Enabling Compositionality for Multicore Timing Analysis

Sebastian Hahn
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
sebastian.hahn@cs.uni-saarland.de

Michael Jacobs
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
jacobs@cs.uni-saarland.de

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

ABSTRACT

Timing compositionality is assumed by almost all multicore timing analyses. In this paper, we show that compositional timing analysis can be incorrect even for simple microarchitectures with in-order execution. We then introduce three approaches to enable sound compositional analysis: two based on analysis and one based on a hardware modification. In the experimental evaluation we explore the strengths and weaknesses of these three approaches. One of the two analysis-based approaches provides an attractive trade-off between analysis cost and precision, enabling sound compositional timing analysis even for microarchitectures with out-of-order execution.

1. INTRODUCTION

Hard real-time embedded systems are an important class of embedded systems in which a violation of timing constraints could result in catastrophic outcomes. In order to reduce size, weight, power, and cost, there is a trend to transition from federated architectures towards integrated architectures, where multiple critical functions are integrated on a single, shared multicore execution platform. As any violation of timing constraints could have catastrophic consequences, it is imperative to prove in advance that all timing constraints are guaranteed to be met; a process we refer to as *timing analysis*.

Timing analysis for single-core architectures has for long been addressed by a two-step approach. First, worst-case execution time (WCET) analysis determines the maximal execution time of each task when run in isolation. Then, schedulability analysis determines whether each task can be guaranteed to meet its deadlines, accounting for all possible *interference* the task might experience due to the execution of other tasks on the same processor.

Timing analysis for multicore architectures faces the challenge that tasks not only interfere on the processor cores, but also on other shared resources, such as shared buses, shared caches, and shared memories. It is an open question, how interference on resources other than the processor cores should best be accounted for. One can distinguish at least three approaches:

1. The *Murphy approach* [5] is to compute a bound on the execution time of a task accounting for all possible interferences on shared resources other than the processor within WCET analysis. This is only feasible if every resource access has bounded latency independently of interference. Where applicable, the Murphy approach can be extremely pessimistic, leading to drastic overestimations [31, 33].
2. In a *fully-integrated timing analysis* [16, 24, 22], tasks running on different cores are simultaneously analysed, precisely capturing all possible interleavings of resource accesses from different cores, as well as latency hiding due to pipelined execution. While this approach promises the highest possible precision, it appears to be practically infeasible for realistic systems due to the enormous number of system states.
3. *Compositional timing analysis* can be seen as a natural extension of the classical two-step approach: low-level analysis computes the "resource demand" of each task for *each* shared resource. In a system with a shared bus, low-level analysis would compute a bound on the task's number of bus accesses and on its execution time on the processor core assuming the task is run in isolation. Given such task characterisations, schedulability analysis [37, 7, 20] then determines, whether each task can be guaranteed to meet its deadlines, accounting for the *interference* it may experience on *each* of the shared resources.

We believe that *compositional timing analysis* is the most promising approach to multicore timing analysis: it avoids the pessimism of the *Murphy approach*, while not suffering from combinatorial explosion as a *fully-integrated timing analysis* would. It also maintains a fairly simple interface between low-level analysis and schedulability analysis, allowing different communities to focus on their particular skills.

However, for contemporary multicore architectures it is neither obvious whether compositional timing analysis is *sound* nor whether it is *precise*: it relies on the assumption of *timing compositionality.* Informally speaking the assumption is that timing contributions from different shared resources can simply be added up to obtain the response time of a task. We provide a more formal definition of timing compositionality in Section 2. In Section 3 we demonstrate that even simple in-order pipelined processors feature *amplifying timing anomalies* which render "naive" compositional timing analysis unsound. On the other hand, we also observe that due to pipelining, modern processors can sometimes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RTNS '16, October 19 - 21, 2016, Brest, France

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4787-7/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2997465.2997471

hide the latency of accesses to shared resources, rendering compositional timing analysis potentially *imprecise*.

We discuss three approaches to enable sound compositional timing analysis with respect to interference on shared resources, where shared buses will serve as an example, in Section 4. One approach is based on a small hardware modification that guarantees timing compositionality, while sacrificing average-case performance under interference. The other two approaches are analysis-based, accounting for the effect of anomalies either in the timing contribution of the core or in the timing contribution of the shared bus. Both approaches are applicable to existing microarchitectures, but they offer different trade-offs between analysis efficiency and precision.

The purpose of the experimental evaluation in Section 5 is to determine the cost of enabling *sound* compositional timing analysis in terms of analysis efficiency and precision. To this end, we use a *semi-integrated analysis* [21] to compute *interference response curves*, i.e., curves that capture how the execution time of a task responds to interference on shared resources. Comparing the three approaches with each other in terms of analysis cost and with the interference response curve in terms of analysis precision reveals their relative strengths and weaknesses.

To summarise, we make the following contributions:

- We show that even simple microarchitectures do not admit "naive" compositional timing analysis as it is found in most approaches in the literature.
- We introduce and evaluate three approaches to enable sound compositional timing analysis, two of which are analysis-based and applicable to existing microarchitectures without modification.

2. BACKGROUND

2.1 Timing Compositionality

Intuitively, timing compositionality enables separate analyses of different aspects of a system's timing behaviour and the subsequent combination of the respective individual analysis results to a sound overall timing guarantee.

There are two motivating factors to employ compositionality during timing analysis. First, the individual separate analyses can be *more efficient* than a single integrated analysis. Second, compared with the Murphy approach, individual analyses offer a richer interface to schedulability analysis compared to single WCET bounds that are independent of co-running tasks and their interference. Schedulability analysis can thus combine the more fine-grained results of the separate analyses to obtain *more precise* results. As an example, Altmeyer et al. [7] demonstrate how to use such a richer interface in a response-time analysis to account for interference on multiple shared resources.

In the following, we provide a formal definition of *timing compositionality* based on [17]. In this definition, a *system configuration* comprises everything that influences the timing behaviour of a system, i.e. the state of all active tasks, the values of registers and memory, and the state of the hardware including pipelines, caches, bus arbitration, and the memory controller. An *execution trace* is a sequence of system configurations formed by the cycle-wise execution of the processor. Thus the execution time of a given trace τ in processor cycles is given by its length $|\tau|$.

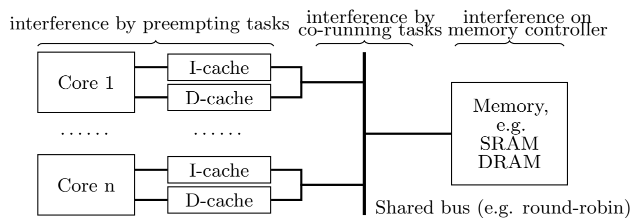


Figure 1: Schematics of a complex contemporary multicore architecture.

Different constituents of the system influence its timing behaviour, e.g. the bus arbiter blocking accesses to the shared bus or the DRAM controller scheduling refreshes. Each constituent's contribution to the system's overall timing is captured by a *contribution function*, which maps execution traces to the constituent's contribution. This contribution is not necessarily expressed by a number of processor cycles, but e.g. by the number of interfering bus accesses. Finally, a *combination* operator calculates an overall timing bound based on the contributions of all constituents:

DEFINITION 1. *Let $\mathcal{T}$ be the set of finite execution traces. A family of contribution functions $(tc_j : \mathcal{T} \to W_j)_{j=1..n}$ and a monotonic combination operator $\bigoplus : \prod_{j=1}^{n} W_j \to \mathbb{N}$ form a timing* decomposition *if*

$$\forall \tau \in \mathcal{T}.\ |\tau| \leq \bigoplus_{j=1}^{n} tc_j(\tau). \tag{1}$$

The following example decomposition matching Figure 1 is similarly chosen in the literature, e.g. in [40, 8, 7]:

- The "ideal" execution time of the tasks running on one of a multicore's cores, assuming the absence of *any* interference $tc_{ideal}(\tau) = i \in W_{ideal} = \mathbb{N}$,
- the number of additional cache misses due to preemptions $tc_{CRPD}(\tau) = cm \in W_{CRPD} = \mathbb{N}$,
- the number of accesses interfering on the shared bus $tc_{bus}(\tau) = ba \in W_{bus} = \mathbb{N}$, and
- the number of DRAM refreshes performed $tc_{DRAM}(\tau) = dr \in W_{DRAM} = \mathbb{N}$.

These contributions are then combined using the following combination operator:

$$\bigoplus(i, cm, ba, dr) := i + mp \cdot cm + bp \cdot ba + rp \cdot dr, \tag{2}$$

where mp, bp, and rp denote the additional timing penalty that one cache miss reload, interfering bus access, or DRAM refresh can contribute to the execution time. An intuitive approach—commonly employed in literature as in [7]—is to choose what we call the *direct effect* as the respective penalty as described in the following. The direct effect penalty for a cache miss due to preemption mp is the latency to load a cache line from the background memory. In work-conserving bus arbitration, each concurrent interfering access can block access to the shared bus at most for the length of the access, i.e. bp is chosen as the background memory latency. The penalty rp is chosen as the refresh latency provided by the DRAM chip datasheet. We will see in Section 3 that the above "naive" decomposition with direct effect penalties can be incorrect even for simple microarchitectures. In Section 4 we will then discuss three remedies to this problem.

In response-time analysis, each contribution tc_j of a decomposition can be approximated by combining the results of corresponding low-level analyses of each involved task. Such analyses focus on the timing contributions of the system constituents during the response time of a given task p. We denote the set of traces from the release of task p until the end of p's execution, including periods in which p is preempted by other tasks, by $\mathcal{T}_p \subseteq \mathcal{T}$. Thus, for the response-time of a given task p, we can derive:

$$\begin{aligned}
\max_{\tau \in \mathcal{T}_p} |\tau| &\leq \max_{\tau \in \mathcal{T}_p} \bigoplus_{j=1}^{n} tc_j(\tau) && \text{Def. compositionality} \\
&\leq \bigoplus_{j=1}^{n} \max_{\tau \in \mathcal{T}_p} tc_j(\tau) && \text{Monotonicity of } \bigoplus \\
&\leq \bigoplus_{j=1}^{n} \mathit{Analysis}_j(p) && \text{Analysis correctness}
\end{aligned}$$

In conventional single-core response-time analysis for fixed-priority scheduling, only $\mathit{Analysis}_{ideal}(p)$ is considered

$$\mathit{Analysis}_{ideal}(p) = R_p = C_p + \sum_{q \in hp(p)} \left\lceil \frac{R_p}{T_q} \right\rceil \cdot C_q, \quad (3)$$

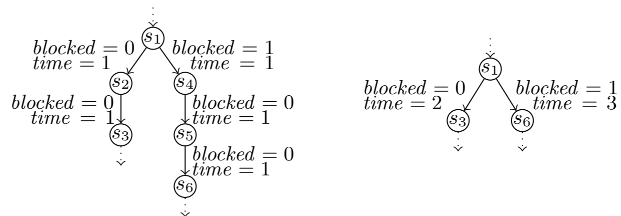
where C_p is a bound on task p's execution time in isolation obtained by a low-level analysis, and T_p is the period of p. In multi-core response-time analysis [7] each $\mathit{Analysis}_j(p)$ similarly aggregates the bounds on the contributions of each task computed by corresponding low-level analyses. In that case, the different analyses, $(\mathit{Analysis}_j(p))_{j=1,\dots,n}$, are coupled through the task's response time R_p.

2.2 Low-Level Timing Analysis

In this section, we describe a slight generalisation of today's de-facto standard approach to WCET analysis. This generalisation [21], which we call *semi-integrated analysis*, allows to analyse a program's execution time in terms of the amount of interference it experiences on shared resources. We will use this generalisation to evaluate the imprecision and possible unsoundness of compositional analysis in Sections 3 and 5. Further, it is the basis of one of the two approaches to enable compositionality by analysis described in Section 4.

We distinguish three analysis phases: 1. *value and control-flow analysis*, 2. *microarchitectural analysis*, and 3. *path analysis*. The first phase computes properties of the program that are independent of the underlying microarchitecture, in particular the possible values of registers and memory cells at different points in the program, as well as loop bounds and other constraints on the set of feasible program paths. This information is used in the two following analysis phases.

As the microarchitecture determines each instruction's timing, *microarchitectural analysis* characterises all possible execution traces of a given task. It is practically infeasible to enumerate them all due to their sheer number. Thus the analysis operates on *abstract microarchitectural states* [43], which each implicitly represent a large set of *concrete* microarchitectural states. In particular, abstract microarchitectural states completely abstract from the values of registers and memory cells, which are characterised in Phase 1. They *do* model the state of the pipeline, and of the caches, for which good abstractions are known. In a *fully-integrated* analysis, abstract states would also comprise the state of the co-running tasks and the exact state of the peripherals such as a DRAM controller. Despite the precision of such an approach, such analyses [16, 24, 22] cannot handle realistic hardware platforms without crippling assumptions—due to the size of the state space. In a *semi-integrated analysis* [21], which we describe here, microarchitectural states further abstract from the co-running cores and the exact state of the peripherals to coarse information about whether the shared bus is blocked or a DRAM refresh is ongoing.



(a) One edge per processor cycle. (b) Compressed chains. One edge represents multiple cycles.

Figure 2: Variants of microarchitectural execution graphs.

Microarchitectural analysis begins with the set of abstract states in which the program under analysis is about to enter the pipeline for execution. Then the analysis successively computes the abstract states that arise within one processor cycle—until the program under analysis has left the pipeline and finished execution. The abstraction introduces *non-determinism* upon analysing the effect of one processor cycle, e.g. whether a cache access is a hit, or a bus access is blocked by another access of a co-running task. Thus, an abstract microarchitectural state can have multiple possible successor states. If two abstract states are sufficiently "similar" they can be joined to a single abstract state to speed up the analysis.

Based on the set of reachable abstract states, microarchitectural analysis generates an abstract *microarchitectural execution graph* (aka prediction-file [42]). The nodes in this graph are abstract states and the edges describe the evolution during a processor cycle as shown in Figure 2a. For efficiency of the following *path analysis*, the resulting graph is compressed by replacing chains of edges by a single edge resulting in Figure 2b (see [42] for details). In the compressed graph, each edge e is annotated with *weights*, in particular the number of processor cycles time_e. To model the impact of interference such as shared-bus blocking, additional edge weights can be used to capture the number of interfering events occurring during the processor cycles modelled by an edge, e.g. the number of interfering bus accesses blocking an access to the shared bus $\mathit{blocked}_e$. For example, in state s_1 in Figure 2a, due to the abstraction of the state of co-running tasks, it is unknown whether bus access will be blocked or not, resulting in non-determinism, where one successor edge is labelled with $\mathit{blocked} = 0$ and the other with $\mathit{blocked} = 1$.

Finally, *path analysis* determines the longest execution trace through the microarchitectural execution graph. The standard approach, *implicit path enumeration* [27], is to encode the longest-path problem as an Integer Linear Program (ILP). For each edge e in the graph, a *frequency* variable x_e is used to model the number of times e is taken. To obtain an upper timing bound, the objective function then is

$$\max \sum_{\text{edge } e} \mathit{time}_e \cdot x_e. \quad (4)$$

The microarchitectural execution graph is encoded via flow conservation constraints for each vertex in the graph. Furthermore, cycles in the graph are bounded using loop bound

information. More details on such ILP formulations can be found in [42].

If an upper bound on the amount of interference I—e.g. on the shared bus—is known, an *interference constraint*

$$\sum_{\text{edge } e} blocked_e \cdot x_e \leq I \tag{5}$$

can be added to exclude paths that experience an infeasible amount of interference [21]. Similar constraints can be formulated to bound the impact of DRAM refreshes or the impact of cache misses due to preemptions, provided the graph is annotated with corresponding weights.

Throughout the paper, we refer to the analysis described above that models interference by non-determinism as a *semi-integrated* analysis. It is apparent that $Analysis_{ideal}$ from Section 2.1, which can assume the absence of interference is more efficient: less non-determinism needs to be considered in the microarchitectural analysis and the ILP path analysis formulation becomes simpler with fewer constraints. In particular, no interference constraint is required.

3. THREATS TO COMPOSITIONALITY

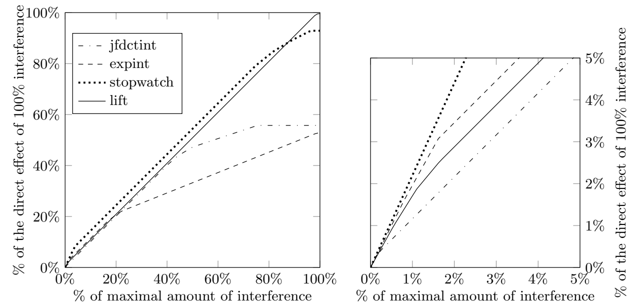
All approaches involved with timing compositionality which we encountered in the literature feature a composition operator that is *linear* w.r.t. interference—similar to the one described in Section 2.1, i.e., each unit of interference is assumed to contribute a fixed penalty of additional execution cycles. This system model assumption is *sound* if a unit of interference can never cause more additional execution time than given by the penalty, and it is *precise* if a unit of interference almost always increases execution time by the penalty.

To validate these two assumptions, we introduce *interference response curves*. The interference response curve of a task bounds its interference-induced additional execution time in terms of the amount of interference that the task experiences during its execution. We use a semi-integrated analysis as described in Section 2.2 to compute such bounds. Here, interference may be quantified by the number of additional cache misses due to preemptions, the number of DRAM refreshes, or the number of blocking accesses at the shared bus. In this paper, we focus w.l.o.g. only on interference caused by shared-bus blocking.

For a given program, the interference response curve is implicitly given by the integer linear program used in the path analysis as described in Section 2.2. A single point on the curve can be computed by choosing a corresponding constant amount of interference I in the ILP constraints.

Figure 3a shows the interference response curves for several programs w.r.t. shared-bus blocking. The underlying hardware configuration—a dual-core machine with round-robin event-driven arbitration—is described in detail in Section 5.2. In round-robin arbitration, a single access can be blocked by at most one concurrent access per co-running core. Thus, there is a maximal amount of interference that can affect the execution of a program. The y-axis shows the additional execution time due to interference, relative to the direct effect of this maximal interference. A curve corresponding to perfect compositionality would have a slope of one with a y-intercept of zero.

Figure 3b zooms in on the interval from 0% to 5% of the maximal amount of interference that affects a given benchmark. We make two observations:



(a) From 0 to max interference. (b) Zoom in on 0% to 5%.

Figure 3: Sampled interference response curves for four programs. 100 samples each. Configuration 4, see Section 5.2.

Observation 1. *For some programs, the additional execution time due to shared-bus interference exceeds the expected direct effect penalty. See Figure 3b.*

Observation 2. *For some programs, a significant portion of the expected additional latency is hidden. See Figure 3a.*

In general, it depends both on the core architecture and the particular shared resource whether a drop in slope of the curve can be observed: Using, e.g., fixed-priority bus arbitration, all interfering accesses can—in the worst case—block a single access, and so very little of their contributions can—in the worst case—be hidden. Under round-robin arbitration, on the other hand, the ability of the core to overlap bus blocking with other computations determines the drop in slope of the curve, as each individual access may only be blocked by a small number of interfering accesses.

The interference response curve can be multi-dimensional modelling the cumulative effects of multiple different sources of interference.

3.1 Amplifying Timing Anomalies

Timing anomalies [29, 34] are a well-known challenge for timing analysis. Due to the non-determinism in the microarchitectural analysis, an abstract state can have multiple successors. One of them is often considered to be the *local worst case* (e.g. shared bus blocked, or additional cache miss) with a bounded *direct effect* on the execution time as described in Section 2.1.

A non-deterministic choice where the local worst case does *not* imply the *global* worst case is known as a timing anomaly. Microarchitectural analysis thus needs to follow all possible successors to obtain a sound upper bound, which can be expensive. Such anomalies are not harmful w.r.t. compositionality and are thus not the focus of this paper.

However, Lundqvist and Stenström [29] found a second type of anomaly. While servicing the local worst case the abstract pipeline state can change in a way that leads to an *indirect effect*, i.e. an additional execution time increase beyond the direct effect penalty. We call a non-deterministic choice, an *amplifying anomaly*, if the overall execution time increases by *more* than the direct effect, i.e. there is a positive indirect effect.

Considering the combination operator presented in Section 2.1, it is apparent that this poses a challenge for timing compositionality. Ignoring indirect effects leads to *unsound* bounds, which is supported by our Observation 1.

The microarchitectural analysis of dynamically-scheduled (a.k.a. out-of-order) processors is known to be prone to amplifying timing anomalies. Lundqvist and Stenström [29]

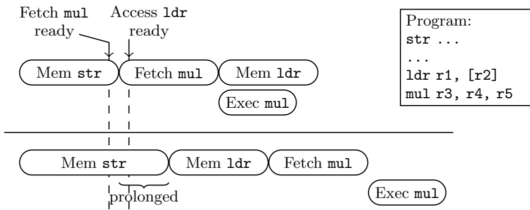


Figure 4: Amplifying timing anomaly upon uncertainty of length of store access **str**, e.g. due to shared-bus blocking.

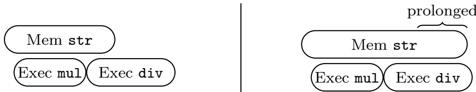


Figure 5: The execution of two subsequent arithmetic instructions hiding the latency of a preceding store.

provide examples of amplifying timing anomalies triggered by uncertainty about the cache behaviour and the varying latencies for cache hits and misses. Such anomalies can also be triggered by uncertainty about the memory latency due to shared-bus blocking.

As one of the contributions of this paper, we observe amplifying timing anomalies *even* in the analysis of *low-complexity* processors comparable to an ARM® Cortex®-M4 [3]. In the following, we provide an example, which we encountered during the microarchitectural analysis of the benchmark program `bsort100.c` taken from [15]. We found this anomaly during the analysis of the in-order pipeline with store buffer described in Section 5.2.

In Figure 4, we show the relevant program snippet and the amplifying anomaly triggered by the uncertainty of the duration of the store memory access **str**. The upper case depicts the situation that **str** finishes fast, while the lower case depicts the situation that the access **str** is prolonged, e.g. by bus blocking. As stores are buffered, **str** can complete and leave the pipeline so that the execution of subsequent instructions can advance in the pipeline. In both cases, the fetch of instruction `mul` becomes ready, but is blocked because the memory is busy with **str**. The data access of the load `ldr` becomes ready during the prolonged part of **str**. After access **str** finished, the fetch of `mul` is started in the upper case as it is the only ready access. In the lower case, there are two ready accesses: the fetch of `mul` and the data access of `ldr`. As in the aforementioned Cortex®-M4 [3], the data access is prioritised over the instruction fetch and thus `ldr` starts. In both cases, `mul` can only execute after it has been fetched. In the upper case, `mul`'s execution can be overlapped with the (independent) load. In the 5-stage in-order pipeline from Section 5.2, the multiplication is performed in the execute stage while the load is concurrently performed in the subsequent memory stage. This overlapping is not possible in the lower case as the load has been performed prior to the `mul` fetch. Finally, the incurred penalty on the execution time is larger than the prolongation of the access. The indirect effect in this example is determined by the execution latency of `mul`.

3.2 Hiding Latencies

In Figure 3a, we observe that there can be significant latency hiding effects. Thus, a timing bound that is derived compositionally may significantly overestimate the bound derived by a semi-integrated analysis.

The reason is that parts of memory accesses—thus also parts of shared-bus blocking—can *overlap* with computations within the processor pipeline. This effectively hides

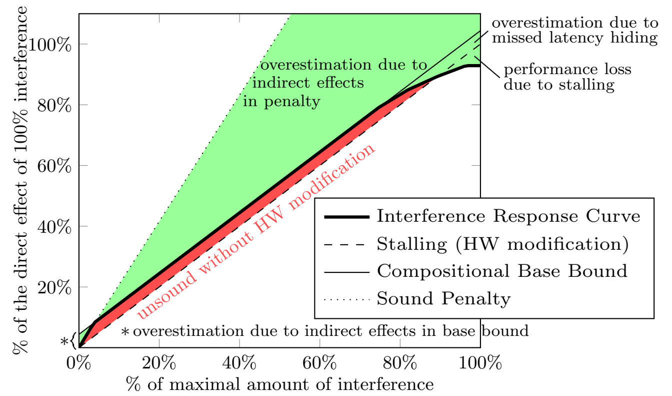


Figure 6: Overview of approaches to obtain a sound decomposition. Different sources of under-/over-estimation distinguished.

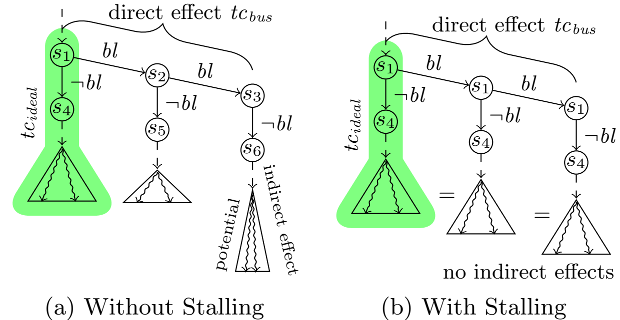


Figure 7: Example: snippet of a cycle-wise microarchitectural execution graph.

parts of the overall latency. The better a microarchitecture can hide latencies, the less precise compositionally derived bounds will be. As an example, consider Figure 5. A store operation is performed in the background memory, while subsequent arithmetic instructions execute concurrently due to pipelining and the presence of a store buffer. For the overall execution time, it is irrelevant whether the store is prolonged or not, as in both cases the memory latency is hidden by the execution of independent operations.

4. ENABLING COMPOSITIONALITY

In the following, we present three different solutions to establish a sound decomposition—ranging from microarchitectural design changes to new analysis techniques. Although we focus on interference by shared-bus blocking, all approaches can be applied for other sources of interference as well. The effects of the three approaches on analysis precision and soundness are summarised in Figure 6.

4.1 By Hardware Design: Pipeline Stalling

The first approach is to *eliminate* indirect effects by hardware design and thus allow the sound use of the decomposition introduced in Section 2.1 using direct effects as penalties. Consider the execution graph snippet in Figure 7a that shows the behaviour of an access that can be blocked (bl) at the bus. The ideal timing contribution tc_{ideal} covers the timing behaviour under the absence of bus blocking (s_4). Timing compositionality is violated if a trace starting from s_5 or s_6 is longer than every trace starting from s_4, i.e. there is an indirect effect caused by bus blocking which is not captured by tc_{ideal}.

A technique to eliminate indirect effects by hardware design is to *stall* the core while a memory access is blocked at the shared bus. If the core is stalled, the state of the core does not change as e.g. instructions cannot advance in the pipeline. In Figure 7b, this results in states s_1, s_2, and s_3—and consequently s_4, s_5, and s_6—being identical. Thus, they exhibit the same subsequent timing behaviour which is soundly covered by tc_{ideal}.

Besides the indirect effects, the stalling approach also eliminates any hiding of the additional latency due to interference. This can degrade average-case performance.

4.2 By Analysis: Sound Penalty

Adjusting the microarchitectural design is often not feasible due to practical or economical reasons. As a second approach, we turn the timing contributions and the combination operator described in Section 2.1 into a sound decomposition. In order to do so, we adjust the penalties used in the combination operator. The bus blocking penalty bp should not only account for the direct effect of one blocking access but also *include* all *indirect* effects possibly caused by this access. The calculation of sound penalties needs an in-depth analysis of the underlying microarchitecture. To the best of our knowledge, no techniques are available to calculate the maximal possible indirect effect for a realisitic microarchitecture. Rather, there are scenarios known as *domino effects* where indirect effects are *not* even bounded by a constant [36]. For such microarchitectures, the *sound penalty* approach cannot be applied.

In reality, indirect effects happen only *rarely* and under specific circumstances. If the indirect effects are incorporated into the penalty, they are *always* taken into account, i.e. for every interfering access. Considering the dotted line in Figure 6, it is apparent that such an approach will lead to a drastic overestimation of the actual timing.

In Section 5, we experimentally determine—per program—the respective minimum penalty required to over-approximate the interference response curve. Sound program-independent penalties would have to be at least as high. For a given program p, we can use an integer linear program—similar to the formulation we present in Section 4.3—to check whether a conjectured penalty is sound. Using this check in a binary search, we can calculate a penalty that is sound for p. Due to space constraints, details on this calculation can be found in our extended technical report [18].

4.3 By Analysis: Compositional Base Bound

Now, we present a sound and reasonably precise approach to obtain timing contributions that can be used in a compositional way during schedulability analysis—without modifying the underlying hardware and even in the presence of domino effects.

The idea is to account for the rare indirect effects in the "ideal" execution time contribution. To this end, we adjust the decomposition from Section 2.1 with the focus on shared-bus blocking as follows:

- The *base* execution time ignoring the direct effects of shared-bus blocking but taking into account *all* possible *indirect effects* $tc_{base}(\tau) = b \in W_{base} = \mathbb{N}$,

- the number of accesses interfering on the shared bus $tc_{bus}(\tau) = ba \in W_{bus} = \mathbb{N}$.

For this decomposition the combination operator is defined as:

$$\bigoplus(b, ba) = b + bp \cdot ba, \tag{6}$$

where bp denotes the *direct effect* timing penalty.

How can $tc_{base}(\tau) = b$ be soundly approximated? In Section 2 we describe a path analysis via Implicit Path Enumeration that uses an ILP to obtain the longest path through the microarchitectural execution graph, satisfying constraints such as the interference constraint. As the ILP is an implicit encoding of the interference response curve, we can modify it to compute a sound approximation of b, which we term a *compositional base bound*. In addition to the ILP variables x_e that model the execution count of edges in the execution graph, we introduce an integer variable i to model the amount of interference. The interference constraint for shared-bus blocking then becomes

$$\sum_{\text{edge } e} \mathit{blocked}_e \cdot x_e \le i. \tag{7}$$

We want to find a value b such that the linear approximation $\bigoplus(b, i) = b + bp \cdot i$ is greater than or equal to the interference response curve for all possible values of i. We obtain the smallest such b, i.e. a sound base bound, by maximising

$$\max\left(\sum_{\text{edge } e} \mathit{time}_e \cdot x_e\right) - bp \cdot i. \tag{8}$$

Intuitively, the first term captures the entire execution time of the program, including interference on the shared resource (under the interference constraint in Equation 7). The second term, $bp \cdot i$, captures the share of execution time that is explained by the direct effects of interference for any amount of interference i, particularly also for a concrete ba. So the difference between the two captures the share of execution time not explained by the direct effects of interference. This includes uninterfered execution but also indirect interference effects. By maximizing over all possible amounts of interference i, the solution to this ILP provides an upper bound on the execution time that is not explained by direct interference effects, i.e., the compositional base bound. Thus the compositional base bound includes the maximum possible indirect effects for the program under analysis. A formal proof of correctness of this approach is given in [18].

This approach can be generalised to multiple dimensions, i.e. to multiple sources of interference such as shared-bus blocking *and* DRAM refreshes *and* cache misses due to preemption. In this case, we introduce integer variables $i_1, \ldots, i_n$ and corresponding interference constraints for each source of interference. The objective function then becomes

$$\max\left(\sum_{\text{edge } e} \mathit{time}_e \cdot x_e\right) - \sum_{k=1}^{n} p_k \cdot i_k, \tag{9}$$

where p_k is the direct effect penalty for interference of type k.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the implications of the proposed approaches on analysis efficiency and precision. A rough qualitative comparison is given in Table 1: If we employ stalling, we efficiently obtain precise bounds by construction, however the actual average-case system performance likely is significantly degraded. Note that a thorough evaluation of actual performance is out of the scope of this

Table 1: Overview of the approaches with their strengths and weaknesses.

Proposed Approach	Analysis Efficiency	Analysis Precision	Average-case Performance
Stalling	✓	✓	×
Sound penalty	✓	×	✓
Compositional base bound	◦	◦	✓
Semi-integrated analysis	×	✓	✓

paper. Due to modelling interference by non-determinism in the microarchitectural analysis, the *compositional base bound* approach is expected to be less efficient than the two other approaches based on stalling or sound penalties. However, it should result in more precise bounds than the approach based on sound penalties, while still sacrificing some precision compared with a *semi-integrated* analysis. In contrast to the other approaches, the *semi-integrated* analysis does not offer a compositional interface. To our knowledge, there is currently no schedulability analysis that can handle such a non-compositional interface. In any case, such a schedulability analysis needs additional runs of the path analysis whenever a different amount of interference is to be considered—potentially inside a fixed-point iteration. The following experimental evaluation sheds light on how significant the differences are quantitatively.

5.1 Experimental Setup

We employ our timing analysis framework `LLVMTA`. The framework allows to run context-sensitive analyses on the binary-level program representation of the LLVM compiler infrastructure. It provides state-of-the-art techniques for microarchitectural analysis [43] and path analysis [42].

Table 3 provides an overview of the benchmark programs used for evaluation. Besides the Mälardalen benchmark suite (M) [15], we use programs generated from models developed in the SCADE suite including the examples delivered with SCADE (S), own models (O), and a model provided by an industrial partner (I).

5.2 Hardware Configurations

We evaluate the proposed approaches on hardware configurations of differing complexity. The processors under analysis feature either two, four, or eight cores. We consider two different types of cores: the simpler type has a five-stage in-order pipeline, while the more complex one has an out-of-order pipeline with Tomasulo dynamic scheduling, both as described in detail in [19]. The out-of-order pipeline has a reorder buffer of size 8 and a 4-entry instruction queue which issues multiple instructions to either the Load-/Store-Unit or one of two arithmetic functional units executing instructions with variable latency. Both pipelines employ static branch prediction, where conditional branches are predicted not taken. The out-of-order pipeline additionally features speculative execution. Optionally, store instructions are passed through a store buffer (a.k.a. write buffer) of size one that allows the pipeline to complete the store instruction while the memory operation is performed in the background.

The cores access the shared bus via private instruction and data caches of size 1 KiB with associativity 2 and LRU replacement policy. As the benchmarks are small, we consequently choose such a small cache size. Upon concurrent instruction and data cache misses, the resulting data access is prioritised over the instruction access. The shared bus employs round-robin event-driven arbitration and is directly

Table 2: Overview of evaluated core configurations.

Configuration	Core Type	1-entry Store Buffer	Mem. Latency
1	in-order	no	5
2	in-order	yes	5
3	in-order	yes	10
4	out-of-order	yes	5
5	out-of-order	yes	10

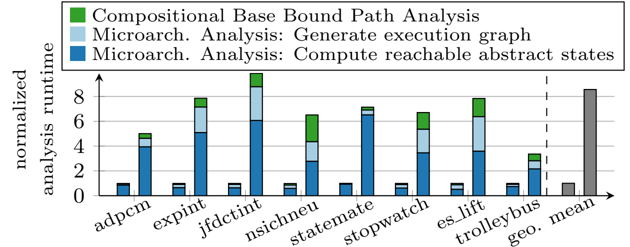


Figure 8: Efficiency: analysis modelling interference by non-determinism versus $Analysis_{ideal}$ assuming the absence of interference. Selected benchmarks and geometric mean over all benchmark programs. Configuration 4, quad core.

connected to an SRAM background memory with a latency of either 5 or 10 processor cycles.

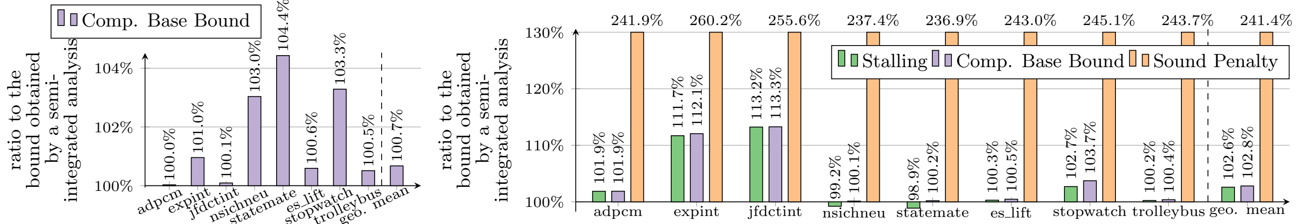
For the evaluation, we choose the five core configurations listed in Table 2 out of the eight possible combinations. Pipelines without store buffers waste the potential of hiding parts of the store latency and are rarely found in real-world processors. Hence, we consider only one configuration without a store buffer.

For the sake of brevity, we focus on the results for selected representative benchmarks and a quad-core processor with core Configuration 4 in Table 2 if not stated otherwise. Configuration 4 featuring out-of-order execution, a store buffer, and a memory latency of 5 exhibits the most potential for overlapping effects: We prefer Configuration 4 over 5 as a larger share of the shorter memory latency can be hidden by overlapping the execution of other instructions. Condensed evaluation results including analysis efficiency—i.e. memory consumption and analysis runtime—and analysis precision for *all* hardware configurations as well as fine-grained results for *each* individual benchmark can be found in our extended technical report [18].

Note, that while we do not precisely model any particular commercial processor, the microarchitectural concepts used in the different configurations can be found in real processors. As an example, the ARM® Cortex®-M4 processor features a three-stage in-order pipeline with an optional pipelined floating-point unit and a store buffer of size one. Data accesses are prioritised over instruction accesses [4, 3]. The ARM® Cortex®-A5 features an in-order pipeline with dynamic branch prediction and a 4-entry store buffer [1]. The ARM® Cortex®-A9 features a complex out-of-order pipeline with dynamic branch prediction and a 4-entry store buffer [2].

5.3 Analysis Efficiency

The *stalling* and *sound penalty* approaches both allow to employ an efficient microarchitectural analysis that can assume the absence of interference. The *compositional base bound* approach as well as the *semi-integrated analysis*, on the other hand, model the interference as non-determinism during microarchitectural analysis. This results in larger execution graphs and more expensive path analysis, as it has to take additional constraints into account.



(a) With no interference.

(b) With maximum interference. For the sound penalty approach, we choose a penalty of 15 cycles per interfering memory access according to Table 3 (forth column).

Figure 9: Ratio of the approaches to the semi-integrated timing bound for a certain amount of interference. Configuration 4, quad-core processor.

We examine the cost of modelling interference via nondeterminism in Figure 8. The first bar represents the analysis assuming the absence of interference, the second bar represents the *compositional base bound analysis*. The numbers are normalised w.r.t. the first bar. The bars are divided into the subphases of the analysis: computing the reachable abstract microarchitectural states, computing the execution graph, and finally the path analysis. The preprocessing phase including value and loop bound analysis took less than 1% of the analysis time and is thus omitted. The analysis runtime increases by a factor of 3.4 to 16.5 with a geometric mean of 8.6 across all benchmarks. The analysis memory consumption increases by a factor of 2.1 to 7.5 with a geometric mean of 4.8 across all benchmarks. The complete evaluation run of the *compositional base bound* approach for all benchmarks and four cores with Configuration 4 took less than 7 hours and 30 minutes and 9.5 GiB memory on an Intel® Core™ i5 machine clocked at 3.3 GHz. Detailed results can be found in the extended technical report [18].

In the context of low-level WCET analysis, it is common for analyses to run for hours for a single benchmark targeting a complex hardware platform. Thus, our evaluation demonstrates that the compositional base bound approach—although more expensive than the other approaches—has practically acceptable analysis cost.

5.4 Analysis Precision

No technique is known to derive sound, general penalties for interfering events on realistic microarchitectures. However, we can compute sound penalties for each particular benchmark as sketched in Section 4.2. The results for the five hardware configurations and all benchmarks are shown in Table 3. The maximally observed penalty was *three* times the memory latency for an out-of-order core with a store buffer. As expected, the impact of indirect effects increases with increasing hardware complexity, and maybe surprisingly with shorter memory latencies. With longer latencies, the pipeline is more likely to *converge* while performing a single memory access, i.e. each stage of the pipeline waits for the memory access to complete. In the converged case, additional interference does not change the pipeline state and cannot cause indirect effects. Thus, shorter memory latencies leave more potential for indirect effects due to additional interference. Except for the simplest configuration, taking the direct effect as penalty is incorrect for many benchmarks.

We evaluate the precision of the proposed approaches against the semi-integrated analysis of the respective benchmark programs at the extremal points, i.e. with no and with maximal interference. The maximal imprecision relative to semi-integrated analysis usually occurs at one of those two points. In Figure 9a, we provide the ratio of the compositional base bound to the timing bound without interference. The ratios reflect the maximal impact of potential positive indirect effects on the overall execution time. The other two approaches to enable compositionality are by construction precise in the case where no interference occurs.

In Figure 9b, we provide the ratio of the timing bounds obtained with each approach to the timing bound obtained using the semi-integrated analysis, under maximal interference. For the sound penalty approach, we use the maximum program-dependent penalty from Table 3. The first observation is that the sound penalty approach is extremely imprecise at maximal interference. The results also show that even at maximal interference the naive decomposition using direct effects as penalty (Section 2.1) is generally unsound (see benchmarks STATEMATE and NSICHNEU) if the hardware is not modified accordingly. If the core employs the stalling mechanism, the presented results are sound and show that stalling can improve worst-case bounds in rare cases by eliminating indirect effects. In general, however, average- and even worst-case system performance is degraded by stalling. The compositional base bound approach is sound without hardware modifications at acceptable precision: At maximal interference, we observe an average overestimation of around 3% relative to the semi-integrated approach. There are, however, exceptions (see benchmarks EXPINT and JFDCTINT) where the overestimation reaches up to 13%.

Table 3: Penalty including indirect effects: Minimum penalty per interfering memory access for each benchmark to obtain a sound overapproximation. Hardware configurations (1,2,3,4,5) as in Table 2.

Benchmark	Penalty in cycles	Benchmark	Penalty in cycles
M **adpcm**	(5, 10, 19, 10, 17)	M **ndes**	(5, 10, 10, 9, 10)
M **bs**	(5, 5, 10, 5, 10)	M **ns**	(5, 8, 10, 9, 28)
M **bsort100**	(5, 6, 14, 7, 10)	M **nsichneu**	(5, 10, 14, 9, 10)
M **cnt**	(5, 10, 20, 8, 10)	M **prime**	(5, 5, 10, 10, 10)
M **compress**	(5, 11, 14, 9, 10)	M **qsort**	(5, 10, 20, 10, 14)
M **crc**	(5, 9, 14, 11, 10)	M **qurt**	(5, 10, 14, 8, 10)
M **edn**	(5, 10, 17, 11, 11)	M **select**	(5, 5, 20, 10, 10)
M **expint**	(5, 9, 20, 10, 10)	M **sqrt**	(5, 7, 10, 7, 12)
M **fdct**	(5, 9, 10, 10, 10)	M **statemate**	(5, 10, 17, 10, 10)
M **fft1**	(5, 9, 10, 10, 10)	M **st**	(5, 10, 17, 9, 10)
M **fibcall**	(5, 5, 10, 5, 10)	M **ud**	(5, 10, 12, 10, 10)
M **fir**	(5, 6, 10, 9, 10)	M **whet**	(5, 10, 10, 9, 10)
M **insertsort**	(5, 5, 10, 13, 10)	S **cruisectrl**	(5, 9, 16, 9, 10)
M **janne**	(5, 6, 10, 9, 10)	S **stopwatch**	(5, 10, 14, 11, 10)
M **jfdctint**	(5, 9, 17, 10, 10)	O **es_lift**	(5, 9, 11, 9, 10)
M **lms**	(5, 10, 10, 12, 14)	S **flight_ctrl**	(5, 10, 15, 13, 10)
M **ludcmp**	(5, 6, 10, 9, 10)	S **pilot**	(5, 10, 16, 13, 25)
M **matmult**	(5, 10, 20, 9, 13)	O **roboDog**	(5, 8, 11, 15, 10)
M **minver**	(5, 10, 20, 10, 16)	I **trolleybus**	(5, 10, 12, 11, 10)

6. RELATED WORK

Lundqvist and Stenström [29] first observed the existence of timing anomalies in dynamically-scheduled microprocessors. As a particular type of anomalies, they described what we call *amplifying timing anomalies*, and demonstrated their presence in processors with out-of-order execution. In this paper, we showed that even processors with *in-order* execution often feature at least one *out-of-order* resource, namely the memory bus. We also observe that in the presence of amplifying timing anomalies, compositional timing analyses that only account for the *direct effect* of an event can be unsound.

Wilhelm et al. [44] distinguish three classes of microarchitectures: *fully timing-compositional architectures*, which do not feature timing anomalies; *compositional architectures with constant bounded effects*, which may exhibit timing anomalies but no *domino effects* [29]; and *non-compositional architectures*, which may exhibit timing anomalies and domino effects. The *sound penalty* approach from this paper is only applicable to the first two classes of microarchitectures, which do not feature domino effects. The other two enablers for compositionality are applicable to all microarchitectures.

De Dinechin et al. [13] describe the KALRAY MPPA®-256, a many-core processor suitable for time-critical computing. Its cores are claimed to be *fully timing-compositional* [44]. We note that due to the presence of a store buffer the cores potentially feature *amplifying timing anomalies* as discussed in Section 3. Schoeberl et al. [38] propose the T-CREST platform, a multicore architecture optimised for the worst-case rather than the average-case. It is conceivable but—to our knowledge—not proven to be fully timing-compositional.

Various methods have been developed to bound the cache-related preemption delay (CRPD) [10, 26, 41, 6], i.e., the cost of additional cache misses induced by preemptive scheduling. All of these methods assume timing compositionality. Our methods to enable compositional analysis w.r.t. bus blocking can analogously be applied w.r.t. additional cache misses.

Recently, a multitude of methods have been proposed for timing analysis of tasks on multicores under interference on shared resources, in particular shared buses with TDMA arbitration [39, 30, 7], dynamic arbitration [32, 37, 12], or a combination of the two [30, 40, 14, 25, 7]. These methods differ in terms of the considered task models, the supported types of shared resources, and concerning analysis precision and efficiency. However, they all rely on timing compositionality, and may be applied to a much broader class of microarchitectures using the approaches to enable compositionality introduced in this paper. The same applies to earlier work accounting for DRAM refreshes [8].

Fully-integrated analyses promise the highest precision and do not require timing compositionality. Kelter and Marwedel [24] describe a microarchitectural analysis that explores all possible interleavings of a parallel, periodic task. Even when tasks on different cores are perfectly synchronised they observe an average slowdown by a factor of 130 compared with [11]. Similarly, Gustavsson et al. [16] employ a model checker to exhaustively explore all possible executions of a parallel program on a simple multicore architecture. However, scaling such approaches to realistic architectures and program sizes appears difficult.

Semi-integrated analysis offers a compromise between compositional and fully-integrated analysis in terms of analysis time and precision and it does not per se require timing compositionality. The general idea is to analyse the execution of a task on a given core under an *abstraction* of the interference on shared resources generated on other cores. Kelter et al. [23] show to precisely account for TDMA arbitration within WCET analysis. This approach was later [11] extended to take into account shared caches, by accounting for the maximal interference of other tasks upon every memory access. This line of work is not immediately applicable to dynamic arbitration policies, which have been shown to provide better worst-case performance [7]. Jacobs et al. [21] present a semi-integrated analysis accounting for interference on shared buses *with* dynamic arbitration. Our proposed compositional base bound approach employs techniques presented in [21]. An open question for future work is how to directly employ "non-murphy" semi-integrated analysis within a multi-core response-time analysis.

An alternative to analysing interference on shared resources is to eliminate such interference. Bui et al. [9] discuss methods to achieve temporal isolation in multicores, including TDMA arbitration of buses. The PTARM [28] provides the illusion of private resources to multiple hardware threads by partitioning resources in space and time, including background memory [35]. We believe that a combination of isolation and interference analysis will be required to extract good worst-case performance out of future multicore architectures.

7. SUMMARY AND CONCLUSIONS

Naive compositional analysis as found in most approaches in the literature is *unsound* even for low-complexity hardware platforms. We demonstrated this by an example of an *amplifying timing anomaly* on a microarchitecture similar to commercial processors such as the ARM® Cortex®-M4.

To bridge the gap between a large body of academic approaches that is assuming timing compositionality and non-compositional commercial hardware platforms, we presented three approaches to enable compositionality. The first approach, *stalling*, requires a hardware modification that is detrimental to average-case performance. The second approach incorporates the indirect effects into a *sound penalty*, leading to high overestimation if the interference is high. The third and most-promising approach incorporates the indirect effects in a *compositional base bound* leading to relatively precise results at acceptable analysis cost. While the second approach is limited to microarchitectures without domino effects, the third approach can be applied to *any* microarchitecture without modification.

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the PEP Project and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

8. REFERENCES

[1] ARM®Cortex®- A5 Processor Technical Reference Manual. page 21 and 136.
[2] ARM®Cortex®- A9 Processor Technical Reference Manual. page 17 and 113.
[3] ARM®Cortex®- M4 Processor Technical Reference Manual. page 22 and 43.
[4] Cortex™-M4 Devices Generic User Guide. page 11.
[5] A. Abel et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, pages 25–43, 2013.

[6] S. Altmeyer, R. I. Davis, and C. 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.

[7] S. Altmeyer et al. A generic and compositional framework for multicore response time analysis. In *RTNS*, pages 129–138, 2015.

[8] P. Atanassov and P. Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *IEEE International Workshop on Application of Reliable Computing and Communication*, 2001.

[9] D. N. Bui et al. Temporal isolation on multiprocessing architectures. In *DAC*, pages 274–279, 2011.

[10] J. V. Busquets-Mataix et al. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTAS*, pages 204–212, June 1996.

[11] S. Chattopadhyay et al. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4s):124:1–124:29, Apr. 2014.

[12] D. Dasari et al. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *TrustCom*, pages 1068–1075, 2011.

[13] B. D. de Dinechin et al. Time-critical computing on a single-chip massively parallel processor. In *DATE*, pages 1–6, 2014.

[14] G. Giannopoulou et al. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *EMSOFT*, pages 63–72. ACM, 2012.

[15] J. Gustafsson et al. The Mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.

[16] A. Gustavsson et al. Towards WCET analysis of multicore architectures using UPPAAL. In *WCET*, pages 101–112, Dagstuhl, Germany, 2010.

[17] S. Hahn et al. Towards compositionality in execution time analysis – definition and challenges. In *Intl. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, December 2013.

[18] S. Hahn et al. Enabling compositionality for multicore timing analysis. Technical report, Saarland University, 2016. Available: http://embedded.cs.uni-saarland.de/EnablingCompositionalityTR.pdf.

[19] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[20] W.-H. Huang et al. MIRROR: Symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *DAC*, June 2016.

[21] M. Jacobs et al. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *RTNS*, 2015.

[22] T. Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, TU Dortmund University, 2015.

[23] T. Kelter et al. Bus-aware multicore WCET analysis through TDMA offset bounds. In *ECRTS*, 2011.

[24] T. Kelter and P. Marwedel. Parallelism analysis: Precise WCET values for complex multi-core systems. In *Formal Techniques for Safety-Critical Systems - Third International Workshop*, pages 142–158, 2014.

[25] K. Lampka et al. A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5):736–773, 2014.

[26] C.-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[27] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[28] I. Liu et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*, pages 87–93, 2012.

[29] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

[30] M. Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, pages 339–349, 2010.

[31] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, July 2014.

[32] R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *DATE*, pages 741–746, March 2010.

[33] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34, 2012.

[34] J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

[35] J. Reineke et al. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011.

[36] J. Reineke and R. Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *WCET*, 2009.

[37] S. Schliecker and R. Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, January 2011.

[38] M. Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449 – 471, 2015.

[39] A. Schranzhofer et al. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, pages 215–224, April 2010.

[40] A. Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, pages 213–222, April 2011.

[41] J. Staschulat et al. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, 2005.

[42] I. J. Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Saarland University, 2010.

[43] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland Univ., 2004.

[44] R. Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 28(7):966–978, 2009.