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

Viet Anh Nguyen, Damien Hardy, Isabelle Puaut

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

Viet Anh Nguyen · Damien Hardy · Isabelle Puaut

Abstract Most schedulability analysis techniques for multi-core architectures assume a single Worst-Case Execution Time (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 instruction or data caches, for which the WCET of a task depends on the cache contents at the beginning of its execution, itself depending on the tasks that were executed immediately before the task under study.

In this paper, we propose two scheduling techniques for multi-core architectures equipped with local instruction and data caches. The two techniques schedule a parallel application modeled as a task graph, and generate a static partitioned non-preemptive schedule, that takes benefit of cache reuse between pairs of consecutive tasks. We propose an exact method, using an Integer Linear Programming (ILP) formulation, as well as a heuristic method based on list scheduling.

The efficiency of the techniques is demonstrated through an implementation of these cache-conscious schedules on a real multi-core hardware: a 16-core cluster of the Kalray MPPA-256, Andey generation. We point out implementation issues that arise when implementing the schedules on this particular platform. In addition, we propose strategies to adapt the schedules to the identified implementation factors.

An experimental evaluation reveals that our proposed scheduling methods significantly reduce the length of schedules as compared to cache-agnostic scheduling methods. Furthermore, our experiments show that among the identified implementation factors, shared bus contention has the most impact.

Viet Anh NGUYEN
Univ Rennes, Inria, CNRS, IRISA
(now with IRT Saint Exupéry, E-mail: viet-anh.nguyen@irt-saintexupery.com)

Damien HARDY
Univ Rennes, Inria, CNRS, IRISA
E-mail: damien.hardy@irisa.fr

Isabelle PUAUT
Univ Rennes, Inria, CNRS, IRISA
E-mail: isabelle.puaut@irisa.fr

1 Introduction

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 cars, airplanes, spacecraft, nuclear plants. With rising demand for applications that are increasingly compute-intensive and parallel, the traditional single-core architectures are no longer a suitable choice for deploying real-time systems. This limitation of single-core architectures is typically referred to as the *power-wall* [49]. To overcome this barrier, leading chip manufacturers have introduced *multi-core platforms*, in which multiple cores are integrated within a single chip. Multi-core platforms have been shown to improve energy-efficiency and performance-per-cost ratio vs. single-core models [15], mainly by exploiting thread-level parallelism. Examples of multi-core architectures include the Kalray MPPA-256 [13], the Tilera Tile CPU line [51], and the Intel Xeon Phi [44].

One of the important challenges of implementing safety-critical parallel applications on multi-core platforms is to guarantee that real-time constraints will be met under all the possible execution conditions of the applications. It is too difficult to precisely estimate the Worst-Case Execution Time (WCET) of tasks that execute on multiple cores simultaneously. Thus, the traditional WCET estimation methods were designed for single-core architectures [52], accounting for program execution paths and the characteristics of the core micro-architecture but ignoring multi-core factors such as caches and buses that may be shared between cores [14, 30]. Additionally, on architectures with local caches, the WCET of a task is affected by the contents of the cache at the beginning of execution, which in turn is affected by the task execution order. The net effect is that the WCET of any particular task can no longer be computed in isolation—it depends on the *execution context* of the task, including previous and concurrent tasks, which is ultimately defined by the task scheduler. While it is possible to continue using the traditional context-independent WCET, the results will be overly pessimistic. The following example illustrates the high degree of variation in WCET that can be caused solely by changes in the execution context.

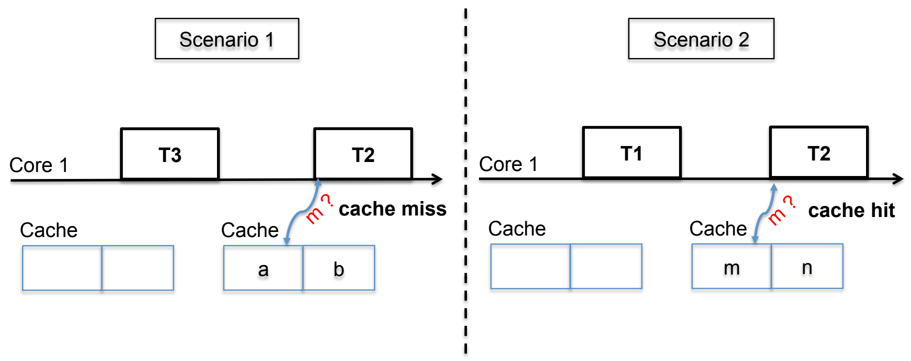


Fig. 1 The influence of scheduling strategies on the WCET of tasks

Motivating example. Let us consider an overly simplified parallel application made of three tasks, named T1, T2, and T3, executing on a dual-core platform, where each core is equipped with a unified private cache containing two lines. T1 and T2 access the same memory block, named m, whereas both the code and data of T3 are independent of tasks T1 and T2. Assuming that the cache is empty at the beginning of execution, we present two scenarios that demonstrate how the scheduling strategy can affect the WCET of each task. In the first scenario, illustrated in the left side of Figure 1, the request of T2 for the memory block m is a cache miss, since at request time, the memory block has not been loaded in the cache yet. In the second scenario, illustrated in the right side of Figure 1, the request of T2 for the memory block m is a cache hit since the memory block was already loaded into the cache by T1. In most machines a cache miss takes one or more orders of magnitude (in cycles) than a cache miss since it requires loading the requested data from main memory. Therefore, the worst-case execution time of T2 in the second scenario is much lower than its worst-case execution time in the first scenario.

This example shows that on architecture equipped with private caches, the execution order of tasks along with their assignment to specific cores can have an impact on the WCET of the tasks. Circularly, the task scheduler needs to know the WCET of each task so that it can determine the feasible sequence of tasks as assigned to specific cores. Therefore task scheduling and WCET estimation for multi-core platforms are inter-dependent—it is a chicken-and-egg problem. By making the scheduling strategies aware of the effect that execution context can have on WCET, we believe that the overall efficiency of parallel applications can be improved.

In this paper, we propose two *cache-aware* scheduling strategies that take advantage of cache reuse between pairs of consecutive tasks. Instead of assigning a single WCET to each task, we assign a set in which each WCET is associated with a task that could potentially precede it on the same core. This *context-sensitive WCET* takes into account the variation in execution time caused by the contents of the cache that remain after the preceding task completes. The objective of our proposed scheduling strategies is to minimize the schedule length (known as the *makespan*) by accounting for cache reuse. Throughout this paper we focus on a single parallel application, modeled as a task graph, in which each node represents a task and each edge represents a dependence relation between two tasks.

To further motivate our work, let us consider an example of scheduling an 8-input Fast Fourier Transform application [4] on a 2-core platform. As shown in Figure 2, in the task graph of the application, 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. In that example, we observe a reduction in WCET of 10.7% on average when taking into account the cache affinity between pairs of tasks that may execute consecutively on the same core. By using the method to be presented in Section 3 to generate the cache-conscious schedule for that application, we observe an 8% reduction in the schedule length vs. its cache-agnostic equivalent.

Once the cache-conscious schedules are generated, our next objective is to implement these schedules on real multi-core hardware using a Kalray MPPA-256 [13]. In the implementation stage, we first identify the implementation challenges that arise when deploying those schedules on the platform, such as shared bus contention, the effect of our time-driven scheduler itself, and the lack of hardware-

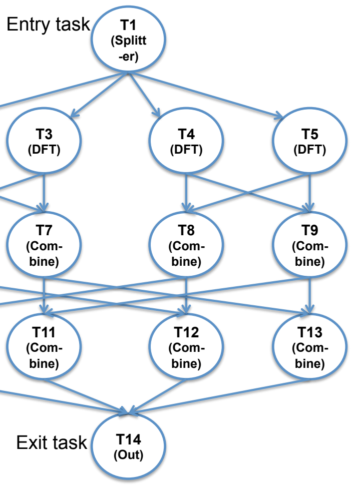


Fig. 2 Task graph of a parallel version of a 8-input Fast Fourier Transform (FFT) application [4]

implemented data cache coherence. These implementation factors require us to make adjustments to our cache-conscious schedules. We thus propose a strategy to adapt the cache-conscious schedules to the identified implementation factors, such that the precedence relations of tasks are still satisfied, and the length of the adapted schedules is minimized.

The main contributions of this paper are as follows:

– We argue the importance of accounting for the effect of private caches on the WCET, and validate our position with experimental results.
– We propose an ILP-based scheduling method and a heuristic scheduling method to statically find a time-driven, partitioned[1], non-preemptive schedule of a parallel application modeled as a directed acyclic graph.
– We provide experimental results showing that the proposed scheduling techniques result in shorter schedules than their cache-agnostic equivalents.
– We identify implementation issues that arise when implementing cache-conscious schedules on the Kalray MPPA-256, and propose strategies for overcoming them.
– We investigate the impact of various implementation factors on cache-conscious schedules.

The rest of this paper is organized as follows. Section 2 gives the overview of our hardware target, and describes the abstract model of the hardware platform, as well as the task model used in cache-conscious schedule generation. Section 3 introduces two cache-conscious scheduling techniques: one based on an Integer Linear Programming (ILP) formulation and a heuristic based on list scheduling. The implementation of cache-conscious schedules on the Kalray MPPA-256 is presented in Section 4, where we describe the execution conditions of an application on the platform and propose our time-driven scheduler implementation. We also identify

[1] Note that although designed for multi-core platforms, our proposed techniques can also be used on a single core to account for reuse among tasks executing on the same core.

the implementation challenges that arise when deploying cache-conscious schedules on the platform, and introduce our strategies for overcoming these issues. In Section 5 we present an experimental evaluation of our proposed scheduling methods and our schedule implementation. Section 6 surveys related work. Finally, we summarize the contents of the paper and provide directions for future work in Section 7.

2 System model

2.1 Hardware model

Our target architecture is the Kalray MPPA-256 [13], more precisely its first generation, named *Andey*. The Kalray MPPA-256 is a clustered many-core platform containing 288 cores which are organized into 16 compute clusters and 4 I/O clusters. These clusters are interconnected with a dual 2D-torus Network on Chip (NoC). In this study we generate cache-conscious schedules and implement them on a single compute cluster. The overview of a Kalray MPPA-256 compute cluster is given as follows.

2.1.1 Overview of a Kalray MPPA-256 compute cluster

A Kalray MPPA-256 compute cluster 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, while the 17th core, referred to as resource manager (RM), manages processor resources for the entire cluster. Additionally, a 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 data (supported by a DMA − Directed Memory Access − engine).

As announced in [13], every core in the Kalray MPPA-256 is fully timing-compositional [53]. 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 default write policy of the data cache is *write-through*. Data flushed from the data cache is not immediately committed to the shared memory—the flushed data is temporally held in a *write buffer*. Since there is no hardware-implemented data cache coherence between cores, the consistency of shared data between cores must be managed at the software level.

Tasks executing on different cores in the same cluster communicate through the shared memory (SMEM), which comprises 16 independent memory banks of 128 KB each, for a total capacity of 2 MB. Each memory bank is associated with a dedicated request arbiter that serves 12 bus masters: the D-NoC Rx interface, the D-NoC Tx interface, the DSU, the RM core, and 8 PE pairs. Each bus master has private paths connected to the 16 memory bank arbiters. The arbitration of memory requests to SMEM's banks is performed in 3 stages, depicted in Figure 3. The first two stages use a round-robin (RR) arbitration scheme. The first stage arbitrates between memory requests from the instruction cache (IC) and the data cache (DC) of each PE in a pair. In the second stage, the requests issued from each PE pair compete against those issued from other PE pairs, the D-NoC Tx, the DSU, and the RM. Finally, at the third stage, the requests compete against

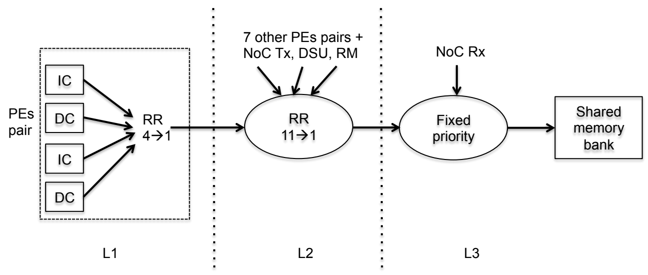


Fig. 3 SMEM memory request flow [13]

those coming from D-NoC Rx under static-priority arbitration, where the requests from D-NoC Rx always have higher priority.

2.1.2 Abstract model of a Kalray MPPA-256 compute cluster

To improve the generality of our cache-conscious scheduling techniques and make them usable on other architectures, we focus on a hardware model that abstracts away as many architectural details of the Kalray MPPA-256 as possible. This abstract model of a Kalray MPPA-256 compute cluster is illustrated in Figure 4. All cores are homogeneous, and each core is equipped with a private instruction cache and a private data cache. Tasks executing on different cores communicate through the shared memory.

Furthermore, we assume that:

- Tasks access the shared bus without contention;
- There is no cost for triggering a task at any specific instant of time;

The overheads for bus contention and task triggering, as well as other hardware-related overheads will be addressed in the implementation stage, to be presented in Section 4.

2.2 Task model

We model an application as a Directed Acyclic Graph (DAG) [23], as illustrated in Figure 2. A node in the DAG represents a task, and an edge represents a

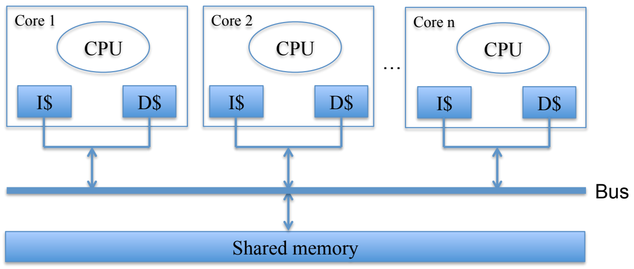


Fig. 4 Abstract model of a Kalray-MPPA 256 compute cluster

precedence relation between the source and target tasks, and may also indicate a transfer of information between them. A task can start executing only when all its direct predecessors have finished executing, 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 the edges (possibly zero) 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 the 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 set of WCET values for a task contains: *(i)* its most pessimistic WCET value, noted $WCET_{\tau_j}$, observed when there is no reuse of cache contents loaded by the task executed before on the same core; *(ii)* its WCET when the task reuses data and/or instruction cache contents from a directly preceding task on the same core. For example, the symbol $WCET_{\tau_i \rightarrow \tau_j}$ represents the WCET of task τ_j when τ_j reuses data and/or instruction cache contents from a directly preceding task τ_i on the same core. Note that the definition of the WCET of a task in this paper differs from the traditional definition, where the WCET of a task is simply the upper bound of its execution time in isolation. In contrast, our definition of a task's WCET is the upper bound of its execution times when taking into account the cache contents left by the task executed immediately before. Also note that for a task τ_j to benefit from cache reuse from a task τ_i, τ_i has to be scheduled before τ_j on the same core (with no task scheduled in between). However, the end time of τ_i needs not coincide with the start time of τ_j. The scheduling algorithm may insert idle time between τ_i and τ_j, for example to respect dependencies between tasks, while still taking benefit of cache reuse.

3 Cache-conscious scheduling algorithms

Our proposed scheduling methods take as inputs (a) the number of cores available, and (b) the DAG of a single parallel application decorated with context-sensitive WCETs information for each task. The result is a time-driven, partitioned, non-preemptive schedule of the application. More precisely, the schedule for each core determines the start and finish times of all tasks assigned to the core. The objective is to find out schedules having the shortest possible length (also known as *makespan*). We introduce two scheduling techniques:

– An ILP formulation which is capable of finding an optimal application's schedule whose the length is minimized under the considered assumptions (see Section 3.1);
– A heuristic method based on list scheduling which is capable of finding a schedule quickly (see Section 3.2). The length of the schedule is usually very close to the optimal one, as demonstrated in our experiments in Section 5.

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 1 Notation used in the proposed scheduling methods

The notation we use to describe our scheduling methods is summarized in Table 1. The first block defines notation for the task graph. A task τ_i is a *direct* predecessor of a task τ_j if there is an edge from τ_i to τ_j in the task graph. A task τ_i is an *indirect* predecessor of a task τ_j if there is an edge from τ_i to τ_j in the transitive closure of the task graph. For instance, in the motivating example of Figure 2, T_1 is a direct predecessor of T_2 and an indirect predecessor of T_{14}. The second block defines integer constants using upper-case letters. Finally, the third block defines variables using lower-case letters.

3.1 Cache-conscious ILP formulation

In this section we present our formula for ILP in the context of cache-conscious scheduling, which we call CILP for "Cache-conscious ILP". Since cores are identical in our abstract model, the execution time of a task is not affected by the properties of the cores. Based on that observation, CILP focuses on constructing sequences of co-located tasks, which includes defining the start time and the finish time of each task in a sequence. Given these sequences, the assignment of tasks to cores is straightforward (each sequence is simply assigned to a core).

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

$$\text{minimize} \quad sl \tag{1}$$

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

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &\quad sl \geq ft_{\tau_j} \end{aligned} \tag{2}$$

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 $wcet_{\tau_j}$:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &ft_{\tau_j} = st_{\tau_j} + wcet_{\tau_j} \end{aligned} \tag{3}$$

In the above equation, variable $wcet_{\tau_j}$ models the variation in WCET of a task caused by private caches, and is computed as follows:

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &wcet_{\tau_j} = f_{\tau_j} * WCET_{\tau_j} + \sum_{\tau_i \in \ nSucc(\tau_j)} o_{\tau_i \to \tau_j} * WCET_{\tau_i \to \tau_j} \end{aligned} \tag{4}$$

The multiplicative term on the left corresponds to the case where task τ_j is the first task running on a core ($f_{\tau_j} = 1$). The summation term on the right 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 of the binary variables among f_{τ_j} and $o_{\tau_i \to \tau_j}$ will be set by the ILP solver, such that exactly one of these WCET values will be assigned to τ_j. The assignment depends solely on the preceding task (if any).

Constraints on the start time of tasks. A task can be executed only when all of its direct predecessors have finished executing. In other words, the start time of a task must be greater than or equal to the finish times of all its direct predecessors.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in dPred(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} \ if \ dPred(\tau_j) \neq \emptyset \\ &st_{\tau_j} \geq 0 \ otherwise \end{aligned} \tag{5}$$

The final term in the above formula indicates that when a task has no predecessor, its start time must be greater than or equal to zero.

When there is a co-located task τ_i scheduled to precede τ_j, such that τ_j cannot start before τ_i finishes. In other words, the start time of τ_j must be greater 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.

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq o_{\tau_i \to \tau_j} * ft_{\tau_i} \end{aligned} \tag{6}$$

In order to linearize equation (6), we use classical big-M notation, which is expressed as:

$$\begin{aligned} &\forall \tau_j \in \tau, \forall \tau_i \in nSucc(\tau_j), \\ &st_{\tau_j} \geq ft_{\tau_i} + (o_{\tau_i \to \tau_j} - 1) * M \end{aligned} \tag{7}$$

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

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

Constraints on the execution order of tasks. A task is preceded by exactly one other task on the same core unless it is the first scheduled task:

$$\begin{gathered}\forall \tau_j \in \tau, \\ \sum_{\tau_i \in nSucc(\tau_j)} o_{\tau_i \rightarrow \tau_j} + f_{\tau_j} = 1\end{gathered} \tag{8}$$

The number of cores used is defined by the number of first scheduled tasks (number of variables f_{τ_j} equal to 1). Since the number of cores is K, the number of cores used has to be at most K:

$$\sum_{\tau_j \in \tau} f_{\tau_j} \leq K \tag{9}$$

Finally, a task has at most one co-located task scheduled immediately succeeding. This is expressed as:

$$\begin{gathered}\forall \tau_i \in \tau,\ if\ nPred(\tau_i) \neq \emptyset \\ \sum_{\tau_j \in nPred(\tau_i)} o_{\tau_i \rightarrow \tau_j} \leq 1\end{gathered} \tag{10}$$

An ILP solver produces results to the mapping/scheduling problem in the form of two sets of variables:

1. Task mapping is defined by variables f_{τ_j} and $o_{\tau_i \rightarrow \tau_j}$, which represent sets of co-located tasks along with the execution order within each set.
2. The static schedule for a core is defined by variables st_{τ_j} and ft_{τ_j}, which represent the start and finish time of the tasks assigned to that core.

3.2 Cache-conscious list scheduling method (CLS)

Finding an optimal solution to a partitioned, non-preemptive scheduling problem for a multi-core architecture is NP-hard [19], and does not scale with large number of tasks, as shown in our experiments (see Section 5). Therefore, we developed a heuristic scheduling method that efficiently produces schedules even for a large number of tasks. This method is based on *list scheduling* (see [24] for a survey of list scheduling methods).

Cache-conscious List Scheduling (CLS) begins with a list of tasks to be scheduled, scanning the list sequentially and scheduling each task without backtracking. For each task, CLS explores every potential core assignment that respects its precedence constraints. The core which allows the earliest finish time of the task is selected and the corresponding schedule is kept.

Task ordering in the list must follow a topological ordering (if τ_i is a direct predecessor or an indirect predecessor of τ_j, τ_i appears before τ_j in the list). To respect precedence constraints, the task sequence must follow a topological ordering. The list order is determined based on two classical metrics, both respecting topological order by construction. They both define for each task a *weight* (tw_{τ_j} for task τ_j), based on the task WCET, as defined below. The *bottom level* metric defines for task τ_j the longest path from τ_j to the exit task (τ_j included), accumulating task weights along the path:

$$\begin{aligned} bottom\_level_{exit} &= tw_{exit} \\ bottom\_level_{\tau_j} &= max(bottom\_level_{\tau_i} + tw_{\tau_j}), \forall \tau_i \in dSucc(\tau_j) \end{aligned} \tag{11}$$

The *top level* metric symmetrically defines for task τ_j the longest path from the entry task to τ_j (excluding τ_j itself):

$$\begin{aligned} top\_level_{entry} &= 0 \\ top\_level_{\tau_j} &= max(top\_level_{\tau_i} + tw_{\tau_i}), \forall \tau_i \in dPred(\tau_j) \end{aligned} \tag{12}$$

The use of the direct successor function *dSucc* in Equation 11 (respectively direct predecessor function *dPred* in Equation 12) guarantees the topological ordering.

What distinguishes CLS from existing scheduling techniques is its consideration of context-sensitive WCET coming from cache reuse. Since a task may have a different WCET for each of its potential predecessors, the weight of a task is defined to approximate the variability of its WCET. The weight tw_{τ_j} of a task τ_j is defined as:

$$tw_{\tau_j} = \frac{1}{K} * min_{\tau_i \in nSucc(\tau_j)}(WCET_{\tau_i \to \tau_j}) + (1 - \frac{1}{K}) * WCET_{\tau_j} \tag{13}$$

This formula integrates the potential for the WCET of task τ_j to be reduced, as well as the diminishing potential for WCET reduction as the number of cores increases.

As will be shown in Section 5.2, neither of the two metrics consistently outperforms the other for all task graphs. For this reason we kept both variations. For convenience, we define shorthand for specific forms of CLS:

- In CLS_BL, tasks are sorted according to the bottom level metric. In case of equality, the first tie-breaker is the top level metric, and remaining ties are broken arbitrarily by task identifier.
- In CLS_TL, tasks are sorted according to the top level metric, with ties broken first by the bottom level metric and then by task identifier.
- CLS indicates the better choice among the bottom level and top level metrics; i.e., the method giving the shorter schedule length for a particular task graph.
- NCLS is the cache-agnostic equivalent of CLS, and indicates the better choice among the bottom level and top level metrics for a system with no consideration of cache reuse. The weight of a task when using NCLS is its WCET ignoring cache reuse.

4 Implementation of cache-conscious schedules on Kalray MPPA-256

In Section 3, only the effects of local caches were considered in the generation of time-driven, cache-conscious schedules. In this section, we point out complications that arise in the implementation of the schedules on a Kalray MPPA-256 compute cluster. We also propose strategies to overcome these implementation factors.

```
void sched(uint64_t triggerTime){
  uint64_t curTimeStamp = 0;
  do
  {
   // get the timing information from the global cycle counter
   curTimeStamp = __k1_read_dsu_timestamp();
    // check the criterion for exiting from the loop
  } while(curTimeStamp < triggerTime);
}
```

Listing 1 The code of the *sched* function

4.1 Assumptions on execution conditions

To limit contention among tasks when accessing the SMEM of the Kalray MPPA-256, we impose the following constraints:

- The code and data of the application must fit into the SMEM of the compute cluster. The Resource Manager (RM) will load the application entirely onto the cluster before the application starts, and will have no further role during execution of the application.
- The application is executed in isolation on a compute cluster to avoid potential contention from the NoC.
- Debug mode is prohibited to avoid contention from the Debug Support Unit (DSU).

These constraints simplify contention on the shared bus, such that contention can only occur between application tasks running on different cores (PEs). In other words, the arbitration of memory requests to the SMEM's banks is simplified from the three stages described in Section 2.1.1 to just one stage, in which access is granted according to an ordinary round-robin policy.

4.2 Time-driven scheduler

The Kalray MPPA-256 provides a timestamp global cycle counter for timing synchronization between cores in the cluster. In order to trigger the execution of a task at a specific instant of time, we implement a *sched* function (see Listing 1) to be invoked just prior to the task. The *sched* function repeatedly checks the task trigger time against the global cycle counter, and starts the task when it detects that its trigger time has been reached.

4.3 Implementation challenges

Several complications arise in the implementation of time-driven, cache-conscious schedules on the Kalray MPPA-256:

4.3.1 Cache pollution caused by the scheduler

As described in Section 4.2, an instance of the scheduling function is interleaved between each pair of consecutive tasks on a given core. But since the data accessed by the scheduling function is unrelated to the task data, it effectively pollutes the cache, thereby attenuating the potential benefit of cache reuse between tasks. The net effect is an increase in the WCET which must be accounted for by our scheduling algorithm.

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 because each memory bank is equipped with only one requests arbiter. Therefore the delay induced by shared bus contention must be taken into account. Note that 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 the tasks *and* the *sched* function executing in parallel with the task.

4.3.3 Delay to the start time of tasks because of scheduler

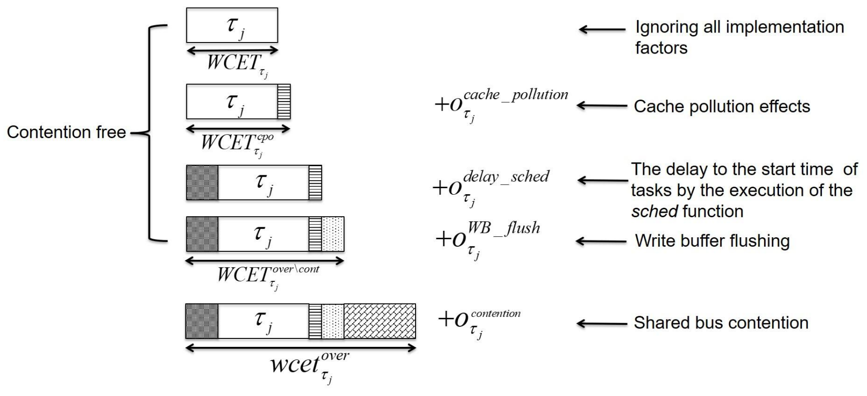
A task starts executing only when the *sched* function which precedes the task terminates. In the worst case, the execution of the task can be postponed by (at most) the amount of time that the *sched* function spent on its last iteration. As a result, there may be a gap between the trigger time of a task and the actual start time of the task (release jitter [27,37]). The delay in the start time of a task affects its finish time, thus requiring the trigger time of every task to be updated such that the precedence relation(s) are maintained.

4.3.4 Lack of hardware-implemented data cache coherence

The Kalray MPPA-256 does not provide hardware support for cache coherence between cores. In a compute cluster, tasks executing on different cores communicate through the SMEM, and data in transit from the cache to the SMEM is temporarily held in a write buffer before being committed to the SMEM. This delay may cause communication between pairs of tasks executing on different cores to fail.

Communication failures can occur in the case that a task is assigned to a different core than its predecessor and starts executing right after the termination of its predecessor. At the time that a task starts executing, the most recent data which the task intends to receive from its predecessor may not have been committed to the SMEM yet. As a result, the task may operate on obsolete data. In order to overcome this issue, all memory stores of the sending task must be committed to the SMEM before its termination. This can be done by inserting synchronization instructions at the end of each task. These instructions, which are available natively in the Kalray MPPA-256, include *__builtin_k1_wpurge()* and *__builtin_k1_fence()*. The former instruction requests the write buffer flush to the shared memory, while the latter waits for all data to be committed to the shared memory.

Additionally, communication failures can occur if the communication buffers between a pair of tasks are not aligned properly. Data misalignment may cause a task to accidentally acquire data that was stored by unrelated computations. This issue can be resolved by aligning all communication buffers on data cache line boundaries.



Contention free
τ_j
$WCET_{\tau_j}$
$WCET^{cpo}_{\tau_j}$
$WCET^{over\setminus cont}_{\tau_j}$
$wcet^{over}_{\tau_j}$
$+o^{cache\_pollution}_{\tau_j}$
$+o^{delay\_sched}_{\tau_j}$
$+o^{WB\_flush}_{\tau_j}$
$+o^{contention}_{\tau_j}$
Ignoring all implementation factors
Cache pollution effects
The delay to the start time of tasks by the execution of the sched function
Write buffer flushing
Shared bus contention

Fig. 5 The overheads induced on the execution of a task

4.4 Adapting our time-driven, cache-conscious schedules

The implementation factors mentioned in Section 4.3 introduce overheads on the execution of tasks. As illustrated in Figure 5, the overheads include:

- Reduction in cache reuse due to cache pollution.
- Delay in the start time of tasks.
- Overhead of flushing the write buffer to the shared memory.
- Shared bus contention.

Section 3 presented algorithms for generating a cache-conscious schedule without accounting for these implementation factors. Here we present a technique, also based on ILP, that updates the tasks trigger times of the initial schedule (shifts them in the future) to account for these overheads. These modifications of trigger times maintain the per-core execution order of tasks of the initial schedule, and ensure that precedence relation constraints between the tasks remain satisfied. We refer to the adjusted schedule as an *adapted cache-conscious schedule*. Figure 6 illustrates the adapted cache-conscious schedule with its adjusted WCET and task trigger time. The technique proposed to shift task trigger times is in the following called ACILP, for *adapted cache-conscious ILP*.

Notation used in the ILP formulation. Extending the set of symbols from Table 1 for managing the task graph, Table 2 introduces the following new symbols:

- The first section of the table defines notation that represents strictly the theoretical factors of cache-conscious scheduling (omitting reference to implementation factors). $B(\tau_j)$ represents the task executing on the same core and immediately preceding τ_j, $C(\tau_j)$ represents the core to which τ_j is assigned, $LT(c_j)$ represents the last task running on core c_j, and c represents the set of cores to which tasks are assigned.

- The second section defines predetermined parameters using upper case letters. The symbols $WCET_{\tau_j}^{over\backslash cont}$ and MR_{τ_j} represent the worst-case execution time of τ_j, and the maximum number of memory requests issued in the execution of τ_j, respectively. As illustrated in Figure 5, both $WCET_{\tau_j}^{over\backslash cont}$ and MR_{τ_j} account for overheads caused by all the implementation factors except for the delay from shared bus contention. The overhead estimation is detailed in Section 5.3.1. Additionally, the symbol $DMEM$ stands for the upper bound of memory access latency in the absence of contention (to be explained in Section 5.1).
- The third section defines variables using lower case letters. Similar to CILP, we use symbols such as sl and ft_{τ_j} for the schedule length and the finish time of τ_j in the *adapted cache-conscious schedule*, respectively. The symbol tt_{τ_j}, represents the trigger time of τ_j. The symbol tt_{τ_j} represents the same concept as st_{τ_j} (start time) in the initial schedule generation. We used a different symbol simply because the values of tt_{τ_j} and st_{τ_j} are not the same ($tt_{\tau_j} \geq st_{\tau_j}$ due to the consideration of implementation overheads). The symbol denoted as $wcet_{\tau_j}^{over}$ (as illustrated in Figure 5) stands for the worst-case execution time of τ_j accounting for shared bus contention. The remaining symbols $o_{\tau_j}^{contention}$, $\delta_{\tau_j}^{c_j}$, δ_{τ_j}, and $intf_{\tau_j}^{c_j}$ focus on contentions affecting τ_j.

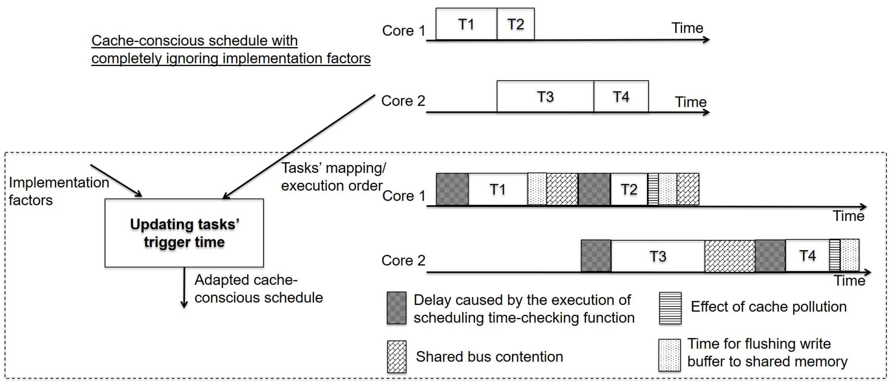

Cache-conscious schedule with completely ignoring implementation factors
Core 1
T1
T2
Time
Core 2
T3
T4
Time
Implementation factors
Tasks' mapping/ execution order
Updating tasks' trigger time
Adapted cache-conscious schedule
Core 1
T1
T2
Time
Core 2
T3
T4
Time
Delay caused by the execution of scheduling time-checking function
Effect of cache pollution
Shared bus contention
Time for flushing write buffer to shared memory

Fig. 6 An example of adapted cache-conscious schedule

Symbol	**Description**	**Data type**
$B(\tau_j)$	Set of co-located task immediately preceding τ_j	set
$C(\tau_j)$	Core to which τ_j is assigned	integer
c	The set of cores to which tasks are assigned	set
$LT(c_j)$	The last running task on core c_j	integer
$WCET_{\tau_j}^{over\backslash cont}$	The worst-case execution time of τ_j, adjusted for all overheads caused by the implementation issues except for shared bus contention	integer
MR_{τ_j}	The total number of memory requests issued by τ_j	integer
$DMEM$	The upper bound of memory access latency, excluding contention delay	integer
sl	The total duration of the adjusted 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
$wcet_{\tau_j}^{over}$	The worst-case response time of τ_j	integer
$\delta_{\tau_j}^{c_k}$	The maximum number of memory accesses issued from core c_k that could 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
$intf_{\tau_j}^{c_k}$	Indicates if the memory accesses from core c_k interfere with the execution of τ_j or not	binary

Table 2 Notations used in the ILP formulation in the *adapted stage*

ILP formulation to account for the implementation factors (ACILP). ACILP retains both the objective function and the constraints between schedule length and task finish time from the original CILP (presented in Section 3.1).

The finish time ft_{τ_j} of τ_j is the sum of its trigger time tt_{τ_j} and its worst-case execution time $wcet_{\tau_j}^{over}$, which now accounts for shared bus contention.

$$\begin{aligned} &\forall \tau_j \in \tau, \\ &ft_{\tau_j} = tt_{\tau_j} + wcet_{\tau_j}^{over} \end{aligned} \tag{14}$$

To complete the ILP formulation, we first present constraints for computing the trigger time of tasks, then we present constraints for computing the worst-case execution time of tasks that account for shared bus contention.

Constraints on task trigger time. If τ_j has any direct predecessors (notated as $\tau_i \in dPred(\tau_j)$), then τ_j cannot begin execution until after those tasks have finished. In order to maintain the precedence relations among tasks, the trigger time of the task must be greater than or equal to the finish time of all its direct predecessors. The same constraint is introduced if τ_j has a co-located task that immediately precedes it (notated as $\tau_i \in B(\tau_j)$).

$$\begin{aligned} \forall \tau_j \in \tau, dPred(\tau_j) \neq \emptyset \vee B(\tau_j) \neq \emptyset, \\ \forall \tau_i \in dPred(\tau_j) \vee \tau_i \in B(\tau_j), \\ tt_{\tau_j} \geq ft_{\tau_i} \end{aligned} \tag{15}$$

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

$$\begin{aligned} \forall \tau_j \in \tau, dPred(\tau_j) = \emptyset \land B(\tau_j) = \emptyset, \\ tt_{\tau_j} \geq 0. \end{aligned} \tag{16}$$

Constraints on the worst-case execution time to account for shared bus contention. As announced in [13], every core in Kalray MPPA-256 is fully timing compositional. The worst-case execution time of τ_j can safely account for the shared bus contention $wcet^{over}_{\tau_j}$ by computing the sum of:

- the worst-case execution time of τ_j, adjusted for overheads caused by all the implementation factors except shared bus contention (denoted as $WCET^{over\setminus cont}_{\tau_j}$);
- shared bus contention induced on τ_j (denoted as $o^{contention}_{\tau_j}$).

$$\begin{aligned} &\forall \tau_j \in \tau, \\ wcet^{over}_{\tau_j} &= WCET^{over\setminus cont}_{\tau_j} + o^{contention}_{\tau_j} \end{aligned} \tag{17}$$

Let us denote δ_{τ_j} the maximum number of memory requests (from all cores) that could delay the execution of τ_j, and $DMEM$ the upper bound of memory access latency in a contention free situation. The shared bus contention delay induced on τ_j is computed as:

$$o^{contention}_{\tau_j} = \delta_{\tau_j} * DMEM, \tag{18}$$

Let us denote as $\delta^{c_k}_{\tau_j}$ the maximum number of memory requests issued from core $c_k \neq C(\tau_j)$ that interfere with the execution of τ_j. Considering memory requests of all cores ($c_k \in c \land c_k \neq C(\tau_j)$) that delay the execution of τ_j, we compute δ_{τ_j} as follows:

$$\delta_{\tau_j} = \sum_{c_k \in c \land c_k \neq C(\tau_j)} \delta^{c_k}_{\tau_j} \tag{19}$$

To compute $\delta^{c_k}_{\tau_j}$, we need to determine whether the memory requests of τ_j compete against those issued from c_k or not (represented as a binary variable $intf^{c_k}_{\tau_j}$).

Since the *sched* function precedes every task, there will always be some code (task or scheduler) executing up to the point that the last task completes (i.e., either the execution of the *sched* function or the execution of a task). In order to ensure that all possible contentions are captured, we always account for potential interference between task τ_j and the operations on core c_k, except when τ_j is triggered after the termination of the last task running on c_k (represented by $LT(c_k)$). If τ_j and $LT(c_k)$ are constrained by a precedence relation, we can predetermine the value of $intf^{c_k}_{\tau_j}$, as follows:

- if τ_j is either a direct predecessor or an indirect predecessor of $LT(c_k)$, τ_j must start executing before $LT(c_k)$. In this case, $intf^{c_k}_{\tau_j} = 1$;
- if τ_j is either a direct successor or an indirect successor of $LT(c_k)$, τ_j has to start executing after the termination of LT_{c_k}. In this case, $intf^{c_k}_{\tau_j} = 0$.

If τ_j and $LT(c_k)$ do not have any precedence relation, the determination of $intf_{\tau_j}^{c_k}$ is presented as following condition:

$$intf_{\tau_j}^{c_k} = \begin{cases} 0 & \text{if} \quad tt_{\tau_j} \geq ft_{LT(c_k)} \\ 1 & \text{if} \quad tt_{\tau_j} < ft_{LT(c_k)} \end{cases} \tag{20}$$

To restate the above condition in classical big-M notation:

$$\begin{aligned} &\forall \tau_j \in \tau, \forall c_k \in c \wedge c_k \neq C(\tau_j), \tau_j \notin allPred(LT(c_k)) \wedge \tau_j \notin allSucc(LT(c_k)) \\ &\quad ft_{LT(c_k)} - tt_{\tau_j} \geq 1 - M * (1 - intf_{\tau_j}^{c_k}) \\ &\quad ft_{LT(c_k)} - tt_{\tau_j} \leq M * intf_{\tau_j}^{c_k} \end{aligned} \tag{21}$$

where M, is a constant[3] greater than any possible $ft_{LT(c_k)}$.

To reduce the computational effort of solving the ILP formulation, when memory requests issued by τ_j are determined to compete against those from c_k, $intf_{\tau_j}^{c_k} = 1$, we assume that all memory requests of τ_j are delayed according to $\delta_{\tau_j}^{c_k} = MR_{\tau_j}$. We formulate $\delta_{\tau_j}^{c_k}$ as follows:

$$\delta_{\tau_j}^{c_k} = intf_{\tau_j}^{c_k} * MR_{\tau_j} \tag{22}$$

5 Experimental evaluation

The experimental evaluation is divided into three parts.

- In the first part, we present experimental conditions. We describe properties of the benchmarks used in the experiment. Additionally, we present the method used for estimating WCETs and the number of cache misses of tasks. Furthermore, we describe experimental environment containing information of the ILP solver and the machine used for running the ILP solver and the proposed heuristic scheduling algorithm.
- In the second part, we evaluate the quality of generated cache-conscious schedules and required time for generating them. The objective is to evaluate the maximum schedule length reduction attainable by our proposed cache-conscious scheduling methods for any multi-core architectures equipped with local caches. Therefore, in this study, we ignore implementation factors, such as hardware sharing, the effects of the time-driven scheduler, and the lack of hardware-implemented data cache coherence. Those implementation issues will be addressed in the third part of the evaluation.
- In the third part, we first validate the functional and temporal correctness of applications when executing on a Kalray MPPA-256 compute cluster. We then quantify the impact of the overheads caused by different implementation factors on *adapted cache-conscious schedules*. Finally, we evaluate performance of our proposed ACILP formulation in both terms of quality of adapted cache-conscious schedules and required time for generating the schedules.

[3] For the experiments, M is the sum of the worst-case execution time of all tasks adjusted for worst-case shared bus contention. Specifically, the total number of memory requests that interfere with the execution of τ_j is equal to $MR_{\tau_j} * (|c| - 1)$, where $|c|$ is the number of cores to which tasks are assigned. This ensures M will be greater than the finish time of any task.

Benchmark (directed acyclic graph)	**No. of tasks**	**No. of Edges**	**Max. graph width**	**Ave. 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 Summary of the characteristics of StreamIt benchmarks in our case studies.

5.1 Experimental conditions

Benchmarks. In our experiments, we use 26 benchmarks of the StreamIt benchmark suite [48]. StreamIt is a programming environment that facilitates the programming of streaming applications, and was selected because it provides benchmarks with explicit parallelism and data transfers. We modified the StreamIt compilation toolchain (code generation step) to obtain task graphs compatible with our task model.

The characteristics of the task graphs are summarized in Table 3. In the table, the maximum width of a task graph is defined as the maximum number of tasks with the same rank[4]. The maximum width defines the maximum parallelism in the benchmark. 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 4. Reported information is the code size for the entire application, the average code size per task, the standard deviation of code sizes (the higher the number, the higher the

[4] The rank of a task is defined as the longest path in terms 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 4 The size of code and communicated data for each benchmark (average μ and standard deviation σ).

variability of the code sizes of tasks in the application), and the average amount of data communicated between tasks.

WCET and the number of cache misses estimation. Many techniques exist for WCET analysis [52] and could be used in our study to estimate WCETs and the gains resulting from cache reuse. At the moment doing the experiment, there is no publicly available WCET analysis tool for the Kalray MPPA-256. Furthermore, WCET estimation is not at the core of our scheduling methods. Therefore, we obtain WCET values by using measurements on a compute cluster of the Kalray MPPA-256. Measurements were performed on one core of the platform, with no activity on the other cores, providing fixed inputs for each task. The execution time of a task is retrieved using the platform's global cycle counter. The effect 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 (10 times in our experiments), the execution time is constant (the same behavior was reported in [29]).

Additionally, in order to record the number of cache misses, we use two performance counters supported by the Kalray MPPA-256. One counts the number of instruction cache misses, the other one counts the number of data cache misses.

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

Experimental environment. We use *Gurobi optimizer* version 6.5 [17] for solving our proposed ILP formulations. 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.

5.2 Evaluation the performance of cache-conscious schedules generation

5.2.1 Context-sensitive WCET information

For each task, we record its execution time when not reusing cache contents, as well as its execution time when executed after any *possible* other task[5]. Note that the way the benefit of cache reuse is evaluated also captures other (minor) hardware effects such as pipeline effects.

Table 5 summarizes the statistical numbers of obtained execution times. This table shows the average and standard deviation of tasks' WCET when having no

[5] A task τ_j can possibly execute after another task τ_i if the sequence $< \tau_i, \tau_j >$ may exist in a valid schedule regarding precedence constraints between tasks. This means that τ_i is neither a direct nor indirect successor of τ_j in the task graph, i.e. $\tau_i \in nSucc(\tau_j)$ according to the notation introduced in Table 1

cache reuse (the higher the standard deviation, the higher the variability of the WCET of tasks in the application). 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:

$$r_{\tau_j} = 100 * \frac{\sum_{\tau_i \in nSucc(\tau_j)} \dfrac{WCET_{\tau_j} - WCET_{\tau_i \to \tau_j}}{WCET_{\tau_j}}}{|nSucc(\tau_j)|} \tag{23}$$

Equation 23 considers for a task τ_j all tasks τ_i that may be scheduled immediately before τ_j by the scheduler, i.e. all tasks in set $nSucc(\tau_j)$ (not successors of τ_j in the task graph), as defined in Table 1.

We observed that tasks with small WCET have important WCET reductions when considering cache reuse. On the other hand, they have low impact on schedule length because of their low WCET. Consequently, we weighted each WCET reduction by its WCET ignoring cache reuse, yielding to the following definition of weighted average reduction:

$$wr = \frac{\sum_{\tau_j \in \tau}(r_{\tau_j} * WCET_{\tau_j})}{\sum_{\tau_j \in \tau} WCET_{\tau_j}} \tag{24}$$

Regarding the cost of estimating context-sensitive WCETs of tasks due to cache reuse, we observed that the worst profiling time is 10 minutes for the most complex benchmark structure *IDCT_2D_reference_fine*. The benchmark contains 548 tasks, and 219238 pairs of tasks that may be executed one after the other (with respect to precedence constraints).

5.2.2 Benefits of cache-conscious scheduling

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:

$$gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100. \tag{25}$$

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.

Results are reported in Figure 7 and Figure 8 for a 16 cores architecture. In Figure 7, 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. As expected, the higher reductions are obtained for the benchmarks with the higher weighted average WCET reduction as defined in Table 5.

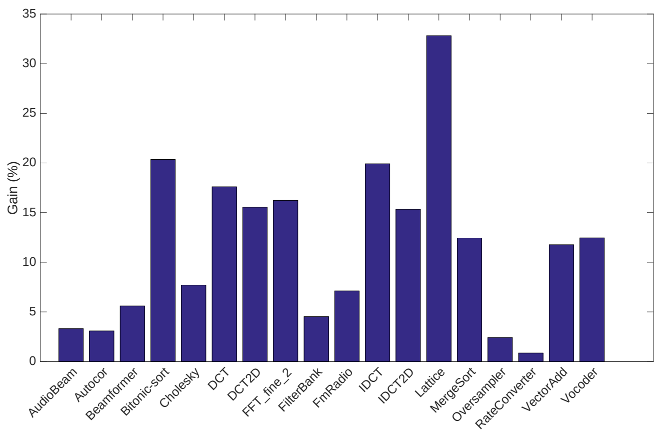


Fig. 7 Gain of CILP as compared to NCILP ($gain = \frac{sl_{NCILP} - sl_{CILP}}{sl_{NCILP}} * 100$) on a 16 cores system

5.2.3 Comparison of exact (CILP) and heuristic (CLS) scheduling techniques

We compare CILP and CLS according to two metrics: quality of generated schedules, estimated through their length (the shorter the better) and time required to generate the schedules. All results are obtained on a 16 cores system.

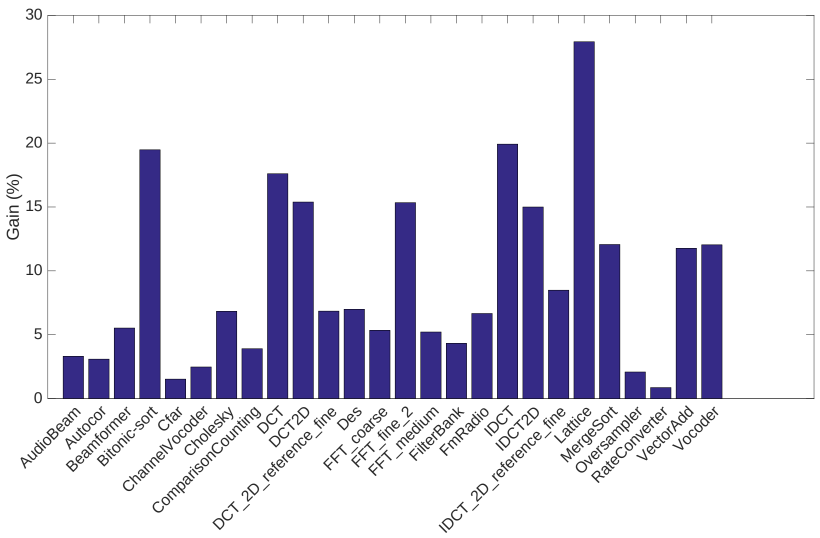


Fig. 8 Gain of CLS as compared to NCLS ($gain = \frac{sl_{NCLS} - sl_{CLS}}{sl_{NCLS}} * 100$) on a 16 cores system

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 6 Comparison of CILP and CLS (schedule length and run time of schedules generation in seconds)

Table 6 gives the length of generated schedules (sl_{CILP} and sl_{CLS}), the run time of schedule generation (in seconds) and the gap (in percent) between the schedules length, computed by the following formula:

$$gap = \frac{sl_{CLS} - sl_{CILP}}{sl_{CILP}} * 100. \qquad (26)$$

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 efficiency and 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 *DES*, *ChannelVocoder*, and *IDCT_2D_reference_fine* CILP is unable to find the optimal solution in 20 hours, which is expected because the problem of finding the optimal solution to a partitioned, non-preemptive scheduling problem on a multi-core architecture is NP-hard [19, 32]. When CILP finds the optimal solution, the gap between CILP and CLS is very small (0.7% on average).

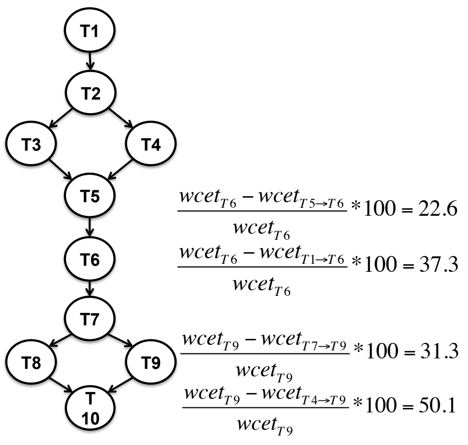

T1
T2
T3
T4
T5
T6
T7
T8
T9
T 10
$\frac{wcet_{T6} - wcet_{T5\to T6}}{wcet_{T6}} * 100 = 22.6$
$\frac{wcet_{T6} - wcet_{T1\to T6}}{wcet_{T6}} * 100 = 37.3$
$\frac{wcet_{T9} - wcet_{T7\to T9}}{wcet_{T9}} * 100 = 31.3$
$\frac{wcet_{T9} - wcet_{T4\to T9}}{wcet_{T9}} * 100 = 50.1$

Fig. 9 The reuse pattern found in the *Lattice* benchmark

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 9) where reuse is higher between indirect predecessors than between direct predecessors. For example, the reduction of the WCET of T6 when executed directly after T1 on the same core[6] (37.3%) is higher than when executed directly after T5 on the same core (22.6%). Similarly, the reduction of the WCET of T9 when executed directly after T4 on the same core (50.1%) is higher than when executed directly after T7 on the same core (31.3%). For such an application, the static sorting of CLS never places indirect precedence-related tasks (for which the higher reuse occurs) contiguously in the list, and then does not fully exploit the cache reuse present in the application.

5.2.4 Impact of the number of cores on the gain of CLS against NCLS

We evaluate the gain in terms of schedule lengths of CLS against its cache-agnostic equivalent when varying the number of cores. The results are depicted in Figure 10 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 schedules 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 schedules length achieved by CLS against NCLS is most of the time significant.

[6] Note that executing T6 directly after T1 on the same core is not a violation of precedence constraints between tasks, provided that tasks T2 to T5 are assigned to another core, and there is an idle time between the end of T1 and the start of T6.

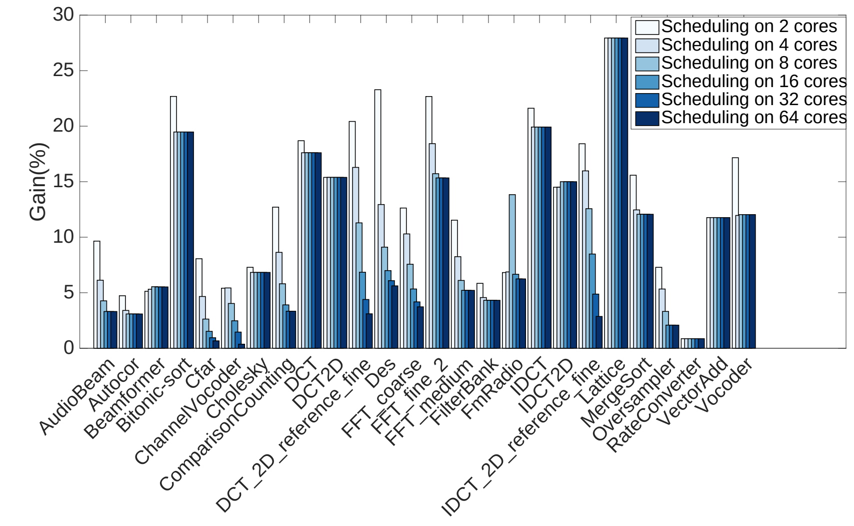


Fig. 10 Impact of the number of cores on the gain of CLS against NCLS

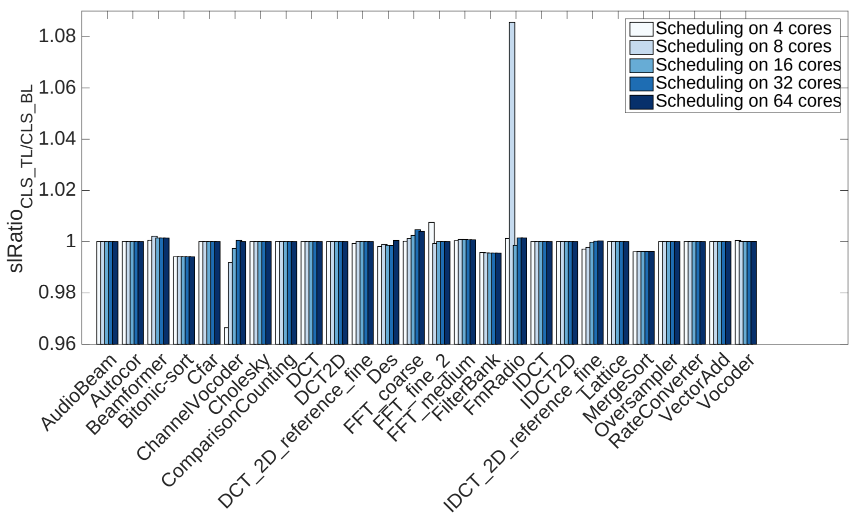


Fig. 11 Comparison of schedule lengths for CLS_TL and CLS_BL

5.2.5 Comparison of schedules length for CLS_TL and CLS_BL

We study the impact of the sorting technique of the list scheduling technique on quality of schedules. Figure 11 depicts the ratio of the length of the schedule generated by CLS_TL to that of CLS_BL as $slRatio_{CLS\_TL/CLS\_BL} = \frac{sl_{CLS\_TL}}{sl_{CLS\_BL}}$ for each benchmark. 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 length of schedules generated by CLS_TL and CLS_BL are most of the time very close to each other if not identical. The explanation is that both CLS_TL and CLS_BL scan tasks in some topological orders. With task graphs having a small width, there is a small number of possible topological orderings of tasks. With task graph having a large width, the WCETs of most tasks having the same rank are correlated since those tasks in the SteamIt benchmarks execute the same piece of code. Therefore, with task graphs having those properties different orderings of tasks results in slight changes in the final schedules length.

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 length 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 paper always used both and selected the best result obtained.

5.3 Evaluation of the cache-conscious schedules implementation

In the evaluation we use four benchmarks [7], named AudioBeam, AutoCor, FmRadio, and MergeSort.

5.3.1 Overheads and number of memory accesses estimation

The overhead induced on τ_j by cache pollution, noted as $o_{\tau_j}^{cache\_pollution}$, is computed by subtracting the worst-case execution time of τ_j when considering the execution of the *sched* function before the task ($WCET_{\tau_j}^{cpo}$) from its worst-case execution time when ignoring the execution of the function ($WCET_{\tau_j}$[8]).

$$o_{\tau_j}^{cache\_pollution} = WCET_{\tau_j}^{cpo} - WCET_{\tau_j} \qquad (27)$$

The $WCET_{\tau_j}^{cpo}$ is estimated according to the execution order of τ_j, which is available at the implementation stage. If τ_i is the task executing on the same core

[7] With those benchmarks we do not have to modify the code of tasks to have a communication buffer per pair of communicating tasks. It is very costly to modify the code of the other benchmarks for having the same property.

[8] Note that the symbol $WCET_{\tau_j}$ has different meaning with the one used in Section 3. Here, $WCET_{\tau_j}$ is predetermined according to its known execution order.

and immediately preceding τ_j, we record the execution time of τ_j when the task is executed according to the following order: $\tau_i \rightarrow sched(0) \rightarrow \tau_j$. Since during the execution of the *sched* function the contents of caches do not change after the first iteration of the function, we pass zero to the input of the function.

Regarding the delay to the start time of a task τ_j, noted as $o_{\tau_j}^{delay\_sched}$, we measure the WCET of one iteration of the *sched* function. For measuring the value, we pass zero to the input of the function, and execute the function in isolation. We observed that $o_{\tau_j}^{delay\_sched} = 258$ cycles. We also observed that the maximum number of cache misses of one iteration of the *sched* function is 10. Since the timestamp of the global cycle is stored at a specific address of the SMEM, one iteration of the function takes one more access to the SMEM to retrieve the information. Therefore, the maximum number of memory requests of one iteration of the *sched* function is 11.

The write buffer is 8-way fully associative (8 bytes per each way), the memory access granularity is 8 bytes, and the memory access latency for accessing 8 bytes in case contention free is 10 cycles [5]. The upper bound of the cost for flushing the write buffer to the SMEM when contention free, noted as $o_{\tau_j}^{WB\text{-}flush}$, is 80 cycles. Besides, the upper bound of the memory accesses for flushing the write buffer to the SMEM is 8.

Additionally, the overhead induced on τ_j due to shared bus contention, noted as $o_{\tau_j}^{contention}$, is a variable in the ACILP formulation. Therefore, its value can be retrieved from the solution file of the Gurobi optimizer (after solving ACILP).

Furthermore, the upper bound of memory access latency when contention free, noted as $DMEM$, is equal to the cost for loading an instruction cache line from the SMEM to the instruction cache. Since an instruction cache line contains 64 bytes and it takes 9 cycles with 8 bytes fetched on each consecutive cycle for accessing the SMEM when contention free, the cost for loading an instruction cache line from the SMEM to the instruction cache is 17 cycles.

5.3.2 Validation of the functional correctness and the timing correctness of the implementation

Functional correctness. For validating the functional correctness of the implementation, we compare the outputs produced by each benchmark when executing in sequential order, i.e., all tasks are executed on one core, and when executing in parallel, i.e., tasks are executed according to their mapping and their scheduling information given in the adapted cache-conscious schedule. We observed the same outputs when executing benchmarks in sequential and when executing in parallel.

Temporal correctness. For validating the temporal correctness of the implementation, we record the actual start time and the actual finish time of every task when executing on a Kalray MPPA-256 compute cluster according their mapping and their schedule. We observed that precedence constraints between tasks are satisfied.

5.3.3 Quantification of the impact of different implementation factors on adapted cache-conscious schedules

The impact of an implementation factor on the adapted cache-conscious schedule is reflected by the fraction of the overall overhead induced on tasks (caused by

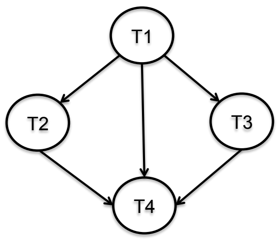


Fig. 12 The schedule graph constructed based on the scheduling information in the adapted cache-conscious schedule as depicted in Figure 6

the factor) to the schedule length. However, tasks are executed in parallel, so that in general the schedule length is not a linear combination of the execution time of a set of tasks. Therefore, for our quantification, instead of relying on the schedule length, we used the length of the longest path of the schedule, which is the accumulation of the execution time of tasks along the path.

In order to determine the longest path of the schedule, we construct a schedule graph based on the scheduling information of tasks in the schedule. In the schedule graph, each node represents a task. Two nodes are connected by an edge if the task represented by the sink node is triggered after the execution of the task represented by the source node. Figure 12 shows the schedule graph constructed based on the scheduling information of tasks in the adapted schedule which was illustrated in Figure 6. The weight of a node is the execution time of a task represented by the node, while the weight of every edge is zero. We use implicit-path enumeration technique (IPET) [25] to find the longest path of the schedule graph. We denote the set of tasks that lay on the longest path as τ^{cp}, and the length of the longest path as sl^{sg}.

The impact of cache pollution on the *adapted cache-conscious schedule*, noted as $oo^{cache\_pollution}$, is quantified as:

$$oo^{cache\_pollution} = \frac{\sum\limits_{\tau_j \in \tau^{cp}} o_{\tau_j}^{cache\_pollution}}{sl^{sg}} \tag{28}$$

The symbols and the quantification of the impact of the other implementation issues on the adapted schedule are done in the same way.

Furthermore, we compute the fraction of the effective execution of tasks (i.e., the execution time of tasks when completely ignoring all implementation issues) to the length of the schedule graph as $\frac{\sum\limits_{\tau_j \in \tau^{cp}} WCET_{\tau_j}}{sl^{sg}}$.

The impact of different implementation issues on the adapted cache-conscious schedules of all benchmarks in the study for 2, 4, 8, and 15 cores[9] is shown in Figure 13. The impact caused by cache pollution is negligible. It is expected since the *sched* function is quite simple so that its execution introduces very small noise on the caches contents. Besides, since the execution of the *sched* function is short, the delay to the start time of tasks due to the execution of the function is small.

[9] Only 15 out of the 16 cores of the cluster are used in our implementation on the Kalray MPPA, because one core is dedicated to the spawning of the tasks on the cluster cores.

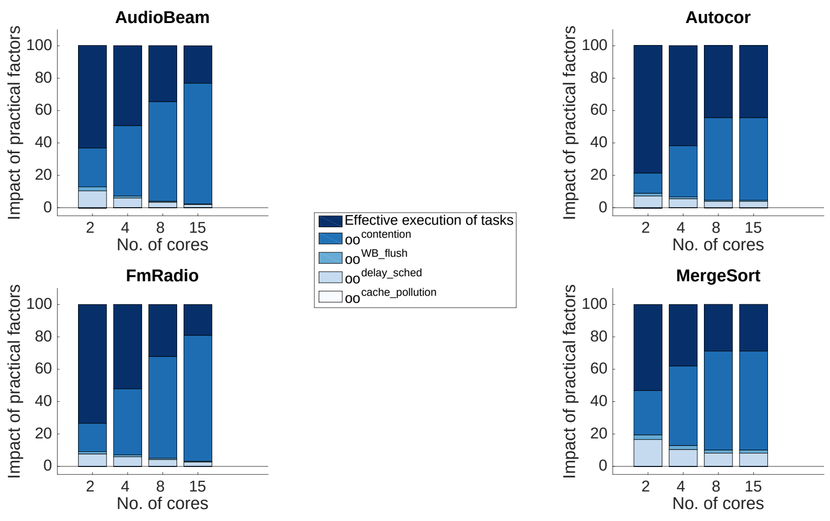


Fig. 13 The impact of different implementation factors on the adapted cache-conscious schedules

Additionally, the overall overhead induced on every task in τ^{cp} by write buffer flushing is very small. The reason is twofold: (i) communicating tasks are likely to be assigned to the same core to benefit from data reuse and in this situation no flush is needed; (ii) the cost for flushing the write buffer to the SMEM is small as compared to the execution time of tasks. As compared to the impact of those implementation factors, shared bus contention has the highest impact on the adapted schedules. The impact of shared bus contention tends to increase when the number of cores increases. The reason is that when the number of cores increases, the number of concurrent tasks tends to increase, which likely introduces more interference to the execution of tasks.

5.3.4 Evaluation performance of ACILP

In this section we evaluate the ability of ACILP to account for contentions, through a comparison with the work [42], having simimar objectives as ACILP. The work described in [42] transforms a contention-free static time-driven schedule to account for interference. In this work, a double fixed-point algorithm is proposed, which iteratively updates the WCET of tasks with contention delays and updates the trigger time of tasks accordingly (with respect to their execution order and their precedence relations) until that information is stable. In their double fixed-point algorithm, every task is forced to be triggered as soon as possible. However, in ACILP tasks at the near end of schedules are considered to be triggered at the time at which shared bus contention induced on the tasks is reduced. With

Benchmark	No. of cores	Schedule length (cycle)		Derivation time (s)		Gain (%)
		ACILP	[42]	ACILP	[42]	
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 7 Performance comparison between ACILP and the double fixed-point algorithm proposed in [42]

that concern, ACILP is expected to generate shorter schedules than the double fixed-point algorithm.

Modifying [42] to take into account the same simple contention model as in our work, in which all memory requests of tasks which are involved in contentions are delayed, we compare schedule length and required time for generating schedules when using ACILP and the double fixed-point algorithm.

All benchmarks in the study are scheduled on 2, 4, 8, and 15 cores. Table 7 presents the length of the adapted schedules and required time for generating the schedules by using ACILP and the double fixed-point algorithm. The gain in term of schedule length reduction, which shows the benefit of ACILP as compared to the double fixed-point algorithm is computed as:

$$gain = \frac{sl_{doublefixed-point} - sl_{ACILP}}{sl_{doublefixed-point}} * 100 \tag{29}$$

Table 7 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 ACILP only has chances to reduce contention induced on tasks at the nearly end of schedules.

Regarding required time for generating the adapted schedules, both ACILP and the double fixed-point algorithm produce schedules very fast. For all benchmarks in the study, the longest solving time of ACILP is 42 seconds, whereas the longest time that the double fixed-point algorithm takes to generate a schedule is 6 seconds, in the case of scheduling FmRadio benchmark on 15 cores.

6 Related work

Schedule generation. Schedulability analysis techniques rely on the knowledge of the WCET of tasks. Originally designed for single-core architectures, static WCET estimation techniques were extended recently to cope with multi-core architectures. Most research has 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 [2, 8, 18, 20, 26]. 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 [31], 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 et al. [39], 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 paper, 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 [11]. 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 paper generate offline time-driven partitioned non-preemptive schedules. Most scheduling strategies surveyed in [11] 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 paper 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. caches 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 [1, 7, 41, 46]; 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 contentions when accessing shared resources (e.g., shared bus, shared caches) [6, 12, 16, 22, 28, 43]. For example, in [12, 43], they jointly perform shared resources contention modeling and tasks mapping/scheduling for multi-core platforms. Ding et al. [12] focus on shared caches contention, whereas Rouxel et al. [43] pay attention to shared bus contention. In those works, tasks are mapped and scheduled, such that the contention delays induced on them are minimized, thus minimizing schedule length. Approaching from a different direction, Martinez et al. [28] modify existed schedules by introducing slack time between the execution of pairs of tasks consecutively assigned to the same core. This modification aims at limiting the contention between concurrent tasks contained in the existing schedules. Besides, some approaches [5,34–36,54] schedule tasks according to predictable execution models that guarantee spatial/temporal isolation between co-running tasks. For example, Becker et al. [5] take the advantage of memory privatization features available in the Kalray MPPA-256 to separately allocate private memory (includes code and data) of tasks. Besides, they design a scheduling policy to schedule the execution of tasks (which comply with a PREM-like model [34]), such that the memory requests of tasks are free from contention. The authors of [22] propose a technique for reducing memory interference using a partitioning of DRAM banks and co-locating memory-intensive tasks on the same processor. Our scheduling solutions in this paper differ from those works because we pay attention to the effect of private caches on tasks' WCETs. On the other hand, because our objective was to concentrate on the effects of private caches, in a first step we used a simple contention model, yielding to contention delays less precise than the ones proposed in [9,10,21]. Using more precise contention models is left for future work. Compared with our previous work [33], in this paper we consider practical, implementation related overheads, that are evaluated on a Kalray MPPA-256 compute cluster.

Having the same interest as us in utilizing data reuse between tasks, in [45] Suhendra et al. jointly consider task scheduling and memory allocating for multi-core systems equipped with scratchpad memory (SPM). In the work, the most frequently accessed data are allocated in SPM, and tasks are scheduled properly to reduce the accesses latency to the off-chip memory. As compared to the approach, our scheduling methods take into account both instruction and data reuse between pairs of tasks, and schedule them in order to get benefit in term of WCET reduction from cache reuse.

Related studies also address the effect of private caches when scheduling tasks on multi-core architectures [38, 47, 50]. 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 us to have better control on cache reuse during scheduling. Furthermore, [38] and [47] focus on single core architectures while our work targets multi-core architectures.

Schedule implementation. In the literature most scheduling techniques for multi-core hardware focus on handling shared resources contention (see [14] for the survey). In this paper, we pointed out that along with shared resources contention, the effect of time-driven scheduler and the lack of hardware-implemented data cache coherence are important factors that need to be considered in the implementation of time-driven, cache-conscious schedules on a multi-core hardware (especially the Kalray MPPA-256).

As compared to the works that address shared resources contention, our intent is not to mitigate the contention, but rather to integrate the contention into existing contention-free schedules. Having the same objective, in [42] Rihani et al. propose the double-fixed point algorithm. As explained in Section 5.3, the algorithm does not produce adapted schedules with shortest schedules length, as our proposed ACILP formulation does.

7 Conclusion

In this paper, we first studied the problem of scheduling a single parallel application on a multi-core platform subjected to the effect of private caches. Two scheduling techniques, including an optimal and a heuristic one, have been proposed to generate static, time-driven, partitioned, non-preemptive schedules. We experimentally showed the benefit in term of schedule length reduction when taking into account context-sensitive WCETs per task (due to cache reuse) in scheduling.

Secondly, we implemented time-driven cache-conscious schedules on a cluster of the Kalray MPPA-256. We pointed out the implementation issues arising when implementing the schedules on the platform, which are summarized as:

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

Additionally, we proposed an ILP formulation to adapt time-driven, cache-conscious schedules to the implementation factors. Experimental validation has shown the functional and the temporal correctness of our implementation and the efficiency of our proposed ILP formulation. Additionally, we observed that shared bus contention is the most impacting factor on the adapted schedules among the other ones.

We see several opportunities to further improve/extend this work. The work presented in this paper currently proceeds in two steps. A schedule that ignores all implementation factors (contention, jitter) is first generated. It is updated in the second step to account for these implementation factors. Proposing an integrated approach, that in particular integrates contention delays from the start, is an interesting direction for future work.

In addition, we can take more benefit from cache reuse between tasks. A task can reuse the workloads of several tasks executed preceding it, but not necessarily of the task executed immediately preceding it. We believe that if those reuses are considered, the advantage of cache-conscious scheduling strategies can be further improved. We envision two approaches to exploit the cache reuse. The first approach is to take into account the reduction in the WCET of a task when executed after several tasks rather than after only the task executed immediately before. Since the number of possible execution orders of tasks needed to be considered increases, estimating context-sensitive WCETs of tasks and scheduling tasks becomes more complex. The second approach is to use cache locking techniques [3, 40] in order to ensure that the useful workloads of tasks are still located in the caches until the task referring to them starts executing. Additionally, tasks scheduling has to be jointly performed with cache locking in order to fully exploit the benefit from cache reuse.

Acknowledgements The authors would like to thank Byron Hawkins and anonymous reviewers for their useful comments on this paper. This work was partially funded by European Unions Horizon 2020 research and innovation program under grant agreement No 688131, project Argo (http://www.argo-project.eu/), and by PIA project CAPACITES (Calcul Parallle pour Applications Critiques en Temps et Sret), reference P3425-146781.

References

1. Abdallah, L., Jan, M., Ermont, J., Fraboul, C.: 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)
2. Altmeyer, S., Davis, R.I., Indrusiak, L., Maiza, C., Nelis, V., Reineke, J.: 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)
3. Arnaud, A., Puaut, I.: Dynamic instruction cache locking in hard real-time systems. In: International Conference on Real-Time Networks and Systems (RTNS), pp. 1–10 (2006)
4. Bahn, J.H., Yang, J., Bagherzadeh, N.: Parallel FFT algorithms on network-on-chips. In: Fifth International Conference on Information Technology: New Generations (ITNG 2008), pp. 1087–1093 (2008)
5. Becker, M., Dasari, D., Nikolic, B., Akesson, B., Nélis, V., Nolte, T.: 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)
6. Calandrino, J.M., Anderson, J.H.: 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)
7. Carle, T., Djemal, M., Potop-Butucaru, D., de Simone, R., Zhang, Z.: 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)
8. Chattopadhyay, S., Roychoudhury, A., Mitra, T.: Modeling shared cache and bus in multicores for timing analysis. In: Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems, SCOPES '10, pp. 6:1–6:10 (2010)
9. Dasari, D., Andersson, B., Nélis, V., Petters, S.M., Easwaran, A., Lee, J.: Response time analysis of cots-based multicores considering the contention on the shared memory bus. In: IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2011, Changsha, China, 16-18 November, 2011, pp. 1068–1075. IEEE Computer Society (2011). DOI 10.1109/TrustCom.2011.146. URL https://doi.org/10.1109/TrustCom.2011.146
10. Dasari, D., Nélis, V.: An analysis of the impact of bus contention on the WCET in multicores. In: G. Min, J. Hu, L.C. Liu, L.T. Yang, S. Seelam, L. Lefèvre (eds.) 14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012, pp. 1450–1457. IEEE Computer Society (2012). DOI 10.1109/HPCC.2012.212. URL https://doi.org/10.1109/HPCC.2012.212
11. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys **43**(4), 35:1–35:44 (2011)
12. Ding, H., Liang, Y., Mitra, T.: Shared cache aware task mapping for WCRT minimization. In: 8th Asia and South Pacific Design Automation Conference, ASP-DAC, pp. 735–740 (2013)
13. Dupont de Dinechin, B., van Amstel, D., Poulhiès, M., Lager, G.: 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)
14. Fernandez, G., Abella, J., Quiñones, E., Rochange, C., Vardanega, T., Cazorla, F.J.: 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)
15. Geer, D.: Industry trends: Chip makers turn to multicore processors. Computer **38**, 11–13 (2005)
16. Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores. In: Proceedings of the Seventh ACM International Conference on Embedded Software, EMSOFT '09, pp. 245–254 (2009)
17. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2015)
18. Hardy, D., Piquet, T., Puaut, I.: 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)
19. Kasahara, H., Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. **33**(11), 1023–1029 (1984)

20. Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., Roychoudhury, A.: Static analysis of multi-core tdma resource arbitration delays. Real-Time Syst. **50**(2), 185–229 (2014)
21. Kim, H., de Niz, D., Andersson, B., Klein, M.H., Mutlu, O., Rajkumar, R.: Bounding memory interference delay in cots-based multi-core systems. In: 20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014, pp. 145–154. IEEE Computer Society (2014). DOI 10.1109/RTAS.2014.6925998. URL https://doi.org/10.1109/RTAS.2014.6925998
22. Kim, H., de Niz, D., Andersson, B., Klein, M.H., Mutlu, O., Rajkumar, R.: Bounding and reducing memory interference in cots-based multi-core systems. Real-Time Systems **52**(3), 356–395 (2016). DOI 10.1007/s11241-016-9248-1. URL https://doi.org/10.1007/s11241-016-9248-1
23. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. In: Journal of Parallel and Distributed Computing, vol. 59, pp. 381–422 (1999)
24. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys **31**(4), 406–471 (1999)
25. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, pp. 456–461 (1995)
26. Liang, Y., Ding, H., Mitra, T., Roychoudhury, A., Li, Y., Suhendra, V.: Timing analysis of concurrent programs running on shared cache multi-cores. Real-time Systems **48**(6), 638–680 (2012)
27. Maaita, A., Pont, M.J.: Using "planned pre-emption" to reduce levels of task jitter in a time-triggered hybrid scheduler. In: Proceedings of the Second UK Embedded Forum (Birmingham, UK, pp. 18–35 (2005)
28. Martinez, S., Hardy, D., Puaut, I.: 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)
29. Nélis, V., Yomsi, P.M., Pinho, L.M.: 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)
30. Nélis, V., Yomsi, P.M., Pinho, L.M., Fonseca, J.C., Bertogna, M., Quiñones, E., Vargas, R., Marongiu, A.: The challenge of time-predictability in modern many-core architectures. In: 14th International Workshop on Worst-Case Execution Time Analysis, *OpenAccess Series in Informatics (OASIcs)*, vol. 39, pp. 63–72 (2014)
31. Nemer, F., Cassé, H., Sainrat, P., Awada, A.: 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)
32. Nemhauser, G.L., Wolsey, L.A.: Integer and combinatorial optimization. Wiley interscience series in discrete mathematics and optimization. Wiley (1999)
33. Nguyen, V.A., Hardy, D., Puaut, I.: 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)
34. Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R.: 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)
35. Perret, Q., Maurère, P., Noulard, E., Pagetti, C., Sainrat, P., Triquet, B.: 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)
36. Perret, Q., Maurère, P., Noulard, E., Pagetti, C., Sainrat, P., Triquet, B.: 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)
37. Phatrapornnant, T., Pont, M.J.: Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling. IEEE Transactions on Computers **55**(2), 113–124 (2006). DOI 10.1109/TC.2006.29
38. Phavorin, G., Richard, P., Goossens, J., Chapeaux, T., Maiza, C.: 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)
39. Potop-Butucaru, D., Puaut, I.: 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)

40. Puaut, I., Decotigny, D.: 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)
41. Puffitsch, W., Noulard, E., Pagetti, C.: Off-line mapping of multi-rate dependent task sets to many-core platforms. Real-Time Systems **51**(5), 526–565 (2015)
42. Rihani, H., Moy, M., Maiza, C., Davis, R.I., Altmeyer, S.: 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)
43. Rouxel, B., Derrien, S., Puaut, I.: Tightening contention delays while scheduling parallel applications on multi-core architectures. ACM Trans. Embed. Comput. Syst. **16**, 164:1–164:20 (2017)
44. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: Second-generation intel xeon phi product. IEEE Micro pp. 34–46 (2016)
45. Suhendra, V., Raghavan, C., Mitra, T.: 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)
46. Tendulkar, P., Poplavko, P., Galanommatis, I., Maler, O.: Many-core scheduling of data parallel applications using SMT solvers. In: 17th Euromicro Conference on Digital System Design, DSD, pp. 615–622 (2014)
47. Tessler, C., Fisher, N.: BUNDLE: real-time multi-threaded scheduling to reduce cache contention. In: IEEE Real-Time Systems Symposium, RTSS, pp. 279–290 (2016)
48. Thies, W., Amarasinghe, S.: 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)
49. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Comput. Surv. **37**(3), 195–237 (2005)
50. Ward, B.C., Thekkilakattil, A., Anderson, J.H.: 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)
51. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.C., Brown III, J.F., Agarwal, A.: On-chip interconnection architecture of the tile processor. IEEE Micro pp. 15–31 (2007)
52. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem: Overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008)
53. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. on CAD of Integrated Circuits and Systems **28**(7), 966–978 (2009)
54. Yao, G., Pellizzoni, R., Bak, S., Betti, E., Caccamo, M.: Memory-centric scheduling for multicore hard real-time systems. Real-Time Systems **48**(6), 681–715 (2012)