Parallel Real-Time Tasks, as viewed by WCET Analysis and Task Scheduling Approaches.

Christine Rochange[1]

1 **University of Toulouse, France**
rochange@irit.fr

Abstract

With the advent of multi-core platforms, research in the field of hard real-time has recently considered parallel software, from the perspective of both worst-case execution time (WCET) and task schedulability (or worst-case response time, WCRT) analyses. These two areas consider task models that are not completely identical and sometimes make different assumptions. This paper draws a brief overview of the state of the art in the timing analysis of parallel tasks and tries to identify points of convergence and divergence between the existing approaches.

1998 ACM Subject Classification D.2.4 Software/Program Verification; C.3 Special-Purpose and Application-based Systems - Real-time and embedded systems

Keywords and phrases multicore, parallel tasks, worst-case execution time analysis, schedulability and worst-case response time analysis

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

After many years of improved single-core processor performance thanks to micro-architectural innovations, a ceiling has been reached due to three limitations: (i) the memory wall (the gap between processor and memory performance is still growing and can only partially be hidden using cache hierarchies), (ii) the instruction-level parallelism wall (finding enough parallelism in a single instruction stream to keep ever more intra-core resources busy becomes challenging) and (iii) the power wall (power consumption exponentially increases with the operating frequency, which comes with thermal dissipation issues). For all these reasons, the design of microprocessors has turned to multicore architectures that integrate several cores on a single chip and share some resources (such as the main memory and the interconnection network) among cores.

This trend, that first concerned servers and desktop systems, spreads now to embedded systems: they exhibit growing performance requirements, mainly to implement new functionalities (e.g. better control of gas emissions in automotive engines), but are subject to drastic constraints on power consumption and thermal dissipation.

Critical systems may not be ready yet to take the step because of the new challenges raised by these multicore architectures. But they will have to in the medium term because they also have increasing computing power needs and because off-the-shelves components likely to meet these needs will all be multicore.

However, if a multicore processor makes it trivially possible to increase performance by allowing higher task rates (several tasks can be run in parallel on different cores), it does not reduce the execution time of a particular task. In that sense, it does not help tasks to meet their deadline. To shorten execution times, parallelising task codes is needed.

Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

It is likely that not all real-time applications can be parallelised because some of them must typically execute a series of actions in sequence. But we can think of applications that perform image processing which generally lends itself well to parallelisation.

As soon as real-time applications are parallelised, the question of their timing analysis arises. This means estimating their worst-case execution time and their worst-case response time (which is related to scheduling decisions). These last few years, a few results have been published in these two areas. Often, contributions to WCET analysis and to schedulability analysis tend to ignore their respective constraints. On one hand, schedulability analysis assumes that tasks can be preempted, migrated from one core to another one, without impairing WCET estimations. On the other hand, WCET analysis assumes that a task will run from end to end on the same core without being interrupted. Recently though, some steps towards higher integration of these two kinds of analyses have been made in the context of single-core platforms. For example, Altmeyer et al. [1] estimate cache-related preemption delays while Zhi-Hua et al. [26] introduce a WCET-aware task mapping and cache partitioning strategy. In this paper, we discuss the few solutions that have been proposed on both sides to handle parallel tasks, and we try to identify how far they fit together.

Please note that this paper focuses on software-level interferences between the threads of a parallel task. We are aware that hardware-level interferences occur when cores share resources. This includes bandwidth sharing that might generate conflicts and incur additional latencies, but also space sharing, such as when multiple cores share a single L2 cache. Although such hardware-related interferences may strongly impact worst-case execution times, we want to keep the focus of this paper on software-level interferences. For this reason, we deliberately ignore hardware-level conflicts, assuming that either they cannot occur (e.g. partitioning strategies are used [24]) or they are accounted for in WCET estimations [6, 3]. Almost all the approaches considered in this paper make a similar assumption.

The paper is organised as follows. In Section 2, we give an overview of parallel programming that support the parallel task models considered in WCET and WCRT analyses. Section 3 deals with scheduling approaches for parallel real-time tasks to be run on multicore platforms. Worst-case execution time analysis techniques for parallel codes are overviewed in Section 4. Section 5 discusses the compatibility of WCET and WCRT analyses and Section 6 summarises the paper.

2 Parallel Programming

The goal of parallel programming is to allow several computations within the same application to be performed simultaneously. To do so, a task is split into subtasks, technically processes or threads. In the following, we only refer to threads for the sake of simplicity. Generally, threads belonging to the same application share data and thus have to communicate in one way or another. Additionally, they sometimes have to synchronise their respective progress to ensure the correctness of the final result.

When the same computation must be done on each element of a set of data, a way to split a task is to split the set of data into subsets and to have each of them processed by a different thread. For example, in matrix-based algorithms, each thread may compute one column of the result matrix. Often, data parallelism leads to parallelising a loop in such a way that each thread executes one part of the loop iterations.

Task parallelism is another kind of parallelism where the application includes several more-or-less independent computations that may exhibit precedence constraints. In such a case, a thread is created for each computation or sequence of computations.

Of course, a hybrid pattern is possible, where the application contains several computations that can be run simultaneously and some of these computations exhibit data parallelism. Software pipelining is a particular parallel programming pattern: each piece of data must undergo the same sequence of computations and one thread is created for each computation (pipeline stage). All the threads execute in parallel and data flow from one thread to the next one.

According to the memory organisation (either a shared memory with a common address space or a distributed memory with local address spaces), the way shared data can be exchanged among threads differs. In the case of a common address space, threads simply access shared variables in memory. However, precautions must be taken to enforce the integrity of data. For example, read-modify-write sequences must be performed atomically to avoid inconsistencies. In the case of a distributed memory, threads communicate though message passing: the thread that hosts the piece of data in its local memory provides it by executing a `send()` primitive, while the thread that requires the data executes a `receive()` primitive.

Two kinds of synchronisations are typically used in parallel programs. Some of them ensure a coherent progress of threads, e.g. that one thread cannot execute beyond a given point X until another thread has reached a given point Y. Mechanisms that implement such progress synchronisations include barriers and conditions. Other synchronisations enforce atomicity for a region of the code, called a critical section, by ensuring that no other thread can execute the same region of code at the same time. This is referred to as mutual exclusion and locks are a simple way to implement it.

3 Scheduling Parallel Tasks

In the field of real-time multiprocessor scheduling, earlier works considered independent sequential tasks [4]. In this paper, we focus instead on scheduling parallel tasks. In this area, a parallel task is defined as a collection of subtasks that can be run in parallel as long as their precedence constraints are fulfilled. Each release of the task generates as many jobs as subtasks and the task instance is considered completed when every related job has been completed. We will first present common parallel task models and the underlying assumptions, then we will provide a brief overview of real-time scheduling approaches for such parallel tasks.

3.1 Parallel Task Model and Assumptions

The most general model used to represent real-time parallel tasks is a directed acyclic graph (DAG) $G = (V, E)$ that exposes subtasks (nodes, $v \in V$) and their precedence constraints (edges, $e \in E$) [2], as illustrated in Figure 1. The WCET of each node/subtask is supposed to be known. The task is characterised by a relative deadline and a period, that apply to the whole DAG. When the task is released, a job is created for each subtask. Jobs are scheduled on several cores in a way that meets precedence constraints and they all must be completed before the deadline. Note that subtask dependencies mentioned in real-time scheduling literature usually denote such precedence constraints exclusively.

This model assumes that the task is mapped to n identical cores that are not shared with other tasks. However the number of cores allocated to the task might not allow running all the ready threads simultaneously.

Other works consider the fork-join model where a task is released as a master thread that creates child threads which can run simultaneously (i.e. that do not exhibit precedence

Figure 1 A parallel task defined as a set of sub-tasks subject to precedence constraints (DAG)

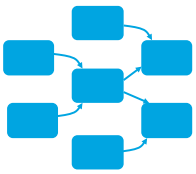
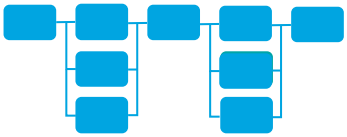


Figure 2 A parallel task following the fork-join model



constraints) before joining [13]. This results in a sequence of sequential/parallel segments, as illustrated in Figure 2. This model assumes that all the child threads in a parallel segment execute the same code (then have the same WCET). All parallel segments have the same number of threads.

A variant of the fork-join model is the multi-frame segment model [23] shown in Figure 3. The main difference resides in the fact that parallel segments may have different numbers of threads.

Note that both the fork-join and multi-frame segment models can be expressed as DAGs.

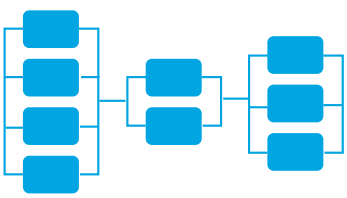
3.2 Scheduling Approaches

There are two main kinds of approaches to scheduling real-time parallel tasks. The first one, called *task decomposition*, transforms a parallel task expressed as a DAG of subtasks into a set of independent tasks that can be handled by multiprocessor scheduling strategies for sequential tasks. The second kind of approach, referred to as *direct scheduling*, considers a parallel task as a set of subtasks that can be scheduled on different cores but must altogether meet the timing constraints of the task. These approaches are summarised below.

3.2.1 DAG Transformation

The goal of DAG transformation techniques is to convert the set of dependent subtasks into independent tasks that can then be scheduled using multiprocessor scheduling approaches for

Figure 3 A parallel task following the multi-frame segment model.



sequential tasks. Each new sequential task has its own computing requirements (WCET) but also its own release offset and its own deadline which are derived from the timing parameters of the original parallel task, taking precedence constraints into account [13, 23, 21].

3.2.2 Direct scheduling

Kato et al. [12] consider the fork-join task model (Fig. 2). They observe that threads in parallel segments should run in parallel to avoid deadlocks (or long delays) that might arise if a thread holding a lock was preempted. For this reason, they explore gang scheduling techniques in the context of real-time parallel tasks. Gang scheduling grants a task with as many cores as threads in its parallel segments. This way, all the threads run simultaneously.

Other works focus on the DAG model (Fig. 1). The federated scheduling strategy allocates a sufficient number of cores to each high-utilization parallel task and schedules its DAG nodes using a greedy algorithm [14]. Global-EDF-based approaches have also been proposed [2, 15].

4 WCET Analysis of Parallel Applications

Scheduling approaches generally summarise a task as an execution segment characterised by a worst-case execution time (often denoted as C_i), together with other attributes such as a period, a deadline or a release time. WCET analysis sees it as a control flow graph (CFG), that expresses theoretically possible execution paths and is built from the executable code. Part of the analysis exploits knowledge of the code semantics.

Research on WCET analysis started more than 20 years ago and investigates two main branches: flow analysis, that computes loop bounds and infeasible paths [8, 17, 11], and low-level analysis, that determines the local WCET of basic blocks (indivisible sequences of instructions) taking the characteristics of the underlying hardware architecture into account [25, 22, 20].

Recently, several contributions have been made to the WCET analysis of parallel tasks. Some of them consider a distributed memory organisation while other ones assume a shared memory and focus on synchronisations. All these works assume that each thread of the parallel application runs on a separate core and that there are enough cores to run all the threads simultaneously. They further consider that all threads are released and start executing at the same time. Note that they all rely on static WCET analysis techniques. However, some of their principles have been adapted to measurement-based analysis with the RapiTime tool [5].

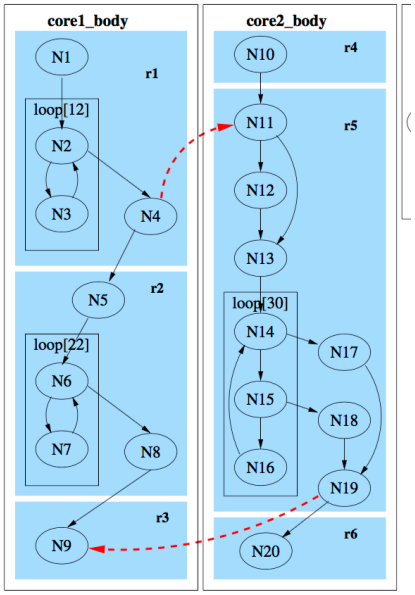
4.1 Communicating Tasks

Potop-Butucaru and Puaut [19] consider two distinct control loops (an infinite loop that typically repeats the following cycle: read sensors, process, write actuators) running on separate cores. They assume that these loops share some data and communicate by message passing.

Their analysis aims at determining the WCET of the longest loop body, taking into account delays due to blocking message passing primitives. To achieve this, they first perform flow and low-level analyses on the control flow graphs of both loops. Then they merge the CFGs by adding edges (dashed red edges in Figure 4) to express precedence between CFG nodes (a thread calling a `receive()` primitive is stalled until the other thread executes the corresponding `send()` primitive). Finally, they compute the worst-case path in the joint CFG using the commonly used IPET method [16].

Figure 4 Joint CFG of parallel control loops (Figure taken from [19])

.



This approach fits well message-passing asymmetric communications but cannot be applied to symmetric communication/synchronisation schemes, when the interleaving of threads is decided dynamically (e.g. when the order in which threads enter a critical section depends on which thread reaches the critical section first).

4.2 Parallel Tasks

Contributions in this category consider parallel tasks using shared memory and focus on delays incurred by synchronisations (barriers, conditions, locks).

Gustavsson et al. [9] apply model-checking techniques to a system of timed automata to compute the WCET of a parallel task composed of synchronising threads running on a multicore processor. Both the behaviour of the hardware executing the code and the conflicts between threads to acquire locks are modeled with timed automata that are then combined. Then properties such as "the clock value must be lower than x for every possible state" are verified considering several values of x selected following a binary search scheme.

Gustavsson et al. [10] introduce a shared-memory parallel programming language and a fix-point analysis able to identify all the possible thread interleavings at critical sections.

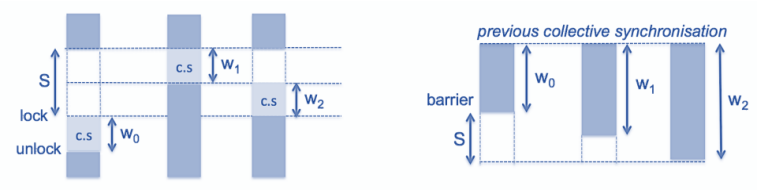
The approach proposed by Ozaktas et al. [18] sticks to the usual static WCET analysis process (flow analysis, low-level analysis and IPET method) and only modifies basic blocks local WCETs to reflect stall times at synchronisation points. The way worst-case stall times (WCST) are estimated is illustrated in Figure 5. It requires being able to compute *partial WCETs*, e.g. the WCET from one point in the code to another one.

The WCST for thread t to enter a critical section (i.e. to acquire a lock) is given by:

$$WCST^t = \sum_{0 \leq i < n, i \neq t} w_i^{cs}$$

where n is the number of parallel threads and w_i^{cs} is the worst-case execution time of thread i

Figure 5 Computing worst-case delays at synchronisations.



in the critical section. If all the threads are identical, we get: $WCST^t = (n-1) \times w^{cs}$. This formula assumes that the lock is granted with a FIFO policy [7].

The WCST at a barrier must reflect the staggered arrival of threads at the synchronisation point. To estimate the maximum delay, we need to refer to a point in the execution where all the threads expected at the barrier formerly synchronised: this point can be another barrier or the point where threads were created since we assume that they all were created simultaneously. Then we can compute:

$$WCST^t = \max_{0 \leq i < n} (w_i - w_t)$$

where w_i is the worst-case execution time of thread i from the previous collective synchronisation to the barrier. All the synchronisation-related stall times in the task can be upper bounded this way, from partial WCET estimates. They are then added to the local WCETs of corresponding basic blocks before computing the WCET of the task, that is the WCET of its master thread.

5 How Do WCRT and WCET Analyses Fit Together?

In this section, we discuss the compatibility of WCRT and WCET analyses for parallel tasks, from three different perspectives: do they consider similar task models, do they manipulate coherent timing data and do they rely on consistent assumptions?

5.1 Various Task Models

Both data parallelism (a subtask is created for each data subset) and task parallelism patterns have been considered in the contributions to the two areas (WCRT and WCET analysis). Schedulability analysis typically represents a parallel task as a DAG of subtasks (see Section 3.1) while worst-case execution time analysis examine its CFG. These different representations reflect different granularity of analysis (subtasks as scheduling units on one side, instructions on the other side).

The main difference between the two areas is the kind of interferences that are considered among subtasks. In WCRT analysis, such interferences are limited to precedence constraints (more precisely, other interferences are assumed to be accounted for in WCET estimations). It can happen that subtasks work on independent data and do not need to synchronise except for enforcing precedence (e.g. using condition signalling): the parallelisation process splits a sequential task into smaller scheduling units to facilitate task scheduling and to provide better opportunity to exploit all the available cores. WCET analysis can support such a model by evaluating each subtask independently. However, the contributions reported in this paper give emphasis to data sharing and to synchronisation delays. This obviously makes

sense for so-called *data-parallel* applications, that perform the same computation on a large set of data: often, the computation performed by one thread uses data allocated to another thread (data sharing, that may require protected access) and threads need to synchronise (e.g. at the end of each iteration of a loop) to produce correct results. Nevertheless, control loops may also exhibit data sharing and synchronisation requirements, as observed within the parMERASA European project[1] on parallel software of the automotive and construction machinery domains [5].

5.2 Different Timing Data

Usually, the results of WCET analysis (WCET estimates) are used as inputs to WCRT analysis. However, when considering parallel tasks, the two domains respectively produce and require different values. For schedulability analysis, most of the approaches require the knowledge of the WCET of each individual subtask. Worst-case execution time analysis instead produces a WCET estimation for the whole task, that is for its master thread, assuming that all the threads will be scheduled on the different cores as a single entity. Depending on the approach used for WCET analysis, extracting the WCET of each thread seems feasible (e.g. by deriving the WCET of a sub-CFG [19, 18]) or more complex [9].

5.3 Inconsistent Assumptions

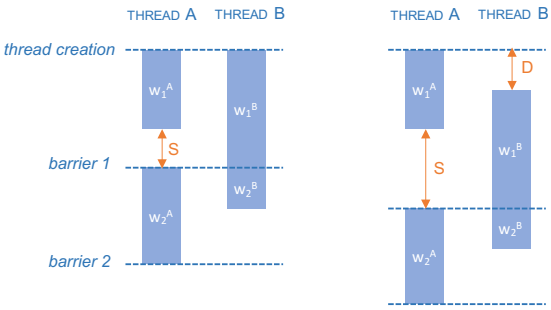
Both domains make assumptions on guarantees given by the other one. WCRT analysis postulates that WCET estimations are safe regardless of scheduling decisions. On the other hand, WCET analysis assumes that all the threads belonging to the same parallel segment are created at the same time and run simultaneously on different cores (which is possible only if the number of threads does not exceed the number of available cores). This assumption is correct with gang-style scheduling strategies [12] that allocate enough cores to execute at the same time all the threads in a parallel segment. But it may not hold with the scheduling approaches that have been discussed more recently [2, 14, 15].
A staggered scheduling of threads in a parallel segment may impair the safeness of WCET bounds. Estimating the worst-case stall time at a barrier requires determining a common former synchronisation point for all the threads expected at the barrier. As shown in Section 4.2, this can be another synchronisation barrier or the point where the threads were created (more precisely, the point where they started executing). If threads are scheduled asynchronously and in a way that is unknown at analysis time, as illustrated in Figure 6, it might be difficult to identify such collective synchronisations. On the leftmost diagram, both threads start simultaneously: the worst-case stall time can be computed as explained in Section 4.2. On the rightmost diagram, thread B starts later than thread A. Unless an upper bound on D can be specified to the WCET analysis, the WCST (S) at barrier 1 cannot be estimated.

6 Conclusion

Several recent papers in the domain of real-time systems deal with parallel tasks, either from the schedulability analysis or the worst-case execution time analysis point of view. The goal of this paper was to understand whether they all consider the same task model and

[1] `www.parmerasa.eu`

Figure 6 Simultaneous vs. staggered thread creation.



make consistent assumptions, and to determine how far the different approaches could be combined. It appears that contributions to WCRT analysis mainly focus sets of subtasks with precedence constraints, while contributions to WCET analysis are more concerned with inter-thread data sharing and progress synchronisation, assuming a trivial scheduling scheme that grants a parallel task with as many cores as threads.

From this overview, it appears that both fields need to cooperate more closely. Some information about subtasks schedules might help to refine WCET estimations, or even to make them safe. Conversely, a more detailed view of subtasks (including information produced by WCET analysis) could be exposed to scheduling approaches.

Acknowledgements The author would like to thank the organizers, the speakers and the participants of the Optimizing Real-Time Systems workshop on *Parallelization of real-time tasks*[2]: they have inspired this paper.

References

1 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 48(5), 2012.

2 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A Generalized Parallel Task Model for Recurrent Real-Time Processes. In *IEEE Real-Time Systems Symposium*, 2012.

3 Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A Unified WCET Analysis Framework for Multicore Platforms. *ACM Transactions on Embedded Computing Systems*, 13(4), 2014.

4 Robert I. Davis and Alan Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4), 2011.

5 Theo Ungerer et al. Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore. *ACM Transactions on Embedded Computing Systems*, 15(3), 2016.

6 Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J Cazorla. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In *Workshop on WCET Analysis*, 2014.

[2] `https://digicosme.lri.fr/tiki-print.php?page=GT+OVSTR`

7 Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat. Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In *Design, Automation & Test in Europe*, 2012.

8 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *IEEE Real-Time Systems Symposium*, 2006.

9 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *Workshop on WCET Analysis*, 2010.

10 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward Static Timing Analysis of Parallel Software. In *Workshop on WCET Analysis*, 2012.

11 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing Switch-Case Code with Abstract Execution. In *Workshop on WCET Analysis*, 2015.

12 Shinpei Kato and Yutaka Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *IEEE Real-Time Systems Symposium*, 2009.

13 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-Core Processors. In *IEEE Real-Time Systems Symposium*, 2010.

14 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher Gill, and Abusayeed Saifullah. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *Euromicro Conference on Real-Time Systems*, 2014.

15 Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher D. Gill, and Chenyang Lu. Global EDF Scheduling for Parallel Real-Time Tasks. *Real-Time Systems*, 51(4), 2015.

16 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *ACM/IEEE Design Automation Conference*, 1995.

17 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *IEEE Conf. on Embedded and Real-Time Computing Systems and Applications*, 2008.

18 Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel Applications. In *Workshop on WCET Analysis*, 2013.

19 Dumitru Potop-Butucaru and Isabelle Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In *Workshop on WCET Analysis*, 2013.

20 Wolfgang Puffitsch. Persistence-Based Branch Misprediction Bounds for WCET Analysis. In *ACM Symposium on Applied Computing*, 2015.

21 Manar Qamhieh, Laurent George, and Serge Midonnet. A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems. In *Intl Conference on Real-Time Networks and Systems*, 2014.

22 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing Predictability of Cache Replacement Policies. *Real-Time Systems*, 37(2), 2007.

23 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Multi-Core Real-Time Scheduling for Generalized Parallel Task Models. *Real-Time Systems*, 49(4), 2013.

24 Vivy Suhendra and Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *45th Annual Design Automation Conference*, 2008.

25 Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models.* PhD thesis, Saarland University, 2005.

26 Gan Zhi-Hua and Gu Zhi-Min. WCET-Aware Task Assignment and Cache Partitioning for WCRT Minimization on Multi-core Systems. In *Intl Symposium on Parallel Architectures, Algorithms and Programming*, 2015.