# Scheduling of parallel applications on many-core architectures with caches: bridging the gap between WCET analysis and schedulability analysis

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

Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut
University of Rennes 1/IRISA, France
anh.nguyen@irisa.fr, damien.hardy@irisa.fr, isabelle.puaut@irisa.fr

## ABSTRACT

Estimating the worst-case execution time (WCET) of parallel applications running on many-core architectures is a significant challenge. Some approaches have been proposed, but they assume the mapping of parallel applications on cores already done. Unfortunately, on architectures with caches, task mapping requires a priori known WCETs for tasks, which in turn requires knowing task mapping (i.e., co-located tasks, co-running tasks) to have tight WCET bounds. Therefore, scheduling parallel applications and estimating their WCET introduce a chicken and egg situation. In this paper, we address this issue by developing an optimal integer linear programming formulation for solving the scheduling problem, whose objective is to minimize the WCET of a parallel application. Our proposed static partitioned non-preemptive mapping strategy addresses the effect of local caches to tighten the estimated WCET of the parallel application. We report preliminary results obtained on synthetic parallel applications.

## 1. INTRODUCTION

Many-core architectures have become mainstream in modern computing systems. Along with them, parallel programming frameworks have been developed to utilize the power of many-core architectures. However, employing many-core architectures in hard real-time system raises many concerns. One significant issue is to precisely estimate the worst-case execution time (WCET) of parallel applications. WCET estimation methods for many-core architecture must take into account not only program paths and architecture (as addressed in WCET estimation methods for uni-core architecture [6]) but also resource contentions (i.e., bus contention and shared memory contention [3]). Moreover, when analyzing the timing behavior of parallel applications, these methods also have to consider the application's properties (i.e., multitasking, inter-task communication and synchronization).

Some approaches reported promising results in estimating the WCET of parallel applications running on many-core architectures. Ozaktas et al. [4] combine the estimated worst-case stall times caused by inter-task synchronization with the tasks' estimated WCETs to compute their worst case response time. Then, the WCET of the parallel application is estimated as the worst case task's response time. In another way, Potop-Butucaru et al. [5] integrate code sections of tasks running on cores as well as communications between them to produce an unified control flow graph. Then, the classical implicit path enumeration technique (IPET [6]) is applied to estimate the WCET of the parallel application. These two methods assume the mapping of the parallel applications on cores a priori known. However, the mapping of the parallel application influences the worst-case response time of the tasks, and hence affects the WCET of the entire parallel application.

As an illustration, let us consider a parallel application containing three tasks *T1, T2,* and *T3* to be mapped onto on a two-core architecture with a private cache on each core. Let us assume that *T1* and *T2* access the same memory block $m$, and that *T1* and *T2* are independent from *T3*. Let us consider two mappings: (i) *T1* and *T2* are assigned to one core and *T2* runs after *T1*, while *T3* is assigned to the other core; (ii) *T1* is assigned to one core, while *T2* and *T3* are assigned to the other core and *T2* runs after *T3*. In the first case, *T2*'s access to block $m$ is a hit because $m$ was loaded by *T1*. Therefore, the WCET of the parallel application in the first case is smaller than in the second case. This small example shows that the WCET of the entire parallel application highly depends on the mapping of its tasks on the cores. This motivates the need for optimal scheduling/mapping of the parallel application to tighten its estimated WCET.

In the literature, many scheduling approaches for parallel applications running on many-core architectures have been proposed [1]. However, most of them consider tasks' WCETs as constant values. As explained above, tasks' WCETs highly depend on the mapping of the applications tasks, due to the effect of private caches. Therefore, scheduling a parallel application without considering the effect of private caches on tasks' WCETs is suboptimal. Consequently, scheduling a parallel application and estimating its WCET are interdependent problems and have to be jointly solved for getting tight estimated WCET of the parallel application.

Ding et al. [2] propose a task scheduling method that minimizes shared cache interferences to tighten estimated WCETs. Their approach is different with us since we consider the effect of private caches in the task scheduling process. Additionally, the communication cost between tasks, which varies depending on task mapping, is not taken into account in [2].

In this paper, we propose a static scheduling solution for an isolated parallel application running on a many-core architecture. Our proposed scheduler not only respects dependence constraints between tasks (i.e., communications and synchronizations) but also takes into account the effect of local caches on tasks' WCETs. We develop an optimal integer linear programming model for solving the task scheduling problem, whose objective is to minimize the WCET of the parallel application. To the best of our knowledge, we are

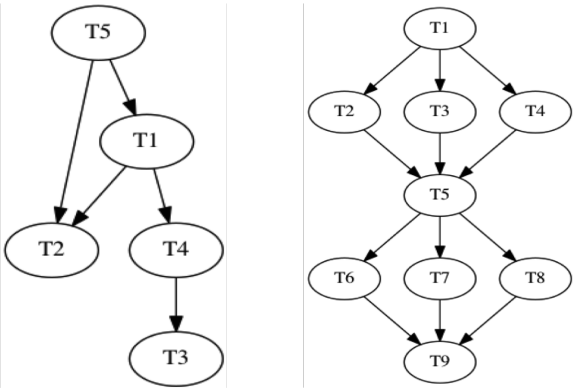



Figure 1: Arbitrary (left) and fork-join (right) task graphs

the first ones considering the effect of private caches on tasks' WCETs when scheduling parallel applications on many-core architectures. Our proposed scheduling approach is a partitioned non-preemptive scheduling approach: tasks are not allowed to be migrated and preempted, which prevents the system from suffering from hard-to predict migration and preemption costs (i.e., mainly caused by losses of working sets stored in local caches).

The paper is organized as follows. Section 2 introduces the application and architecture model, and presents the problem formulation. Section 3 presents an ILP formulation for solving the identified scheduling problem. Section 4 gives preliminary experimental results. Finally, we summarize the content of paper and give directions for future work.

## 2. MODEL AND PROBLEM FORMULATION

**Application model.** The parallel application is represented as a directed acyclic task graph (as illustrated in Fig. 1). Following the terminology used in [2], in these graphs, nodes represent tasks (i.e., pieces of code without communication or synchronization inside), and edges represent communications or synchronizations (precedence relations) between pairs of tasks. For each edge, the volume of transmitted data (zero for synchronizations) is known. For example, in the task graph illustrated in Fig. 1, the arrow from node *T5* to node *T1* means that *T1* is not authorized to execute before *T5* ends. We consider two instances of the task models: (1) arbitrary model, which does not constrain communications and synchronizations between tasks; (2) the popular fork-join model.

**Architecture model.** Our proposed scheduler applies to many-core architectures equipped with private caches, including the one depicted in Fig. 2. In the figure, cores are homogeneous and have a private cache. Our model can deal with any type of cache (instruction cache, data cache, and unified cache).

**Problem formulation.** Our scheduling method takes as input the task graph of an isolated parallel application and the following information: (a) the communication costs between tasks (when running on the same core, and when running on different cores); (b) tasks' WCETs when running alone as well as tasks' WCETs when running immediately after another task on the same core (to consider the effect of private caches). As a result the method produces a static partitioned non-preemptive schedule that determines on which cores each task is assigned, as well as a static

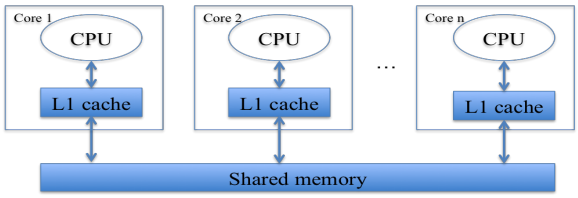



Figure 2: An example of many-core architecture with private caches

schedule on each core. The produced schedule minimizes the WCET of the parallel application.

## 3. ILP FORMULATION OF TASK SCHEDULING/MAPPING PROBLEM

Due to space limitations, only the main ILP constraints are presented hereafter. In the ILP formulation, we use uppercase letters for constants, and lowercase letters for variables to be calculated by the ILP solver. The solution is a set of variables that indicates static task mapping on cores and static task scheduling on each core.

**Base constraints for task mapping and scheduling.** We define a 0-1 variable $m_i^k$ to indicate whether task $t_i$ is assigned to core $k$ or not. Since the proposed scheduler is partitioned, each task is mapped to exactly one core, therefore:

$$\sum_{k \in K} m_i^k = 1. \tag{1}$$

In equation (1), $K$ represents the set of cores. Besides, we define a 0-1 variable $o_{j \to i}$ to determine whether task $t_i$ runs right after task $t_j$ or not, and a 0-1 variable $f_i^k$ to decide whether task $t_i$ is the first task running on core k or not. Since the produced schedule is non-preemptive, a task has at most one task running right after it, thus:

$$\sum_{i \in T - \{j\}} o_{j \to i} \leq 1. \tag{2}$$

In equation (2), $T$ represents the set of tasks. Additionally, one core has at most one first-running task, therefore, the following constraint is introduced:

$$\sum_{i \in T} f_i^k \leq 1. \tag{3}$$

**Further constraints for task mapping/scheduling.** The objective of the scheduling problem is to minimize the WCET of a parallel application. Let us represent the WCET of the parallel application by an integer variable $wcet_{pro}$, the objective function is described as:

$$\text{minimize} \quad wcet_{pro}. \tag{4}$$

The WCET of the parallel application has to be larger than or equal to the latest finish time of any of its tasks. If the latest finish time of task $t_i$ is represented by integer variable $lft_i$, the following constraint is introduced:

$$wcet_{pro} \geq lft_i, \forall t_i \in T. \tag{5}$$

In the following, we present the ILP constraints for computing the latest finish time of tasks and for computing the WCET of tasks by considering the effect of local caches.

**Constraints on tasks' latest finish times.**

The latest finish time of $t_i$ ($lft_i$) is the sum of its latest start time (denoted as $lst_i$) and its worst case execution time (denoted as $wcet_i$):

$$lft_i = lst_i + wcet_i. \quad (6)$$

In equation (6), $wcet_i$ is a variable introduced to integrate the variations of tasks' WCETs due to the effect of local caches (as explained later). The latest start time of $t_i$ ($lst_i$) is the sum of its latest ready time (denoted as $lrt_i$ which is calculated in considering its running order) as well as the worst communication delay with its predecessors (denoted as $wc_i$):

$$lst_i = lrt_i + wc_i. \quad (7)$$

In equation (7), the worst communication delay of $t_i$ with its predecessors ($wc_i$) is computed by considering the predecessors' allocations, i.e.,whether they are allocated on the same core or different cores (as explained later). The latest ready time of $t_i$ ($lrt_i$) is calculated by considering two cases: (1) $t_i$ is the first task running on a core; (2) $t_i$ runs right after another task on the same core.

In the first case, if $t_i$ has some predecessor, its latest ready time has to be equal to or larger than the latest finish time of its predecessors since $t_i$ cannot be executed before the completion of its predecessors. Otherwise, its latest ready time is greater than or equal to zero. Let's denote the set of predecessors of $t_i$ as $pred(t_i)$. The latest ready time of $t_i$ is expressed as:

$$\begin{aligned} lrt_i &\geq 0 \\ lrt_i &\geq lft_j, \forall t_j \in pred(t_i). \end{aligned} \quad (8)$$

In the second case, if $t_i$ is assigned to the same core as $t_j$ and runs immediately after $t_j$, then the latest ready time of $t_i$ is larger than or equal to the latest finish time of $t_j$, $lrt_i \geq lft_j$. Therefore, the latest ready time of $t_i$ in the second case is calculated according to the following constraint:

$$lrt_i \geq o_{j\to i} * lft_j. \quad (9)$$

Since (9) is a quadratic form, we linearize (9) as:

$$lrt_i \geq lft_j + (o_{j\to i} - 1) * M, \quad (10)$$

with $M$ the sum of all tasks' WCETs when running alone plus all communication costs between pairs of tasks when running on different cores, such that $M$ is guaranteed to be higher than the latest finish time of any tasks.

Let us denote the communication cost between $t_i$ and $t_j$ when they are placed on the same core and different cores as $C^s_{j\to i}$ and $C^d_{j\to i}$, respectively. The worst communication delay of $t_i$ ($wc_i$) with its predecessors is calculated as:

$$wc_i \geq s_{i,j} * C^s_{j\to i} + (1 - s_{i,j}) * C^d_{j\to i}, \forall j \in pred(t_i). \quad (11)$$

In equation (11), $s_{i,j}$ is a 0-1 variable to indicate whether two tasks $t_i$ and $t_j$ are assigned to the same core or not.

**Constraints on tasks' WCETs.**

To account for the variability of tasks' WCETs due to private caches, two cases have to be considered when calculating the WCET of a task $t_i$ (variable $wcet_i$): (1) $t_i$ is the first task running on a core; (2) $t_i$ runs right after another task. Let's denote by $WCET_i$ the WCET of $t_i$ when running alone, and $WCET_{j\to i}$ as the WCET of $t_i$ when running right after $t_j$ on the same core. In the first case, the WCET of $t_i$ is equal to its WCET when running alone, $wcet_i = WCET_i$. In the second case, the WCET of $t_i$ is equal to its WCET when running right after another task, $wcet_i = WCET_{j\to i}$. The WCET of $t_i$ is calculated as:

$$wcet_i = \sum_{j\in T-\{i\}} o_{j\to i} * WCET_{j\to i} + \sum_{k\in K} f^k_i * WCET_i. \quad (12)$$

# 4. EXPERIMENTAL RESULTS

Our scheduling approach was evaluated on synthetic task graphs of isolated parallel applications. The communication cost between two tasks $t_i$ and $t_j$ when running on the same core ($C^s_{j\to i}$) and different cores ($C^d_{j\to i}$) is generated randomly with the constraint $C^s_{j\to i} < C^d_{j\to i}$. The WCET of task $t_i$ when running right after task $t_j$ is calculated according to the following equation:

$$WCET_{j\to i} = WCET_i - r_{i,j} * WCET_i. \quad (13)$$

In equation (13), in order to address the effect of local caches on tasks' WCETs, $r_{i,j}$ ($0 \leq r_{i,j} < 1$) is randomly chosen according to the relation between $t_i$ and $t_j$; the range of $r_{i,j}$ in case $t_j$ is direct predecessor of $t_i$ is higher than that in case $t_j$ is indirect predecessor of $t_i$ and that in case $t_j$ and $t_i$ are independent.

In order to evaluate the performance of the proposed scheduler, we compare the WCET values obtained by our proposed scheduling method (S_CACHE), a random scheduling method (S_RAND) and scheduling method without taking into account the effect of private caches (S_NOCACHE). The smaller the WCET, the better the scheduling method. For S_RAND, we first randomly allocate tasks to cores, then tasks scheduling on each core is calculated such that communication/synchronization constraints are respected. We generate 10 schedules using S_RAND and report the best, average and the worst of the estimated WCETs. For S_NOCACHE, we apply the proposed ILP formulas for getting the schedule, but the WCET of a task when running right after another task on the same core is considered to be equal to its WCET when running alone ($WCET_{j\to i} = WCET_i$); when estimating the WCET of the entire parallel application, tasks' WCETs are re-evaluated by considering the effect of private caches. We use CPLEX version 12.5 as ILP solver.

For space considerations, we provide results for two examples of task graphs only (see Fig. 3). In the example of fork-join graph (illustrated in Fig.3), the communication cost between two tasks $t_1$ and $t_3$ when running on the same core is 247785 cycles, and when running on different cores is 376633 cycles. In our example, the range of $r_{i,j}$ in case $t_j$ is direct predecessor of $t_i$ is set to [0.6;0.9], the range of $r_{i,j}$ in case $t_j$ is indirect predecessor of $t_i$ is set to [0.2;0.5], and the range of $r_{i,j}$ in case $t_i$ and $t_j$ are independent is set to [0;0.1].

Fig. 4(b) gives the static schedule obtained by our method for our example of fork-join graph on a two-core architecture equipped with a private cache per core. In the schedule, up arrows denote ready time of tasks ($lrt_i$), while down arrows denote the finish time of tasks ($lft_i$). Colored boxes represent communications (in this specific case, there is no overlap between communications and computations, but overlaps may happen in the general case).

Fig. 4(a) compares estimated WCETs obtained when using the different scheduling methods for our example of arbitrary graph and fork-join graph. We normalize all results with respect to the WCET value obtained by S_CACHE. Our scheduling method generates schedules that lead to the smallest estimated WCET. Moreover, the WCET obtained

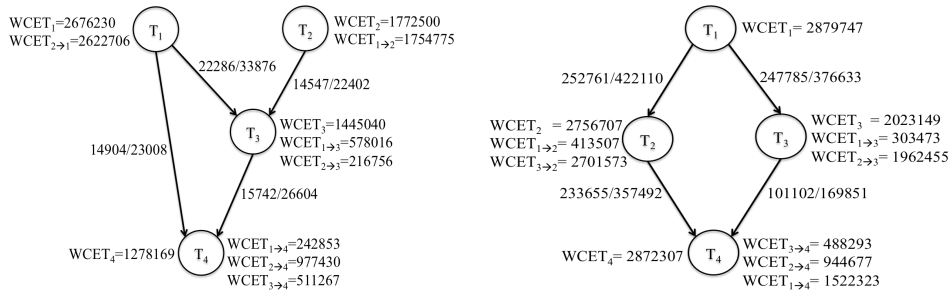
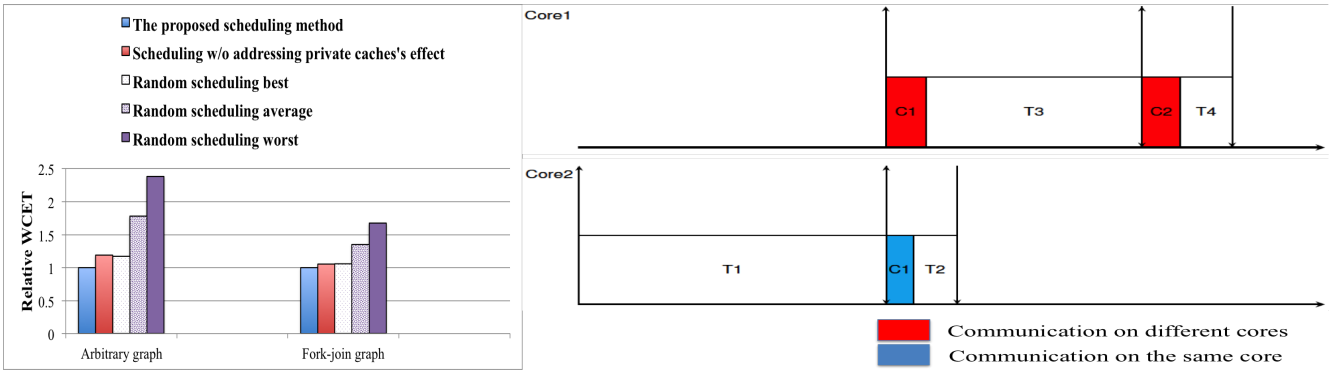



Figure 3: Our example of arbitrary graph and fork-join graph.




(a) (b)

Figure 4: (a) WCET comparison between different scheduling methods and (b) scheduling graph of tested fork-join graph.

by S_CACHE is less than 43% in our example of arbitrary graph and 26% in our example of fork-join graph when compared with the average results of S_RAND. The sizes of the test graphs is small, leading to a small solution space, explaining why S_RAND finds a schedule as good as S_CACHE (i.e., in our example of fork-join graph, the best WCET obtained by S_RAND is more than 5% when compared to the WCET obtained by S_CACHE). Additionally, compared to S_NOCACHE, we achieve 16% reduction in WCET in our example of arbitrary graph and 5% reduction in WCET in our example of fork-join graph, which shows the interest of considering the effect of private caches on tasks' WCETs in task scheduling. Furthermore, the runtime of our scheduling approach for these two graphs is very small (10 milliseconds on a 3GHz Intel Core i7 CPU with 16GB of RAM).

## 5. CONCLUSION

In this paper, we have developed an ILP formulation for finding an optimal schedule for a parallel application on a many-core architecture. Experimental results show the advantage of the proposed scheduler when considering the effect of private caches on tasks's WCETs. In the future, we will investigate the scalability of the proposed scheduling strategy by applying it to synthetic graphs with larger size, as well as to real applications. Additionally, we will address the effect of shared resource interferences (i.e., shared bus) in task scheduling.

## 6. ACKNOWLEDGMENTS

This work was supported by PIA project CAPACITES (Calcul Parallèlle pour Applications Critiques en Temps et Sûreté), reference P3425-146781). The authors would like to thank Benjamin Rouxel for comments on earlier versions of this paper.

## 7. REFERENCES

[1] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys*, 2011.

[2] H. Ding, Y. Liang, and T. Mitra. Shared cache aware task mapping for wcrt minimization. *Asia and south pacific - Design automation conference (ASP-DAC)*, 2013.

[3] 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. *International workshop on worst-case execution time analysis (WCET)*, 2014.

[4] H. Ozaktas, C. Rochange, and P. Sainrat. Minimizing the cost of synchronisations in the wcet of real-time parallel programs. *International workshop on software and compiler for embedded systems (SCOPES)*, 2014.

[5] D. Potop-Butucaru and I. Puaut. Integrated worst-case execution time estimation of multicore applications. *International workshop on worst-case execution time analysis (WCET)*, 2013.

[6] R. Whilhelm, 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 surveys of tools. *ACM transactions on embedded computing systems*, 2008.