Leibniz Transactions on Embedded Systems

Volume 1 | Issue 1 | June 2014

Vol. 1, Issue 1 ISSN 2199-2002 http://www.dagstuhl.de/lites

ISSN 2199-2002

Published online and open access by
the European Design and Automation Association (EDAA) / EMbedded Systems Special Interest Group (EMSIG) and Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.
Online available at
http://www.dagstuhl.de/dagpub/2199-2002.

Publication date
June 2014

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier
10.4230/LITES-v001-i001

Aims and Scope
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Contact
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de
http://www.dagstuhl.de/lites

Contents

Regular Papers

Foreword

Alan Burns

University of York, UK
alan.burns@york.ac.uk

Digital Object Identifier 10.4230/LITES-v001-i001-a000
Published 2014-06-10

I would like to welcome all readers to the first issue of this new not-for-profit open access journal: the Leibniz Transactions on Embedded Systems (LITES). Unless you have come across this journal by accident then you will already understand the key role that embedded systems have in modern life. One can hardly think of a single human activity that is not underpinned by such systems; transport, entertainment, supply lines for supermarkets, health care and drug production, energy production and transmission, robotic manufacturing, control systems and communication media of all kinds are now dependent on the fusion of embedded hardware and software. For researchers in this domain this provides great opportunities but also responsibilities. We need to make sure that society can justifiable rely on technology that is increasing beyond the understanding of most ordinary people. Computer-based technologies have been described as modern magic; it follows that we are therefore magicians. But the spells we cast must be based on sound principles, solid theory and demonstrable performance.

One of the influences that embedded and other IT technology has had in the last decade is in publishing itself. Online services are now the norm. And early and open access to publicly funded research is now rightly demanded by Government bodies and related funding councils. This new journal has been created to meet this challenge. All papers are open access, with copyright being retained by the authors. Moreover, only a small fee is charged to authors due to low operational overheads and the support of Google and the Klaus Tschira Stiftung. But the lack of a physical page limit in an online-only journal does not mean that quality is undermined. All papers are thoroughly reviewed, with only the best work, in terms of originality and rigour, being accepted. Our aim is to evolve an excellent and effective venue for publish scholarly articles. To help achieve this aim LITES benefits greatly from having the name and reputation of Schloss Dagstuhl behind it.

The volume of research material produced world-wide relating to embedded systems has lead to the spawning of many conferences and workshops, special issues and focused publications. In LITES we intend to cater for the broadest collection of relevant topics. We currently have subject editors to cover: the design, implementation, verification, and testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; applications in the avionics, automotive, telecommunication, medical and production domains; cyber-physical systems; high performance and real-time embedded systems; and hybrid systems. This is an impressive list, but it is not exhaustive. New areas will emerge and new editors will be appointed.

LITES obtains its governance from EDAA (European Design and Automation Association) and EMSIG (Embedded Systems Special Interest Group) as a joint endeavour with Schloss Dagstuhl. EDAA/EMSIG appoint the Editor-in-Chief (EiC) and the subject area editors. The terms for editors is four years, renewable once. All editorial work is done voluntarily.

The first few issues of the journal will contain standard papers that have been through the review process. Later, comments on previously published papers will be allowed and commentaries

included that will help the reader trace forward the influence of each paper. Comments will be reviewed; commentaries will just need to be passed by the EiC.

I hope that as a reader you will find the papers in this journal of interest and often inspirational. As a researcher I hope you will consider it as a worthy place to entrust your work. All the editorial team will work towards building up the reputation of the journal. I hope the community at large will be part of that journey.

I am proud to be the founding EiC of this journal, but I promise not to include editorials in future issues. The papers are quite capable of introducing themselves.

Alan Burns

A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays

Will Lunniss[1], Sebastian Altmeyer[2], and Robert I. Davis[1]

1 **Department of Computer Science**
University of York
York, UK
{wl510,rob.davis}@york.ac.uk

2 **Computer Systems Architecture Group**
University of Amsterdam
Amsterdam, The Netherlands
altmeyer@uva.nl

Abstract

In multitasking real-time systems, the choice of scheduling algorithm is an important factor to ensure that response time requirements are met while maximising limited system resources. Two popular scheduling algorithms include *fixed priority* (FP) and *earliest deadline first* (EDF). While they have been studied in great detail before, they have not been compared when taking into account *cache related pre-emption delays* (CRPD). Memory and cache are split into a number of blocks containing instructions and data. During a pre-emption, cache blocks from the pre-empting task can evict those of the pre-empted task. When the pre-empted task is resumed, if it then has to re-load the evicted blocks, CRPD are introduced which then affect the schedulability of the task.

In this paper we compare FP and EDF scheduling algorithms in the presence of CRPD using the state-of-the-art CRPD analysis. We find that when CRPD is accounted for, the performance gains offered by EDF over FP, while still notable, are diminished. Furthermore, we find that under scenarios that cause relatively high CRPD, task layout optimisation techniques can be applied to allow FP to schedule tasksets at a similar processor utilisation to EDF. Thus making the choice of the task layout in memory as important as the choice of scheduling algorithm. This is very relevant for industry, as it is much cheaper and simpler to adjust the task layout through the linker than it is to switch the scheduling algorithm.

2012 ACM Subject Classification Software and its engineering, Software organization and properties, Software functional properties, Correctness, Real-time schedulability

Keywords and phrases Real-Time Systems, Fixed Priority, EDF, Pre-emptive, Scheduling, Cache Related Pre-emption Delays

Digital Object Identifier 10.4230/LITES-v001-i001-a001

Received 2013-08-22 **Accepted** 2014-03-03 **Published** 2014-04-14

1 Introduction

Today's real-time applications are complex systems built up of a large number of interacting tasks running on hardware with non-deterministic performance enhancing features such as caches, pipelines and out-of-order execution. To manage the available resources efficiently, scheduling algorithms are used to determine which task should run and at which time in order to fulfil the functional and temporal requirements of the system. The scheduling algorithms are often *pre-empting*, in that they allow important tasks to interrupt less important tasks before they have finished. Two popular scheduling algorithms for real-time systems are *fixed priority* (FP) and

earliest deadline first (EDF). FP scheduling uses statically defined priorities to run the task with the highest priority first. In comparison, EDF is a dynamic scheduling algorithm that schedules the task with the earliest absolute deadline first. EDF is an optimal scheduling algorithm without pre-emption costs, whereas FP is not, and is therefore typically able to schedule tasksets at a higher processor utilisation than FP [20]. However, despite the significant performance benefits over FP, EDF is not widely used in commercial real-time operating systems.

In real-time systems, and especially hard real-time systems, the schedulability of each task must be known in order to verify that the timing requirements will be met. The schedulability of a taskset is determined using information about the scheduling algorithm, the arrival pattern of tasks and the tasks' *worst-case execution times*. Worst-case execution times are typically obtained assuming no pre-emption. However, in pre-emptive multi-tasking systems, caches introduce additional *cache related pre-emption delays* (CRPD) caused by the need to re-fetch blocks belonging to the pre-empted task which were evicted from the cache by the pre-empting task. These CRPD effectively increase the worst-case execution time of the tasks. It is therefore important to be able to calculate, and therefore account for, CRPD when determining if a system is schedulable or not.

In 2005, Buttazzo [13] performed a detailed study of FP and EDF scheduling. This work covered both schedulability under a variety of scenarios, in addition to practical implementation considerations. Results showed that the FP scheduling algorithm introduces more pre-emptions than EDF, especially at high processor utilisation levels. This leads to FP performing worse than EDF. Yet, FP has an advantage over EDF, in that it is generally simpler to implement in commercial kernels which do not provide explicit support for timing constraints. Despite being a very detailed study, these comparisons where done under the assumption that there were no pre-emption costs due to CRPD.

In this paper we build on the work by Buttazzo [13] and use state of the art CRPD analysis for FP [3] and EDF [22] to perform a comprehensive study of these two popular scheduling algorithms when accounting for CRPD.

1.1 Related Work on CRPD

Analysis of CRPD uses the concept of *useful cache blocks* (UCBs) and *evicting cache blocks* (ECBs) based on the work by Lee et al. [18]. Any memory block that is accessed by a task while executing is classified as an ECB, as accessing that block may evict a cache block of a pre-empted task. Out of the set of ECBs, some of them may also be UCBs. A memory block m is classified as a UCB at program point $\mathcal{P}$, if (i) m may be cached at $\mathcal{P}$ and (ii) m may be reused at program point $\mathcal{Q}$ that may be reached from $\mathcal{P}$ without eviction of m on this path. In the case of a pre-emption at program point $\mathcal{P}$, only the memory blocks that are (i) in cache and (ii) will be reused, may cause additional reloads. The maximum possible pre-emption cost for a task is determined by the program point with the highest number of UCBs. For each subsequent pre-emption, the program point with the next smallest number of UCBs can be considered. Altmeyer and Burguière [1] presented a tighter definition of UCBs however, we only need the basic concept for this paper.

Depending on the approach used, the CRPD analysis combines the UCBs belonging to the pre-empted task(s) with the ECBs of the pre-empting task(s). Using this information, the total number of blocks that are evicted, which must then be reloaded after the pre-emption, can be calculated and combined with the cost of reloading a block to then give the CRPD.

A number of approaches have been developed for calculating the CRPD when using FP pre-emptive scheduling. They include Lee et al. [18] UCB-Only approach, which considers just the pre-empted task(s), and Busquets et al. [12] ECB-Only approach which considers just the pre-empting task. Approaches that consider the pre-empted and pre-empting task(s) include Tan and Mooney [26] UCB-Union approach, Altmeyer et al. [2] ECB-Union approach, and an alternative

approach by Staschulat et al. [25]. Finally, there are advanced multiset based approaches that consider the pre-empted and pre-empting task(s) by Altmeyer et al. [3], ECB-Union Multiset, UCB-Union Multiset, and a combined multiset approach.

There has been less work towards developing CRPD analysis for EDF pre-emptive scheduling. In 2007, Ju et al. [17] considered the intersection of the pre-empted task's UCBs with the pre-empting task's ECBs. However, this method for handling nested pre-emptions can lead to significant pessimism as each pair of tasks is considered separately. In 2013, Lunniss et al. [22] adapted a number of approaches for calculating CRPD for FP to work with EDF. Including the ECB-Only, UCB-Only, UCB-Union, ECB-Union, ECB-Union Multiset, UCB-Union Multiset and combined multiset CRPD analysis for FP given by Busquets et al. [12], Lee et al. [18], Tan and Mooney [26], and Altmeyer et al. [2, 3].

A different methodology was used by Bastoni et al. [8]. Instead of focusing on how to calculate an upper bound on the CRPD, they used measurements on real hardware to estimate a lower bound on the CRPD and *cache related migration delays* for data caches in a multi-processor system.

CRPD can have a significant effect on schedulability, and can also vary dramatically depending on a number of factors. In particular, the CRPD is highly dependent on how tasks are placed in cache. As the layout of tasks in memory determines how they are positioned in cache, choosing a sensible layout can have a big impact on the CRPD caused due to pre-emptions. In 2012, Lunniss et al. [21] presented an approach that uses a *Simulated Annealing* algorithm to optimise the layout of tasks to increase system schedulability when using FP scheduling.

1.2 Organisation

The paper is organised as follows. Section 2 introduces the system model, terminology and notation used. Existing schedulability tests and CRPD analysis are outlined in Section 3 for FP scheduling, and in Section 4 for EDF scheduling. Section 5 briefly covers optimising task layout to reduce CRPD. Section 6 compares FP and EDF with CRPD analysis using a set of case studies. In Section 7, we investigate the effect of a variety of configuration parameters in a series of evaluations using synthetic tasksets. Finally, we conclude in Section 8.

2 System Model, Terminology and Notation

This section describes the system model, terminology, and notation used in the rest of the paper.

We assume a single processor system, running a statically defined taskset under either pre-emptive FP or pre-emptive EDF scheduling. The system comprises a taskset Γ made up of a fixed number of tasks $(\tau_1, \ldots, \tau_n)$ where n is a positive integer. In the case of FP, each task has a unique fixed priority and the priority of task τ_i, is i, where a priority of 1 is the highest and n is the lowest. Each task, τ_i may produce a potentially infinite stream of jobs that are separated by a minimum inter-arrival time or period T_i. Each task has a relative deadline D_i, and each job of a task has an absolute deadline d_i which is D_i after it is released. In the case of EDF, each task has a unique task index ordered by relative deadline from smallest to largest. In the case of a tie when assigning the unique task indices, an arbitrary choice is made. Each task also has a worst case execution time C_i (determined for non-pre-emptive execution). In this paper, we consider tasks with *constrained* deadlines. (Task deadlines may be referred to as *constrained* deadlines, i.e. $D_i \leq T_i$ or *implicit* i.e. $D_i = T_i$). We assume a discrete time model. We define T_{max} as the largest period of any task in the taskset, and similarly D_{max} as the largest relative deadline of any task in the taskset. Each task has a utilisation U_i, where $U_i = C_i/T_i$, and each taskset has a utilisation U which is equal to the sum of its tasks' utilisations.

A taskset is said to be *schedulable* with respect to a scheduling algorithm if all valid sequences of jobs generated by the taskset can be scheduled by the algorithm without any missed deadlines. A taskset is *feasible* if there exists some scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the taskset without any missed deadlines. A scheduling algorithm is said to be *optimal* with respect to a task model if it can schedule all of the feasible tasksets that comply with the task model.

Each task τ_i has a set of UCBs, UCB_i and a set of ECBs, ECB_i represented by a set of integers. If for example, task τ_1 contains 4 ECBs, where the second and fourth ECBs are also UCBs, these can be represented using $\text{ECB}_1 = \{1, 2, 3, 4\}$ and $\text{UCB}_1 = \{2, 4\}$. The *block reload time* (BRT) is the time taken to load a block from memory into cache. This cost is incurred every time that a UCB has to be reloaded after a pre-emption. We assume that the remaining context switch costs, i.e., pipeline and scheduler related costs are subsumed in the execution time bound of each task. Furthermore, we assume that the OS resides in a different cache partition and therefore scheduler operations do not cause CRPD.

We use the term *cache utilisation* to describe the ratio of the total size of the tasks to the size of the cache. A cache utilisation of 1 means that the tasks fit exactly in the cache, whereas a cache utilisation of 5 means the total size of the tasks is 5 times the size of the cache.

We focus on instruction only caches. In the case of data caches, the analysis would either require a write-through cache or further extension in order to be applied to write-back caches. We assume that tasks do not share any code. We also assume a direct mapped cache, but the work extends to set-associative caches with the LRU replacement policy[1]. In the case of set-associative LRU caches, a single cache-set may contain several UCBs. For example, $\text{UCB}_1 = \{2, 2, 4\}$ means that task τ_1 has two UCBs in cache-set 2 and one UCB in cache set 4. As one ECB suffices to evict all UCBs of the same cache-set, multiple accesses to the same set by the pre-empting task do not appear in the set of ECBs. A bound on the CRPD in the case of LRU caches due to task τ_j directly pre-empting τ_i is thus given by the intersection $\text{UCB}_i \cap' \text{ECB}_j = \{m | m \in \text{UCB}_i : m \in \text{ECB}_j\}$, where the result is a multiset that contains each element from UCB_i if it is also in ECB_j. A precise computation of CRPD in the case of LRU caches is given in Altmeyer et al. [4]. The equations provided in this paper can be applied to set-associative LRU caches with the above adaptation to the set-intersection.

3 CRPD Analysis for FP Scheduling

In this section, we give an overview of FP scheduling and schedulability analysis for it. We then cover the state of the art CRPD analysis for FP scheduling, by Altmeyer et al. described in detail in [3].

Under FP scheduling, the sets of tasks that can pre-empt each other are based on the statically assigned fixed task priorities. Using the fixed priorities, we can define the following sets of tasks for determining which tasks can pre-empt each other. $\text{hp}(i)$ and $\text{lp}(i)$ are the sets of tasks with higher and lower priorities than task τ_i, and $\text{hep}(i)$ and $\text{lep}(i)$ are the sets containing tasks with higher or equal and lower or equal priorities to task τ_i. Additionally, $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$ is used to represent all tasks that can have CRPD caused by task τ_j pre-empting them, which affects the response time of task τ_i. In other words, it is the set of tasks that may be pre-empted by task τ_j and have at least the priority of task τ_i.

[1] The concept of UCBs and ECBs cannot be applied to the FIFO or PLRU replacement policies as shown by Burguière [11].

Schedulability tests are used to determine if a taskset is schedulable, i.e. all the tasks will meet their deadlines given the worst-case pattern of arrivals and execution. For a given taskset, the response time R_i for each task τ_i, can be calculated and compared against the tasks' deadline, D_i. If every task in the taskset meets its deadline, then the taskset is schedulable. The equation used to calculate R_i is defined as [5]:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil (C_j) \,. \tag{1}$$

Equation (1) can be solved using fixed point iteration. Iteration continues until either $R_i^{\alpha+1} > D_i$ in which case the task is unschedulable, or until $R_i^{\alpha+1} = R_i^{\alpha}$ in which case the task is schedulable and has a worst-case response time of R_i^{α}.

Note the convergence of (1) may be speeded up using the techniques described in [14].

3.1 CRPD Analysis

To account for the CRPD, a component $\gamma_{i,j}$ is introduced into (1). There are a number of different methods that can be used to compute $\gamma_{i,j}$ described by Altmeyer et al. in [3]. Depending on the method used, $\gamma_{i,j}$ represents either a single pre-emption, or multiple pre-emptions and is calculated using the cost incurred when reloading a block, the *block reload time* (BRT), multiplied by the number of blocks which may need to be reloaded after each pre-emption.

We will now summarise the *combined multiset* approach, which has been shown to dominate all other approaches [3]. For worked examples of the analysis, see Section 4 ECB-Union and Multiset Approaches of Altmeyer et al. [3].

In the combined multiset approach, $\gamma_{i,j}$ represents the total cost of all pre-emptions due to jobs of task τ_j executing within the response time of task τ_i. Incorporating $\gamma_{i,j}$ into (1) gives a revised equation for R_i:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil C_j + \gamma_{i,j} \right) . \tag{2}$$

$\gamma_{i,j}$ is then calculated using two separate approaches, the *UCB-Union multiset*, and *ECB-Union multiset* which are described below. The key concept behind them is to calculate the cost of each individual pre-emption by jobs of task τ_j that could occur within the response time of task τ_i. By calculating the cost of each pre-emption, the analysis is able to account for the fact that intermediate tasks in a nested pre-emption will often be pre-empted less than the lowest priority task. Consider the following example with three tasks shown in Figure 1.

In the example, the total cost of all jobs of task τ_1 pre-empting task τ_3 within the response time of task τ_3 is equal to the cost of task τ_1 pre-empting task τ_2 and task τ_3 once (nested pre-emption), and task τ_3 on its own twice. It is therefore important to recognise that the cost of one task pre-empting another is highly dependent on any intermediate tasks that may be involved in a nested pre-emption. To calculate the number of pre-emptions, we use $E_j(R_i)$ to denote the

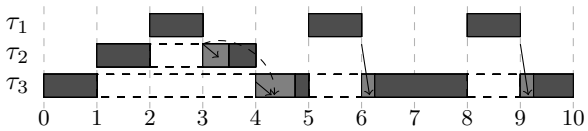

Figure 1 The pre-emption cost of all jobs of task τ_1 pre-empting task τ_2 can only contribute once to the total pre-emption cost of task τ_1 pre-empting τ_3 during the response time of τ_3.

maximum number of jobs of task τ_j that can execute during the response time, R_i, of task τ_i. For our model, $E_j(R_i) = \lceil R_i/T_j \rceil$.

3.1.1 ECB-Union Multiset

The ECB-Union multiset approach computes the union of all ECBs that may affect a pre-empted task during a pre-emption by task τ_j. Specifically, it accounts for nested pre-emptions by assuming that task τ_j has already been pre-empted by all tasks of a higher priority.

The first step is to calculate the number of UCBs that task τ_j could evict when pre-empting an intermediate task, τ_k. This is given by calculating the intersection of the UCBs of the pre-empted task, task τ_k, with the set of ECBs belonging to the pre-empting tasks $\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h$ to give:

$$\left| \text{UCB}_k \cap \left(\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| . \tag{3}$$

Note $h \in \text{hp}(j) \cup \{j\}$ is used to account for the case when tasks can share priorities.

The ECB-Union multiset approach recognises that task τ_j cannot pre-empt each intermediate task τ_k more than $E_j(R_k)E_k(R_i)$ times during the response time of task τ_i. Therefore, the next step is to form a multiset $M_{i,j}$ that contains the cost of task τ_j pre-empting task τ_k (3) repeated $E_j(R_k)E_k(R_i)$ times, for each task $\tau_k \in \text{aff}(i,j)$ hence:

$$M_{i,j} = \bigcup_{k \in \text{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \left| \text{UCB}_k \cap \left(\bigcup_{h \in \text{hp}(j) \cup \{j\}} \text{ECB}_h \right) \right| \right) . \tag{4}$$

As only $E_j(R_i)$ jobs of task τ_j can execute during the response time of task τ_i, the maximum CRPD is obtained by summing the $E_j(R_i)$ largest pre-emptions, i.e. the $E_j(R_i)$ largest values in $M_{i,j}$:

$$\gamma_{i,j}^{\text{ecb-m}} = \text{BRT} \cdot \sum_{l=1}^{E_j(R_i)} M_{i,j}^l , \tag{5}$$

where $M_{i,j}^l$ is the l^{th} largest integer value from the multiset $M_{i,j}$.

3.1.2 UCB-Union Multiset

The UCB-Union multiset approach accounts for the effects of nested pre-emptions by assuming that the UCBs of any tasks that could be pre-empted, including nested pre-emptions, by task τ_j are evicted by the ECBs of task τ_j. The first step is to form a multiset $M_{i,j}^{\text{ucb}}$ containing $E_j(R_k)E_k(R_i)$ copies of the UCB_k of each task $\tau_k \in \text{aff}(i,j)$ that could be pre-empted by task τ_j and has at least the priority of task τ_i. This multiset reflects the fact that jobs of task τ_j cannot evict the UCBs of jobs of task τ_k within the response time of task τ_i more than $E_j(R_k)E_k(R_i)$ times. Hence:

$$M_{i,j}^{\text{ucb}} = \bigcup_{k \in \text{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \text{UCB}_k \right) . \tag{6}$$

The second step is to form a separate multiset $M_{i,j}^{\text{ecb}}$ containing $E_j(R_i)$ copies of the ECB_j of task τ_j. This multiset reflects the fact that there can be no more than $E_j(R_i)$ jobs of task τ_j

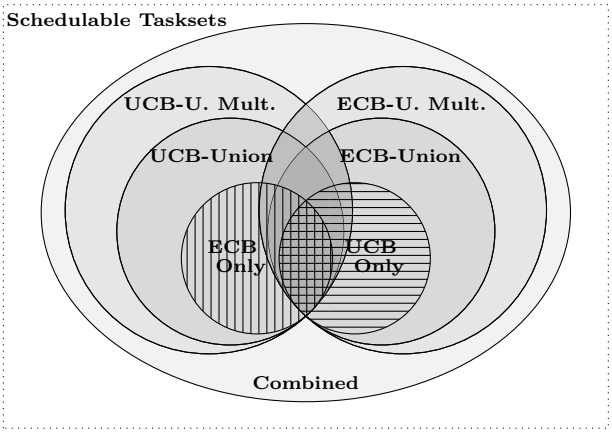

Figure 2 Venn diagram showing the relationship between different approaches for CPRD analysis under FP scheduling.

within the response time of task τ_i, each of which can evict cache blocks in the set ECB_j:

$$M_{i,j}^{\text{ecb}} = \bigcup_{E_j(R_i)} (\text{ECB}_j)\,. \tag{7}$$

$\gamma_{i,j}^{\text{ucb-m}}$ is then given by the size of the multiset intersection between $M_{i,j}^{\text{ucb}}$ and $M_{i,j}^{\text{ecb}}$:

$$\gamma_{i,j}^{\text{ucb-m}} = \text{BRT} \cdot \left| M_{i,j}^{\text{ucb}} \cap M_{i,j}^{\text{ecb}} \right|\,. \tag{8}$$

3.1.3 Combined Multiset

The ECB-Union multiset and UCB-Union multiset approaches are incomparable, meaning that each approach can find different sets of tasksets schedulable. Because of this property, they can be combined together to form a combined approach:

$$R_i = \min(R_i^{\text{ucb-m}}, R_i^{\text{ecb-m}})\,. \tag{9}$$

The response time for every task is calculated using each approach and then the minimum is taken, because of this, the combined approach can deem some tasksets schedulable that are not schedulable by either approach on its own.

3.2 Comparison of Approaches

Figure 2 shows a Venn diagram that conveys the relationship between a number of different approaches for calculating CRPD under FP scheduling [3]. However, it does not include the method by Staschulat et al. [25] because it is incomparable to them. Specifically, while it typically deems a lower number of tasksets schedulable, it could potentially find a taskset schedulable that is not schedulable by any of the other approaches. Aside from the approach by Staschulat et al. [25], it can be seen that the combined multiset approach dominates all other approaches. See Altmeyer et al. [3] for a detailed comparison between each approach.

4 CRPD Analysis for EDF Scheduling

In this section, we give an overview of EDF scheduling and schedulability analysis for it. We then cover the state of the art CRPD analysis for EDF scheduling, by Lunniss et al. [22].

EDF is a dynamic scheduling algorithm which always schedules the job with the earliest absolute deadline first. In pre-emptive EDF, any time a job arrives with an earlier absolute deadline than the current running job, it will pre-empt the current job. When a job completes its execution, the EDF scheduler chooses the pending job with the earliest absolute deadline to execute next.

In 1973, Liu and Layland [20] gave a necessary and sufficient schedulability test that indicates whether a taskset is schedulable under EDF iff $U \leq 1$, under the assumption that all tasks have implicit deadlines ($D_i = T_i$). In the case where $D_i \leq T_i$ this test is still necessary, but is no longer sufficient.

In 1974, Dertouzos [15] proved EDF to be optimal among all scheduling algorithms on a uniprocessor, in the sense that if a taskset cannot be scheduled by pre-emptive EDF, then this taskset cannot be scheduled by any algorithm.

In 1980, Leung and Merrill [19] showed that a set of periodic tasks is schedulable under EDF iff all absolute deadlines in the interval $[0, \max\{s_i\} + 2H]$ are met, where s_i is the start time of task τ_i, $\min\{s_i\} = 0$, and H is the hyperperiod (least common multiple) of all tasks' periods.

In 1990 Baruah et al. [6, 7] extended Leung and Merrill's work [19] to sporadic tasksets. They introduced $h(t)$, the processor demand function, which denotes the maximum execution time requirement of all tasks' jobs which have both their arrival times and their deadlines in a contiguous interval of length t. Using this they showed that a taskset is schedulable iff $\forall t > 0, h(t) \leq t$ where $h(t)$ is defined as:

$$h(t) = \sum_{i=1} \max \left\{ 0, 1, + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right\} C_i \,. \tag{10}$$

Examining (10), it can be seen that $h(t)$ can only change when t is equal to an absolute deadline, which restricts the number of values of t that need to be checked. In order to place an upper bound on t, and therefore the number of calculations of $h(t)$, the minimum interval in which it can be guaranteed that an unschedulable taskset will be shown to be unschedulable must be found. For a general taskset with arbitrary deadlines t can be bounded by L_a [16]:

$$L_a = \max \left\{ D_1, \ldots, D_n, \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} \right\} . \tag{11}$$

Spuri [24] and Ripoll et al. [23] showed that an alternative bound L_b, given by the length of the synchronous busy period can be used. Where L_b is computed by solving the following equation using fixed point iteration:

$$w^{\alpha+1} = \sum_{i=1}^{n} \left\lceil \frac{w^{\alpha}}{T_i} \right\rceil C_i \,. \tag{12}$$

There is no direct relationship between L_a and L_b, which enables t to be bounded by $L = \min(L_a, L_b)$. Combined with the knowledge that $h(t)$ can only change at an absolute deadline, a taskset is therefore schedulable under EDF iff $U \leq 1$ and:

$$\forall t \in Q : h(t) \leq t \,, \tag{13}$$

where Q is defined as:

$$Q = \{ d_k \mid d_k = kT_i + D_i \wedge d_k < \min(L_a, L_b), k \in \mathbb{N} \} \,. \tag{14}$$

In 2009, Zhang and Burns [27] presented their Quick convergence Processor-demand Analysis (QPA) algorithm which exploits the monotonicity of $h(t)$. QPA determines schedulability by starting with a value of t that is close to L, and then iterating back towards 0 checking a significantly smaller number of values of t than would otherwise be required.

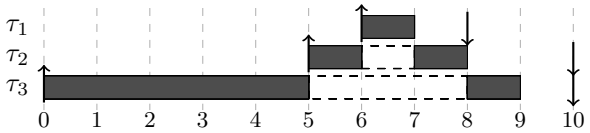

Figure 3 Example schedule showing how the scheduler chooses which task should execute. Task τ_3 is released at $t = 0$. At $t = 5$, task τ_2 is released, pre-empting τ_3 as although it has the same absolute deadline, it has a lower task index. At $t = 6$, task τ_1 is released, pre-empting task τ_2. At $t = 7$, τ_1 completes, the scheduler then chooses to resume task τ_2 as although it has the same absolute deadline as task τ_3, it has the lower task index.

4.1 CRPD Analysis

Due to the undefined behaviour of EDF when two or more jobs have the same absolute deadline, an assumption needs to be made before we can tightly calculate CRPD for EDF. In the case where two or more jobs have the same absolute deadline, Lunniss et al. [22] assume the scheduler always picks the job belonging to the task with the lowest unique task index, see Figure 3. This has the benefit of minimising the number of pre-emptions. In the case where jobs of two tasks have the same absolute and relative deadlines, it ensures that they cannot pre-empt each other. Furthermore, it ensures that after a pre-emption, the task that was pre-empted last is resumed first.

Following the analysis of Lunniss et al. [22], we now define a number of terms with respect to EDF scheduling. Some of the terms are also present in the analysis for FP, but have slightly different meanings under EDF. Assuming any task τ_j with a relative deadline $D_j < D_i$ can pre-empt task τ_i, the set of tasks that may have a higher priority, and can pre-empt task τ_i, under EDF is:

$$\text{hp}(i) = \{\tau_j \in \Gamma \mid D_j < D_i\}\,. \tag{15}$$

The set of tasks that can be pre-empted by jobs of task τ_j in an interval of length t, $\text{aff}(t, j)$ is based on the relative deadlines of the tasks. It captures all of the tasks whose relative deadlines are greater than the relative deadline of task τ_j excluding tasks whose deadlines are larger than t as they do not need to be included when calculating $h(t)$. This gives:

$$\text{aff}(t, i) = \{\tau_j \in \Gamma \mid t \geq D_i > D_j\}\,. \tag{16}$$

To determine how many pre-emptions can occur, we use $P_j(D_i)$ to denote the maximum number of jobs of task τ_j that can pre-empt a single job of task τ_i:

$$P_j(D_i) = \max\left(0, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right)\,. \tag{17}$$

Finally, we also use $E_j(t)$ to denote the maximum number of jobs of task τ_j that can have both their release times and their deadlines in an interval of length t:

$$E_j(t) = \max\left(0, \left\lfloor \frac{t - D_j}{T_j} \right\rfloor\right)\,. \tag{18}$$

We now summarise CRPD analysis for EDF by Lunniss et al. [22] using the *combined multiset* approach as it has been shown to dominate all other approaches for calculating CRPD for EDF. This approach is based on the combined multiset approach for FP as described in Section 3.1, and as such the equations and the intuition behind them are similar. The difference is to do with

which tasks pre-empt each other and the timeframe used to determine which jobs to include in the calculation.

CRPD analysis can be integrated into the EDF schedulability test by introducing an additional parameter, $\gamma_{t,j}$ to represent the CRPD. In this case, $\gamma_{t,j}$ represents the cost of the maximum number $E_j(t)$ of pre-emptions by jobs of task τ_j that have their release times and absolute deadlines in an interval of length t. It is therefore included in (10) as follows:

$$h(t) = \sum_{j=1}^{n} \left(\max \left\{ 0, \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \right\} C_j + \gamma_{t,j} \right) . \tag{19}$$

$\gamma_{t,j}$ can then be calculated using two different methods and the lowest value of the two used to calculate the processor demand. These methods calculate the cost of each possible individual pre-emption by task τ_j that could occur during an interval of length t.

4.1.1 ECB-Union Multiset

The ECB-Union multiset approach computes the union of all ECBs that may affect a pre-empted task during a pre-emption by task τ_j. Specifically, it accounts for nested pre-emptions by assuming that task τ_j has already been pre-empted by all other tasks that may pre-empt it. The first step is to form a multiset $M_{t,j}$ that contains the cost:

$$\left| \mathrm{UCB}_k \cap \left(\bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h \right) \right| \tag{20}$$

of task τ_j pre-empting task τ_k repeated $P_j(D_k)E_k(t)$ times, for each task $\tau_k \in \mathrm{aff}(t, j)$. Hence:

$$M_{t,j} = \bigcup_{k \in \mathrm{aff}(t,j)} \left(\bigcup_{P_j(D_k)E_k(t)} \left| \mathrm{UCB}_k \cap \left(\bigcup_{h \in \mathrm{hp}(j) \cup \{j\}} \mathrm{ECB}_h \right) \right| \right) . \tag{21}$$

As there are only $E_j(t)$ jobs of task τ_j with release times and deadlines in an interval of length t, the maximum CRPD is obtained by summing the $E_j(t)$ largest values in $M_{t,j}$:

$$\gamma_{t,j}^{\text{ecb-m}} = \mathrm{BRT} \cdot \sum_{l=1}^{E_j(t)} M_{t,j}^{l} , \tag{22}$$

where $M_{t,j}^{l}$ is the l^{th} largest integer value from the multiset $M_{t,j}$.

4.1.2 UCB-Union Multiset Approach

The UCB-Union multiset approach accounts for the effects of nested pre-emptions by assuming that the UCBs of any tasks that could be pre-empted, including nested pre-emptions, by task τ_j are evicted by the ECBs of task τ_j. The first step is to form a multiset $M_{t,j}^{\text{ucb}}$ containing $P_j(D_k)E_k(t)$ copies of the UCB_k of each task $\tau_k \in \mathrm{aff}(t, j)$. This multiset reflects the fact that jobs of task τ_j cannot evict the UCBs of jobs of task τ_k that have both their release times and deadlines in an interval of length t more than $P_j(D_k)E_k(t)$ times. Hence:

$$M_{t,j}^{\text{ucb}} = \bigcup_{k \in \mathrm{aff}(t,j)} \left(\bigcup_{P_j(D_k)E_k(t)} \mathrm{UCB}_k \right) . \tag{23}$$

The second step is to form a separate multiset $M^{\text{ecb}}_{t,j}$ containing $E_j(t)$ copies of the ECB_j of task τ_j. This multiset reflects the fact that there are at most $E_j(t)$ jobs of task τ_j that have their release times and deadlines in an interval of length t, each of which can evict cache blocks in the set ECB_j:

$$M^{\text{ecb}}_{t,j} = \bigcup_{E_j(t)} (\text{ECB}_j) \,. \tag{24}$$

$\gamma^{\text{ucb-m}}_{t,j}$ is then given by the size of the multi-set intersection between $M^{\text{ucb}}_{t,j}$ and $M^{\text{ecb}}_{t,j}$:

$$\gamma^{\text{ucb-m}}_{t,j} = \text{BRT} \cdot \left| M^{\text{ucb}}_{t,j} \cap M^{\text{ecb}}_{t,j} \right| \,. \tag{25}$$

4.1.3 Combined Multiset Approach

The ECB-Union Multiset and UCB-Union Multiset approaches provide upper bounds that are incomparable, therefore, $h(t)$ can be calculated at each stage of the QPA algorithm using both approaches and the minimum value taken to form a combined approach:

$$h(t) = \min \left(h(t)^{\text{ucb-m}}, h(t)^{\text{ecb-m}} \right) \,. \tag{26}$$

As the processor demand is calculated using each approach, for each interval t, the combined approach can deem some tasksets schedulable that are not schedulable by either approach on its own.

4.1.4 Effect on Task Utilisation and $h(t)$ Calculation

As the multiset approaches effectively inflate the execution time of task τ_j by the CRPD that it can cause in an interval of length t, the upper bound L, used for calculating the processor demand $h(t)$, must be adjusted. This is achieved by calculating an upper bound on the utilisation due to CRPD that is valid for all intervals of length greater than some value L_c. This CRPD utilisation value is then used to inflate the taskset utilisation and thus compute an upper bound L_d on the maximum length of the synchronous busy period. This upper bound is valid provided that it is greater than L_c, otherwise the actual maximum length of the busy period may lie somewhere in the interval $[L_d, L_c]$, hence we can use $\max(L_c, L_d)$ as a bound.

The first step is to assign $t = L_c = 100T_{\max}$ which limits the overestimation of the CRPD utilisation $U^\gamma = \gamma_t / t$ to at most 1%. Next, γ_t is calculated using (22) for ECB-Union Multiset and (25) for UCB-Union Multiset. However, in (21) and (23) & (24), $E^{\max}_x(t)$ is substituted for $E_x(t)$ to ensure that the computed value of U^γ is a valid upper bound for all intervals of length $t \geq L_c$:

$$E^{\max}_x(t) = \max \left(0, 1 + \left\lceil \frac{t - D_x}{T_x} \right\rceil \right) \,. \tag{27}$$

If $U + U^\gamma \geq 1$, then the taskset is deemed unschedulable, otherwise an upper bound on the length of the busy period can be computed via a modified version of (12):

$$w^{\alpha+1} \leq \sum_{\forall j} \left(\frac{w^\alpha}{T_j} + 1 \right) C_j + w^\alpha U^\gamma \tag{28}$$

rearranged to give:

$$w \leq \frac{1}{(1 - (U + U^\gamma))} \sum_{\forall j} U_j T_j \,. \tag{29}$$

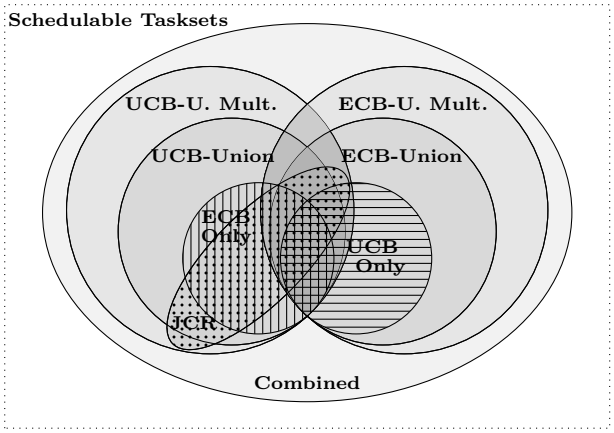

Figure 4 Venn diagram showing the relationship between different approaches for CPRD analysis under EDF scheduling.

Then, substituting in $T_{\max}$ for each value of T_j the upper bound is given by:

$$L_d = \frac{U \cdot T_{\max}}{(1 - (U + U^{\gamma}))} \,. \tag{30}$$

Finally, $L = \max(L_c, L_d)$ can then be used as the maximum value of t to check in the EDF schedulability test.

4.2 Comparison of Approaches

Figure 4 shows a Venn diagram that conveys the relationship between the different approaches for calculating CRPD under EDF scheduling [22]. Note that JCR represents the approach of Ju et al. [17]. It can be seen that the combined multiset approach dominates all other approaches, see Lunniss et al. [22] for detailed comparisons of each approach.

5 Task Layout

The layout of tasks in memory determines how they are positioned in cache, which then affects the CRPD that occurs during pre-emptions. Figure 5 shows an example layout of five tasks in cache. If scheduled under FP, task τ_1 has the highest priority, so its UCBs can never be evicted as it cannot be pre-empted. Task τ_2 and τ_3's UCBs are safe from eviction as they are not mapped to the same location in cache as higher priority task's ECBs. However, task τ_4's UCBs could be evicted by task τ_1, and τ_5's UCBs could be evicted by task τ_1, τ_2 or τ_4. This layout could be improved by shifting task τ_5 so that its UCBs can only be evicted by task τ_3.

In 2012, Lunniss et al. [21] presented an approach that uses *Simulated Annealing* to optimise the layout of tasks to increase system schedulability. It does so by changing the order of tasks in memory, which can be implemented in practice by presenting the tasks to the linker in the desired order. The approach is driven by the schedulability of the taskset, favouring layouts that allow the taskset to be scheduled at a higher utilisation. While this approach was originally used for FP scheduling, it can also be applied to the EDF scheduling algorithm by switching the schedulability test used. We therefore use this approach to optimise the layout of the tasksets to make each scheduling algorithm as competitive as possible.

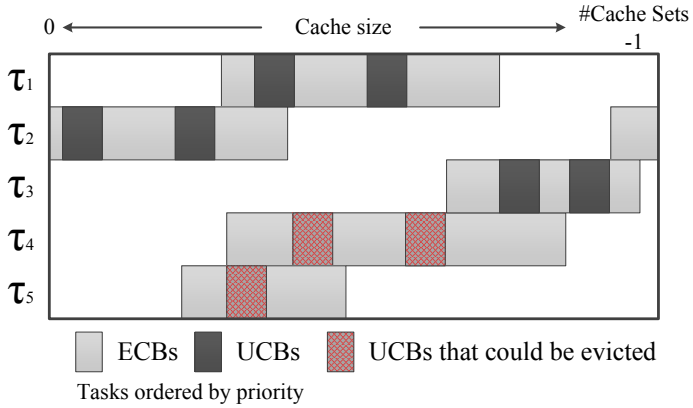

Figure 5 Example layout showing how the position of tasks in cache affects whether their UCBs could be evicted during a pre-emption.

6 Case Studies

In this section we compare the different approaches for calculating CRPD using a set of case studies based on PapaBench[2], the Mälardalen[3] benchmark suite and a set of SCADE[4] tasks (partially provided by SCADE, partially from our own SCADE models). In all cases the system was set up to model an ARMv7[5] processor clocked at 100 MHz with a 2 KB direct-mapped instruction cache and a line size of 8 Bytes, giving 256 cache sets, 4 Byte instructions, and a BRT of 8 μs.

6.1 Single Taskset Case Study

PapaBench is a real-time embedded benchmark based on the software of a GNU-license UAV, called Paparazzi. PapaBench contains two sets of tasks, *fly-by-wire* and *autopilot*. In this paper we used the *autopilot* tasks, for which the WCETs, periods, UCBs, and ECBs were collected using aiT[6] – see Table 1. We made the following assumptions in our evaluation to handle the interrupt tasks:

- Interrupts have higher priority than the normal tasks, but they cannot pre-empt each other
- Interrupts can occur at any time
- All interrupts have the same deadline which must be greater than or equal to the sum of their execution times in order for them to be schedulable
- The cache is disabled whenever an interrupt is executing and enabled again after it completes

In the case of FP scheduling, the interrupts can be modelled as normal tasks with no UCBs or ECBs. Due to the interrupts having the same deadline which is large enough to accommodate the interrupts execution times, no other changes need to be made to the analysis. For EDF scheduling, a number of adjustments must be made to correctly account for the interrupts not being able to pre-empt each other. First we modify equation (19) to exclude interrupts when calculating the *processor demand*, $h(t)$. We then calculate the execution time of each interrupt in the interval t using equation (2) of [10]. The result of which is then added onto the result of the modified version

[2] `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`
[3] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`
[4] Esterel SCADE `http://www.esterel-technologies.com/`
[5] `http://www.arm.com/products/processors/cortex-m/cortex-m3.php`
[6] AbsInt `http://www.absint.com/ait/`

of (19), giving the processor demand for both tasks and interrupts. We then adjust the upper bound L used when checking $h(t)$. This is implemented by substituting $U = U^{tasks} + U^{interrupts}$ into equation (30). Note that we leave U^{γ} to represent the utilisation of the CRPD caused by just tasks as we assume that the cache is disabled while the interrupts are executing and as such they cannot cause any CRPD.

We assigned a deadline of 2 ms to all of the interrupt tasks, and implicit deadlines i.e. $D_i = T_i$, to the normal tasks. We then calculated the total utilisation for the system and then effectively scaled the clock speed in order to reduce the total utilisation to the target utilisation for the system. We used the number of UCBs and ECBs obtained via analysis, placing the UCBs in a group at a random location in each task.

In each evaluation, the taskset utilization not including pre-emption cost was varied from 0.025 to 1 in steps of 0.001. For each utilization value, the schedulability of the taskset was determined under both FP and EDF. Specifically, we compared each scheduling algorithm (i) assuming no pre-emption cost, (ii) using CRPD analysis using the standard task layout, and (iii) using CRPD analysis after optimising the task layout using Simulated Annealing as described in [21]. The standard task layout is obtained by ordering tasks sequentially in memory based on their unique task indices.

Table 2 shows the breakdown utilisation for the single taskset based on PapaBench. There are a few interesting points to note. Firstly the breakdown utilisation is very high for both FP and EDF, this is due to the nearly harmonic periods and small range of task periods, with EDF outperforming FP. Secondly, the CRPD is very low when scheduled using either FP or EDF due to the small number of UCBs. As the CRPD is very low, the layout optimisation makes little to no difference.

6.2 Multiple Taskset Case Studies

The single taskset case study provides one specific example based on the PapaBench tasks and their periods. The remaining case studies used tasksets generated by randomly selecting tasks from a set of benchmarks. In the case of the PapaBench tasks, we treated the interrupts as normal tasks. We obtained tasksets by randomly selecting 10 tasks from Table 1 (PapaBench benchmarks), or 10 tasks from Table 3 (Mälardalen and SCADE benchmarks) or 15 tasks from the two tables (Mixed benchmarks). Using the UUnifast algorithm [9], we calculated the utilisation, U_i of each task so that the utilisations added up to the desired utilisation level for the taskset. Based on the target utilisation and task execution times, T_i was calculated such that $C_i = U_i T_i$. We used $D_i = y + x(T_i - y)$ to generate the constrained deadlines, where x is a random number between 0 and 1, and $y = \max(T_i/2, 2C_i)$. This generates constrained deadlines that are no less than half the period of the tasks. Note, allowing deadlines to be as small as C_i would result in tasks that were unschedulable once CRPD were introduced. We used the number of UCBs and ECBs obtained using aiT, placing the UCBs in a group at a random location in each task.

We generated 1000 tasksets for the multiple taskset case studies, and evaluated them using the same method as the single taskset case study, except that we varied the utilisation excluding pre-emption costs from 0.025 to 1 in steps of 0.0125.

6.2.1 PapaBench Benchmark

The tasks in the PapaBench benchmarks are simple, short control tasks with limited computations and data accesses. Figure 6 shows the percentage of tasksets that were deemed schedulable by each approach for the 1000 tasksets of cardinality 10 that we randomly selected from Table 1. The results are similar to those obtained using the single taskset PapaBench case study. Specifically,

Table 1 Execution times, periods and number of UCBs and ECBs for the tasks from PapaBench. (ms = milisecond)

Task	Name	UCBs	ECBs	WCET	Period
I4	interrupt_modem	2	10	0.303 ms	100 ms
I5	interrupt_spi_1	1	10	0.251 ms	50 ms
I6	interrupt_spi_2	1	4	0.151 ms	50 ms
I7	interrupt_gps	3	26	0.283 ms	250 ms
T5	altitude_control	20	66	1.478 ms	250 ms
T6	climb_control	1	210	5.429 ms	250 ms
T7	link_fbw_send	1	10	0.233 ms	50 ms
T8	navigation	10	256	4.432 ms	250 ms
T9	radio_control	0	256	15.681 ms	25 ms
T10	receive_gps_data	22	194	5.987 ms	250 ms
T11	reporting	2	256	12.222 ms	100 ms
T12	stabilization	11	194	5.681 ms	50 ms

Table 2 Breakdown utilisation under the different approaches for the single PapaBench taskset.

	Breakdown Utilisation
EDF – No Pre-emption Cost	0.999
FP – No Pre-emption Cost	0.977
EDF – Optimised Layout	0.985
EDF – Standard Layout	0.985
FP – Optimised Layout	0.970
FP – Standard Layout	0.969

Table 3 Execution times and number of UCBs and ECBs for the largest benchmarks from the Mälardalen Benchmark Suite (M), and SCADE Benchmarks (S). (s = second, ms = milisecond)

Source	Description	UCBs	ECBs	WCET
M	adpcm	24	226	5.541 s
M	compress	25	114	3.664 s
M	edn	56	98	244.9 ms
M	fir	28	50	21.53 ms
M	jfdctinit	40	162	62.53 ms
M	ns	17	26	73.38 ms
M	nsichneu	53	256	149.6 ms
M	statemate	3	256	77.96 ms
S	cruise control system	25	107	1.959 s
S	flight control system	70	256	2.138 s
S	navigation system	45	82	1.409 s
S	stopwatch	58	130	3.786 s
S	elevator simulation	40	114	1.586 s
S	robotics systems	68	256	4.311 s

Table 4 Weighted schedulability measures for the mixed case study shown in Figure 8. The higher the weighted schedulability measure, the more tasksets deemed schedulable by the approach.

	Weighted Schedulability
EDF – No Pre-emption Cost	0.922
FP – No Pre-emption Cost	0.855
EDF – Optimised Layout	0.830
EDF – Standard Layout	0.771
FP – Optimised Layout	0.784
FP – Standard Layout	0.747

EDF outperformed FP as it deemed a higher number of tasksets schedulable at each utilisation level. Because the range of execution times is relatively small, so is the typical range of task periods for the generated tasksets, hence the number of pre-emption is also relatively small. Further, the number of UCBs is small, resulting in low CRPD. Therefore, the task layout optimisation was only able to make a small improvement, but did so for both FP and EDF.

6.2.2 Mälardalen and SCADE Benchmarks

The second multiple taskset case study was based on tasks from the Mälardalen and SCADE benchmarks, shown in Table 3. Compared to the tasks from PapaBench, these tasks have higher execution times, high amounts of computation, and a larger number of UCBs. Figure 7 shows the percentage of tasksets that were deemed schedulable by each approach for the 1000 tasksets of cardinality 10 that we randomly selected from Table 3. As with the PapaBench benchmarks, EDF outperformed FP scheduling as it has a higher percentage of schedulable tasksets at each utilisation level. Likewise, because the range of task periods was also relatively small, CRPD is minimised.

6.2.3 Mixed Benchmarks

The third multiple taskset case study was based on a mixture of the small and short PapaBench tasks, and the large and long Mälardalen and SCADE tasks. Here the tasksets had 15 tasks each, and represent systems with background tasks combined with short control tasks. As we mixed tasks from both tables, it also allowed us to generate tasksets with a higher number of tasks.

The results, shown in Figure 8, show that when a taskset contains tasks with a wide range of periods, CRPD can become a significant factor in the schedulability of the taskset. This is because short high priority tasks are able to pre-empt long running low priority tasks multiple times.

While EDF still outperformed FP, the gain in schedulability of using EDF over FP was diminished once CRPD was taken into account. Optimising the task layout resulted in a significant improvement for both FP and EDF, showing the task layout optimisation can be effectively applied to both EDF and FP scheduling. Furthermore, by optimising the task layout, FP was able to schedule a similar number of tasksets to EDF with the standard layout. In other words, in cases where the CRPD is relatively high, selecting an optimised task layout can be as effective as changing the scheduling algorithm. The results are summarised in Table 4 using weighted schedulability measures [8], see Section 7.2 for details. They show that for these tasksets, FP with an optimised layout achieved a weighted measure of 0.784, outperforming EDF with the standard layout as it achieved a weighted measure of 0.771.

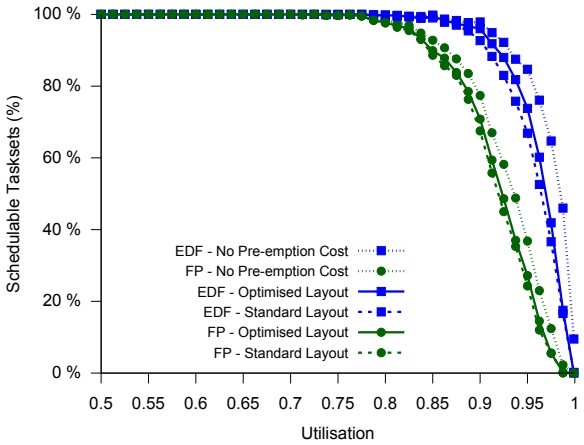

Figure 6 Percentage of schedulable tasksets at each utilisation level for the PapaBench benchmark for tasksets of cardinality 10.

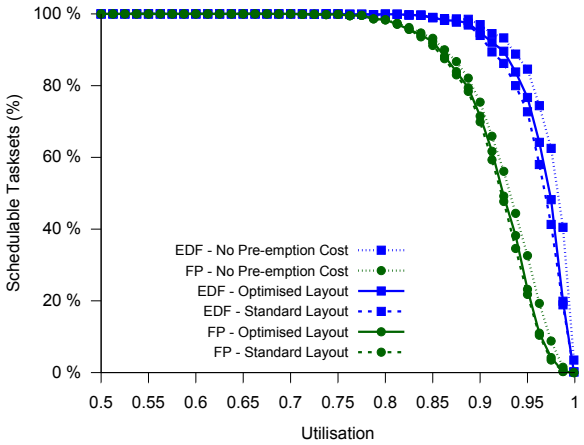

Figure 7 Percentage of schedulable tasksets at each utilisation level for the Mälardalen and SCADE benchmarks for tasksets of cardinality 10.

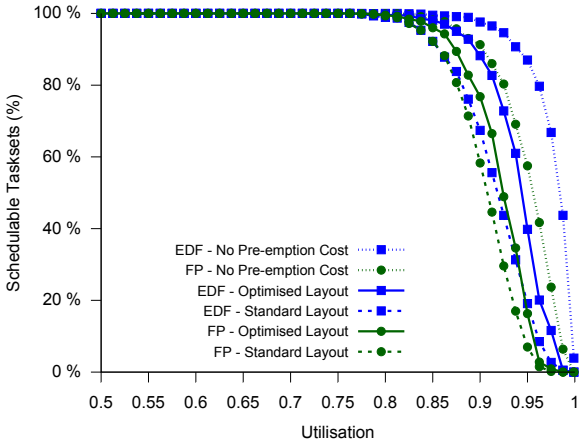

Figure 8 Percentage of schedulable tasksets at each utilisation level for the mixed case study with tasks randomly selected from both the PapaBench and Mälardalen and SCADE benchmarks (taskset cardinality 15).

7 Evaluation

In addition to the case studies based on the PapaBench, Mälardalen and SCADE benchmarks, we evaluated FP and EDF with CRPD analysis using synthetically generated tasksets. This enabled us to investigate the effect of varying key parameters under each scheduling algorithm.

The UUnifast algorithm [9] was again used to calculate the utilisation, U_i of each task so that the utilisations added up to the desired utilisation level for the taskset. Task periods T_i, were generated at random between 5 ms and 500 ms according to a log-uniform distribution. C_i was then calculated via $C_i = U_i T_i$.

We generated two sets of tasksets, one with implicit deadlines and one with constrained deadlines. In the following section, we present the results for constrained deadline tasksets. In general, the results for implicit deadline tasksets gave a higher number of schedulable tasksets for every approach compared to the constrained deadline tasksets. Additionally, the task layout had a similar or slightly larger effect on schedulability in relation to the chosen scheduling algorithm.

The UCB percentage for each task was based on a random number between 0 and a maximum UCB percentage specified for the evaluation. UCBs were split into N groups (where N was chosen at random between 1 and 5), and placed at a random starting point within the task's ECBs.

7.1 Baseline Configuration

To investigate the effect of key cache and taskset configurations we varied the following parameters:

- Cache utilisation (default of 10)
- Maximum UCB percentage (default of 30%)
- Number of tasks (default of 15)
- Block Reload Time (BRT) (default of 8 μs)
- Period range (default of [5, 500] ms)

We used $1,000$ randomly generated tasksets for the evaluation.

In addition to testing the different analyses as done for the case study, we also performed a simulation of the schedule for the tasksets[7]. For FP, the simulation tested each task τ_i in turn by releasing it at time $t = 0$. It then released all of the other tasks that have a higher priority than task τ_i, sorted by lowest to highest priority, at $t = 1$, $t = 2$, $t = 3$, etc. If all tasks were schedulable it also performed the same test but instead of staggering the other tasks, released them at random. For EDF, it is more complicated to generate the worst case arrival pattern. The first step is to determine the interval that needs to be checked, L, which can be achieved by using equation (30). Then for each task τ_i in turn, we scheduled a job of task τ_i so that it has a deadline at $t = L$. We then scheduled a job of all of the other tasks, sorted by longest to shortest deadline, so that they have their deadlines at $t = L - 1$, $t = L - 2$, $t = L - 3$ etc... Based on the final jobs' deadlines, we then calculated when the first jobs for each task need to be released. If all tasks are schedulable, we repeated the process using $t = L - 1$ for all of the other tasks' jobs, and also using a random schedule.

The results for the baseline configuration are shown in Figure 9 and are summarised in Table 5 using weighted schedulability measures. The results follow a similar pattern to the results from the case study. EDF outperformed FP finding a higher number of tasksets schedulable. The results for the simulations show that the CRPD affects both FP and EDF, with the CRPD being slightly lower for EDF. Specifically, the simulation shows that CRPD reduced the weighted measure by at least 0.129 for EDF ($0.925 - 0.795$) and 0.141 for FP ($0.774 - 0.633$) in this case. However, once

[7] Note that the simulation effectively provides a necessary, but not sufficient test of schedulability.

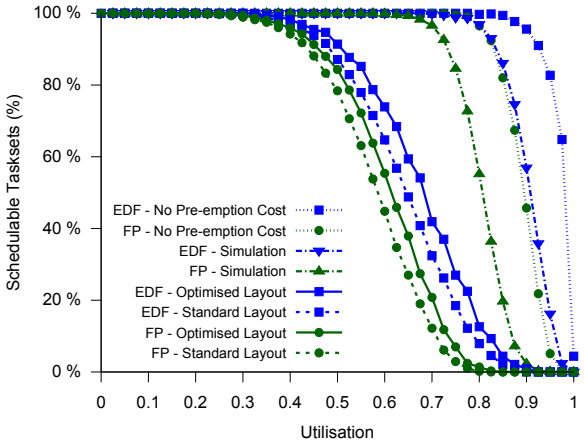

Figure 9 The percentage of schedulable tasksets at each utilisation level for the baseline configuration (taskset cardinality 15).

Table 5 Weighted schedulability measures for the baseline configuration study shown in Figure 9. The higher the weighted schedulability measure, the more tasksets deemed schedulable by the approach.

	Weighted Schedulability
EDF – No Pre-emption cost	0.925
EDF – Simulation	0.796
FP – No Pre-emption cost	0.774
FP – Simulation	0.633
EDF – Optimised layout	0.455
EDF – Standard layout	0.413
FP – Optimised layout	0.369
FP – Standard layout	0.336

the CRPD obtained via analysis is taken into account, the performance gains of using EDF over FP are diminished. This is most likely caused by increased pessimism in the CRPD analysis for EDF. The results for the layout optimisation showed that it was able to make improvements to the schedulability of tasksets scheduled under both FP and EDF.

7.2 Weighted Schedulability

Evaluating all combinations of different parameters is not possible. Therefore, the majority of our evaluations focused on varying one parameter at a time. To present the results, weighted schedulability measures [8] are used. This allows a graph to be drawn which shows the weighted schedulability, $W_l(p)$, for each method used to obtain a layout l as a function of parameter p. For each value of p, this measure combines the data for all of the generated tasksets τ for all of a set of equally spaced utilisation levels, where the utilisation is based on no pre-emption cost.

The schedulability test returns a binary result of 1 or 0 for each layout at each utilisation level. If this result is given by $S_l\ (\tau, p)$, and $u(\tau)$ is the utilisation of taskset τ, then:

$$W_l(p) = \frac{\left(\sum_{\forall_\tau} u(\tau) S_l(\tau, p)\right)}{\sum_{\forall \tau} u(\tau)}. \tag{31}$$

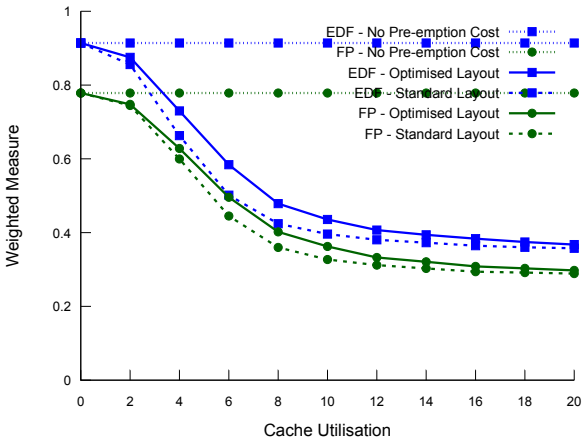

Figure 10 Weighted measure for varying the cache utilisation from 0 to 20 in steps of 2.

The benefit of using a weighted schedulability measure is that it reduces a 3-dimensional plot to 2 dimensions. Individual results are weighted by taskset utilisation to reflect the higher value placed on a being able to schedule higher utilisation tasksets.

7.2.1 Cache Utilisation

As the cache utilisation increases the likelihood of tasks evicting each other from cache increases, this causes higher CRPD reducing the number of schedulable tasksets. It can be seen in Figure 10 that task layout optimisation is effective for FP and EDF across the same range of cache utilisations. In both cases it becomes less effective once the cache utilisation becomes high. We note that because the number of tasks is fixed, that the average size of the tasks is equal to the cache utilisation divided by the number of tasks. This means that as the cache utilisation increases, so does the size of the tasks and therefore, the number of UCBs. This in turn makes it harder to find an improved layout.

7.2.2 Maximum UCB Percentage

As the maximum UCB percentage increases, the CRPD increases resulting in a reduction in the number of tasksets that are deemed schedulable, as can be seen in Figure 11. With a low percentage of UCBs, the CRPD is low which means there is little benefit from layout optimisation. When the UCB percentage is very high, there are so many conflicts that there is very little that can be done to improve the layout. When the maximum UCB percentage is around 40–60%, there is a notable amount of CRPD, but there is also room for the task layout algorithm to optimise the layout. This allows FP using an optimised task layout to schedule a similar number of tasksets as EDF using the standard layout.

7.2.3 Number of Tasks

When varying the number of tasks, Figure 12, we scaled the cache utilisation to keep the average size of tasks constant based on a cache utilisation of 10 for 15 tasks. This is because it would be unrealistic for the size of tasks to decrease as more tasks are added to the system. Hence with 8 tasks the cache utilisation is equal to 5.33, whereas for 32 tasks, it is equal to 21.33. As the number of tasks increases, it becomes harder the schedule all tasks which leads to a decrease

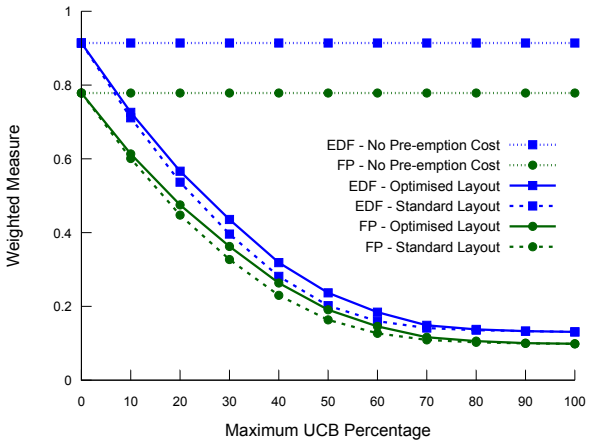

Figure 11 Weighted measure for varying the maximum UCB percentage from 0 to 100 in steps of 10.

in the overall weighted measure. The task layout optimisation performs best when there is a moderate number of tasks, as there are enough conflicts that optimising the layout can give an improvement, but not so many that there is nothing that can be done to avoid the conflicts.

7.2.4 Block Reload Time

As the block reload time is increased, it becomes more costly to reload a block, which causes an increase in CRPD. It can be seen in Figure 13 that as the block reload time is increased, the analysis that takes into account the pre-emption cost shows a decrease in the overall weighted measure. We note that as the cost of reloading a block increases, the potential gains of optimising the layout increase. Once the block reload time exceeds 14 μs, using an optimised layout under FP scheduling outperforms using a standard layout under EDF scheduling.

7.2.5 Period Range

We also investigated the effect of the scaling factor used to generate task periods to simulate tasksets with shorter to longer execution times. We varied the scaling factor, w, from 0.5 to 10 and hence the range of task periods given by $w[1, 100]$ ms. A lower scaling factor resembles tasks with shorter execution times, as seen in the PapaBench benchmark, and a higher scaling factor resembles tasks with higher execution times and commensurately longer periods, as seen in the Mälardalen and SCADE benchmarks. The results in Figure 14 show the layout optimisation performs best when task periods are relatively short, as that is when the pre-emption costs are highest. Once the period range is greater than [10, 1000] ms, the relative pre-emption costs are low enough that performing the layout optimisation only makes a very small improvement on the schedulability of the tasksets.

8 Conclusion

The EDF scheduling algorithm is an optimal scheduling algorithm for single processors however, it has been largely disregarded by industry. Whereas FP, despite offering lower theoretical schedulable processor utilisation, is relatively popular with many commercial real- time operating systems supporting it.

Previous work by Buttazzo [13] has compared the two algorithms, but it did not take into account CRPD which can have a significant effect on the schedulability of a taskset.

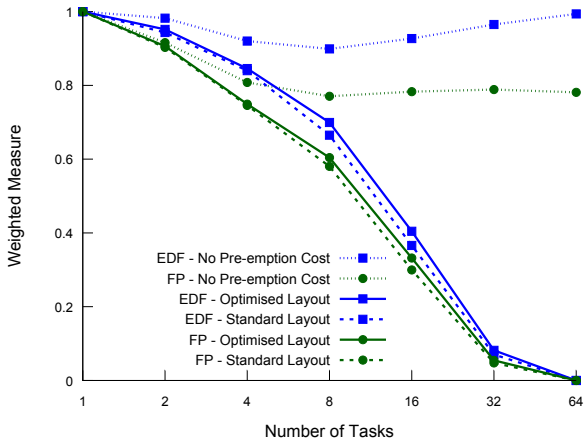

Figure 12 Weighted measure for varying the number of tasks from 2^0 to 2^6 while maintaining a constant ratio of number of tasks to cache utilisation.

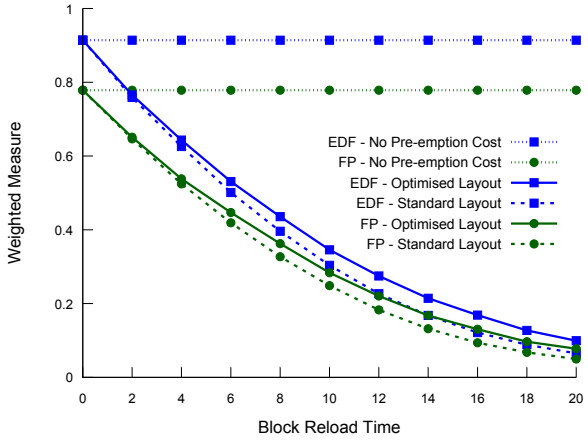

Figure 13 Weighted measure for varying the block reload time from 0 to 20 μs in steps of 2.

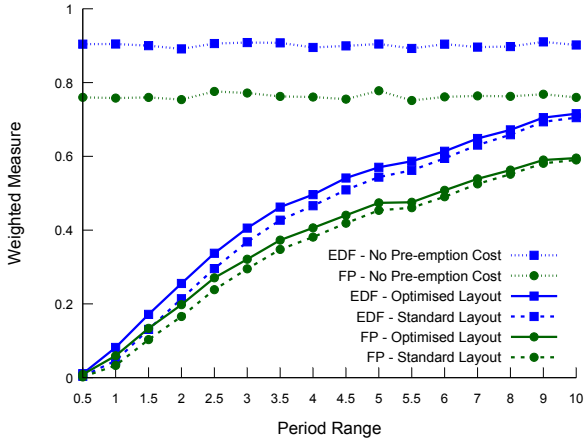

Figure 14 Weighted measure for varying the scaling factor used to generate periods, w, in $w[1, 100]$ ms, from 0.5 to 10.

The main contributions of this paper are:

- Performing a detailed comparison of FP and EDF taking into account CRPD using state-of-the-art CRPD analysis [3, 22].
- Showing the feasibility of simple, yet effective, task layout optimisation techniques for EDF.
- Finding that when CRPD is considered, the performance gains offered by EDF over FP, while still significant, are somewhat diminished. This is most likely due to greater pessimism in the CRPD analysis for EDF than FP.
- Discovering that in configurations that cause relatively high CRPD, optimising task layout can be just as effective as changing the scheduling algorithm from FP to EDF.

We investigated the effects of performing task layout [21] optimisation based on Simulated Annealing under both FP and EDF scheduling algorithms. We found that in the scenarios that cause the pre-emption cost to be relatively high in relation to task execution times, applying task layout optimisation to a system scheduled using FP scheduling can allow it to be schedulable at a similar processor utilisation compared to using EDF scheduling with a standard layout. This is important in an industrial setting as it is considerably simpler and cheaper to control the task layout via the linker, than it is to change the scheduler. Nevertheless, our evaluations showed that changing to an EDF scheduler and optimising the task layout provides a gain over FP scheduling. Although this gain was not as pronounced as the advantage that EDF has over FP when pre-emption costs are not accounted for via analysis.

In the future we plan to further investigate techniques for CRPD analysis, and to apply them in an industrial context comparing the results of analysing a real system with those obtained via measurement.

Grant Information. This work was partially funded by the UK EPSRC through the Engineering Doctorate Centre in Large-Scale Complex IT Systems (EP/F501374/1), the UK EPSRC funded MCC (EP/K011626/1), and the European Community's ARTEMIS Programme and UK Technology Strategy Board, under ARTEMIS grant agreement 295371-2 CRAFTERS.

References

1 Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1–3, 2009*, pages 109–118. IEEE Computer Society, 2009. doi:10.1109/ECRTS.2009.21.

2 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 – December 2, 2011*, pages 261–271. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.31.

3 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):499–526, 2012. doi:10.1007/s11241-012-9152-2.

4 Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2010, Stockholm, Sweden, April 13–15, 2010*, pages 153–162. ACM, 2010. doi:10.1145/1755888.1755911.

5 Neil C. Audsley, Alan Burns, Mike F. Richardson, Ken Tindell, and Andrew J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=238595&isnumber=6134.

6 Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium – 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 182–190. IEEE Computer Society, 1990. doi:10.1109/REAL.1990.128746.

7 Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990. doi:10.1007/BF01995675.

8 Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays: Empirical approxim-

ation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, OSPERT 2010*, pages 33–44, 2010. URL: http://www.mpi-sws.org/~bbb/papers/pdf/ospert10.pdf.

9 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

10 Björn B. Brandenburg, Hennadiy Leontyev, and James H. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24–26 August 2009*, pages 273–283. IEEE Computer Society, 2009. doi:10.1109/RTCSA.2009.37.

11 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *9th International Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1–3, 2009*, volume 10 of *OASICS*, pages 1–11. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2009. doi:10.4230/OASIcs.WCET.2009.2285.

12 José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *2nd IEEE Real-Time Technology and Applications Symposium, RTAS 1996, June 10–12, 1996, Boston, MA, USA*, page 204. IEEE Computer Society, 1996. doi:10.1109/RTTAS.1996.509537.

13 Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29(1):5–26, 2005. doi:10.1023/B:TIME.0000048932.30002.d9.

14 Robert I. Davis, A. Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008. doi:10.1109/TC.2008.66.

15 Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

16 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical report, INRIA, 1996. URL: http://hal.inria.fr/inria-00073732.

17 Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, April 16–20, 2007, Nice, France*, pages 1623–1628. ACM, 2007. URL: http://dl.acm.org/citation.cfm?id=1266366.1266723.

18 Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong-Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Trans. Computers*, 47(6):700–713, 1998. doi:10.1109/12.689649.

19 Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980. doi:10.1016/0020-0190(80)90123-4.

20 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

21 Will Lunniss, Sebastian Altmeyer, and Robert I. Davis. Optimising task layout to increase schedulability via reduced cache related preemption delays. In *20th International Conference on Real-Time and Network Systems, RTNS 2012, Pont a Mousson, France – November 08–09, 2012*, pages 161–170. ACM, 2012. doi:10.1145/2392987.2393008.

22 Will Lunniss, Sebastian Altmeyer, Claire Maiza, and Robert I. Davis. Integrating cache related preemption delay analysis into EDF scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9–11, 2013*, pages 75–84. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531081.

23 Ismael Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996. doi:10.1007/BF00365519.

24 Marco Spuri. Analysis of deadline scheduled real-time systems. Technical report, INRIA, 1996. URL: http://hal.inria.fr/inria-00073920.

25 Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *17th Euromicro Conference on Real-Time Systems, ECRTS 2005, 6–8 July 2005, Palma de Mallorca, Spain*, pages 41–48. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.26.

26 Yudong Tan and Vincent John Mooney III. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1), 2007. doi:10.1145/1210268.1210275.

27 Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009. doi:10.1109/TC.2009.58.

TLM.open: a SystemC/TLM Front-end for the CADP Verification Toolbox*

Claude Helmstetter

Verimag – CNRS
2 Avenue de Vignate, 38610 Gières, France
claude.helmstetter@gmail.com

Abstract

SystemC/TLM models, which are C++ programs, allow the simulation of embedded software before hardware low-level descriptions are available and are used as golden models for hardware verification. The verification of the SystemC/TLM models is an important issue since an error in the model can mislead the system designers or reveal an error in the specifications. An open-source simulator for SystemC/TLM is provided but there are no tools for formal verification.

In order to apply model checking to a SystemC/TLM model, a semantics for standard C++ code and for specific SystemC/TLM features must be provided. The usual approach relies on the translation of the SystemC/TLM code into a formal language for which a model checker is available.

We propose another approach that suppresses the error-prone translation effort. Given a SystemC/TLM program, the transitions are obtained by executing the original code using g++ and an extended SystemC library, and we ask the user to provide additional functions to store the current model state. These additional functions generally represent less than 20% of the size of the original model, and allow it to apply all CADP verification tools to the SystemC/TLM model itself.

2012 ACM Subject Classification Hardware, Hardware validation, Functional verification, Transaction-level verification

Keywords and phrases Model checking, Verification, Simulation, SystemC, Transactional modeling

Digital Object Identifier 10.4230/LITES-v001-i001-a002

Received 2013-03-01 **Accepted** 2013-04-25 **Published** 2014-04-25

1 Introduction

The design of abstract models written in SystemC/TLM has become more common in the development of embedded systems. These models allow the simulation of the embedded software before the hardware RTL description is available, and are used as golden models for hardware verification. The verification of the SystemC/TLM models is an important issue since an error in the model can mislead the system designers or reveals an error in the specifications.

ASI (*Accellera Systems Initiative*, previously OSCI: *Open SystemC Initiative*) provides an open-source simulator for SystemC/TLM and a library SCV (*SystemC Verification*) to ease test generation. However, ASI does not provide tools for formal verification. Moreover, while the SystemC specification allows many schedules for a given test case, the ASI simulator always exhibits the same schedule. Thus, even if an execution leads to the expected result, another execution with a different schedule may be erroneous. To find these kind of bugs, many publications have experimented with the use of model checking.

The problem of verifying C++ and SystemC codes could be avoided by writing transactional models in a formal language directly. However, in order to model embedded systems at the

* This work was achieved when the author was working for INRIA Montbonnot – VASY team. Moreover, this work was sponsored by the French government and by Conseil Général de l'Isère as part of the Multival project (pôle de compétitivité Minalogic).

transaction level, engineers of industrial companies prefer to use SystemC/TLM. One reason is that SystemC/TLM provides all the useful features directly, like shared memory and transactional communication channels. Another reason is that a SystemC/TLM program is mainly C++ code, so engineers can learn SystemC/TLM quickly, and existing C code can be reused easily.

In order to apply model checking to a SystemC/TLM program, the usual approach relies on the translation of the SystemC/TLM code into a formal language for which a model checker is available. A lot of languages and tools have been tested so far (see Subsection 3.1.2). Nonetheless, there have been few successes with industrial case studies.

We propose another approach that suppresses the translation effort. Basically, an explicit model checker must be able to execute transitions and store states. Given a SystemC/TLM program, we assume that the states are the places where processes yield back to the scheduler. Consequently, transitions correspond to pieces of C++ code delimited by yield points: either `wait` statements or `return` statements from the process main function. We obtain the transitions by executing the original code using `g++` and a SystemC library, as in any simulator. Storing the state could be done by copying the whole memory used by the simulator, but would be inefficient. Therefore, we ask the user to provide additional functions to store the current state and restore a previous state. Part of the state, including the SystemC kernel, is stored automatically; so in general the user can only store the SystemC module data members.

Following this approach, we have developed a new front-end for the CADP tool suite. The CADP tool suite includes many tools useful for formal verification and bug finding; the main tool is an explicit model checker. This article does not introduce a new verification technique (we did not change anything in CADP) except a pragmatic and efficient way to use existing tools to verify programs written in a language that has not been designed to ease formal verification. The new front-end we have developed is not fully automatic since the user must provide some additional functions; these additional functions generally represent less than 20% of the size of the original model.

The model checking technique is known to be limited by the state space explosion. Because we rely on this technique and there are no changes in the core algorithm, we are limited in this area. Nevertheless, model checking has been applied to many real-life case studies (over 150 using CADP in many application fields[1] most of the times, using model checking allowed to verify properties or discover bugs. We have written our new front-end in a way that avoids to make the state space explosion even worse by adding intermediate states and transitions, which was the case using a previous approach [16]. Experiments which were first made with benchmarks, then with a single SystemC module, and finally with a basic system, show that we can indeed find bugs and prove some properties on real-life TLM models.

The remainder of this article is organized as follows. We present briefly SystemC and TLM in Section 2. Section 3 gives an overview of the related work and presents the existing CADP toolbox. Section 4 describes our technique to connect SystemC/TLM with CADP. The performances of `TLM.open` are evaluated in Section 5 and Section 6 concludes this article.

2 SystemC and TLM

SystemC [1] is a C++ library published by the *Accellera Systems Initiative* (ASI) and defined by an IEEE standard which provides classes to describe heterogeneous systems composed of hardware and software. The architecture of a system is defined by a set of *modules* connected by synchronous or asynchronous *ports* and *channels* (`sc_module`, `sc_port`, ...). Each module contains zero, one, or

[1] Case studies achieved using the CADP toolset: `http://www.inrialpes.fr/vasy/cadp/case-studies`.

various *processes* (`SC_THREAD` or `SC_METHOD`) describing the system's behavior. SystemC processes interact using shared memory or communication channels and are synchronized using SystemC events (`sc_event e`, `e.notify()`, `wait(e)`) with timing annotations (`sc_time t`, `wait(t)`).

Each SystemC process is a C++ method that is executed by the SystemC scheduler communicates with other processes using shared memory and may explicitly suspend itself by executing a `wait` statement. When the process is resumed by the scheduler, its execution continues from the `wait` statement. Each SystemC process is *eligible* or *running* or *waiting* for a SystemC event. There is, at most, one running process simultaneously. If the running process notifies an event, then all processes waiting for this event move from waiting to eligible.

The *Transaction Level Modeling* (TLM) library [10] built upon SystemC, provides a *transaction* mechanism that encapsulates communication protocols (data transfer and synchronization) between modules and accelerates both model design and simulation. Using a transaction, a process in an *initiator* module can directly call the methods exported by a *target* module. Thus, a process can read many values from a memory, or set many registers of a peripheral without any costly inter-process synchronization (no context switch is required). At the TLM level of abstraction, processes inside the same module communicate using SystemC events and shared variables. A TLM model can be *timed* or *untimed*: a timed model contains timing annotations (`sc_time t`, `wait(t)`) whereas an untimed model does not. An untimed model includes more possible behaviors than a timed model, increasing the coverage, but also the cost, of the verification.

Because SystemC and TLM are C++ libraries, simulating a SystemC/TLM model does not require a dedicated SystemC/TLM parser. A SystemC/TLM model is parsed and compiled as with any C++ program, using a regular C++ compiler, such as `g++`.

3 Related Works

3.1 Verification of SystemC/TLM models

In order to provide formal verification for SystemC/TLM programs, two approaches were investigated: stateless model checking of a SystemC/TLM program and a translation of a SystemC/TLM program into a language for which a stateful model checker is available.

3.1.1 Stateless model-checking

A stateless model-checker explores the set of all the possible executions of a given program without storing the states. Because the states are not stored, a stateless model-checker can execute the same transition many times. Moreover, if the program under verification has at least one possible execution that does not terminate then the stateless model-checker will not terminate either. However, stateless model-checkers have benefits: 1. naive stateless model-checkers are easy to implement because one just needs to modify the functions used for non-deterministic choices; 2. their memory consumption is limited (linear in terms of execution lengths).

Many stateless model checking tools have been implemented for SystemC/TLM programs [15, 21, 2]. In order to reduce the number of executions explored, these model-checkers select a subset for the possible executions; this subset is guaranteed to detect all the errors of a particular family; such as all the assertion failures or all the deadlocks. All these stateless model-checkers implement dynamic partial order reduction [5]; the selection of the executions explored is based on the analysis of detailed execution traces. The dynamic partial order algorithm was specifically adapted for the particularities of the SystemC scheduling policy.

In particular, [14, 15] show how to validate programs with loose timing annotations encoded by bounded intervals. This technique extracts a finite subset from the infinite set of the timings allowed

by the specification. Given a program that always terminates and without non-deterministic data choices, this technique detects all the assertion failures and the deadlocks.

These tools give interesting results for small and medium sized industrial examples. Using SCRV [13], a synchronization error was found in a model of a video decoder provided by STMicroelectronics. However, stateless model-checkers can only be applied to terminating programs without non-deterministic unbounded data inputs.

3.1.2 Translate then verify

For programs that do not terminate, a second approach was investigated. The idea was to translate the SystemC/TLM program to be verified into another language, and then verify the translated program using an existing stateful model checker. This approach has first been applied to the RTL level SystemC descriptions [4, 11].

Many translations and languages have been proposed for the validation of transactional models, as in [25], which translates SystemC/TLM programs into finite state machines (FSM), similarly [20], which describes abstraction techniques and a translation from SystemC/TLM to labeled Kripke structures. Most of these translations are manual, the first exception being the LusSy tool chain [24], which automatically translates TLM models into synchronous automata with variables; it provides some simple abstraction techniques (e.g., abstract address representation). The LusSy tool chain has been connected to many model checkers, including symbolic model checkers based on BDD or SAT. Some minor examples have been successfully verified, but industrial examples face the state space explosion problem. There are now other automatic translation tools starting from SystemC, including [17] that can translate SystemC/TLM models into Uppall models, and allow verification of liveness properties and timing constraints. [12] translates TLM models into sequential C programs, in order to use verification tools dedicated to software.

The state space explosion problem appears mainly because TLM models are mostly asynchronous. Thus, after each transition, there are many valid scheduling choices that should be explored. It is therefore suitable to use the model checkers for asynchronous programs as these model checkers have been specifically optimized to fight state space explosions arising from asynchrony. Translation of TLM programs into Promela [27] has allowed TLM models to be verified using the SPIN checker which uses partial orders to reduce state spaces. Since then other translations into Promela have been presented [22, 3], allowing the verification of larger models. Note that the translation defined in [3] was implemented in an automatic tool where other back-ends were available.

Furthermore, we firstly proposed a translation of TLM to LOTOS [16, 26] that enables verification of the benchmark of [27] for a slightly greater number of processes than using SPIN. The translation was fully manual, preventing the approach to scale up.

Next, [7] presented both an extension of our TLM to LOTOS translation and an application to an industrial case study. As shown in Figure 1, part of the SystemC/TLM code was translated into LOTOS while data-types and related operations were kept as C++ code, thus strongly reducing the translation effort. This paper showed that some properties can be checked on industrial code, but the amount of manual work still limited the efficiency of the approach.

3.2 The CADP verification toolbox

Our goal is to detect synchronization errors between asynchronous processes of SystemC/TLM programs or to guarantee that the communication protocol they use is correct. For this kind of task one of the best techniques is model checking and in particular explicit model checking.

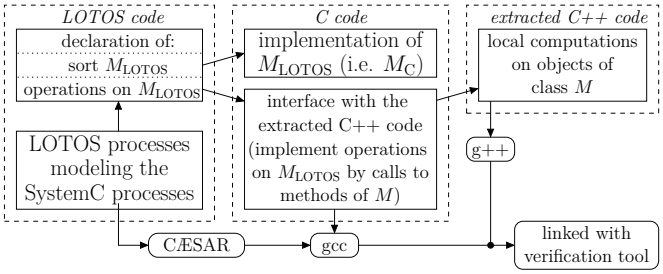

Figure 1 Verification of hybrid LOTOS/C++ models.

$R \leftarrow \{\text{initial state}\}$ *//set of remaining states*
$E \leftarrow \emptyset$ *//set of explored states*
while $(\exists x \in R)$ **do**
  foreach transition $x \rightarrow y$ **do**
    add the transition $x \rightarrow y$ **to the LTS**
    if $(y \notin E \cup R)$ **then** $R \leftarrow R \cup \{y\}$
  $R \leftarrow R \setminus \{x\}$
  $E \leftarrow E \cup \{x\}$
end while

Figure 2 Basic algorithm for LTS generation.

A well-known explicit model-checker is SPIN. In this work, we investigated the use of another model-checker; namely CADP.

CADP ("Construction and Analysis of Distributed Processes") [9] is a toolbox for the validation of communication protocols and distributed systems.

The usual entry point for CADP is the language LOTOS. The ISO standard LOTOS [18] (*Language Of Temporal Ordering Specification*) is a process algebra used to describe asynchronous concurrent processes communicating and synchronizing by rendezvous on gates. This language is well suited for designing communication protocols.

The semantics of a LOTOS specification is formally defined by a *state graph*, also called an LTS (labeled transition system) – i.e. a set of states and transitions labeled by gates and offers between states.

CADP [8] includes a compiler from LOTOS to LTS with many tools exploiting the LTS for simulation as well as model checking of modal μ-calculus formulae, equivalence checking, test generation and performance evaluation. The LOTOS to LTS compiler generates the LTS by executing all transitions of the system under verification; each visited state is recorded, so that each transition is executed once only. The algorithm is shown in Figure 2.

4 Connecting SystemC/TLM with formal methods

4.1 The architecture of TLM.open

The CADP toolbox architecture is similar to the GNU compiler tool suite, with many front-ends and back-ends. There is one front-end per input language; the front-end reads a program and implements some basic analysis (e.g., type checking). Then there is one back-end per CADP feature, such as simulation, LTS generation, or on-the-fly property checking. The most used front-ends are `caesar.open` which manages LOTOS programs and `bcg_open` which reads compressed

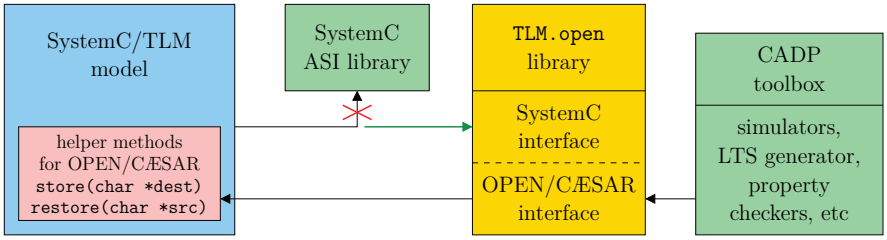

Figure 3 Overview of the verification framework. The model is linked with the SystemC `TLM.open` library instead of the ASI library.

and explicit LTS (`bcg` stands for *"binary coded graphs"*). All CADP front-ends connect with CADP back-ends using the OPEN/CÆSAR interface [6].

In this section, we present `TLM.open` which is a new CADP front-end allowing the use of the same back-ends as `caesar.open` and `bcg_open`. `TLM.open` is a C and C++ library that implements two interfaces: the SystemC interface and the OPEN/CÆSAR interface. The architecture of `TLM.open` is shown in Figure 3.

A SystemC/TLM program communicates with `TLM.open` through the SystemC interface as a SystemC/TLM program communicates with a SystemC simulator. The library `TLM.open` provides the same classes as the ASI SystemC simulator, including the `sc_module`, `sc_port`, `sc_event`, `sc_signal`, etc.

The OPEN/CÆSAR interface provides the operators required by the CADP model-checker itself. In order to implement the algorithm described by Figure 2, the following operators are required and must be provided by the `TLM.open` front-end:

- generation of the initial state
- enumeration and simulation of the transitions starting from a given state
- efficient storage of a state (requires comparison and hash functions).

To simulate a transition, `TLM.open` executes the corresponding C++ code of the SystemC/TLM program. This C++ code is compiled with an unmodified C++ compiler such as `g++`. `TLM.open` does not parse the C++ code itself and does not produce LOTOS code.

The most difficult task is to store and restore the states of the SystemC/TLM program. The person who writes and verifies the SystemC/TLM program, called *user* in this paper, has to provide some additional functions that allow `TLM.open` to store the states of each SystemC module. To date, these additional functions have had to be written by hand. Thus, our approach is not fully automatic.

When TLM.open is used with the LTS generator of CADP, the result is an LTS with two kinds of transitions. Here, *offers* are only used to add information to the transitions, and have no impact on synchronizations or communications.

- `TE` transitions indicate that time has elapsed; the offer gives the duration and the list of events triggered and processes awakened. For example, "TE !o(+41ms, VGAC.compute)" means that the SystemC clock has advanced 41 ms and the process "VGAC.compute" is now awake.
- `EXEC` transitions represent the execution of a SystemC process; the offers name the executed SystemC process, the inputs of this process if the special `rand()` function of `TLM.open` was called (cf. Section 4.3.2) and the outputs generated using the overloaded `puts()` function. For example, "EXEC !VGAC.compute !o(image updated, IRQ sent)" means that the SystemC scheduler has executed the process "VGAC.compute" and this process has printed two messages "image updated" and "IRQ sent".

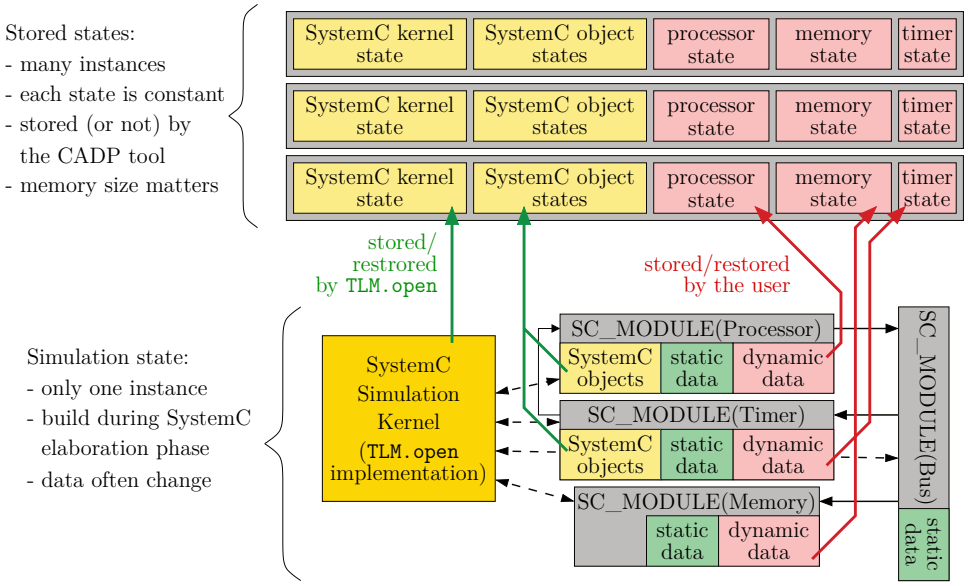

Figure 4 Memory layout: simulation state and stored states.

4.2 Storing and restoring program states

The `TLM.open` library includes a SystemC simulator. The state of this simulator consists of the state (i.e., the current value) of each object that has been instantiated. Some objects are described by SystemC classes (such as: `sc_event`, `sc_signal`, ...) and others are described by user classes. SystemC modules are hybrid: some class members are inherited from the base class `sc_module` but other members are defined by the user.

A *stored state* contains a copy of each value of the simulator state that may change during the simulation. A stored state must be as small as possible and does not use the same types as the simulator states: constant values are not stored, Boolean values can be grouped in one byte using bit fields.

All objects which are defined by a SystemC class are stored automatically by the `TLM.open` library. The other objects are stored using callback functions implemented by the user. Each SystemC module must provide the following functions:

- `size_t size() const`: number of bytes needed to store a copy of the SystemC module.
- `size_t alignment() const`: specify whether padding bytes are needed.
- `void store(char *dest) const`: store the current state of the module in `dest`.
- `void restore(const char *src)`: restore the state of the module according to the copy stored in `src`.

The `store()` function must generate a canonical representation, so that state comparison can be done using `memcmp()` and hash functions can be generated automatically.

Implementing the functions `size()` and `alignment()` generally requires only one line of code for each. The `store()` function implementation contains two lines of code per module member on average; similarly the `restore()` function. Implementing these functions requires some manual work, but less than translating the whole model into another language.

Theoretically, generating automatically the `store()` and `restore()` functions should be simpler than translating the whole code, because it is not necessary to manage the code describing the behavior. However, such a generator would have to parse and manage a large part of C++, and the generated functions would likely be less efficient than those hand-written.

4.2.1 Storing modules using flat state

The user has many possibilities to implement the `store` and `restore` functions. The basic solution is to define a new `struct` type with one field per member of the SystemC module that is not constant and not managed directly by `TLM.open`. To store the state, the user filled this new type by copying values from the C++ class, and inversely, the user filled the C++ class by copying values from the `struct` type when the state must be restored. This is shown in the example below.

```
SC_MODULE(Foo) {
  sc_event e;      // state stored by the tlm.open library
  bool flag;       // dynamic data
  uint32_t data;   // dynamic data
  const sc_time period; // static data, not stored
  ... // module implementation

  // code below is used only by TLM.open
  struct State { // container type
    bool flag; uint32_t data;
    void set(Foo *f) const {f->flag=flag; f->data=data;}
    void set(const Foo *f) {flag=f->flag; data=f->data;}
  };
  size_t size() const {return sizeof(State);}
  size_t alignment() const {return 4 /*alignmentof(State)*/;}
  void store(char *dst) const {
    reinterpret_cast<State*>(dst)->set(this);}
  void restore(const char *src) {
    reinterpret_cast<const State*>(src)->set(this);}
}; // Foo
```

In this example, storing the state of an instance of `Foo` requires 8 bytes (i.e., `sizeof(Foo::State)`). If a program contains n modules $M_1, \ldots, M_n$, each module being stored using a type M_i`::State`, then each state stored consumes at least $\Sigma_{i=1}^{n}$`sizeof(`M_i`::State)` bytes.

4.2.2 Storing module using hierarchical state

Most of the time, a transition modifies the state of only one or two modules. If storing a module consumes a lot of memory, it is then mostly better to use a hierarchical state. Using hierarchical states, the main state contains a pointer to the module state instead of the module state itself. When a transition is executed and the module has not been modified, then the new stored state contains only a pointer to the previously stored value.

Moreover, checking whether the module has been modified by the last transition is not enough. Even if the module has been modified, it is possible that we already have a copy of its current state. At the end of a transition, we search all the previous states of this module, which are stored in a container (hash table or binary tree). If this module state is encountered for the first time, then it is added to this container, or else we reuse the existing module state.

Here is how the `Foo` state could be recorded using a hierarchical state:

```
typedef std::set<const State*, StateCmp> state_set;
static state_set foo_states;
size_t Foo::size() const {return sizeof(State*);}
```

```
void Foo::store(char *dst) const {
  State *s = new State(); s->set(this);
  std::pair<state_set::iterator,bool> p = foo_states.insert(s);
  if (!p.second) delete s; //This Foo state already exists,
                           // so we reuse the previous version.
  *reinterpret_cast<const State**>(dst) = *p.first;}
```

To compare two states of the whole program, we just need to compare the pointers because identical module states are never stored in distinct memory locations.

In some cases, hierarchical states can significantly reduce the memory consumption. Moreover, whereas the OPEN/CÆSAR interface requires the main state to have a fixed size, hierarchical states allow a module to be stored whose size is not statically bound.

Internally, for all objects that are stored automatically, the `TLM.open` library uses a flat state for all objects except SystemC threads (`SC_THREAD`). Moreover, storing the state of a thread is done by copying its execution stack. Note that when yielding, the QuickThreads library used by SystemC pushes the register contents and the program counter on top of the thread stack. As thread stack sizes vary during simulation because stacks may become large,and because at most one thread stack is modified during a transition, the hierarchical state technique here is more efficient than flat states.

4.3 Implementation of the OPEN/CÆSAR interface

4.3.1 Generation of the initial state

The generation of the initial state faces a technical problem. Moreover, SystemC and CADP do not use the same control flow:

- A SystemC simulator creates the initial state by calling the function `sc_main`, which is implemented by the user, and the simulation starts when the user **calls back** the function `sc_start` from the `sc_main function`.
- A CADP back-end creates the initial state by calling the function `CAESAR_START_STATE`, which is implemented by the front-end and the verification starts after the function `CAESAR_START_STATE` **returned**.

Thus, the CADP back-end calls the function `CAESAR_START_STATE` of `TLM.open`, and this function calls `sc_main`. The function `CAESAR_START_STATE` must return when `sc_start` is called, before `sc_main` returns. If one returns in advance of `sc_start` using a `return` statement or a C long jump or a C++ exception then the modules allocated on the stack are destroyed before they are used. The solution is to execute the `sc_main` function in a separated thread, which has its own stack and suspends this thread when `sc_start` is called. As we do not need real concurrency, this thread is implemented using the collaborative QuickThreads library, which is used to implement the SystemC threads too.

4.3.2 Enumerating the transitions

The key function of the OPEN/CÆSAR interface is `CAESAR_ITERATE_STATE`. This function must enumerate the transitions starting from a given stored state x. A transition is defined by a label s (a C string) and the successor state y.

There is at least one transition per eligible process. Assuming all transitions are deterministic, the `TLM.open` library behaves as follows:

1. A SystemC process is selected.

2. The simulator is set according to the stored state x, by calling the `restore` function of each stored object (either a user function for modules, or a `TLM.open` library function for other SystemC objects).
3. The transition is executed, until the elected process yields back to the scheduler.
4. The new simulator state is stored in y, by calling the `store` function of each stored objects. The label s is created using the name of the elected process, and the outputs generated by the user using the `puts()` function.
5. This transition is sent to the back-end.
6. If there is another eligible thread, then go back to step 1.

If no processes are eligible, then `TLM.open` can let the time elapse until a process is awoken, just like a regular SystemC simulator. In this case, a specific transition is generated with the label "TE". If no process can be awoken by a time elapse, then this means that x is in a deadlock state. In order to simulate inputs or a non determinism, the `TLM.open` library provides a `rand(int MAX)` function. From the user point of view, this function returns a number between 0 and `MAX`. In case of a simulation, an implementation would choose a number randomly. On the contrary, model checking requires an enumeration of all values. In order to generate the full LTS, each time `TLM.open` encounters a call to `rand()`, it records that another transition exists for the same process for which values have already been tried. Thus, the same code will be executed MAX+1 times, generating as many LTS transitions. Because a transition may call `rand()` many times, `TLM.open` uses a stack to remember its position in the transition tree. Thus, all input combinations are finally generated (e.g., "`x=rand(2); y=rand(3); wait();`" generates $3 \times 4 = 12$ transitions).

4.4 Features and limitations

Most SystemC, TLM and C++ features can be used normally. However, some features require special care. As aforementioned, the functions `puts()` and `rand()` have a special meaning when used with `TLM.open`.

4.4.1 The `sc_stop` function

SystemC provides a function `sc_stop()` to stop the simulation. Because all states that can be reached using a simulator must be reached using `TLM.open`, calling `sc_stop()` may not stop the generation of the LTS. With respect to the SystemC specifications, the effect of executing `sc_stop()` in a transition $x \to y$ is to eliminate all the successors of y. If other transitions are pending, then they are explored normally.

4.4.2 Recording the current time

A SystemC simulator, such as the ASI simulator, records the current date. The user can read this data using the function `sc_timestamp()`. Because this value is stored in the state, the state space becomes infinite for all programs containing a timed instruction in an unbounded loop. An example of such program is:

```
SC_THREAD(compute); ...
void compute() {
  while (true) {wait(1,SC_SEC);}}
```

To allow the verification of this program, `TLM.open` provides an option to record only relative durations. This option disables the function `sc_timestamp()`. Using this option, the LTS of the program above has only two states and two transitions: a transition with gate "EXEC" leads from the initial state to the second state and another transition with gate "TE" leads back to the initial state.

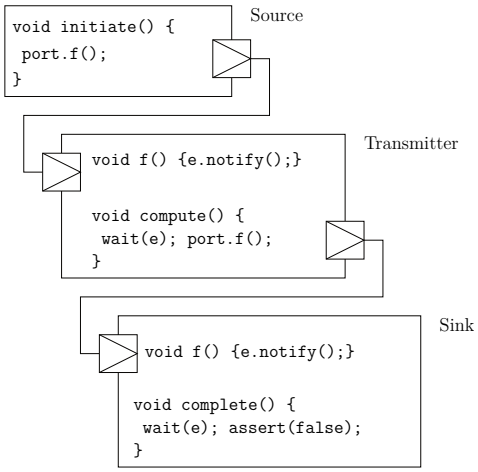

Figure 5 Source code of the `chain` benchmark for $n = 1$.

4.4.3 Pointers and dynamic allocations

It is perfectly safe to use pointers when verifying a SystemC/TLM program with `TLM.open`. Both the pointer and the pointed value must be stored (respectively restored) when the module is stored (resp. restored).

However, dynamic allocations should not be used because a transition can be executed many times, calling `new` creates a memory leak (a second object will be created if the transition is executed again), and calling `delete` corrupts the memory (memory can be freed twice). There is one exception: a `new` statement can be used safely if the corresponding `delete` statement is found in the same transition.

If using dynamic allocation is necessary, then the user must define its own memory allocator. Next, the user must provide `store` and `restore` functions to manage the state of the memory allocator itself. Thus, when a state is restored, the memory allocator knows which objects are allocated and which memory locations are available.

5 Examples

5.1 The `chain` benchmark

We evaluate our new front-end on the benchmark proposed in [27] and reused in [16]. This benchmark consists of a chain of interrupt transmitter modules, whose length is parametrised by n. Modules communicate through transactions, and processes synchronize with events.

Figure 5 presents the SystemC original benchmark for $n = 1$. To increase n, one adds a transmitter module between the last transmitter and the sink module. There are always $n + 2$ threads (functions named `compute` and `process`) and $n + 1$ events (private attribute e of each module).

It is very easy to use `TLM.open` to verify this benchmark because the modules do not contain any dynamic members. Thus, the `store` and `restore` functions can be left empty. The state of the SystemC events and of the SystemC threads (possibly including local variables) are stored automatically.

We have also tried `TLM.open` on a modified version of this benchmark. The modified version uses the `SC_METHOD` instead of the `SC_THREAD`. Using `SC_METHOD` makes the code more difficult to read, but accelerates the simulation and reduces the memory consumption. When replacing a

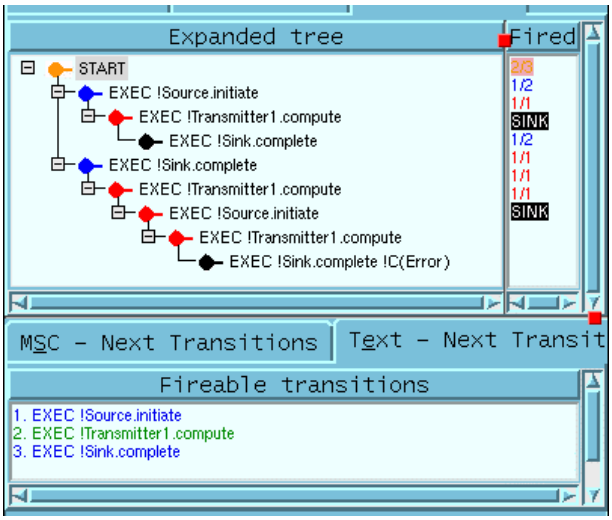

Figure 6 Screen-shot of the OCIS simulator of CADP (`chain` benchmark, $n = 1$).

Table 1 Results of the experiments using `TLM.open`.

$n =$	3	7	11	15	17	19	21
LTS generation (`SC_THREAD`)	1.1 s	1.2 s	2.3 s	35.3 s	193 s	844 s	4314 s
LTS generation (`SC_METHOD`)	1.1 s	1.1 s	1.5 s	11.8 s	62 s	268 s	1445 s
state number	62	1022	16,382	262,142	1,048,574	4,194,302	16,777,214
state number after minimization	47	767	12,287	196,607	786,431	3,145,727	n.a.

`SC_THREAD` by a `SC_METHOD`, local variables have generally to be replaced by module members, and thus must be stored and restored by the user callback methods.

Among the CADP tools that can be used, there is `ocis`, an interactive simulator with backtracking. Figure 6 provides a screen-shot of this tool. Also, for small values of n, the LTS can be fully generated and displayed (cf. Figure 7).

Table 1 presents the results for the generation of the full LTS, using a Macbook machine with 4 GB of memory. For comparison, [27] verifies this benchmark up to $n = 15$ (47 seconds), and [16] verifies this benchmark up to $n = 19$ (8293 seconds for $n = 19$, 60.2 seconds for $n = 15$). These results show a significant improvement compared to the previous approach based on the translation into Promela or LOTOS. The efficiency of `TLM.open` can be explained by two points:

- One transition in the LTS corresponds exactly to one SystemC transition (i.e., the execution of a process between two `wait` statements.) There are no additional transitions used to mimic the behavior of the SystemC scheduler.
- The memory size of a state is kept as small as possible, allowing the model checker to store more states.

The modified benchmark in which `SC_THREAD`s have been replaced by `SC_METHOD`s gives identical LTSes. However, the generation is three times faster, and the memory consumption is reduced: for $n = 19$, generating the LTS for the original benchmark requires 650 MB whereas the modified version requires only 387 MB.

In this experiment, `SC_THREAD`s are stored using hierarchical states. We have tried another

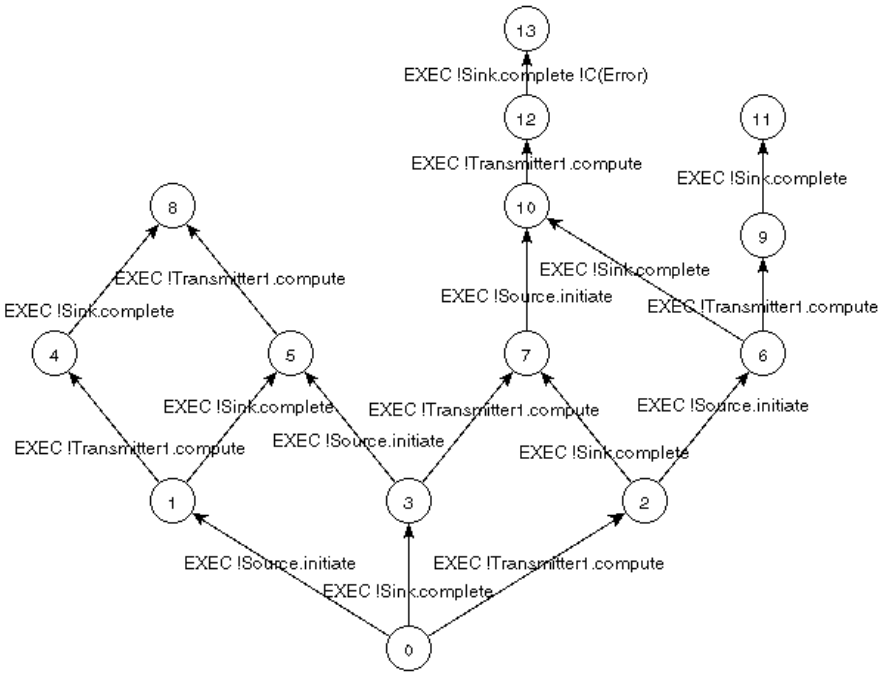

Figure 7 The LTS of the `chain` benchmark for $n = 1$ (output of `bcg_edit`).

implementation using only flat states. Using the original benchmark with `SC_THREAD`s, the flat state technique leads to an explosion of the memory consumption: 700 MB instead of 3.5 MB for $n = 12$ and the LTS generation is about 1.5 times slower.

5.2 The LusSy benchmark

The thesis [23] describes another SystemC/TLM benchmark, which is similar to the `chain` benchmark. The main difference is that the LusSy benchmark uses real transactions which are routed by a bus model.

It is worth noting that LusSy has a special interpretation of the timing annotations [23]. `TLM.open` provides an option to mimic the semantics of LusSy. This allows a greater number of schedules than the official specification, because it considers that all durations are equal.

Using `TLM.open`, instrumenting this benchmark with `store()` and `restore()` functions is trivial for all modules but the bus model because the whole state is contained in `SC_THREAD` stacks, which are automatically stored and restored. The bus model requires about 40 additional lines of code, used for storing and restoring the list of pending transactions. When using LusSy, no additional code is needed. However, LusSy does not use the bus model code. Indeed, LusSy is currently restricted to a few bus models for which a corresponding automaton model has been manually provided. Modeling a bus using automata requires more work and knowledge than adding `store()` and `restore()` functions. Therfore using LusSy is not easier than using `TLM.open`.

Table 2 provides the results obtained with `TLM.open`. It appears that `TLM.open` uses less memory than LusSy combined with SMV. Thus, `TLM.open` can verify this benchmark up to $n = 18$, whereas LusSy does not work over $n = 13$ (with a common memory limit fixed at 512 MB). For $n = 12$, `TLM.open` needs only two seconds where LusSy+SMV spends over one hour.

Table 2 Results for the LusSy benchmark verification using `TLM.open`.

n	memory consumption	time
$n = 15$	30.8 MB	11.3 sec
$n = 16$	64.6 MB	24.1 sec
$n = 17$	136.1 MB	52.6 sec
$n = 18$	289.8 MB	116.3 sec

5.3 Application to a timer

This subsection illustrates the features provided by `TLM.open` by showing how it can be applied to a simple but realistic example. We consider a timer with two registers; PERIOD and ACK.

- Writing a non-null value to PERIOD starts the timer.
- When enabled, the timer generates an interrupt periodically.
- Writing to ACK acknowledges the interrupt.
- Writing 0 to PERIOD stops the timer.

We have 4 TLM models for this timer. The first comes from the SimSoC project [19]; the second is identical to the first with a bug fix; the third and the fourth were provided respectively by an engineer and a PhD student.

In order to verify the first TLM model, which contains 80 lines of code, we had to write 17 additional lines of code to implement the `store()` and `restore()` functions. The timer verification requires the design of an environment modeling the commands generated by the embedded software. For this example, we decided to describe the environment using SystemC code. Here, the code of the environment process:

```
void compute () {
  switch (rand(5)) {
  case 0: puts(‘‘stop’’); port->write(Timer::PERIOD_REG_OFFSET,0); break;
  case 1: puts(‘‘start’’); port->write(Timer::PERIOD_REG_OFFSET,5); break;
  case 2: puts(‘‘ack’’); port->write(Timer::ACK_REG_OFFSET,1); break;
  case 3: {
    std::ostringstream oss;
    oss <<‘‘read_period:’’ <<port->read(Timer::PERIOD_REG_OFFSET);
    puts(oss.str().c_str());
    break;}
  case 4: {
    std::ostringstream oss;
    oss <<‘‘read_ack:’’ <<port->read(Timer::ACK_REG_OFFSET);;
    puts(oss.str().c_str());
    break;}
  case 5: puts(‘‘wait’’); next_trigger(5,sc_core::SC_MS); return;
  }
  next_trigger(sc_core::IMMEDIATE_WAKE_UP);
}
```

Note that we trigger the timer with only one specific period. Explicit model-checking does not permit the verification of this model for all values of the period. Thus, we have to assume that the presence of bugs does not depend on this particular period.

The last line uses a special feature of `TLM.open`: the process yields back to the scheduler but remains eligible. This statement is similar to the `yield()` statement introduced in [15]. The rationale of this statement is to break critical sections that would exist in the model but not in the real system.

Firstly, we applied on-the-fly property checking. The property checker of CADP revealed an error in the first version: for some particular schedules, the timer could generate an interrupt after it was stopped. A counter-example was automatically shown, allowing us to fix the bug. Another minor bug was found in the third version.

Secondly, we applied equivalence checking. We generated the LTS of each timer TLM model, we hid the internal transitions and we minimized them according to the branching equivalence. We got the proof that the second and the forth version are bisimilar modulo branching equivalence. It means that if one contains an error, the other contains the same error. Obviously, the first and third versions are not bisimilar, since they contain distinct errors.

5.4 Application to a basic system

In order to evaluate the behavior of `TLM.open` on a system made of many modules, we studied a basic system that was originally used for practical work. This system was implemented on FPGA. It contains a MicroBlaze processor, a VGA controller, plus the usual and mandatory peripherals: bus, memory, timers, interrupt controller. In the SystemC/TLM model, the user can model the processor, by either using a native wrapper or an instruction set simulator (ISS). The embedded software compute images and manage the configuration of the peripherals.

For the validation of the embedded software, we decided to use the native wrapper instead of the ISS. On the one hand, there is nothing that prevents us using the native wrapper for this software (i.e., no inline assembly code and no dynamic code loading). On the other hand, using the ISS would multiply the number of states: 1 state per binary instruction with the ISS instead of one state per explicit synchronization point with the native wrapper.

Thus, we have instrumented all modules with `store()` and `restore()` methods. Then, we changed the output functions so that traces are added to LTS labels instead of sent to the terminal. Finally, we made some simplifications: 1. we have disconnected the graphical library used by the VGA module, which means that during model checking we do not display the simulated VGA screen. 2. We have simplified the TLM protocol so that it no longer uses a transaction pool because the transaction pool mechanism is only a trick to make simulations a little faster.

Using `TLM.open`, we generated the LTS corresponding to this basic SystemC model for one embedded software. Using the `bcg_min tool` of CADP, the LTS can be reduced to a minimal LTS. This minimal LTS is small enough to be read by human. By observing this LTS, we notice that in some cases the processor was receiving an interruption before any other module raised one. The rationale was a missing `dont_initialize()` in the SystemC code. Because the occurrence of this error depends on the scheduling, this bug had not been noticed before we used `TLM.open`. After fixing this bug, we generated the minimized LTS again. This second LTS is represented by Figure 8.

Finally, we tried to add some errors in the embedded software, such as changing an initial value or disabling a register write in order to verify that all errors can be discovered during model checking. For each error, we got either a SystemC error message (such as assertion failure coming from the TLM code) during LTS generation, or an LTS that was not equivalent to the reference once (equivalence checked using the CADP tool `bisimulator`).

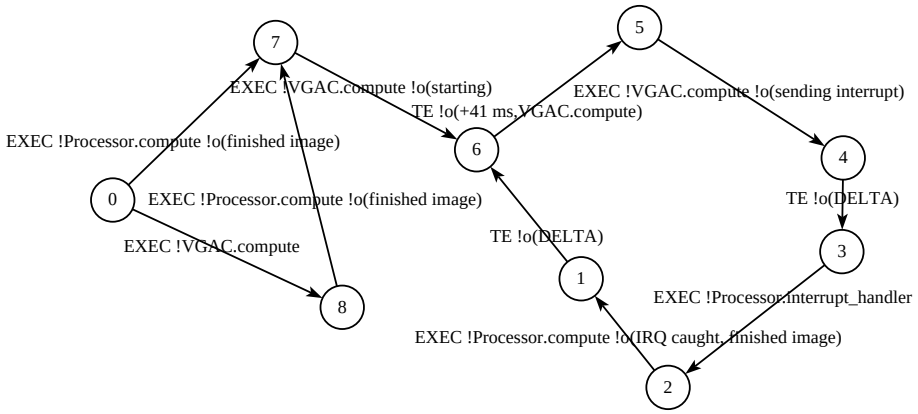

Figure 8 LTS generated for the basic system, after minimization.

6 Conclusion

We presented a new framework for the verification of SystemC/TLM programs. Our new SystemC/TLM front-end avoids the need to translate the whole SystemC/TLM program into another language. Compared to approaches based on manual translation, the verification using `TLM.open` is much simpler: there are less lines of code to write and the engineers do not need to learn a new modeling language. Moreover, `TLM.open` allows better scaling than previous works. Thanks to the numerous tools of CADP, it is now possible to check complex properties and to test the equivalence of two SystemC/TLM programs.

Note that `TLM.open` can be used with pure SystemC programs also (i.e., programs not using TLM). The rationale of calling our tool `TLM.open` instead of `SystemC.open` is related to the abstraction level: the CADP verification toolbox is optimized for asynchronous processes. SystemC/TLM models use asynchronous processes, but SystemC programs modeling a system at a lower level of abstraction use synchronous processes. In order to verify synchronous processes, symbolic model-checker based on BDD or SAT, are in general more efficient than CADP. Thus, `TLM.open` can be used for pure SystemC programs, but is not likely to be the most efficient tool.

As explained in [7], the most difficult task when verifying a SystemC/TLM program is to extract an abstract model that is simple enough to be formally verified. Thus, the main further work is to integrate `TLM.open` in the design flow in such a way that this task becomes simple and safe. Additionally, it would help to automatize the generation of the `store()` and `restore` methods.

References

1 Accellera Systems Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual.*, 2011. URL: `http://www.accellera.org`.

2 Nicolas Blanc and Daniel Kroening. Race analysis for SystemC using model checking. In *2008 International Conference on Computer-Aided Design (ICCAD'08), November 10–13, 2008, San Jose, CA, USA*, pages 356–363. IEEE, 2008. URL: `http://doi.acm.org/10.1145/1509456.1509540`.

3 Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software model checking SystemC. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(5):774–787, 2013. `doi:10.1109/TCAD.2012.2232351`.

4 Rolf Drechsler and Daniel Große. Reachability analysis for formal verification of SystemC. In *2002 Euromicro Symposium on Digital Systems Design (DSD 2002), Systems-on-Chip, 4–6 September 2002, Dortmund, Germany*, pages 337–340. IEEE Computer Society, 2002. `doi:10.1109/DSD.2002.1115387`.

5 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005, pages 110–121. ACM, 2005. doi:10.1145/1040305.1040315.

6 Hubert Garavel. Open/cæsar: An OPen software architecture for verification, simulation, and testing. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings*, pages 68–84. Springer, 1998. Full version available as INRIA Research Report RR-3352. doi:10.1007/BFb0054165.

7 Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an industrial SystemC/TLM model using LOTOS and CADP. In *7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), July 13–15, 2009, Cambridge, Massachusetts, USA*, pages 46–55. IEEE Computer Society, 2009. doi:10.1109/MEMCOD.2009.5185377.

8 Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

9 Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3–7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, July 2007. doi:10.1007/978-3-540-73368-3_18.

10 Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems.* Springer, June 2005. ISBN 0-387-26232-6.

11 Daniel Große and Rolf Drechsler. CheckSyC: an efficient property checker for RTL SystemC designs. In *International Symposium on Circuits and Systems (ISCAS 2005), 23–26 May 2005, Kobe, Japan*, volume 4, pages 4167–4170. IEEE, May 2005. doi:10.1109/ISCAS.2005.1465549.

12 Daniel Große, Hoang M. Le, and Rolf Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26–28 July 2010*, pages 113–122. IEEE Computer Society, 2010. doi:10.1109/MEMCOD.2010.5558643.

13 Claude Helmstetter. *Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis.* PhD thesis, INPG, Grenoble, France, March 2007. URL: http://tel.archives-ouvertes.fr/tel-00350929.

14 Claude Helmstetter, Florence Maraninchi, and Laurent Maillet-Contoz. Test coverage for loose timing annotations. In *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26–27, and August 31, 2006, Revised Selected Papers*, volume 4346, pages 100–115. Springer, August 2006. doi:10.1007/978-3-540-70952-7_7.

15 Claude Helmstetter, Florence Maraninchi, and Laurent Maillet-Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods in System Design*, 35(2):152–189, 2009. doi:10.1007/s10703-009-0075-z.

16 Claude Helmstetter and Olivier Ponsini. A comparison of two SystemC/TLM semantics for formal verification. In *6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2008), June 5–7, 2008, Anaheim, CA, USA*, pages 59–68. IEEE Computer Society, June 2008. doi:10.1109/MEMCOD.2008.4547687.

17 Paula Herber, Marcel Pockrandt, and Sabine Glesner. Transforming SystemC transaction level models into UPPAAL timed automata. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11–13 July, 2011*, pages 161–170. IEEE, 2011. doi:10.1109/MEMCOD.2011.5970523.

18 ISO/IEC. Lotos – a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Genève, September 1989.

19 Vania Joloboff and Claude Helmstetter. SimSoC: A SystemC TLM integrated ISS for full system simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on.* IEEE, 2008. doi:10.1109/APCCAS.2008.4746381.

20 Daniel Kroening and Natasha Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11–14 July 2005, Verona, Italy, Proceedings*, pages 101–110. IEEE, 2005. doi:10.1109/MEMCOD.2005.1487900.

21 Sudipta Kundu, Malay K. Ganai, and Rajesh Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8–13, 2008*, pages 936–941. ACM, 2008. doi:10.1145/1391469.1391706.

22 Kevin Marquet, Matthieu Moy, and Bertrand Jeannet. Efficient Encoding of SystemC/TLM in Promela. In *Workshop on Design, Analysis and Tools for Integrated Circuits and Systems at the International MultiConference of Engineers and Computer Scientists 2011, DATICS-IMECS*, pages 1039–1044, 2011. URL: http://www.iaeng.org/publication/IMECS2011/IMECS2011_pp1039-1044.pdf.

23 Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level.* PhD thesis, INPG, Grenoble, France, December 2005. URL: http://www-verimag.imag.fr/~moy/phd/.

24 Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: an open tool for the

analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 10(2–3):73–104, 2005. doi:10.1007/s10617-006-9044-6.

25 Bernhard Niemann and Christian Haubelt. Formalizing TLM with communicating state machines. In *Forum on specification and Design Languages, FDL 2006, September 19–22, 2006, Darmstadt, Germany, Proceedings*, pages 285–293. ECSI, 2006.

26 Olivier Ponsini and Wendelin Serwe. A schedulerless semantics of TLM models written in SystemC via translation into LOTOS. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26–30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-68237-0_20.

27 Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1–3, 2007, Proceedings*, volume 4595 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2007. doi:10.1007/978-3-540-73370-6_14.

Randomized Caches Considered Harmful in Hard Real-Time Systems

Jan Reineke

Saarland University
Saarbrücken, Germany
reineke@cs.uni-saarland.de

Abstract

We investigate the suitability of caches with randomized placement and replacement in the context of hard real-time systems. Such caches have been claimed to drastically reduce the amount of information required by static worst-case execution time (WCET) analysis, and to be an enabler for measurement-based probabilistic timing analysis. We refute these claims and conclude that with prevailing static and measurement-based analysis techniques caches with deterministic placement and least-recently-used replacement are preferable over randomized ones.

2012 ACM Subject Classification Computer systems organization∼Real-time system architecture, Theory of computation∼Caching and paging algorithms, Hardware∼Safety critical systems
Keywords and phrases Real-time systems, caches, randomization, WCET analysis
Digital Object Identifier 10.4230/LITES-v001-i001-a003
Received 2013-12-04 **Accepted** 2014-05-28 **Published** 2014-06-10

1 Introduction

Recent work has promoted the use of randomized caches in hard real-time systems [4, 20, 22, 23, 21, 7, 5, 25]. Along with randomized microarchitectures, this line of work proposes static probabilistic timing analysis (SPTA) and measurement-based probabilistic timing analysis (MBPTA). Caches are a major challenge in the timing analysis of traditional, deterministic microarchitectures. A key feature of randomized microarchitectures are caches with randomized placement and replacement. Such caches have been claimed to drastically reduce the amount of information required by WCET analyses. To quote Kosmidis et al. [23]: "The key benefit of embracing PTA (probabilistic timing analysis) is that execution timing becomes dramatically less dependent on execution history, with drastic reduction in the amount of information required to obtain tight WCET estimates in comparison to other timing analysis approaches."

In this paper, we critically assess these claims both in the context of static and measurement-based analysis. Specifically, we compare the precision of static cache analyses for caches with *least-recently-used* (LRU) replacement and with randomized replacement provided the *same* amount of information, i.e. the information stated to be sufficient for the analysis of randomized caches. Among deterministic caches we restrict our attention to those with LRU replacement, as it is widely considered to be the most predictable replacement policy, and it has been demonstrated to be efficiently implementable [1, 8]. Our analysis demonstrates that, with simple, state-of-the-art analyses, deterministic LRU replacement is preferable over random replacement. We also observe that, with its current restrictions, MBPTA is equally applicable to LRU caches as it is to randomized ones.

Regarding random placement, we show that it is impossible to assign non-zero hit probabilities to individual memory accesses that are *independent* of the outcome of other accesses. This means that caches with random placement are *not* amenable to the prevailing SPTA approach that relies on independence, as execution time profiles (ETPs) of individual instructions are convolved.

Finally, we provide a class of memory access sequences that is problematic for MBPTA under random placement. On these sequences, which may occur in practice due to loops, MBPTA either fails, is incorrect, or highly imprecise.

We provide the necessary background about probabilistic timing analysis in Section 2. Then, we introduce deterministic and randomized caches in Section 3. We assess the suitability of random placement and replacement for use in hard real-time systems in Sections 4 and 5. Finally, we briefly summarize our findings in Section 6.

2 Static and Measurement-Based Timing Analysis

The goal of static and measurement-based timing analysis for deterministic microarchitectures is to compute tight upper bounds on the worst-case execution times (WCET) of programs. The goal of timing analyses for randomized microarchitectures is slightly different: in such microarchitectures, very high execution times are possible, but—hopefully—only with a very low probability. Thus, timing analyses for such microarchitectures compute *exceedance functions*. These exceedance functions determine upper bounds on the probability of exceeding any given execution time. From such a function, and a probability threshold p, an execution time can be obtained that is exceeded only with a probability of e.g. $p = 10^{-12}$.

2.1 Static Probabilistic Timing Analysis

The de facto standard approach to static timing analysis (STA) for deterministic microarchitectures divides analysis into two main parts [31]:

1. *Low-level analysis*, which determines execution-time bounds for basic blocks (or other small contiguous program fragments) based on an accurate model of the underlying microarchitecture.
2. *Path-level analysis*, which determines an upper bound on the execution time of the program as a whole based on constraints on the control flow, e.g. loop bounds, and the execution-time bounds for basic blocks determined by low-level analysis.

A critical assumption of this approach is that the bounds obtained for a basic block during low-level analysis hold for *all possible* execution histories leading to the respective basic block. As execution times may depend heavily on the execution history, low-level analysis is often made context sensitive, e.g. by distinguishing the first iteration of a loop from the following ones.

While so far less studied and thus less developed, static probabilistic timing analysis (SPTA) follows a similar approach [4]:

1. For each instruction in the program, an *execution time profile* (ETP), i.e., a discrete probability distribution over the instruction's possible execution times, is derived. This step corresponds to the low-level analysis in STA.
2. To arrive at an ETP for a sequence of instructions the ETPs of all instructions in the sequence are combined by convolution. If multiple different execution paths are possible, their ETPs can be merged conservatively [4]. This roughly corresponds to path-level analysis in STA. From an ETP, a corresponding exceedance function can then be determined easily.

A critical assumption for SPTA to be sound is that the ETPs derived in step one are *independent* of each other. Only if they are independent, can they be soundly combined by convolution to arrive at an ETP for a sequence of instructions.

2.2 Measurement-based Probabilistic Timing Analysis

Measurement-based probabilistic timing analysis (MBPTA) derives exceedance functions for the execution time of a program from measurements. MBPTA as described by Cucu et al. [5] is

based on Extreme Value Theory (EVT). In this approach, a series of end-to-end execution-time measurements is performed. The measurement results are used to estimate the parameters of an extreme value distribution, the Gumbel distribution. Measurements and estimation of the Gumbel distribution are interleaved until the distribution is considered to have converged [5]. The thus obtained Gumbel distribution then immediately induces an exceedance function.

Applicability of this approach relies on two assumptions:

1. The execution-time measurements can be modeled by independent and identically-distributed (i.i.d.) random variables.
2. The maximum of a sample of these i.i.d. random variables converges in distribution to the Gumbel distribution.

To satisfy the first assumption, Cucu et al. [5] propose a number of changes to the microarchitecture to eliminate the dependence of execution times on input data. For instance, input-dependent memory accesses must bypass the cache. They also initially limit their approach to single-path programs, which they later [5] show how to relax.

The satisfaction of the second assumption is validated during the analysis of a particular program by statistical tests.

3 Deterministic and Randomized Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To reduce management overhead and to profit from spatial locality, data is not cached at the granularity of words, but at the granularity of so-called *memory blocks*. To this end, main memory is logically partitioned into the set of equally-sized memory blocks $\mathcal{B} = \{0, \ldots, n\}$. Blocks are cached as a whole in *cache lines* of the same size. The size of a memory block varies from one processor to another, but is usually between 32 and 128 bytes.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache (a *cache hit*) or not (a *cache miss*). To enable an efficient look-up, each memory block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* k of the cache.

The *placement policy* determines the cache set a memory blocks maps to. In Section 3.2 we describe common deterministic and randomized placement policies.

Since caches are usually much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. In Section 3.1 we describe common deterministic and randomized replacement policies.

The performance of a cache depends on the *temporal* and *spatial locality* of the memory accesses. In Section 3.3, we describe two notions of locality that are approximated by state-of-the-art static (probabilistic) cache analyses.

3.1 Replacement Policies

Usually, replacement policies treat each cache set separately, so that accesses to a particular cache set do not influence replacement decisions in other cache sets. While exceptions to this rule exist, they have been identified as particularly unsuitable for real-time systems [17]. Thus, in the following, we only consider replacement policies treating each cache set separately.

Well-known *deterministic* replacement policies in this class are *least-recently used* (LRU), used in various Freescale processors such as the MPC603E and the TriCore17xx, as well as the recent Kalray MPPA 256; *pseudo-LRU* (PLRU), a cost-efficient variant of LRU, used in the

Freescale MPC750 family and multiple Intel microarchitectures; *most-recently used* (MRU), also known as *not most-recently used* (NMRU), another cost-efficient variant of LRU, used in the Intel Nehalem; *first-in first-out* (FIFO), also known as ROUND ROBIN, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC55xx family

Logically, LRU orders cached memory blocks by the recency of their last use, from most- to least-recently-used. Upon a miss, the least-recently-used block is evicted. Among deterministic policies, LRU is generally accepted as the most predictable policy [28]. Thus, in the following, among deterministic policies, we restrict our attention to LRU, which has been shown to be efficiently implementable [1], and which is used in the Kalray MPPA 256 [8] for predictability.

Quiñones et al. [25, 20, 21, 4] have promoted the use of *randomized* caches in real-time systems. They have focused on a policy, which we will call Random in the following, that was introduced by Belady [2]. Upon a miss, Random chooses the block to evict randomly and uniformly among the k cache lines of the cache set. Thus, upon a miss, a cached block—in the cache set that the accessed block maps to—is evicted with probability $\frac{1}{k}$. This policy is also referred to as *evict-on-miss* in the literature [7].

Several commercial processors are claimed to employ random replacement, e.g. the ARM720T, the ARM940T, the ARM11xx, and the Freescale MPC7450. However, most processor documentations are inconclusive about the exact meaning of "random". Such caches could be based on hardware random number generators that generate random numbers from a physical process, such as thermal noise, or they could employ deterministic pseudo-random number generators. The well-documented MPC7450 [18] allows to choose between two random policies for its second-level caches [10]: "The simpler one uses a modulo counter that is incremented on each clock cycle and whose value determines the cache line to replace."

To achieve independence between cache-miss probabilities, Cazorla et al. [4] have also proposed the *evict-on-access* policy, which evicts a block uniformly at random upon each memory access, rather than upon each cache miss. In the following, we limit our attention to Random, as it provably dominates *evict-on-access* in terms of the induced exceedance function on any workload.

Randomized policies have been studied extensively in the context of competitive analysis [3]. Policies such as Mark and Equitable have been shown to have smaller competitive ratios than *any* deterministic policy. These results concern the *expected performance* of a policy, rather than the performance achieved with high probability, which would be of greater interest in the hard real-time setting. However, recent results by Komm et al. [19] suggest that randomized policies can also be shown to be competitive "with high probability".

3.2 Placement Policies

A placement policy can be formalized as a mapping from memory blocks to cache sets: $place : \mathcal{B} \to \{0, \ldots, s-1\}$, where s is the number of cache sets.

The most common *deterministic* placement policy for set-associative caches is *modulo* placement: $place_{modulo}(b) = b \bmod s$. The number of sets s is usually a power of two, so that the cache set of a block is simply determined by its $\log s$ least significant bits.

In *random* placement the mapping $place_{random}$ from memory blocks to cache sets is chosen randomly from the set of all mappings $\mathcal{B} \to \{0, \ldots, s-1\}$. For static-analysis purposes it is convenient if the mapping is chosen uniformly at random from this set. Then, the probability of block b mapping to cache set t, $P(place_{random}(b) = t)$, is $\frac{1}{s}$. Note, that $place_{random}$ needs to be *fixed* for the entire execution of a program. Otherwise, it would not be possible to locate memory blocks that were cached earlier under a different mapping.[1]

[1] Changing the mapping at runtime requires either flushing of cache contents or their migration.

Kosmidis et al. present an approximation of random placement in hardware [20, 21] based on a parametric hash function and in software [22] on top of conventional caches with deterministic placement. The crucial difference between the hardware and the software solution is that the software solution can only randomize mapping at the granularity of "memory objects", i.e., memory entities normally stored in consecutive memory addresses such as functions, basic blocks, or arrays.

3.3 Notions of Locality

Caches rely on *locality* in the memory access sequences generated by programs. Different ways of capturing locality have been proposed over time. Two notions of locality particularly relevant to LRU and Random replacement are the *reuse distance* and the *stack distance* of a memory access.

The *reuse distance* of a memory access to block b is the number of memory accesses between the current and the previous access to block b. The first access to a block has reuse distance ∞. As an example, we have annotated each memory access in the following sequence with its reuse distance:

$$a^\infty, b^\infty, b^0, a^2, c^\infty, d^\infty, d^0, c^2, b^5, a^5.$$

In contrast to the reuse distance, the *stack distance* of an access to block b is defined as the number of *distinct* memory blocks accessed between the current and the previous access to block b. The stack distance of a block is sometimes also referred to as the *age* of the block. The first access to a block has stack distance ∞. In the sequence from above, the stack distances are as follows:

$$a^\infty, b^\infty, b^0, a^1, c^\infty, d^\infty, d^0, c^1, b^3, a^3.$$

Note that the stack distance of any access is less than or equal to its reuse distance.

We will see later how hit and miss probabilities of a memory access can be given based on its reuse and stack distance for both randomized and deterministic caches.

3.4 Cache Analysis in Static (Probabilistic) Timing Analysis

Cache analysis is an important part of low-level analysis. In STA, its purpose is to classify memory accesses in the program as either definite hits or definite misses. Sometimes, an access may result in a hit *or* a miss depending on the execution history leading to the access. As a consequence of such inherent uncertainty or uncertainty due to analysis imprecision, cache analysis may also classify an access as "unknown". Due to timing anomalies [24, 29] it is not always safe to simply assume a cache miss in case of uncertainty.

Similarly, in SPTA [4], a probability needs to be attached to the hit and the miss case for each memory access. Current SPTAs assume microarchitectures in which hits and misses have a fixed, context-independent cost and thus cache-related timing anomalies may not occur. As a consequence, it is sufficient to determine a lower bound h on the hit probability of an access, which induces an upper bound of $1 - h$ on its miss probability. Together with hit and miss latencies $hit_{latency}$ and $miss_{latency}$ we get the following ETP for a memory instruction with hit probability h:

$$\begin{pmatrix} hit_{latency} & miss_{latency} \\ h & 1-h \end{pmatrix}.$$

The issue of timing anomalies in the pipeline is orthogonal to that of using deterministic or randomized caches. In order not to mix the two issues, we compare deterministic[2] and randomized caches in the context of SPTA, i.e., in terms of deriving lower bounds on the hit probability of a memory access.

[2] Caches with LRU replacement do not exhibit timing anomalies.

4 Deterministic versus Random Replacement in Fully-Associative Caches

4.1 In Static Probabilistic Timing Analysis

In a cache with LRU replacement, an access b with stack distance $sd(b)$ less than the associativity k is a hit, otherwise it is a miss:

$$P(hit_{LRU}(b)) = \begin{cases} 1 & : sd(b) < k \\ 0 & : sd(b) \geq k \end{cases} \tag{1}$$

Note, that the hit probabilities of different memory accesses in a sequence are *independent*. Static cache analyses thus determine upper bounds on the stack distance of each memory access to guarantee cache hits. Such analyses are called *must analyses* [9]. Analogously, *may analyses* [9] determine lower bounds on stack distances to guarantee cache misses.

In the case of a fully-associative cache there are two challenges for may and must analyses:

1. The initial state of the cache is unknown. Thus, must analyses conservatively assume an upper bound $> k$ on the stack distance of any block at program start. Similarly, may analyses assume a lower bound of 0.
2. Logically, memory accesses are at the granularity of words, not memory blocks. Thus a *value analysis* needs to determine for pairs of memory accesses whether they refer to the same memory block or not.[3] This is trivial for instruction caches, but may be very hard for input-dependent data accesses.

In a cache with Random replacement the situation is different. In contrast to LRU, the hit probability of an access cannot be given purely in terms of its stack or reuse distance. Zhou [32] observes that the hit probability of an access b to a block that has been accessed before is

$$P(hit_{Random}(b)) = \left(1 - \frac{1}{k}\right)^m \tag{2}$$

where m is the number of cache misses between access b and the previous access to the same memory block. Clearly m is bounded from above by b's reuse distance $rd(b)$, so

$$P(hit_{Random}(b)) \geq \left(1 - \frac{1}{k}\right)^{rd(b)} \tag{3}$$

This formula correctly underestimates the hit probability of an individual access. Unfortunately, however, hit probabilities computed with the formula above are *not* independent of each other. Thus, the convolution of corresponding ETPs may underestimate the probability of observing a given number of misses. Consider, e.g., the access sequence a, b, c, a, b, c and a cache with associativity 2. Clearly, *at least one* miss must occur on the final three accesses of the sequence, as the first access to c will evict either a or b. Yet, the convolution of the ETPs obtained from Equation (3) yields a non-zero probability of having *no* misses on those three accesses, because the hit probability of each individual access is greater than zero[4]. As a consequence, the probability of observing four or more cache misses on the entire sequence, which is 1, would be underestimated.

[3] Note, that it is not necessary to determine which memory block is referred to by a memory access. It is sufficient to determine for pairs of accesses whether they refer to the same block or not. This is exploited by *relational cache analysis* [16].

[4] According to Equation (3), it is $\frac{1}{4}$.

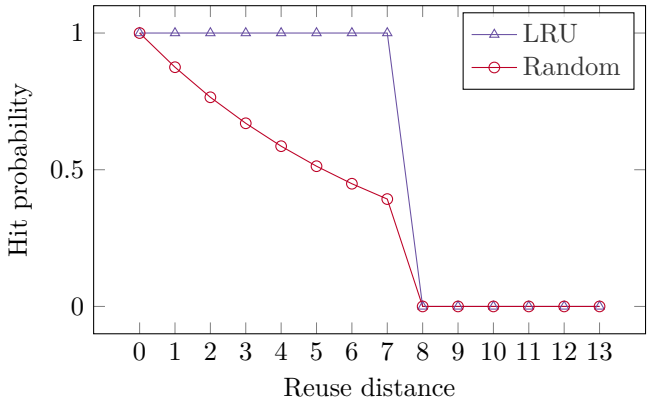

Figure 1 Lower bounds on hit probabilities of memory accesses for LRU and Random in terms of reuse distances for a cache of associativity 8 based on Equations 1 and 4.

Davis et al. [7] provide the following formula, which bounds the hit probability of an access b, *independently* of whether preceding accesses hit or miss, in terms of the reuse distance of b:

$$P(hit_{Random}(b)) \geq \begin{cases} \left(1 - \frac{1}{k}\right)^{rd(b)} & : rd(b) < k \\ 0 & : rd(b) \geq k \end{cases} \tag{4}$$

The more optimistic formula for hit probabilities given in [20] has been refuted in [6]. Based on the reuse distance, the above formula is the most precise hit probability that holds independently of the outcome of previous memory accesses.

For associativity 8, Figure 1 illustrates the hit probabilities of LRU and Random in terms of reuse distances. Remember that by definition the stack distance $sd(b)$ of an access is less than or equal to its reuse distance $rd(b)$. With this in mind, comparing the hit probabilities for LRU and for Random from Equations 1 and 4, we make the following two observations:

▶ Observation 1. With the *same information* about an access sequence, i.e. upper bounds on the reuse distances of accesses, the hit probabilities for LRU are always greater than or equal to the hit probabilities for Random.

▶ Observation 2. In case of LRU, and in contrast to Random, current cache analyses can profit from bounding stack distances, which can be arbitrarily lower than reuse distances.

▶ Conclusion 1. With simple, state-of-the-art analysis methods, LRU replacement is preferable over Random replacement in static (probabilistic) timing analysis.

In general, we note that the state (or the state distribution in case of randomized caches) of *any* cache is a function of the history of memory accesses. This holds independently of whether the cache is deterministic or randomized. More precisely, the state of any fully-associative cache depends, at least, on the suffix of the history of memory accesses containing k distinct blocks, where k is the associativity of the cache. Among all policies, randomized or deterministic, the state of an LRU-controlled cache depends on the *shortest* suffix of the access history. Thus, LRU requires the least information about the access history to fully determine its state [28].

Other Deterministic Policies: FIFO, PLRU, and MRU. Common deterministic policies such as FIFO, PLRU, and MRU have been found to be less predictable than LRU [28]. In contrast to LRU, the exact hit probability of an access cannot be determined in terms of stack or reuse

distances for these policies. The best lower bounds on hit probabilities that can be given based on stack distances for FIFO, PLRU, and MRU are:[5]

$$P(hit_{FIFO}(b)) \geq \begin{cases} 1 & : sd(b) < 1 \\ 0 & : sd(b) \geq 1 \end{cases} \tag{5}$$

$$P(hit_{PLRU}(b)) \geq \begin{cases} 1 & : sd(b) < \log_2 k + 1 \\ 0 & : sd(b) \geq \log_2 k + 1 \end{cases} \tag{6}$$

$$P(hit_{MRU}(b)) \geq \begin{cases} 1 & : sd(b) < 2 \\ 0 & : sd(b) \geq 2 \end{cases} \tag{7}$$

Static (probabilistic) timing analyses that are based purely on stack (or reuse) distances thus yield worse results under FIFO than under Random, MRU, PLRU, or LRU. MRU and PLRU are incomparable with Random. Depending on the benchmark, and the resulting distribution of reuse distances, MRU and PLRU may yield better results than Random and vice versa.

It should be noted that there are access sequences (and programs that generate such sequences) on which Random outperforms LRU and other deterministic policies with a very high probability. A prime example for such sequences are so-called "payroll sequences", i.e. sequences of the form $(a_1, \ldots, a_{k+1})^*$, where k is the associativity of the cache. On such sequences LRU incurs cache misses only. Similar sequences can be constructed for *every* deterministic policy.

To profit from Random replacement in such cases, more sophisticated analysis techniques are required. Such analyses will likely have to derive conditional hit probabilities and combine results for individual memory accesses in a different way than convolution, which is not required in case of LRU. Similarly, sophisticated static analyses have recently been proposed for FIFO [11, 12, 15], PLRU [13], and MRU [14]. Yet, while being more complex than Ferdinand's analysis for LRU [9], they still do not quite achieve the same level of precision.

Stack Distance versus Reuse Distance. As an example where reuse and stack distances may differ a lot, consider instruction accesses following the execution of a small, nested loop:

```
1 x = 0
2 y = 0
3 for i in [1, 1000]:
4     for j in [1, i]:
5         x = x+1
6     y = y+1
```

Assume for simplicity that the instructions implementing each line of the program occupy exactly one memory block. Thus, the program's instructions occupy exactly six memory blocks. Then, except for the first access, the stack distance of each instruction access for `y = y+1` is three. Yet, the corresponding reuse distance depends on the value of `i`. In the last iteration of the outer loop the reuse distance of the instruction access for `y = y+1` is 2001.

In practice, the relation between reuse and stack distances likely varies strongly within benchmarks and from one benchmark to another. Unfortunately, we were not able to find empirical data concerning their relation, with one exception: Sen and Wood [30] plot the distribution of reuse distances for accesses of a given stack distance (Figure 3 in [30]) for an online transaction processing application.

[5] This follows immediately from the competitiveness of the respective policies relative to LRU [26].

4.2 In Measurement-Based Probabilistic Timing Analysis

MBPTA derives WCET estimates from a series of execution-time measurements. Cache performance depends on the initial state of the cache and on the sequence of memory accesses provided to the cache. Cucu et al. [5] flush the cache upon program start and eliminate all input-dependent memory accesses from the program. Thus, on a given path through the program, the sequence of memory accesses is the same for different program inputs. Then, a number of end-to-end execution-time measurements is performed on each program path. By nature of the approach, the analysis results apply only to those program paths for which measurements have been performed.

In such a scenario, i.e., flushed cache and no input-dependent memory accesses, the requirements of MBPTA, namely independence and identical distribution of execution times on a given program path, are also met by a conventional cache with LRU replacement: the cache behavior will be identical on each measurement; it follows a degenerate probability distribution. Independence is thus trivial. In fact, the same argument applies to *any* deterministic cache replacement policy.

Thus, under the conditions described above, a *single* measurement will reveal the worst case—in terms of cache performance. This compares with having to perform hundreds of measurements in case of a randomized cache [5].

▶ Conclusion 2. Deterministic replacement yields more efficient MBPTA than Random replacement.

For LRU, the empty state is the worst-case initial state for any memory access sequence [27]. Therefore, measurements obtained starting with a flushed cache yield upper bounds on the number of cache misses under any initial cache state for LRU. In this case, flushing the cache would only be required during the measurement-based analysis and could in principle be disabled during normal operation, assuming the microarchitecture features no timing anomalies [24, 29].

5 Deterministic versus Random Placement in Set-Associative Caches

5.1 In Static Probabilistic Timing Analysis

In a set-associative cache with s cache sets, the placement policy partitions the stream of memory accesses into s substreams, each of which is processed by one of the s cache sets. For set-associative caches, it is convenient to define the reuse and the stack distance of an access based on the subsequence the access belongs to. In other words, the *reuse distance* of an access to block b is the number of memory accesses between the current and the previous access to block b within the *same* cache set. Let the *stack distance* be defined analogously for set-associative caches.

Then we get the same hit probabilities in terms of stack and reuse distance for LRU and Random as in case of a fully-associative cache.

The additional difficulty in static cache analysis with *deterministic placement* is thus to determine which memory accesses map to the same cache set. This is again trivial for instruction accesses, but may be very difficult for data accesses. Note, however, that it is *not* required to determine the absolute cache set an access maps to, as demonstrated by relational cache analysis [16].

Randomized placement [20, 21] promises to reduce the analysis effort for set-associative caches, as two memory blocks will only collide in the cache with a certain probability. If the placement function is chosen randomly from a uniform distribution over all possible placement functions, then the probability of any two blocks to map to the same cache set is $\frac{1}{s}$. Based on this assumption, Kosmidis et al. [20] derive the following hit probability for a direct-mapped cache in terms of the

stack distance[6] of an access:

$$P(hit_{Random}(b)) = \left(1 - \frac{1}{s}\right)^{sd(b)} \tag{8}$$

This formula is correct. However, as in the case of Zhou's formula for random replacement, hit probabilities determined in this way are *not* independent. Consider the following access sequence:

$$a, b, a, b, a, b, a, b, a, b$$

As the placement is chosen randomly at program start and does not change during program execution, there are only two possibilities: either a and b systematically collide in the cache or they do not. They collide with probability $\frac{1}{s}$. Thus with a probability of $\frac{1}{s}$ all ten memory accesses will be cache misses, and with a probability of $1 - \frac{1}{s}$ only two (compulsory) misses will occur. In the example, each access's miss probability is $1 - (1 - \frac{1}{s})^1 = \frac{1}{s}$. Assuming independence of these miss probabilities would incorrectly yield a probability of $\frac{1}{s^8} \ll \frac{1}{s}$ of observing ten misses.

Unfortunately, *no* non-zero hit probability can be assigned to an access based on its stack distance that is *independent* of other whether previous accesses resulted in hits or misses. Yet, independence is required by the current SPTA approach. To see this, consider arbitrarily long sequences of the form $a, b, a, b, a, b, \ldots$ No matter how long the sequence, with a probability of $\frac{1}{s}$ all accesses will miss in a direct-mapped cache. For any non-zero hit probability p assigned to each individual access there is a length n of the sequence, such that the convolution of the ETPs based on p will underestimate the probability of incurring n misses.

▶ Observation 3. In case of random placement, no mutually independent hit probabilities greater than zero can be assigned to individual memory accesses with stack distances greater than zero.

Observation 3 immediately implies the following conclusion:

▶ Conclusion 3. Random placement requires complex static analyses that take into account conditional hit probabilities. Random placement is thus not amenable to current analysis approaches.

5.2 In Measurement-Based Probabilistic Timing Analysis

As we have seen in the previous section, with random placement there are cases in which we observe very few misses with a high probability p and very many misses with a low probability $1 - p$, with *no* cases in between the two extremes.

If p is sufficiently close to 1, MBPTA is unlikely to ever observe the case of very many misses. Then, its observations are indistinguishable from a case in which many misses are in fact impossible. Consider as an example[7] the same sequence $\sigma_{slow} = a, b, a, b, a, b, \ldots$ as above, which may be generated by a loop, and a very large direct-mapped cache with $s = 10^6$ cache sets. The probabilities of the two possible execution times are depicted in Figure 2.

Even $10000 = 10^4$ measurements will only reveal the worst case with a probability of $1 - (1 - 1/10^6)^{10^4} < 1\%$. In other words, with a probability greater than 99%, all measurements will yield exactly two cache misses, and thus the sequence would be indistinguishable from the sequence $\sigma_{fast} = a, b, b, b, b, \ldots$, which will yield exactly two misses independently of the placement.

---

[6] Here, $sd(b)$ refers to b's stack distance among *all* memory accesses, i.e., *not* to its stack distance among blocks mapping to the same cache set.

[7] While this example is slightly construed, due to the unrealistically high number of cache sets, the sequence $a, b, c, d, a, b, c, d, \ldots$ in a 3-way set-associative cache with a more realistic $s = 10^2$ leads to similar results, yet is more difficult to analyze precisely.

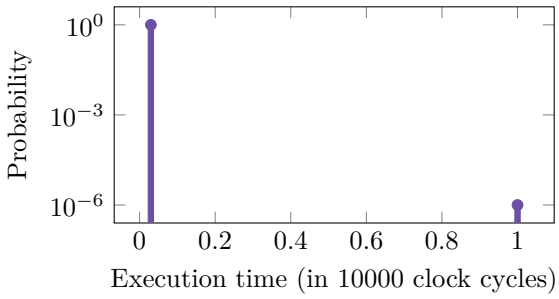

Figure 2 Execution-time distribution on example sequence with random placement.

MBPTA is used to estimate execution times that are only exceeded with a very low probability, such as 10^{-12}, ideally without performing 10^{12} measurements. In our example, with a probability of $10^{-6} \gg 10^{-12}$, the execution time for sequence σ_{slow} will be very high, as all memory accesses will result in cache misses.

MBPTA bases its estimates solely on measurement results. It would thus generate the *same* execution-time distribution for programs that generate the sequences σ_{slow} and σ_{fast} with a high probability. In such a situation there are three possibilities:

1. MBPTA correctly estimates the execution-time distribution for the sequence σ_{slow}.
2. MBPTA incorrectly underestimates the execution-time distribution for the sequence σ_{slow}.
3. Based on the measurement results, the statistical tests in MBPTA reject such programs.

The two latter cases are clearly undesirable. In the first case, MBPTA's estimate would have to be the same for the sequence σ_{fast}. However, a correct estimate for σ_{slow} is necessarily extremely pessimistic for σ_{fast}, which never exhibits more than two cache misses. This leads us to our final conclusion:

▶ Conclusion 4. Random placement is not suitable for MBPTA.

6 Summary

We have critically assessed the suitability of randomized caches for use in hard real-time systems. We observe that when used in SPTA, state-of-the-art cache analyses deliver better hit probabilities for LRU than for Random replacement with the *same* amount of information. With the restrictions currently imposed upon the use of randomized caches in MBPTA, i.e., no input-dependent memory accesses, deterministic caches with LRU replacement can also be safely employed in MBPTA. This comes with the additional benefit of requiring only a single measurement to identify the worst-case cache performance.

Further, we have shown that non-trivial hit probabilities under random placement are *not* independent and can thus not be safely combined by convolution. We have also identified simple access sequences, which may be generated by simple loops, on which MBPTA must—by construction—be either unsound, extremely pessimistic, or fail to produce an estimate at all.

Despite the negative results obtained in this paper, we believe that randomization might have a place in microarchitectures for real-time systems. A benefit over deterministic microarchitectures may be an increase in *robustness*. It is future work to rigorously analyze the benefits of randomization in this direction.

Acknowledgements. I would like to thank the anonymous reviewers for their helpful remarks. This work was supported by the Deutsche Forschungsgemeinschaft as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS).

References

1 Bryan D. Ackland, Alex Anesko, Douglas M. Brinthaupt, Steven J. Daubert, Asawaree Kalavade, Joseph Knobloch, E. Micca, Mallik Moturi, Chris J. Nicol, Jay H. O'Neill, Joseph H. Othmer, Eduard Säckinger, Kanwar J. Singh, J. Sweet, Christopher J. Terman, and Joseph Williams. A single-chip, 1.6 billion, 16-b mac/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, 35(3):412–423, March 2000. doi:10.1109/4.826824.

2 Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi:10.1147/sj.52.0078.

3 Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, New York, NY, USA, 1998.

4 Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems*, 12(2s):94:1–94:26, May 2013. doi:10.1145/2465787.2465796.

5 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS'12, Pisa, Italy*, pages 91–101, Washington, DC, USA, July 2012. IEEE Computer Society. doi:10.1109/ECRTS.2012.31.

6 Robert I. Davis. Improvements to static probabilistic timing analysis for systems with random cache replacement policies. In *2013 4th Real-Time Scheduling Open Problems Seminar, RTSOPS'13*, July 2013.

7 Robert I. Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems, ECRTS'13, Paris, France*, pages 168–179, July 2013. doi:10.1109/ECRTS.2013.27.

8 Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition, DATE'14, Dresden, Germany*, pages 513–518, March 2014. doi:10.7873/DATE2014.110.

9 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999. doi:10.1023/A:1008186323068.

10 Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2012. URL: https://www.epubli.de/shop/buch/Static-Cache-Analysis-for-Real-Time-Systems-Daniel-Grund-9783844216998/13092.

11 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *Static Analysis, 16th International Symposium, SAS'09, Los Angeles, CA, USA*, pages 120–136. Springer, August 2009. doi:10.1007/978-3-642-03237-0_10.

12 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS'10, Brussels, Belgium*, pages 155–164. IEEE Computer Society, July 2010. doi:10.1109/ECRTS.2010.8.

13 Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET'10, Brussels, Belgium*, pages 23–35. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, July 2010. doi:10.4230/OASIcs.WCET.2010.23.

14 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China*, pages 55–64. IEEE, April 2012. doi:10.1109/RTAS.2012.31.

15 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485362.

16 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *24th Euromicro Conference on Real-Time Systems, ECRTS'12, Pisa, Italy*, pages 102–111. IEEE Computer Society, July 2012. doi:10.1109/ECRTS.2012.14.

17 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. doi:10.1109/JPROC.2003.814618.

18 Freescale Semiconductor Inc. MPC7450 RISC microprocessor family reference manual, rev. 5, 2005.

19 Dennis Komm, Rastislav Kràlovic, Richard Kràlovic, and Tobias Mömke. Randomized online algorithms with high probability guarantees. In *31st International Symposium on Theoretical Aspects of Computer Science, STACS'14, Lyon, France*, pages 470–481. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, March 2014. doi:10.4230/LIPIcs.STACS.2014.470.

20 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 513–518. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485416.

21 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Efficient

cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013. doi:10.1109/TC.2013.182.

22 Leonidas Kosmidis, Charlie Curtsinger, Eduardo Quiñones, Jaume Abella, Emery D. Berger, and Francisco J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 603–606. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485435.

23 Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. Achieving timing composability with measurement-based probabilistic timing analysis. In *2013 16th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing, ISORC'13*, 2013.

24 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA*, pages 12–21. IEEE Computer Society, December 1999. doi:10.1109/REAL.1999.818824.

25 Eduardo Quiñones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using randomized caches in probabilistic real-time systems. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland*, pages 129–138. IEEE Computer Society, July 2009. doi:10.1109/ECRTS.2009.30.

26 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, 2008. URL: http://rw4.cs.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalysis.pdf.

27 Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. *ACM Transactions on Embedded Computing Systems*, 12(1s):42, 2013. doi:10.1145/2435227.2435238.

28 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

29 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis, WCET'06, Dresden, Germany*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, March 2006. doi:10.4230/OASIcs.WCET.2006.671.

30 Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS'13, Pittsburgh, PA, USA*, pages 279–292. ACM, June 2013. doi:10.1145/2465529.2465756.

31 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008. doi:10.1145/1347375.1347389.

32 Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Network and Parallel Computing, IFIP International Conference, NPC'10, Zhengzhou, China*, pages 144–154. Springer, 2010. doi:10.1007/978-3-642-15672-4_13.