 15 cores. The figure shows that the effect caused by cache pollution is negligible. It is expected since the *sched* function is quite simple so that its execution introduces very small impact on the contents of caches. Besides, since the execution of the *sched* function is short, the impact of the delay to the start time of tasks caused by the execution of the function on the length of schedules graph is small. Additionally, the overall overhead induced on the execution of every task in τ^{cp} by write buffer flushing is very small. The reason is that the upperbound of the cost for flushing the write buffer to the shared memory is inexpensive and communicating tasks are likely to be assigned to the same core to benefit from data reuse.

As compared to the aforementioned practical issues, shared bus contentions issue has the highest impact on the length of schedule graphs. The effect of shared bus contention tends to increase when the number of cores increases. It can be explained that when the number of cores increases the number of concurrent tasks tends to increase, which introduces more contentions to the execution of tasks.

4.6.2.3 Evaluation the performance of ACILP

We evaluate the performance of ACILP in both terms of the length of adapted cache-conscious schedules generated by using ACILP and the required time for solving ACILP. For the evaluation, we apply the double fixed-point algorithm proposed in [94] to produce adapted cache-conscious schedules with considering the contention model presented in Section 4.4.3. We then compare the length of adapted cache-conscious schedules generated

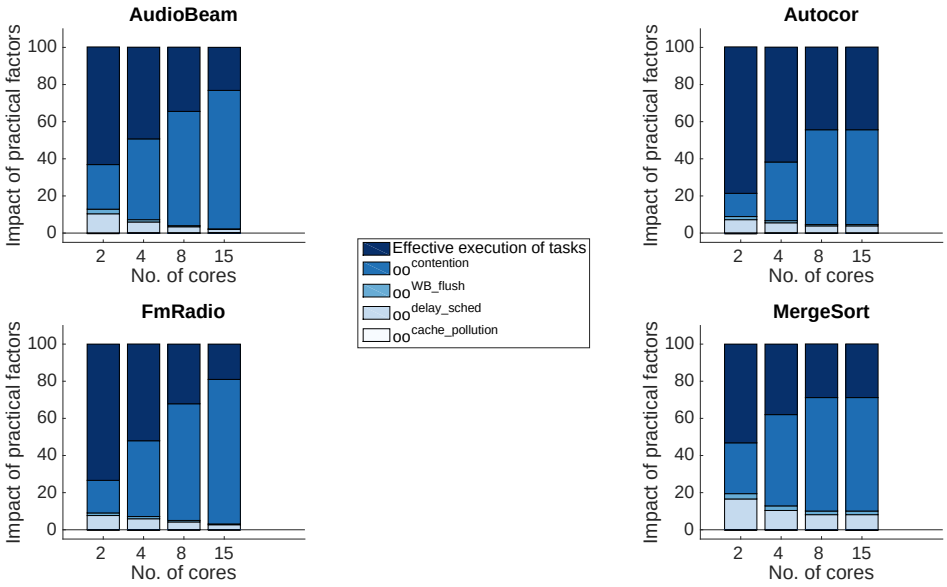


Figure 4.13: The fraction of the overall overhead by each practical issue to the length of schedule graphs

by using the double fixed-point algorithm and the solving time of the algorithm to those resulted by using ACILP.

Sharing the same interest with us, the double fixed-point algorithm proposed in [94] transforms a contention free static time-driven schedule to account for interference. The algorithm iteratively updates the WCET of tasks with contention and the trigger time of tasks accordingly with respect to their execution order and their precedence relation until those information are unchanged. The main difference between the double fixed-point algorithm and ACILP is that in the double fixed-point algorithm every task is forced to be triggered as soon as possible. In ACILP, according to our contention model and the property of fork-joint task graphs, the trigger time of tasks at the nearly end of schedules are considered to be optimized to limit the overhead induced on those tasks by shared bus contention.

All benchmarks in the study are schedules on 2, 4, 8, and 15 cores. Table 4.4 presents the length of adapted cache-conscious schedules and the required of time for generating the schedules by using ACILP and the double fixed-points algorithm.

The gain in term of schedule length reduction, which shows the benefit of ACILP as

Benchmark	No. of cores	Schedule length (cycle)		Scheduling time (s)		Gain (%)
		ACILP	[94]	ACILP	[94]	
AudioBeam	2	32278	32278	< 1	< 1	0
	4	**38868**	39251	< 1	1	0.98
	8	**53190**	53467	< 1	< 1	0.52
	15	78516	78516	4	< 1	0
Autocor	2	28579	28579	< 1	< 1	0
	4	27938	27938	< 1	< 1	0
	8	33330	33330	< 1	< 1	0
	15	33330	33330	< 1	< 1	0
FmRadio	2	125750	125750	< 1	1	0
	4	**100005**	100308	< 1	2	0.3
	8	**99709**	99997	4	3	0.29
	15	**129643**	130102	42	6	0.35
MergeSort	2	28002	28002	< 1	< 1	0
	4	32401	32401	< 1	< 1	0
	8	**41195**	41457	< 1	< 1	0.63
	15	**41195**	41457	< 1	< 1	0.63

Table 4.4: Performance comparison between ACILP and the double fixed-points algorithm proposed in [94]

compared to the double fixed-points algorithm is computed as:

$$gain = \frac{sl_{doublefixed-point} - sl_{ACILP}}{sl_{doublefixed-point}} * 100 \tag{4.14}$$

Table 4.4 shows that ACILP has slight gains in some cases, i.e., the highest gain is 0.98% when scheduling AudioBeam on 4 cores, and is never inferior to the double fixed-point algorithm. The result is expected since with our contention model ACILP only has chances to reduce the interference delay induced on tasks at the nearly end of schedules by optimizing their trigger time. However, the pessimism of contention model relieves the burden of the ILP solver in solving ACILP.

Regarding the required time for generating adapted cache-conscious schedules, both ACILP and double fixed-point algorithm produce schedules very fast. All solutions found by ACILP are optimal. For all benchmarks in the study, the longest solving time of ACILP is 42 seconds, whereas the longest scheduling time of the double fixed-points algorithm is 6 seconds when scheduling FmRadio benchmark on 15 cores.

4.7 Related work

In the literature most studies for multi-core hardwares focus on handling shared resources contentions (see [9] for the survey). However, in Section 4.3 we've shown that in the implementation of time-driven cache-conscious schedules on a multi-core hardware, especially on a compute cluster of Kalray MPPA-256 machine, besides shared resources contention other

practical issues such as cache pollution, the delay to the start time of tasks caused by the execution of the scheduling time-checking function, and the absence of hardware-implemented data cache coherence have to be paid great attention. To the best of our knowledge, we are the first ones designing and implementing time-driven cache-conscious schedules on a compute cluster of Kalray MPPA-256 machine.

The work proposed in [94] shares the same interest with us in term of integrating contentions on existed static time-driven contention-free schedules. As shown in Section 4.6.2.3 our proposed solution based on ILP formulation has slight gains in term of schedules length reduction as compared to the double-fixed points algorithm proposed in [94]. According to our study, shared resource contention is the most important issue needed to be handled for improving the quality of schedules for multi-core platforms.

Many scheduling approaches have been proposed to manage shared resources contention for multi-core platforms. Becker et al. [86] propose an execution model to completely avoid shared memory contention. In the work, they take the advantage of memory privatization features available in Kalray MPPA-256 machine to allocate the memory (includes code and data) of tasks and design a scheduling policy to schedule each phase of tasks (i.e., read, execution, write) such that the memory requests in each phase are free from contention. In [81], Rouxel et al. jointly perform shared resources contention modeling and tasks mapping/scheduling. In [82], Martinez et al. attempt to reduce contentions that exist in existing schedules by introducing slack time between the execution of pairs of tasks consecutively assigned to the same core, which limits the contention between concurrent tasks. As compared to the aforementioned works, our intent is not to tackle shared resource contentions, but rather to integrate the contention into existed contention-free schedules.

4.8 Summary

In this chapter, we addressed practical challenges arising when implementing time-driven cache-conscious schedules on a Kalray MPPA-256 compute cluster. We also proposed an ILP formulation for adapting time-driven cache-conscious schedules to the identified practical issues, as well as a strategy for generating applications' code. The experimental results showed the validity of the functional correctness and the timing correctness of our implementation. Besides, we showed the benefit in term of schedules length reduction of our proposed ILP formulation as compared to the double fixed-point algorithm proposed in [94]. The results also revealed that shared bus contention is the most impacting factor on the length of adapted cache-conscious schedules.

Chapter 5

Conclusion

Summary of contributions. In this PhD work, we first studied the problem of scheduling a single parallel application on a multi-core platform subjected to the effect of private caches. We aim at minimizing the schedule length of the application by leveraging cache reuse between tasks. Two cache-conscious scheduling techniques have been proposed to generate static time-driven partitioned non-preemptive schedules. Those techniques contain an optimal scheduling method which is based on ILP formulation, and a heuristic scheduling method which is based on list scheduling. Experimental results have shown that the proposed cache-conscious scheduling approaches produce better schedules (in term of schedules length reduction) than their cache-agnostic equivalent. Additionally, the proposed heuristic scheduling method shows a good trade-off between efficiency and the quality of generated schedules.

Secondly, we implemented time-driven cache-conscious schedules on the Kalray MPPA-256 machine, a clustered many-core platform. We presented our time-driven scheduler implementation, and pointed out the practical issues arising when implementing time-driven cache-conscious schedules on a cluster of the machine. Those issues include:

- cache pollution and the delay to the start time of tasks caused by the execution of the time-driven scheduler;
- shared bus contention;
- absence of hardware-implemented data cache coherence.

Besides, we proposed an ILP formulation to adapt time-driven cache-conscious schedules to the identified practical factors, such that precedence relations between tasks are still satisfied, and the length of adapted schedules are minimized. Moreover, we proposed a strategy for generating the code of applications to be executed on the machine according to adapted cache-conscious schedules. Experimental validation has shown the functional and the temporal correctness of our implementation. Furthermore, we showed that our proposed ILP formulation generates adapted cache-conscious schedules very fast. Additionally, we observed that shared bus contention is the most impacting factor.

Perspectives for future work. We see several opportunities to further improve/extend the PhD work. First of all, we can further benefit from cache reuse between tasks. A task can reuse the workloads of several tasks executed before it, but not necessarily of the task executed immediately before it. We believe that if those reuses are considered, the advantage of cache-conscious scheduling strategies in term of schedules length reduction can be further improved. We envision two approaches to exploit the cache reuse.

The first approach simply takes into account the reduction in the WCET of a task when executed after several tasks (i.e., for example, two tasks) rather than after only the task executed immediately before. Since the number of possible execution orders of tasks needed to be considered increases, this approach has to spend more effort on estimating context-sensitive WCETs of tasks, as well as finding schedules.

The second approach uses cache locking techniques [105, 106] in order to ensure that the useful workloads of tasks are still located in the cache until the task referring to them starts executing. That approach requires efforts in analyzing the memory footprints of tasks, as well as properly designing cache locking and scheduling strategies.

Additionally, extending our scheduling problem to deal with contentions on shared hardware resources is also an interesting direction. In this PhD work, we assume worst-case contentions occur between concurrent tasks, i.e., every memory access of a task is delayed if the task contends with concurrent tasks. We believe that the estimated worst-case contention delays can be tightened in several ways. First, we can consider the overlap time between concurrent tasks to estimate more precisely the number of memory accesses of tasks that are possibly delayed. Second, we can take into account the memory layout of tasks, i.e., at which banks the code and data of tasks are located, to limit the set of tasks which are possible contended against each other. In the Kalray MPPA-256 machine, tasks which have accesses to different memory banks do not contend against each other since different memory banks have different memory requests arbiters. Third, we can benefit from cache reuse between tasks to reduce the number of memory accesses of tasks since the number of memory accesses is equal to the number of caches misses. For that approach, we can simply integrate formulas that compute shared resources contentions delays into our cache-conscious ILP formulation.

Furthermore, in this PhD work, we leverage workloads reuse between tasks for the reduction of tasks' WCETs. We envision that workloads reuse between tasks can be also leveraged for the reduction of tasks' workloads loading time form shared memories to local memories. In PREM-like models, the entire workloads of tasks are loaded from the shared memory to the local memory (i.e., caches, or scratchpad memories) before the execution of tasks [84, 85]. If the workloads reuse between tasks are exploited, and tasks are scheduled properly, such that tasks sharing the same workloads are assigned to the same core and executed consecutively, the workloads loading time of the later executed task can be reduced much, thus reducing schedules length. In that approach, local memories allocation and task scheduling should be jointly performed to achieve optimal solutions.

Bibliography

[1] Marketsandmarkets.com, "Embedded systems market by hardware (mpu, mcu, application specific ic / application specific standard product, dsp, fpga, and memory), software (middleware and operating system), application, and geography - global forecast to 2023," 2017.

[2] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," ACM Comput. Surv., vol. 37, no. 3, pp. 195–237, 2005.

[3] D. Geer, "Industry trends: Chip makers turn to multicore processors," Computer, vol. 38, pp. 11–13, 2005.

[4] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in Proceedings of the Conference on Design, Automation & Test in Europe, DATE '14, pp. 97:1–97:6, 2014.

[5] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," IEEE Micro, pp. 15–31, 2007.

[6] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," IEEE Micro, pp. 34–46, 2016.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem: Overview of methods and survey of tools," ACM Trans. Embed. Comput. Syst., vol. 7, no. 3, pp. 36:1–36:53, 2008.

[8] V. Nélis, P. M. Yomsi, L. M. Pinho, J. C. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu, "The challenge of time-predictability in modern many-core architectures," in 14th International Workshop on Worst-Case Execution Time Analysis, vol. 39 of OpenAccess Series in Informatics (OASIcs), pp. 63–72, 2014.

[9] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in multicore hardware shared resources: Understanding of the state of

the art," in 14th International Workshop on Worst-Case Execution Time Analysis, OpenAccess Series in Informatics (OASIcs), pp. 31–42, 2014.

[10] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 365–376, 2010.

[11] J. H. Bahn, J. Yang, and N. Bagherzadeh, "Parallel FFT algorithms on network-on-chips," in Fifth International Conference on Information Technology: New Generations (ITNG 2008), pp. 1087–1093, 2008.

[12] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," Computer, vol. 21, pp. 10–19, 1988.

[13] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer Publishing Company, Incorporated, 3rd ed., 2011.

[14] 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," 2007.

[15] N. Holsti, "Analysing switch-case tables by partial evaluation," in 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007, 2007.

[16] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in Proceedings of the Second International Symposium on Programming, pp. 106–130, 1976.

[17] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, pp. 456–461, 1995.

[18] AbsInt Angewandte Informatik GmbH, "ait wcet analyzers."

[19] Tidorum Ltd, "Bound-t time and stack analyser."

[20] 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), pp. 35–46, 2010.

[21] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in 17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017), 2017.

[22] Rapita Systems Ltd, "Rapitime white paper - worst-case execution time analysis," 2008.

[23] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," ACM Computing Surveys, vol. 31, no. 4, pp. 406–471, 1999.

[24] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," ACM Computing Surveys, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[25] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. P. amd M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," IEEE Trans. Comput., vol. 47, pp. 700–713, 1998.

[26] H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemptions," ACM Trans. Embed. Comput. Syst., vol. 10, pp. 1–34, 2011.

[27] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," IEEE Trans. on Indus. Infor., vol. 9, pp. 3–15, 2013.

[28] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Is semi-partitioned scheduling practical?," in Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems, pp. 125–135, 2011.

[29] I. Ahmad and A. Ghafoor, "Semi-distributed load balancing for massively parallel multicomputer systems," IEEE Trans. Softw. Eng., vol. 17, pp. 987–1004, 1991.

[30] R. R. Schaller, "Moore's law: Past, present, and future," IEEE Spectr., vol. 6, pp. 52–59, 1997.

[31] R. Ramanathan, "Intel multi-core processors: Making the move to quad-core and beyond," tech. rep., Technology@Intel Magazine, 2006.

[32] J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," ACM Trans. Comput. Syst., vol. 4, pp. 273–298, 1986.

[33] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," SIGOPS Oper. Syst. Rev., vol. 5, pp. 2–11, 1996.

[34] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, Parallel Programming in OpenMP. Morgan Kaufmann, 2000.

[35] B. Lewis, D. J. Berg, and S. M. Press, Multithreaded Programming With PThreads. Prentice Hall PTR, 1998.

[36] S. Altmeyer, R. I. Davis, L. 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 and Networks Systems, pp. 129–138, 2015.

[37] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in RTSS, 2007.

[38] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha,

C. Silva, J. Sparsø, and A. Tocchi, "T-crest: Time-predictable multi-core architecture for embedded systems," J. Syst. Archit., pp. 449–471, 2015.

[39] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in Proceedings of the 44th Annual Design Automation Conference, pp. 264–265, 2007.

[40] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, and K. Keeton, "A case for intelligent ram," IEEE Micro, vol. 17, no. 2, pp. 34–44, 1997.

[41] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 75–84, 1991.

[42] M. Kowarschik and C. Wei, An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms, pp. 213–232. Springer Berlin Heidelberg, 2003.

[43] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., 2011.

[44] P. Panda, G. Patil, and B. Raveendran, "A survey on replacement strategies in cache memory for embedded systems," in Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), IEEE, pp. 12–17, 2016.

[45] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Syst. J., pp. 78–101, 1966.

[46] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," Commun. ACM, vol. 28, no. 2, pp. 202–208, 1985.

[47] J. Reineke, Caches in WCET analysis. PhD thesis, Univeristät des Saarlandes, 2008.

[48] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in Proceedings of the 20th IEEE Real-Time Systems Symposium, pp. 12–21, 1999.

[49] C. Rochange, "An overview of approaches towards the timing analysability of parallel architecture," in Bringing Theory to Practice: Predictability and Performance in Embedded Systems, pp. 32–41, 2011.

[50] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, "Timing analysis of concurrent programs running on shared cache multi-cores," Real-time Systems, vol. 48, no. 6, pp. 638–680, 2012.

[51] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS, pp. 68–77, 2009.

[52] J. Yan and W. Zhang, "Wcet analysis for multi-core processors with shared l2 instruction caches," in Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 80–89, 2008.

[53] B. Lesage, D. Hardy, and I. Puaut, "Shared data caches conflicts reduction for wcet computation in multi-core architectures," in 18th International Conference on Real-Time and Network Systems, pp. 80–89, 2010.

[54] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multicore platforms," ACM Trans. Embed. Comput. Syst., vol. 13, pp. 1–29, 2014.

[55] M. Jacobs, S. Hahn, and S. Hack, "Wcet analysis for multi-core processors with shared buses and event-driven bus arbitration," Proceedings of the 23rd International Conference on Real Time and Networks Systems, pp. 193–202, 2015.

[56] T. Kelter and P. Marwedel, "Parallelism analysis: Precise WCET values for complex multi-core systems," Sci. Comput. Program., pp. 175–193, 2017.

[57] B. C. Ward, J. L. Herman, C. J. Chirstopher, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems, pp. 157–167, 2013.

[58] M. Caccamo, M. Cesatu, R. Pellizzoni, E. Betti, R. Dudko, and R. Mancuso, "Real-time cache management framework for multi-core architectures," in Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 45–54, 2013.

[59] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 99–108, 2011.

[60] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in 20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014, pp. 155–166, 2014.

[61] R. Pellizzoni and H. Yun, "Memory servers for multicore systems," in 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016, pp. 97–108, 2016.

[62] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," IEEE Trans. Computers, pp. 562–576, 2016.

[63] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin, "The multi-resource server for predictable execution on multi-core platforms," in 20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014, pp. 1–12, 2014.

[64] S. Chattopadhyay, A. Roychoudhury, and T. Mitra, "Modeling shared cache and bus in multi-cores for timing analysis," in Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems, SCOPES '10, pp. 6:1–6:10, 2010.

[65] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Static analysis of multi-core tdma resource arbitration delays," Real-Time Syst., vol. 50, no. 2, pp. 185–229, 2014.

[66] 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 Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS '11, pp. 269–279, 2011.

[67] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu, "WCET analysis of a parallel 3d multigrid solver executed on the MERASA multi-core," in 10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium, pp. 90–100, 2010.

[68] H. Ozaktas, C. Rochange, and P. Sainrat, "Automatic wcet analysis of real-time parallel applications," in 13th Workshop on Worst-Case Execution Time Analysis (WCET 2013), pp. 11–20, 2013.

[69] H. Ozaktas, C. Rochange, and P. Sainrat, "Minimizing the cost of synchronisations in the wcet of real-time parallel programs," in Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems, pp. 98–107, 2014.

[70] D. Potop-Butucaru and I. Puaut, "Integrated worst-case execution time estimation of multicore applications," in 13th International Workshop on Worst-Case Execution Time Analysis, vol. 30, pp. 21–31, 2013.

[71] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," IEEE Trans. Comput., vol. 21, pp. 137–146, 1972.

[72] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," IEEE Trans. Softw. Eng., vol. 19, pp. 139–154, 1993.

[73] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," IEEE Trans. Comput., vol. 33, no. 11, pp. 1023–1029, 1984.

[74] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," SIAM J. Comput., vol. 18, pp. 244–257, 1989.

[75] T. F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," IEEE Trans. Parallel Distrib. Syst., vol. 10, pp. 1179–1191, 1999.

[76] T. Carle, M. Djemal, D. Potop-Butucaru, R. de Simone, and Z. Zhang, "Static mapping of real-time applications onto massively parallel processor arrays," in Proceedings of the 2014 14th International Conference on Application of Concurrency to System Design, ACSD '14, pp. 112–121, 2014.

[77] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-core scheduling of data parallel applications using SMT solvers," in 17th Euromicro Conference on Digital System Design, DSD, pp. 615–622, 2014.

[78] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul, "Reducing the contention experienced by real-time core-to-i/o flows over a tilera-like network on chip," in 28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016, vol. 86-96, 2016.

[79] H. Ding, Y. Liang, and T. Mitra, "Shared cache aware task mapping for WCRT minimization," in 8th Asia and South Pacific Design Automation Conference, ASP-DAC, pp. 735–740, 2013.

[80] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in 21st Euromicro Conference on Real-Time Systems, pp. 194–204, 2009.

[81] B. Rouxel, S. Derrien, and I. Puaut, "Tightening contention delays while scheduling parallel applications on multi-core architectures," ACM Trans. Embed. Comput. Syst., vol. 16, pp. 164:1–164:20, 2017.

[82] S. Martinez, D. Hardy, and I. Puaut, "Quantifying wcet reduction of parallel applications by introducing slack time to limit resource contention," in Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017, pp. 188–197, 2017.

[83] 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, no. 6, pp. 681–715, 2012.

[84] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in Proceedings of the 23rd International Conference on Real Time and Networks Systems, pp. 87–96, 2015.

[85] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in 21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015, pp. 285–296, 2015.

[86] 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," in 28th Euromicro Conference on Real-Time Systems, ECRTS, pp. 14–24, 2016.

[87] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Mapping mixed-criticality applications on multi-core architectures," in Proceedings of the Conference on Design, Automation & Test in Europe, pp. 1–6, 2014.

[88] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in Proceedings of the Seventh ACM International Conference on Embedded Software, EMSOFT '09, pp. 245–254, 2009.

[89] T. Liu, Y. Zhao, M. Li, and C. J. Xue, "Task assignment with cache partitioning and locking for wcet minimization on mpsoc," in Proceedings of the 2010 39th International Conference on Parallel Processing, pp. 573–582, 2010.

[90] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-conscious offline real-time task scheduling for multi-core processors," in 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), pp. 14:1–14:22, 2017.

[91] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," in Journal of Parallel and Distributed Computing, vol. 59, pp. 381–422, 1999.

[92] V. Nélis, P. M. Yomsi, and L. M. Pinho, "The variability of application execution times on a multi-core platform," in 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), OpenAccess Series in Informatics (OASIcs), pp. 1–11, 2016.

[93] Gurobi Optimization, Inc., "Gurobi optimizer reference manual," 2015.

[94] 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 Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS '16, pp. 67–76, 2016.

[95] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in International Conference on Real Time and Networks Systems, RTNS '15, pp. 129–138, 2015.

[96] F. Nemer, H. Cassé, P. Sainrat, and A. Awada, "Improving the worst-case execution time accuracy by inter-task instruction cache analysis," in IEEE Second International Symposium on Industrial Embedded Systems, SIES, pp. 25–32, 2007.

[97] W. Puffitsch, E. Noulard, and C. Pagetti, "Off-line mapping of multi-rate dependent task sets to many-core platforms," Real-Time Systems, vol. 51, no. 5, pp. 526–565, 2015.

[98] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Mapping hard real-time applications on many-core processors," in Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS '16, pp. 235–244, ACM, 2016.

[99] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 37–47, 2016.

[100] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '06, pp. 401–410, 2006.

[101] B. C. Ward, A. Thekkilakattil, and J. H. Anderson, "Optimizing preemption-overhead accounting in multiprocessor real-time systems," in Proceedings of the 22Nd International Conference on Real-Time Networks and Systems, RTNS '14, pp. 235:235–235:243, 2014.

[102] G. Phavorin, P. Richard, J. Goossens, T. Chapeaux, and C. Maiza, "Scheduling with preemption delays: Anomalies and issues," in Proceedings of the 23rd International Conference on Real Time and Networks Systems, RTNS '15, pp. 109–118, 2015.

[103] C. Tessler and N. Fisher, "BUNDLE: real-time multi-threaded scheduling to reduce cache contention," in IEEE Real-Time Systems Symposium, RTSS, pp. 279–290, 2016.

[104] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 28, no. 7, pp. 966–978, 2009.

[105] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in Proceedings of the 23rd IEEE Real-Time Systems Symposium, pp. 114–123, 2002.

[106] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," in International Conference on Real-Time Networks and Systems (RTNS), pp. 1–10, 2006.

Publications of the authors

[1] Viet Anh Nguyen, Damien Hardy, Isabelle Puaut, **Scheduling of parallel applications on many-core architectures with caches: bridging the gap between WCET analysis and schedulability analysis.** 9th Junior Workshop on Real Time computing, in conjunction with RTNS 2015, Lille, France, November 2015 (**best junior paper award**).
[2] Viet Anh Nguyen, Damien Hardy, Isabelle Puaut. **Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors** . In 29th Euromicro Conference on Real-Time Systems (ECRTS), 2017.

List of Tables

List of Figures

Abstract

Nowadays, real-time applications are more compute-intensive as more functionalities are introduced. Multi-core platforms have been released to satisfy the computing demand while reducing the size, weight, and power requirements. The most significant challenge when deploying real-time systems on multi-core platforms is to guarantee the real-time constraints of hard real-time applications on such platforms. This is caused by interdependent problems, referred to as a chicken and egg situation, which is explained as follows. Due to the effect of multi-core hardware, such as local caches and shared hardware resources, the timing behavior of tasks are strongly influenced by their execution context (i.e., co-located tasks, concurrent tasks), which are determined by scheduling strategies. Symmetrically, scheduling algorithms require the Worst-Case Execution Time (WCET) of tasks as prior knowledge to determine their allocation and their execution order.

Most schedulability analysis techniques for multi-core architectures assume a single WCET per task, which is valid in all execution conditions. This assumption is too pessimistic for parallel applications running on multi-core architectures with local caches. In such architectures, the WCET of a task depends on the cache contents at the beginning of its execution, itself depending on the task that was executed before the task under study. In this thesis, we address the issue by proposing scheduling algorithms that take into account context-sensitive WCETs of tasks due to the effect of private caches.

We propose two scheduling techniques for multi-core architectures equipped with local caches. The two techniques schedule a parallel application modeled as a task graph, and generate a static partitioned non-preemptive schedule. We propose an optimal method, using an Integer Linear Programming (ILP) formulation, as well as a heuristic method based on list scheduling. Experimental results show that by taking into account the effect of private caches on tasks' WCETs, the length of generated schedules are significantly reduced as compared to schedules generated by cache-unaware scheduling methods.

Furthermore, we perform the implementation of time-driven cache-conscious schedules on the Kalray MPPA-256 machine, a clustered many-core platform. We first identify the practical challenges arising when implementing time-driven cache-conscious schedules on the machine, including cache pollution caused by the scheduler, shared bus contention, delay to the start time of tasks, and the absence of data cache coherence. We then propose our strategies including an ILP formulation for adapting cache-conscious schedules to the identified practical factors, and a method for generating the code of applications to be executed on the machine. Experimental validation shows the functional and the temporal correctness of our implementation. Additionally, shared bus contention is observed to be the most impacting factor on the length of adapted cache-conscious schedules.

Keywords: real-time scheduling, cache-conscious schedules, time-driven cache-conscious schedules implementation, ILP, list scheduling, multi-core architectures, Kalray MPPA-256

Résumé

Les temps avancent et les applications temps-réel deviennent de plus en plus gourmandes en ressources. Les plateformes multi-cœurs sont apparues dans le but de satisfaire les demandes des applications en ressources, tout en réduisant la taille, le poids, et la consommation énergétique. Le challenge le plus pertinent, lors du déploiement d'un système temps-réel sur une plateforme multi-cœur, est de garantir les contraintes temporelles des applications temps réel strict s'exécutant sur de telles plateformes. La difficulté de ce challenge provient d'une interdépendance entre les analyses de prédictabilité temporelle. Cette interdépendance peut être figurativement liée au problème philosophique de l'œuf et de la poule, et expliqué comme suit. L'un des pré-requis des algorithmes d'ordonnancement est le Pire Temps d'Exécution (PTE) des tâches pour déterminer leur placement et leur ordre d'exécution. Mais ce PTE est lui aussi influencé par les décisions de l'ordonnanceur qui va déterminer quelles sont les tâches co-localisées ou concurrentes propageant des effets sur les caches locaux et les ressources physiquement partagées et donc le PTE.

La plupart des méthodes d'analyse pour les architectures multi-cœurs supputent un seul PTE par tâche, lequel est valide pour toutes conditions d'exécutions confondues. Cette hypothèse est beaucoup trop pessimiste pour entrevoir un gain de performance sur des architectures dotées de caches locaux. Pour de telles architectures, le PTE d'une tâche est dépendant du contenu du cache au début de l'exécution de la dite tâche, qui est lui-même dépendant de la tâche exécutée avant et ainsi de suite. Dans cette thèse, nous proposons de prendre en compte des PTEs incluant les effets des caches privés sur le contexte d'éxecution de chaque tâche.

Nous proposons dans cette thèse deux techniques d'ordonnancement ciblant des architectures multi-cœurs équipées de caches locaux. Ces deux techniques ordonnancent une application parallèle modélisée par un graphe de tâches, et génèrent un planning statique partitionné et non-préemptif. Nous proposons une méthode optimale à base de Programmation Linéaire en Nombre Entier (PLNE), ainsi qu'une méthode de résolution par heuristique basée sur de l'ordonnancement par liste. Les résultats expérimentaux montrent que la prise en compte des effets des caches privés sur les PTE des tâches réduit significativement la longueur des ordonnancements générés, ce comparé à leur homologue ignorant les caches locaux.

Afin de parfaire les résultats ainsi obtenus, nous avons réalisé l'implémentation de nos ordonnancements dirigés par le temps et conscients du cache pour un déploiement sur une machine Kalray MPPA-256, une plateforme multi-cœur en grappes (clusters). En premier lieu, nous avons identifé les challenges réels survenant lors de ce type d'implémentation, tel que la pollution des caches, la contention induite par le partage du bus, les délais de lancement d'une tâche introduits par la présence de l'ordonnanceur, et l'absence de cohérence des caches de données. En second lieu, nous proposons des stratégies adaptées et incluant, dans la formulation PLNE, les contraintes matérielles ; ainsi qu'une méthode permettant de générer le code final de l'application. Enfin, l'évaluation expérimentale valide la correction fonctionnelle et temporelle de notre implémentation pendant laquelle nous avons pu observé le facteur le plus impactant la longeur de l'ordonnancement: la contention.

Mot clé: ordonnancement temps-réel, ordonnancements conscient du cache, implémentation de planning ordonnancé par le temps et conscient du cache, PLNE, ordonnancement

par liste, architectures multi-cœur, Kalray MPPA-256