Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures

Thomas Carle
Université Paul Sabatier, IRIT, CNRS
Toulouse, France
thomas.carle@irit.fr

Hugues Cassé
Université Paul Sabatier, IRIT, CNRS
Toulouse, France
casse@irit.fr

Abstract

We introduce a unified WCET analysis and scheduling framework for real-time applications deployed on multicore architectures. Our method does not follow a particular programming model, meaning that any piece of existing code (in particular legacy) can be re-used, and aims at reducing automatically the worst-case number of timing interferences between tasks. Our method is based on the notion of *Time Interest Points* (TIPs), which are instructions that can generate and/or suffer from timing interferences. We show how such points can be extracted from the binary code of applications and selected prior to performing the WCET analysis. We then represent real-time tasks as sequences of time intervals separated by TIPs, and schedule those tasks so that the overall makespan (including the potential timing penalties incurred by interferences) is minimized. This scheduling phase is performed using an Integer Linear Programming (ILP) solver. Preliminary results on state-of-the-art benchmarks show promising results and pave the way for future extensions of the model and optimizations.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Multicore architecture, WCET, Time Interest Points

Digital Object Identifier 10.4230/OASIcs.WCET.2018.3

1 Introduction

The advent of multicore architectures in embedded real-time systems raises multiple challenges for the community. For single-task (single-threaded) applications running on single-core architectures, the computation of safe-yet-precise Worst-Case Execution Time (WCET) bounds is a mature research domain, in which the complexity of hardware acceleration mechanisms (e.g. branch predictors) and of programs semantical properties (e.g. infeasible execution paths) must be mitigated in the analysis in order for the problem to remain tractable. On single-core machines, using preemptions to implement multi-task applications additionally incurs Cache-Related Preemption Delays (CRPDs) [2]: since multiple tasks share the instruction and data caches, a preemptive task can invalidate cache lines still needed by preempted tasks. This leads to additional timing penalties that were not present in the analysis of single-task applications.

For applications running on multicore architectures, deriving WCET bounds for the tasks running on each core becomes even more complex. Indeed, logically independent tasks can cause or suffer from *timing interferences* induced by the execution of tasks running simultaneously on other cores. For architectures where multiple cores share caches, the same

effect as CRPD can be observed. However, caches are not the only source of contention in multicore architectures, and subtler timing interferences between tasks can be generated in other shared elements such as the interconnect.

We consider that closely integrating WCET analysis and Time-Triggered (TT) scheduling can be a pragmatic and efficient way of coping with this increasing complexity by reducing the temporal instability of the applications. Existing models [15, 6] have shown that this approach yielded good results, but they require the analyzed applications to be written in a particular fashion. On the other hand, we propose a unified, code-analysis centric approach targetting arbitrary applications, and thus suited for legacy applications. Our technique analyses each task's code in isolation, and pinpoints all instructions that can generate or suffer from timing interferences. We call these particular instructions Time Interest Points (TIPs). Our method abstracts each task of the system into a sequence of code segments delimited by two (not necessarily consecutive) TIPs. Each segment's execution duration is stabilized by injecting a busy-wait loop before the ending TIP, directly in the binary code. Each segment is represented by its duration and the worst number of TIPs executed on any control flow path contained in the code of the segment. The objective of our approach is to schedule the segment sequences according to the real-time (e.g. periods and deadlines) and functional (data dependencies) constraints of their respective tasks, while reducing the number of possible timing interferences. In this paper, we propose an ILP formulation of the scheduling constraints in order to formally expose the problem. Since this paper presents a preliminary investigation of this model we will only focus on applications composed of two tasks running at the same frequency, yet the proposed approach can be easily extended to more general task systems (e.g. multi-periodic dependent tasks). Our approach does not rely on a particular programming model, and can be used on existing code without re-writing it. It works at binary level, allowing the analysis and the automatic code injection in pre-compiled code, and freeing our analysis from any programming language constraint.

This paper is divided as follows: Section 2 gives a presentation of existing work in the domain, Section 3 formally presents the problem and Section 4 details our method. Finally, Section 5 provides a proof-of-concept and Section 6 concludes.

2 Related work

2.1 Multicore interference analysis frameworks

Several Worst Case Response Time analysis frameworks [1] for multicore architectures have been devised in the past years. Their goal is to provide a schedulability criterion for a multi-task real-time system prior to its deployment, in particular for task systems scheduled using a non TT policy (e.g. fixed priority or EDF). The objective is to derive an exact or conservative bound on the number of timing interferences that can occur on each task, and to apply timing penalties to their WCETs accordingly. In [3], the analysis framework is based on the analysis of all possible execution traces of the task system on a given architecture, and allows a very high level of precision in the modeling of the architecture components, raising the concerns of the authors about the complexity of their analysis. Alternatively, the authors of [14, 16] propose an analysis method based on real-time Calculus for applications running on multicore architectures: tasks are approximated as sequences of time intervals containing the minimum and maximum number of potential interferences that can occur for the task on these intervals. However, to the best of our knowledge, the authors do not provide methods to obtain such abstractions from actual code. Our method uses an intermediate representation that is very close to the one defined in [14] and refined in [16]. However our

model differs in several points. First, instead of verifying the schedulability of the system, we use this representation to derive a schedule of the tasks. Second, in our method each code portion corresponding to a segment in the representation is temporized using busy-wait loops so that it executes for exactly the segment duration. Finally, our method targets the general model of multicore architectures with starvation-free interconnects, instead of the more restricted model of TDMA interconnect based architectures of [16].

2.2 Multicore extensions of the PREM model

The PREM [13] model was designed to avoid timing interferences for applications running on single core architectures connected to peripherals. The main idea is to separate the application into phases of three types: Read phases perform reads in the memory to preload the application code and the needed data, Execution phases perform the task calculation using only the instructions and data present in the cache, and Write phases update the values of modified variables in the main memory. The phases of the application can then be statically scheduled so that no Read or Write phase occurs when a peripheral uses the bus[1]. This model is extended to multicore architectures with scratchpad memories [15] and caches [6] by separating each task in three phases (Read/Exec/Write for the REW model or Acquisition/Execution/Restitution for the AER model) and by scheduling them statically so that memory phases from two or more cores never happen simultaneously. Each phase is time-triggered following the pre-computed starting dates. These methods work at the granularity of tasks, meaning that each task is composed of exactly one Read, one Execution and one Write phase. The Read (or Acquisition) phase prefetches all the data and instructions *potentially* required for the execution of the task in the local L1 cache or scratchpad, even though they may not be actually needed during the execution. To do so, it must be clear what data will *potentially* be read or written, as well as what code *may* be executed, by the task. This is defined by the programmer, using for example a system-level language such as PRELUDE [12] or *wrapper functions*. Tasks whose memory requirements exceed the capacity of the cache or scratchpad have to be manually divided into smaller subtasks. By contrast, our method works at a finer grain level and does not require any programmer's intervention.

3 Problem setting and formalism

In this section we define the formalism that will be used to describe our model and method throughout the paper.

3.1 Architecture

Our model focuses on multicore architectures composed of N cores, each of them connected to a private L1 cache. Each L1 cache is connected to the main memory through a starvation-free interconnect.

Each core has a programmable timer that can wake up a task sequencer (implementing a schedule computed off-line) using an interrupt through a direct link (not going through the shared interconnect).The core can program or rearm the timer through the shared interconnect. Moreover, each core also has a time stamp counter register *tsc_reg* (or an equivalent register) which counts CPU clock cycles with a fine granularity. These architecture traits are present in commercial off-the-shelf microprocessors such as the Aurix Tricore [10] or multicore ARMv8A [4].

[1] In the PREM model, peripherals such as sensors are allowed to write to the main memory.

3.2 Real-time tasks

We consider real-time applications modeled under the form of *non-preemptive* mono-periodic task systems. Formally, we denote $\mathcal{T} = \{\tau_i | 1 \leq i \leq n\}$ a task system composed of n tasks. Each task $\tau_i \in \mathcal{T}$ is characterized by:

- its period[2] $\tau_i.p \in \mathbb{N}$,
- its deadline $\tau_i.d \in \mathbb{N}$ (when $\tau_i.p = \tau_i.d$, the task is said to have an implicit deadline),

In the scope of this paper, we assume that each task runs on a separate core: this simplifies the scheduling ILP system, and at the same time allows us to apply our technique in situations where interferences are more likely to appear. This model is simple, yet complex enough to capture the traits of real-time applications with regard to multicore timing interferences.

3.3 WCET Computation

The identification of TIPs and the proposed scheduling method require not only WCET computation by static analysis but also intermediate results such as the analysis of the data cache. To this end, we use the Implicit Path Enumeration Technique (IPET) [11] approach which is made of three passes: (a) the path analysis, (b) the accelerator mechanism analysis and (c) the time analysis.

The path analysis consists in parsing all executions of the program. In order to increase the precision of the analysis, the IPET is performed on the binary form of the program and therefore, a compact and complete representation of a task is the *Control Flow Graph* (CFG). A CFG is a graph $G = \langle V, E, \nu, \omega \rangle$ where the nodes set V is composed of *Basic Blocks* (BB). A BB is a sequence of instructions in which only the first instruction can be targeted by a branch and only the last instruction can be a branch. $E \subseteq V \times V$ is the set of edges representing sequential execution or branches of the program. $\nu, \omega \in V$ are special empty BBs ensuring that G contains exactly one entry point (ν) and one exit point (ω).

The second analysis (b) aims at estimating the impact of accelerator mechanisms such as caches or branch predictors: these statistically improve the execution of the program (*hit*), but they do not work all the time (*miss*). A very common approach to support them is to statically compute abstract states (including all possible hardware states) and to assign a category representing their behavior. For example, for data caches [7], we distinguish four categories: *Always Hit* (AH), *Always Miss* (AM), *Persistent* (PE) or *Not-Classified* (NC). NC is the most imprecise case and a fall-back when the cache behavior is too complex. PE is a bit smarter and arises in loops: it means that the first access may cause a *miss* but the following accesses will cause *hits*. Notice that only memory instructions classified as AH are guaranteed to not generate interferences.

The last pass (c) computes the duration of BBs and weaves together (1) the WCET expression as the sum of all BBs durations multiplied by their occurrence counts on the WCET path, and (2) the constraints representing the execution paths and the effects of the accelerator mechanisms. The result gives an ILP system whose maximization provides the WCET.

3.4 TIPsGraph

We define TIPsGraphs as an intermediate representation in order to transform the CFG representing the control flow of a task into a sequence of time intervals representing the timing aspects of the task execution.

[2] In the scope of this paper we only target mono-periodic systems, so all tasks have the same period.

A TIPsGraph for task τ_i, $G_{TIPs}(\tau_i) = \{V_{TIPs}(\tau_i), E_{TIPs}(\tau_i)\}$ is composed of TIPs $t \in V_{TIPs}(\tau_i)$ and of edges $e \in E_{TIPs}(\tau_i)$.

TIPs $t \in V_{TIPs}(\tau_i)$ are instructions of task τ_i which can create or suffer from interferences in a multicore execution context, or *pivot instructions* which represent flow disjunctions (i.e. conditional branches) and junctions in the CFG. Pivot instructions allow our algorithm to encapsulate *if* and *loop* constructs into a single TIPsGraph edge, and thus to restrain the complexity of the subsequent ILP system.

Typically, TIPs can be:

- Memory instructions (stores and loads), when the static analysis cannot guarantee that they will always result in AH,
- Memory instructions addressing shared variables, or data residing in a cache block that can be written by another task,
- Instructions for which the static analysis cannot guarantee that they will always result in a hit in the instruction cache,
- Pivot instructions.

Instructions falling in the first and third categories can generate interferences for other tasks or suffer from interferences from other tasks on the interconnect (e.g. memory bus). Instructions falling in the second category are subject to interferences due to cache coherence maintenance. In the scope of this paper, we will focus on instructions falling in the first and last categories only, although the extraction of TIPsGraphs including TIPs falling in the other two categories is performed using the same algorithm. The reason for this restriction is that increasing the number of TIPs in the system dramatically complexifies the ILP system that we use for scheduling. Consequently, for the scope of this paper we consider that the tasks code is preloaded into the Instruction caches (or equivalently in private ScratchPad Memories) when the system is powered up.

An edge $e \in E_{TIPs}(\tau_i)$ is characterized by:

- its source TIP instruction $e.src \in V_{TIPs}(\tau_i)$,
- its destination instruction $e.dst \in V_{TIPs}(\tau_i)$,
- the worst-case number $e.\mu$ of TIPs encountered on any control-flow path linking $e.src$ to $e.dst$,
- e.WCET: the WCET of control-flow paths linking $e.src$ to $e.dst$.

3.5 Temporal segments sequence

Each task τ_i is represented as a sequence of time intervals (or segments) $\{(d_{i.j}, \mu_{i.j})_{0 \leq j < n_i}\}$, n_i being the number of segments that compose τ_i. These sequences are used to generate the ILP system which ultimately produces the tasks schedule. A time interval $ti_{i.j}$ is characterized by its duration $d_{i.j}$, as well as the worst case number of non-AH memory accesses $\mu_{i.j}$ performed during the execution of the segment. An important point is that a segment is characterized by an exact duration, and not by a WCET: in order to effectively reduce conflicts on the interconnect through careful scheduling of the tasks, we must know in advance when a task accesses memory. In order to suppress the temporal instability inherent to unbalanced control-flow paths and to the conservatism of our WCET estimation technique, stabilization loops are injected automatically in the binary code before the end of each segment. These loops poll the *tsc_reg* of their core until a pre-computed date is reached. Once it has been reached, the normal execution flow resumes. This technique has been introduced in the PREM model [13] to stabilize the duration of the whole Execution phase of each task.

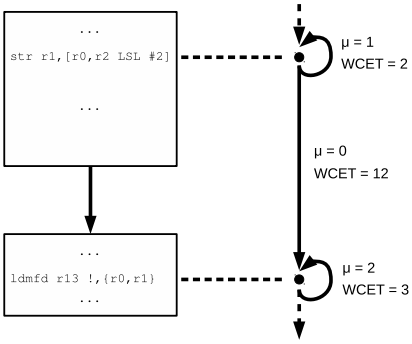

Figure 1 Example of graph extraction for a linear sequence of BBs.

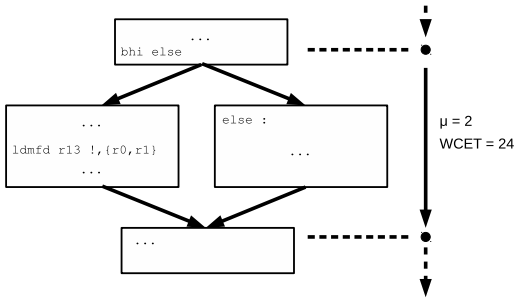

Figure 2 Example of graph extraction for a non-linear control structure.

The segments are straightforwardly obtained from a TIPsGraph by translating each edge in the graph into a segment.

4 Multicore WCET analysis using TIPs

In this section, we describe how TIPsGraphs are extracted from the CFG of tasks and then transformed into sequences of temporal segments. We also explain how the ILP scheduling system is generated from a set of temporal segments sequences.

4.1 Extracting a TIPsGraph from the CFG of a task

The extraction of the TIPsGraph of a task τ_i is performed by exploring the task's CFG from the entry point to the exit point. During this exploration, the extraction algorithm can be in one of two situations: either it is exploring a linear sequence of BBs without pivot instruction, or it has reached a pivot which marks a disjunction in the control flow. In this second case, a subprocedure looks for the matching junction in the graph and creates an edge between the disjuncting pivot and its matching join pivot (see 4.1.2).

4.1.1 Linear sequence of Basic Blocks

As long as the exploration procedure has not encountered a pivot instruction, it goes through the instructions of the program in sequence. If an instruction *inst* is a memory instruction (`str`/`ldr`/`stm`/`ldm` in ARM instruction set) which is not guaranteed to result in a hit (non-AH) in the data cache, the procedure creates a corresponding TIP new_TIP in $V_{TIPs}(\tau_i)$, and an edge e in $E_{TIPs}(\tau_i)$ from the last encountered TIP $last\_TIP$ to the current TIP, with $e.\mu = 0$ since no memory operation is performed between the two TIPs, and e.WCET equal to the WCET of the code portion between $last\_TIP$ and new_TIP. In order to reduce the number of extracted TIPs, the procedure then regroups all non-AH memory instructions directly following new_TIP in the code as part of the same TIP (if such instructions are present). It computes the number μ' of all non-AH memory accesses performed by the instruction(s) grouped in the TIP, as well as the WCET of the corresponding instruction(s) WCET$'$, and creates an edge e', in which $e'.src = e'.dst = new\_TIP$, $e'.\mu = \mu'$ and e'.WCET = WCET$'$. This self-edge looping on the TIP accounts for the duration of the instruction(s) represented by the TIP, which generate traffic on the interconnect. The procedure then resumes the exploration of the instructions in sequence. Figure 1 illustrates

this process: the boxes on the left represent BBs in the CFG of a task, and the graph in the right is the part of the TIPsGraph corresponding to this part of the CFG. The *str* instruction in the top is analyzed as non-AH, so a TIP (a node) is created in the TIPsGraph. The self-edge on this TIP is labeled with $\mu = 1$ because the *str* instruction only performs one non-AH memory access. The WCET label for this edge corresponds to the WCET of this *str* instruction[3]. The next non-AH memory access found by the procedure is made by the *ldmfd* instruction at the bottom. A TIP is added to represent this instruction in the TIPsGraph, and an edge links it to the previous TIP.

If a pivot instruction p is reached, the procedure creates a corresponding TIP in $V_{TIPs}(\tau_i)$, as well as an edge e from the last encountered TIP to p, with $e.\mu = 0$ and $e.WCET = WCET(last\_TIP, p)$. The procedure then follows the algorithm described in the next section. Finally, when the procedure reaches the end of the CFG, it returns $G_{TIPs}(\tau_i)$.

4.1.2 Non-linear control structures

When a pivot instruction p is reached, it necessarily marks a disjunction in the control flow (an *if* branch or the start of a loop). In order to analyze the disjoint part of the CFG as a whole, the procedure first looks for the unique pivot instruction p' that marks the corresponding junction of the control flow paths, and puts it in $V_{TIPs}(\tau_i)$. This instruction is the first instruction of the first BB that (a) is a (direct or transitive successor of the BB containing p *and* (b) dominates the exit point (ω) of the CFG. Then the procedure explores all control flow paths between p and p', in order to find the maximum number of non-AH memory instructions μ_{max} present on any path linking p to p'. Finally, it creates an edge e in $E_{TIPs}(\tau_i)$, with $e.src = p$, $e.dst = p'$, $e.\mu = \mu_{max}$ and $e.\text{WCET} = \text{WCET}(p, p')$. The procedure then resumes the linear exploration of the CFG described in Section 4.1.1.

The exploration of non-linear control structures is illustrated by Figure 2. The *bhi* instruction is a pivot which opens a disjoint section of the CFG. The procedure adds a TIP corresponding to this pivot in the TIPsGraph. After this, it looks for the first instruction after the disjoint portion of the CFG (the first instruction of the bottom BB) and creates a corresponding TIP. Then an analysis is performed on the two paths: the maximum number of non-AH memory accesses on either paths is 2: the left path executes a *ldmfd* instruction performing two memory accesses, both of which were labeled non-AH by the cache analysis. On the other hand, the path on the right makes no non-AH memory access. The WCET of the section between the *bhi* instruction and the first instruction in the BB at the bottom was found to be 24 time units. This WCET does not necessarily correspond to the left path.

4.2 From a TIPsGraph to a temporal segments sequence

Once a TIPsGraph containing all TIPs of a task has been extracted, its translation into a sequence of temporal segments is straightforward: the graph is traversed from its starting node to its end node, passing by each edge exactly once, with a priority given to self-edges. When traversing an edge e, it is translated into a segment s with $s.d = e.WCET$ and $s.\mu = e.\mu$. Figure 3 shows how a segment sequence is obtained from the TIPsGraph of a task τ_1: the TIPsGraph section considered in this example starts by a TIP on the left. The first edge e_0 to be translated into a segment is a self-edge: a segment $ti_{1.0}$ is created with

[3] In the figures, the WCETs are given in arbitrary time units.

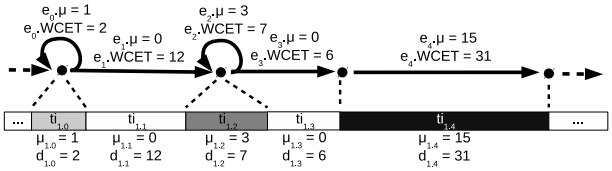

Figure 3 Example of temporal segment sequence extraction.

$d_{1.0} = e_0.WCET = 2$ and $\mu_{1.0} = e_0.\mu = 1$. Then a second segment $ti_{1.1}$ with $\mu_{1.1} = 0$ is extracted from edge e_1, and so on. In the figure, the density of the color of the segments reflects the number of TIPs they contain: the higher the μ, the darker the segment.

We will now present how such sets of sequences are translated into ILP variables and constraints in order to schedule the task system.

4.3 Multicore scheduling using ILP

In this section, we present the variables and constraints that are used to model our scheduling problem in ILP. Multiple objective functions can be used, optimizing different aspects, but overall the constraints presented here remain the same regardless of the optimization criterion. Finally, some constraints make use of ∞ : these constraints are encoded using a sufficiently large integer number (i.e. at least one order of magnitude larger than the variables of the system)[4].

For each task τ_i in our system, we first introduce two sets of variables : $\{s_{i.j} | 0 \leq j < n_i\}$ and $\{\gamma_{i.j} | 0 \leq j < n_i\}$, which represent respectively the start time and the number of interferences for each segment $ti_{i.j}$. In addition to these variables, we define $s_{i.n_i}$ as the end date of the last temporal segment of τ_i (i.e. the end date of $ti_{i.n_i-1}$). Using these variables, the following constraints impose the sequential execution of τ_i and the application of deadline $\tau_i.d$ (c_{inter} represents the cost of an interference):

$$s_{i.0} \geq 0 \tag{1}$$

$$s_{i.n_i} \leq \tau_i.d \tag{2}$$

$$\forall j : 0 \leq j < n_i, s_{i.j+1} = s_{i.j} + d_{i.j} + c_{inter} \times \gamma_{i.j} \tag{3}$$

The tricky part concerns the evaluation of $\gamma_{i.j}$ which depends on the segments of tasks running on other cores, $k.l$ (segment l of task k), that overlap the execution of segment $i.j$. Variable $\chi_{i.j-k.l} \in \{0, 1\}$ asserts whether $i.j$ and $k.l$ overlap. In this case, $i.j$ undergoes at most $min(\mu_{i.j}, \mu_{k.l})$ interferences from $k.l$. In fact, considering all segments of τ_k overlapping $i.j$, our conservative approximation is that $i.j$ suffers in the worst case from the sum of interferences generated by each overlapping segment of core k, with at most $\mu_{i.j}$ interferences in total. The interferences with τ_k are recorded in $\gamma_{i.j-k}$ and, as exposed below, $\gamma_{i.j}$ is the sum of interferences of τ_i with all other tasks:

$$\gamma_{i.j} = \sum_{0 \leq k < n \wedge k \neq i} \gamma_{i.j-k} \text{ , with: } \qquad \gamma_{i.j-k} = min\left(\mu_{i.j}, \sum_{0 \leq l < n_k} \mu_{k.l} \times \chi_{i.j-k.l}\right)$$

The formulation of $\gamma_{i.j-k}$ cannot be translated as is in the ILP system because of the min

[4] As a result, $\infty \times 0 = 0$

but we can rewrite it as:

$$\gamma_{i.j-k} \leq \mu_{i.j} \tag{4}$$

$$\gamma_{i.j-k} \leq \left(\sum_{0 \leq l < n_k} \mu_{k.l} \times \chi_{i.j-k.l} \right) \tag{5}$$

$$\gamma_{i.j-k} \geq \mu_{i.j} - \infty \times (1 - \alpha_{i.j-k}) \tag{6}$$

$$\gamma_{i.j-k} \geq \left(\sum_{0 \leq l < n_k} \mu_{k.l} \times \chi_{i.j-k.l} \right) - \infty \times \alpha_{i.j-k} \tag{7}$$

$$0 \leq \alpha_{i.j-k} \leq 1 \tag{8}$$

Eq. (4) and (5) enforce the selection of the minimum but, according to the trend of the objective function, a possible value for $\gamma_{i.j-k}$ could be 0. This is prevented by the variable $\alpha_{i.j-k}$ and Eq. (6) and (7) which ensure that either $\mu_{i.j}$, or the sum of $\mu_{k.l}$ is selected.

To detect overlapping and define $\chi_{i.j-k.l}$, we have to compare start and end dates of segments of tasks running on different cores, $i.j$ and $k.l$:

$$\theta_{i.j-k.l} \iff s_{k.l} \leq s_{i.j} < s_{k.l+1} \text{ , and} \qquad \theta_{k.l-i.j} \iff s_{i.j} \leq s_{k.l} < s_{i.j+1}$$

Considering the trend to minimize $\gamma_{i.j}$, $\theta_{i.j-k.l}$ (and symmetrically $\theta_{k.l-i.j}$) can be viewed as the selection of exactly one of the following constraints:

$$s_{k.l} \leq s_{i.j} < s_{k.l+1} \ (\theta_{i.j-k.l} = 1); \quad s_{i.j} < s_{k.l} \ (\theta_{i.j-k.l} = 0); \quad s_{k.l+1} \leq s_{i.j} (\theta_{i.j-k.l} = 0)$$

Introducing the cancellation variable $\beta_{i.j-k.l}$, the ILP formulation becomes:

$$s_{k.l} \leq s_{i.j} + \infty \times (1 - \theta_{i.j-k.l}) \tag{9}$$

$$s_{i.j} < s_{k.l+1} + \infty \times (1 - \theta_{i.j-k.l}) \tag{10}$$

$$s_{i.j} < s_{k.l} + \infty \times (1 - \beta_{i.j-k.l}) \tag{11}$$

$$s_{k.l+1} \leq s_{i.j} + \infty \times (\beta_{i.j-k.l} + \theta_{i.j-k.l}) \tag{12}$$

$$0 \leq \beta_{i.j-k.l} + \theta_{i.j-k.l} \leq 1 \tag{13}$$

Eq. (9) and (10) apply only if $\theta_{i.j-k.l} = 1$ (overlapping of segments $i.j$ and $k.l$). When $\theta_{i.j-k.l} = 0$, only one constraint between Eq. (11) and (12) holds, depending on the value of $\beta_{i.j-k.l} \in \{0, 1\}$. The last constraint ensures that $\beta_{i.j-k.l}$ and $\theta_{i.j-k.l}$ are not both set to 1 at the same time.

Notice that $\theta_{i.j-k.l}$ and $\theta_{k.l-i.j}$ can be set to 1 together when the segments start at the same date ($s_{i.j} = s_{k.l}$). Finally, $\chi_{i.j-k.l}$ is defined as:

$$0 \leq \chi_{i.j-k.l} \leq 1 \tag{14}$$

$$\chi_{i.j-k.l} \geq \theta_{i.j-k.l} \tag{15}$$

$$\chi_{i.j-k.l} \geq \theta_{k.l-i.j} \tag{16}$$

At this point, we have presented all the models and algorithms required to apply our method. In the next section, we present our preliminary results on realistic applications.

Table 1 Summary of applications profiles.

bench	WCET in isolation (in clock cycles)	# segments	# TIPs	longest segment (in clock cycles)	max TIPs in a segment
edn	416221	70	5882	208056	3400
insertsort	2968	30	13	2796	1
fibcall	942	18	69	761	60

5 Proof-of-concept

We developed a prototype application[5] based on the OTAWA [5] WCET analyzer and applied it on three benchmarks from the *Mälardalen* [8] suite: *edn*, *fibcall* and *insertsort*. These benchmarks exhibit common traits of embedded applications, and as we will see, they show very different profiles in terms of WCET and of number of memory accesses.

The first result of our analysis method is that we are able to exhibit and analyze a safe and refined timing profile of memory accesses of these applications. These profiles can also be used to extract precise arrival curves suited for methods such as [14]. We summarize key points in Table 1.

These three applications show varied profiles in number of segments, overall size and number of TIPs. Yet, one common trait is that each of them has one segment that lasts around half of its total WCET or more (WCETs and segment lengths are given in number of processor cycles). This is the result of aggregating *ifs* and *loops* inside one segment. However, we are currently working on adding more precision to the analysis of such constructs, and in particular on allowing the extraction of segments delimited by a chosen number of loop iterations.

Once this profiling is done, our prototype calls CPLEX [9] to schedule tasks two-by-two on separate cores, minimizing the makespan of the task system. We chose to fix the interference cost c_{inter} to 10 processor cycles, because it is approximately the cost of accessing the shared data scratchpad in the Aurix Tricore architecture. The result for *insertsort* and *fibcall* with this objective function is an interference-free schedule in which *insertsort* begins its execution at date 0 and finishes at date 2968. *fibcall* starts at date 170 and finishes at date 1112. Without our method the worst-case of 13 interferences should have been assumed, incurring a total additional duration of 130 cycles, which is more than a 10% overhead for *fibcall*. This preliminary experiment on real applications illustrates the possibility to reduce the number of timing interferences without having to re-write existing code, as well as the necessity to define precise analysis models in order to do so. We also tried to apply our method on application pairs featuring *edn*, but CPLEX failed to provide a solution. We believe this is linked to this application having a too long overall WCET, which increases dramatically the feasible region to be explored. These experiments convince us that our method should rely on efficient scheduling heuristics rather than on ILP solvers if we are to successfully deal with large tasks and/or large task systems.

[5] Following this proof-of-concept, a complete analysis application is now under development.

6 Conclusion and future work

In this paper we proposed a novel approach for the WCET analysis of applications running on multicore architectures. This method is particularly well-suited for legacy applications, since it can be fully automated, requires no re-writing of existing code, and works directly at the binary level. It is based on the notion of Time Interest Points, which are instructions in the binary code that potentially cause or suffer from timing interferences on the interconnect. Our method extracts such TIPs and abstracts the application tasks as sequences of TIPs separated by temporal segments. In order to increase the timing stability of this representation, waiting loops are automatically injected at the end of each segment. These sequences of temporal segments are then scheduled in order to minimize the application makespan. In order to illustrate how this approach works, we implemented a prototype application and applied it on benchmarks from the *Mälardalen* suite. Our preliminary results (application profiling and scheduling) lead to the following conclusions:

- This method is technically feasible and promising, especially for the analysis of legacy code,
- Our next efforts should target the definition of fast-yet-efficient scheduling heuristics, to free our method from the limitations inherent to ILP and allow the resolution of larger systems as well as the introduction of new kinds of TIPs in our problems (e.g. Instruction Cache TIPs),
- In order to aggressively reduce the number of interferences, we must break down large temporal segments that represent *ifs* and *loops*. For example, we want to make it possible to extract temporal segments as specified chunks of loop iterations.

References

1. A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, 2013.
2. S. Altmeyer and C. Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 2011.
3. S. Altmeyer, R. I. Davis, L. Soares Indrusiak, C. Maiza, V. Nélis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *RTNS*, 2015.
4. ARM. *ARM Cortex-A Series – Programmer's Guide for ARMv8 - A*, v1.0 edition, 2015.
5. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, 2010.
6. G. Durieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In $ERTS^2$, 2014.
7. C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. *Lecture notes in computer science*, 1998.
8. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET*, 2010.
9. IBM. Cplex user's manual. `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.0/ilog.odms.studio.help/pdf/usrcplex.pdf`, 2016.
10. Infineon. *AURIX TC27x D-Step (32-Bit Single-Chip Microcontroller) User's Manual, v2.2*, 2014.
11. Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1995.

12 C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3), 2011. doi:10.1007/s10626-011-0107-x.

13 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. *RTAS*, 2011.

14 R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. *DATE*, 2010.

15 B. Rouxel, S. Derrien, and I. Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst.*, 2017.

16 A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. *RTAS*, 2010.