Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master Thesis

Defining Compositionality in Execution Time Analysis

submitted by
Sebastian Hahn

submitted
November 2014

Supervisors
Prof. Dr. Jan Reineke
Prof. Dr. Reinhard Wilhelm

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Timely operation of hard real-time systems is a correctness criterion. Guarantees, i.e. upper bounds on the execution times of a program running on a specific microarchitecture, are obtained by static timing analysis. To prevent over-provisioning of system resources, the computed bounds should be as tight as possible. For current processors, precise timing analysis is difficult mainly due to subtle interferences between processor features that affect the overall timing behaviour. To still obtain precise results, state-of-the-art approaches perform a detailed but computationally expensive exploration of possibly arising system states – thereby capturing the effects of these interferences.

Future systems pose new challenges. As an example, multi-core processors introduce interference on shared resources between (functionally independent) programs running on different cores. This will eventually render the above approach infeasible in terms of analysis runtime and memory consumption. As a consequence, recent approaches to timing analysis assume so-called *timing compositionality* of the underlying system. This allows for a decoupling into several, less complex, individual analyses.

In this thesis, we present and discuss an updated version of our definition of timing compositionality [18]. How to achieve timing compositionality *in general* is an unsolved question. We are interested in the construction of compositional analyses even for complex processors, as well as hardware designs that allow for precise and compositional analyses by construction. We present ideas to approach the challenges that arise in the context of compositional timing analysis.

Remarks on previous publications

The first part of this thesis has already been published in [18] with a preliminary version of the compositionality definition. The publication includes Section 1, Section 2, a previous version of Section 3, and parts of Section 7 of this thesis. The remaining sections present novel and unpublished material.

Acknowledgments

First of all, thanks to my supervisors Jan Reineke and Reinhard Wilhelm providing an excellent working and research environment - including many valuable discussions on ideas and details within the scope of compositionality.

Further thanks go to

(a) Michael Jacobs for interesting discussions and conversations - not limited to compositionality;

(b) Florian Haupenthal, Barbara Dörr, Christoph Mallon, and Jörg Herter for $\subseteq$ { regular tea meetings, lively conversations, reading and correcting parts of the thesis };

(c) the people of the compiler design/real-time and embedded systems lab;

(d) the anonymous reviewers of [18] for their comments.

This work was supported by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) 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.

Last but not least, I would like to express my special gratitude to my entire family for their love and support throughout my life.

Contents

1 Introduction

In general-purpose computing, the *fast* execution of programs in *most cases* is a *desirable* property. For safety-critical, hard real-time (embedded) systems, *timely* program execution in *all cases* is strictly *required* [36]. In such a hard real-time setting, static analysis methods are therefore employed to derive *guarantees* for the timing behaviour of a program – prior to the deployment of the system.

Analysis results at high precision are required to prove the system's timeliness without over-provisioning its resources. The timing of a program depends not only on its inputs such as sensor values, but also on the state of the underlying hardware platform, e.g. cache contents. A memory access during program execution, e.g., can be served within a few cycles if the requested memory block is cached, while taking hundreds of cycles otherwise. In order to obtain precise results, this *microarchitectural influence* on the execution time has to be considered during analysis.

Modern microprocessors have several performance-enhancing features such as complex pipelining, caching, branch prediction, and speculation. Each of these features enlarges the microarchitectural state that has to be considered during analysis to obtain tight timing bounds. Additionally most of these processor features are also highly interdependent [20] – often enough in a subtle way. These interdependencies cause *interferences* between processor features during program execution – e.g. a speculative memory access influences the cache contents and thereby the cache's timing behaviour. Timing analysis for such microprocessors has become a complex task due to these interferences.

To obtain tight timing bounds, state-of-the-art approaches employ a highly-integrated, non-compositional analysis that simultaneously keeps track of all the interferences caused by interdependencies. They explore the space of whole microarchitectural states that can evolve during program execution and search for the longest path. Such approaches allow to precisely capture the detailed execution behaviour of a program – at the cost of significant analysis effort. To allow for a more compact representation of microarchitectural states, abstractions are employed. Efficient and sufficiently precise abstractions have been proposed for some isolated features, e.g. caches [4], while abstractions for other components and their complex interplay are still to be found. The integrated approach using abstractions, as implemented in the industry-strength tool aiT [13] by AbsInt GmbH, is successfully applied to programs that execute uninterruptedly and in isolation – even on complex processors [33]. Despite the employed abstractions, the state space exploration is still very expensive in terms of analysis time and memory consumption. Thus, any change to the analysis setting or any additional processor feature might render this approach computationally infeasible.

The need for compositionality by two examples In modern and future embedded systems, tasks are scheduled preemptively as this increases the overall schedulability compared to non-preemptive scheduling. This introduces additional interferences, as the preempting task might evict useful cache contents that has to be reloaded by the preempted task.

Multi-core platforms are emerging also in the embedded domain as they offer a better performance-energy ratio and reduce the total weight compared to multiple single-core computers. As a consequence, several programs are grouped together to execute concurrently on different cores sharing common resources such as buses and memory. Thus, due to resource sharing, interferences with an impact on the timing behaviour between functionally independent programs are introduced [1]. Keeping track of this increasing amount of *interferences* in an *integrated* analysis will lead to state space explosion and will finally render the above approach *infeasible*.

For that reason, complexity, there is a need for a *compositional* view of (analyses of) the timing behaviour of a system – moving away from the integrated, non-compositional view. Recently, efficient and precise analyses have been proposed that focus on the (timing) behaviour of selected features – not of the whole system at once. Examples are the analysis of shared buses in a multi-core system [31, 32] as well as analyses for preemptively scheduled systems [5]. The inherent, underlying assumption is that the system allows for such a decoupling of analyses. This assumption is referred to as *timing compositionality*.

Our Contributions and Overview

Up to now, timing compositionality is a term whose meaning is solely based on intuition without a rigorous, formal definition. We examine existing approaches that assume "timing compositionality" with respect to their intuitive understanding of compositionality in Section 2. Based on our findings, we present a unified formal definition of timing compositionality (Section 3). A preliminary version of the definition has been given in [18]. Following our definition, we want to soundly replace the analysis of a whole system by a combination of independent analyses that focus only on individual features. We discuss *timing compositional architectures* as introduced by [37] and highlight the differences to our definition of compositionality in Section 3.4.

In the second part – beginning with Section 4 – we focus on the exploitation of compositionality in execution-time analysis. We describe the integrated, state-of-the-art approach as well as the compositional approach to timing analysis and we discuss their respective advantages and shortcomings. We argue that compositional analyses are possible even for complex processors. Coming at the price of overestimation, compositionality potentially enables more efficient analyses.

In addition, compositionality might even improve precision by enabling more powerful analyses focusing on individual components, that were prohibitive in an integrated analysis setting due to their complexity. Concrete example applications for compositionality are presented in Section 4.6.

How to achieve timing compositionality in general is an unsolved question. In Section 4.7, we identify challenges for the design of compositional *analyses*. Section 5 provides ideas towards a compositional analysis framework, including the presentation of a generic, compositional analysis based on counting so-called timing accidents.

Third, we discuss the microarchitectural influence on compositionality in more detail from the point of view of *hardware design*. In Section 6, we present a microarchitectural design guideline that, given a decoupling (e.g. into cache and pipeline behaviour), enables precise and compositional analyses by construction. In return, such designs might degrade performance as they restrict the way latencies can be hidden.

Related work is described in Section 7. We conclude and summarise challenges and open problems that are subject to future work in Section 8.

2 Timing Compositionality by Examples

Recently, approaches have been proposed that make use of a compositional rather than an integrated view on the timing behaviour of systems. This enables focusing on the analysis of selected features of a system in isolation while maintaining overall soundness – given timing compositionality and soundness of the individual analyses. We give several examples where timing compositionality is assumed or required.

2.1 Resource-Sharing Systems

Schranzhofer et al. [31, 32] are concerned with the analysis of the interference on a shared bus in a resource-sharing system. As an example, consider a multi-core system with a shared memory that is accessed via a shared bus as depicted in Figure 1. A task executes on one core and can access the shared memory through the shared bus. Each resource/bus access might be blocked until access is granted by the arbiter (e.g. TDMA arbitration [31] or adaptive arbitration [32]).

Timing compositionality enables the decoupled analysis of the timing contributions of selected features and allows to combine the individual results to a globally safe result. First, an upper bound on the execution times of the task under consideration $exec^{max}$ (excluding resource accesses) as well as an upper bound on the number of resource accesses μ^{max} are computed. With these bounds in mind, the authors search for the worst distribution of accesses and execution time to maximise the overall blocking time B that the considered task can incur. In case of TDMA arbitration, the calculation of the overall blocking time B depends on the static bus schedule and the initial offset of the task's start time w.r.t. the bus schedule. For other arbitration policies such as Round-Robin, the calculation of B additionally depends on $exec^{max}$ and μ^{max} of the tasks running on the other cores. In the given scenario, a globally safe bound is computed by

$$exec^{max} + \mu^{max} \cdot a + B,$$

where the constant a bounds the access time to the resource once access is granted.

The authors explicitly assume a *fully timing compositional architecture* in the sense of [37], i.e. an architecture without timing anomalies (Section 3.4). In general, the absence of timing anomalies allows to prune parts of the abstract state space that an analysis has to consider and thus affects the efficiency of analyses. The presence of timing anomalies, however, does not generally preclude timing compositionality in the sense of decoupled analyses, e.g. as they are described in the previous paragraph. In contrast to timing compositionality, the absence of *any* timing anomaly is not strictly required by the above approach and only constrains the usable hardware platforms. Most known analyses for modern microprocessors

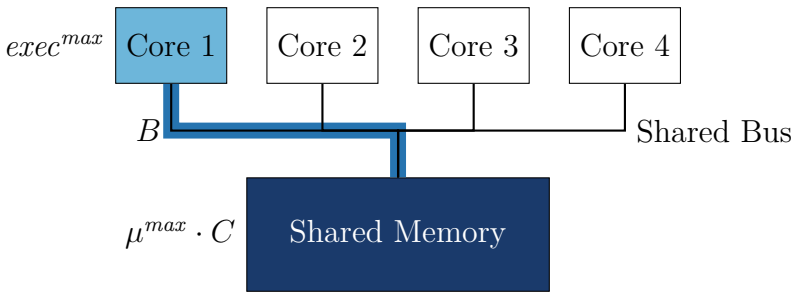


Figure 1: Multi-core system with shared bus and memory. Compositional view of the worst-case response time of a task running on Core 1.

(except for very simple ones) exhibit timing anomalous behaviour. Therefore, having a "fully timing compositional architecture" is a quite strong and possibly overly restrictive assumption. Assuming timing compositionality (see Section 3.2) of the bus blocking time w.r.t. execution and resource access time is already sufficient.

A detailed discussion of the term "timing compositional architecture" as well as further details on the relationship between timing anomalies and compositionality can be found in Section 3.4.

2.2 Preemptively Scheduled Systems

Altmeyer et al. [5] present a response time analysis for systems with fixed priority, preemptive scheduling in the presence of caches. An example schedule of two tasks is depicted in Figure 2. The worst-case response time of task 2 is prolonged by task 1 preempting it.

The response time of a task i is decoupled into (a bound of) its execution time(s) E_i without preemption, (a bound of) the execution time(s) E_j of tasks j possibly preempting it, and the preemption cost $\gamma_{i,j}$, i.e. the additional execution time of task i due to preemption by task j (and tasks with higher priority than j). In [5] and [6], Altmeyer et al. focus on the computation of the preemption cost that results from evicting useful cache blocks by preempting tasks. Other effects, not related to the cache, are considered constant and are assumed to be incorporated in the execution time bound. Their analyses approximate the number of additional cache misses a due to preemption in the worst-case and thus yield $\gamma_{i,j} = a \cdot \text{BRT}$. The block reload time (BRT) – the penalty that one additional cache miss contributes to the overall execution time – is assumed to be bounded by a constant. This does not always have to be the case: one cache miss could trigger a chain reaction in other parts of the system, whose timing effect is only bounded by the length of the program's execution path (so-called domino effects). Note, that chain reactions

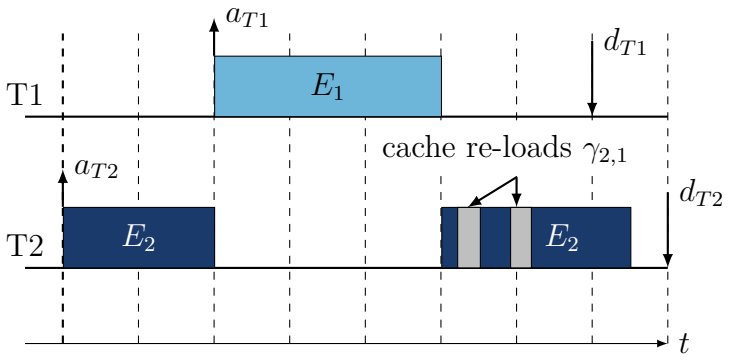


Figure 2: Compositional view on preemptively scheduled systems: Task 2 is preempted by task 1. Task 1 evicted useful cache contents resulting in additional cache re-loads. a_{Ti} arrival time of task i, d_{Ti} deadline of task i, E_i execution time of task i, $\gamma_{i,j}$ additional cost of task i due to preemption by task j.

within the cache are already captured by a.

The calculation of the timing contributions E_i, E_j, and $\gamma_{i,j}$ entails possible over-approximations. As an example consider the block reload time (BRT) that soundly approximates the timing contribution of one additional cache miss incurred at *any* point during execution. Depending on the execution context, e.g. the number of outstanding bus accesses to main memory, the time needed for one reload varies. Necessarily, a sound BRT might overestimate the actual reload time.

In case of multiple preemptions, the overall preemption cost depends on the number of preemptions, which again depends on the response time. Therefore, the authors employ a fixed point iteration scheme to compute a valid response time [5]. During an iteration step, the results of the previous iteration step are used to calculate n_j the number of preemptions by task j.

Timing compositionality is required to allow for a decoupling of the computation of the preemption costs and the execution time of the tasks without preemption. The combination of the individual analysis results

$$E_i + \sum_{j \text{ preempts } i} n_j \cdot (E_j + \gamma_{i,j})$$

leads to a globally sound result – in case of timing compositionality.

2.3 Dynamic Random Access Memory (DRAM)

The use of DRAM (compared to Static RAM) complicates timing analysis as the behaviour of the DRAM controller has to be additionally modelled. One

complication arises from DRAM refreshes that prolong ongoing memory accesses and appear (in general) asynchronously to program execution. A possibility to account for these DRAM refreshes in timing analysis is described by Atanassov et al. in [7]. Let t be a bound on the execution time assuming no DRAM refreshes, n the maximum number of refreshes occurring during program execution and d the maximal delay caused by a refresh. They claim that the execution time with enabled refreshes t_{DRAM} is bounded by $t + n \cdot d$. Thus, they implicitly assume timing compositionality because the penalty due to DRAM refreshes $n \cdot d$ and the execution time without refreshes t are independently computed and then summed up to obtain a valid bound. Note, that the computation of the maximal number of refreshes n might involve a fixed point iteration as it depends on t_{DRAM}.

2.4 The Essence

The examples presented above highlight the intuitive understanding of timing compositionality and thus form the basis for our formal definition in the next section. Therefore, we summarise the key insights gained in this section.

First, we have a *system*, e.g. a program executed on a multi-core processor, whose timing is of interest to us, i.e. the execution time of the program running on the multi-core. The timing of the system is *decomposed* into several *timing contributions* that capture different shares of the system's timing behaviour – e.g. the execution time without bus accesses, the resource access time and the bus blocking time. Next, the timing contributions are approximated separately by individual analyses. The individual analysis results are finally *combined* to a *sound* upper bound on the system's timing.

3 Timing Compositionality – Definition

In the previous section, we introduced the intuition behind timing compositionality. In this section, we want to present our formal definition and discuss it in detail.

3.1 Foundations

First of all, we need a notion of time. The set of possible timings is denoted by T. There are several possible choices for T. Using $T := \mathbb{N}_0$, we can model discrete processor cycles, while $T := \mathbb{R}^+$ could be used to model dense time (e.g. in case of multiple clock domains).

Timing Behaviour We consider a system, such as a program executed on a multi-core processor, whose timing is of interest to us. In general, the system's timing depends on its internal state (e.g. the processor's cache contents) as well as on the inputs to the system (e.g. sensor values). In analogy to finite state machines [2], we denote the pair of state and remaining input by the term *configuration*. The set of possible system configurations is denoted by C.

A description of the system's timing behaviour is given by a function $C \to T$. Provided a configuration c, the function returns a (bound on) the overall system's timing, i.e. the time needed by the system to reach a *final* configuration starting in c. Final means e.g. the termination of the program started in configuration c, or a configuration such that all tasks were successfully scheduled. Note that there are several possible functions that describe the system's timing behaviour in a sound way.

The system's timing is decomposed into individual *timing contributions* that capture a specific share of the system's timing. A timing contribution is again described by a function that returns a (bound on) a specific share of the overall system's timing starting in a given configuration.

A timing contribution does not necessarily depend on all the information that is available from a given configuration. E.g. to compute the execution time without bus accesses, no information about the bus is needed. Therefore, a timing contribution is associated with a set of subconfigurations C_i keeping the relevant information from the system configurations. A timing contribution is thus described by a function $tc_i : C_i \to T$.

To obtain a subconfiguration from a given system configuration, we use a *projection function* $p_i : C \to C_i$ that extracts the relevant parts of a system configuration. Subconfigurations of different timing contributions can share common information: E.g. to compute the resource access time and the execution time

without bus accesses, information about the executed program is needed in both cases.

The system's timing can also be seen as a timing contribution function $tc : C \to T$.

Decomposition of Timing Behaviours Before defining timing compositionality, we need a notion of decomposition of a system's timing into individual timing contributions. For exemplary decompositions, refer to Section 2.1 and 2.2.

Definition 3.1. Let $tc : C \to T$ describe the timing of the system under consideration. Furthermore, let $(tc_i : C_i \to T)_{i=1..n}$ be timing contributions that capture shares of the system's timing. We call $(tc_i)_{i=1..n}$ together with a family of projection functions $(p_i : C \to C_i)_{i=1..n}$ and a combination function $\bigoplus : T^n \to T$, a *decomposition* of tc.

The combination function captures the way individual timing contributions are combined in order to safely bound the system's timing. In the examples presented in Section 2, the combination is given by the addition operator, i.e. the individual timing contributions are added up to obtain an overall timing bound. Other combination functions are possible. E.g. in case parts of the system work in parallel and independently from each other, the combination function might compute the maximum of the timing contributions. In general, the combination function can be more complex and is determined by the chosen decomposition. For practical purposes (see Section 4.4), the combination function should be monotonic, i.e. a larger timing contribution should not lead to a smaller combined result.

3.2 Timing Compositionality

Definition 3.2 (Timing Compositionality)**.** Let $tc : C \to T$ describe the timing of the system under consideration. Furthermore, let the timing contribution functions $(tc_i : C_i \to T)_{i=1..n}$ together with configuration projection functions $(p_i : C \to C_i)_{i=1..n}$ and combination operator $\bigoplus : T^n \to T$ be a decomposition of $tc : C \to T$. We call the decomposition *timing compositional* if and only if

$$\forall c \in C.\ tc(c) \leq \bigoplus_{i=1}^{n} tc_i(p_i(c)).$$

Dependence on Decomposition Timing compositionality as defined above is a property that depends on a chosen *decomposition* of the system's timing behaviour. For one system's timing, there might exist multiple decompositions into different timing contribution functions, possibly associated with different projection functions and a different combination operator. Compositionality states that specific shares

of the overall system's timing, as described by timing contribution functions, can be considered separately. Which shares can be considered separately depends on the chosen decomposition.

There exist trivial decompositions (e.g. $n = 1$, $C_1 = C$ and $tc_1 = tc$) such that compositionality becomes a weak statement. Thus the significance of timing compositionality strongly relies on the significance of the decomposition. A significant decomposition will have non-trivial timing contribution functions. Furthermore, a significant decomposition will allow to obtain a conservative estimate of the system's timing more efficiently by separate estimations of timing contributions. (See paragraph on Complexity Reduction)

Nesting Note that the definition also captures nesting, i.e. that the definition is again applicable to any timing contribution $tc_i : C_i \to T$ (if considered as "system timing") and so on. Thus compositionality can be employed at different levels within a system.

Complexity Reduction Timing compositionality aims at reducing the computational complexity of analysing the timing of an entire system by combining the results of individual, less complex analyses of timing contributions. The projection functions $(p_i : C \to C_i)_{i=1..n}$ capture the information need to compute the timing contributions and thereby the amount of information that individual analyses have to consider. In case a projection function p_i is the identity function ($p_i(c) = c$), the timing contribution tc_i requires entire system configurations as input. It has the same information need as the system's timing tc, indicating a complex analysis of timing contribution tc_i. In general, the smaller the subconfigurations $p_i(c) \in C_i$, the better for the complexity of the individual analyses.

Approximation So far, timing compositionality (Definition 3.2) makes a statement solely about the correctness of the combined timings with respect to the system's timing. However, there might exist several different timing-compositional decompositions of one system's timing into timing contributions. For some decompositions, the combination of the timing contributions might approximate the overall system's timing quite closely, while the system's timing is considerably overestimated for other decompositions. To capture the precision of a decomposition, we refine our definition of timing compositionality as follows.

Definition 3.3 ((μ, α)-Timing Compositionality)**.** Let $tc : C \to T$ describe the timing of the system under consideration. Furthermore, let the timing contribution functions $(tc_i : C_i \to T)_{i=1..n}$ together with configuration projection functions $(p_i : C \to C_i)_{i=1..n}$ and combination operator $\bigoplus : T^n \to T$ be a decomposition of $tc : C \to T$. We call the decomposition (μ, α)-*timing compositional* where $\mu \in \mathbb{R}_{\geq 1}$, $\alpha \in \mathbb{R}_0^+$ if and only if

$$\forall c \in C.\ tc(c) \leq \bigoplus_{i=1}^{n} tc_i(p_i(c)) \leq \mu \cdot tc(c) + \alpha.$$

The additional inequality restricts the Definition 3.2 of timing compositionality because μ and α are finite constants.

Precision If a decomposition is (μ, α)-timing compositional, we have an upper bound on the overestimation of the combined timing contributions compared to the system's timing. The values μ and α are thus a measure of the *precision* of a specific decomposition: the system's timing is never overestimated by more than a factor of μ and an additive constant α. For a given decomposition and timing contribution functions, we are interested in the minimal μ and α such that compositionality still holds. Furthermore, a decomposition that permits small constants μ and α within reasonable complexity is preferable.

Next, we introduce some specific notions of timing compositionality based on Definition 3.3. Consider a (μ, α)-timing-compositional decomposition of a system's timing. We call the decomposition

- *timing compositional with constant-bounded effects* in case $\mu = 1$ and
- *fully timing compositional* in case $\mu = 1$ and $\alpha = 0$.

In case of a fully-timing-compositional decomposition, the two inequalities in Definition 3.3 imply equality,

$$\forall c \in C.\ tc(c) = \bigoplus_{i=1}^{n} tc_i(p_i(c)).$$

3.3 Compositionality and Composability

Timing compositionality and timing composability are two important properties with applications in timing analysis. In the following, we present examples to illustrate the respective properties as well as their relationship.

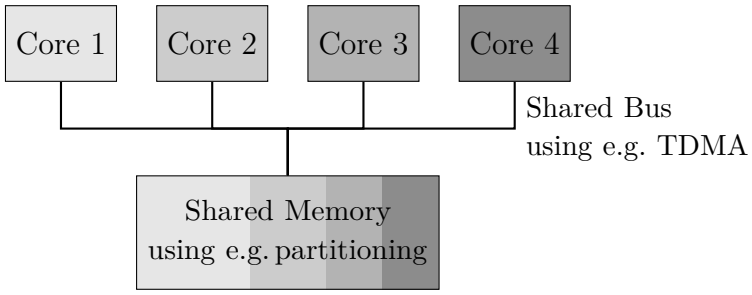


Figure 3: Multi-core system with shared resources: Composability by Temporal Isolation.

Compositionality Timing compositionality as defined in the previous section is a property of a *decomposition* of a given system's timing behaviour. It states that the timing behaviour of the system as a whole can be inferred from timing contributions of parts of the system and the type of composition. This enables a modular view on the system's timing behaviour.

As an example, consider the multi-core setting in Figure 1 together with the analysis presented in Section 2.1. The response time of the program running on Core 1 is decomposed into the computation time, the memory access time and the bus blocking time. Given correct timing contribution functions, this decomposition is *compositional*.

Composability In contrast, timing composability is a property of a system's timing behaviour within a larger environment. It states that the timing behaviour of the system is independent of the behaviour of the surrounding environment. The system's timing behaviour is thus not influenced by the integration with other systems. As a consequence, it can be analysed in isolation.

Consider the multi-core platform in Figure 3. The timing behaviour of one core, i.e. the worst-case response time of the program running on the respective core, is *composable*, if the timing behaviour is independent of the behaviour (e.g. accesses to the shared resources) of the other cores. Thus, composability allows for a separate verification of the timing behaviour of (the program running on) one core without knowledge about the behaviour of the other cores.

How to achieve Composability (for Multi-Core Processors) It is desirable, that the timing of the program running on one core is not influenced by other programs running on other cores – the timing is then *composable*. However, for current multi-core architectures this is not true: Interference on the shared bus as well as possible state changes of the shared memory (e.g. a cache) influence

the timing of programs. One way to achieve composability is to enforce *temporal isolation* at the implementation level as depicted in Figure 3. Several approaches to temporal isolation have been proposed such as TDMA arbitration of the shared bus and partitioning of the cache. A survey on the interference in multicores as well as techniques to achieve temporal isolation is given in [1]. Akesson et al. [3] and Goossens et al. [14] provide an overview of how to achieve temporal isolation in a system-on-chip setting. Another way to achieve timing composability at the analysis level is to conservatively account for the possible interference by the environment. As an example, round-robin arbitration does not achieve temporal isolation at the implementation level, yet, the latency of bus accesses can be bounded independently of the number of interfering accesses.

Interplay between Compositionality and Composability There is an interplay between the properties compositionality and composability. Consider the two previous examples together: the compositional decomposition of the response time of the program running on Core 1 (Figure 1), and the composable behaviour of the cores in the multi-core system (Figure 3). As the cores operate in a composable fashion, the bus blocking time only depends on the behaviour of the core under consideration and no longer on the behaviour of the other three cores. Thus, the computation of the bus blocking time is simplified.

3.4 Timing Compositional Architectures

Previously, there have been attempts towards defining a notion of compositionality. In [37], Wilhelm et al. give definitions for *timing compositional architectures* based on the notion of so-called timing anomalies and domino effects. A timing anomaly describes a situation during analysis where the locally worst choice (cache miss) does not lead to the globally worst timing. If the effect of the local choice on the global result cannot be bounded by a constant, the anomalous situation is called domino effect. A definition of timing anomalies and domino effects as well as concrete examples can be found in [28]. Fully timing compositional architectures are then defined as architectures whose abstract model does not exhibit timing anomalies (nor domino effects). In case there are timing anomalies but no domino effects, the architecture is classified as compositional with bounded effects – and non-compositional otherwise.

Compared to our definition, their notion of timing compositionality is a property of a model of a system and not of a behavioural decomposition of a model of a system, which we consider an important aspect. Our definition of compositionality is always meant with respect to a specific underlying decomposition of a system's timing behaviour into timing contributions. In contrast to the definition of timing

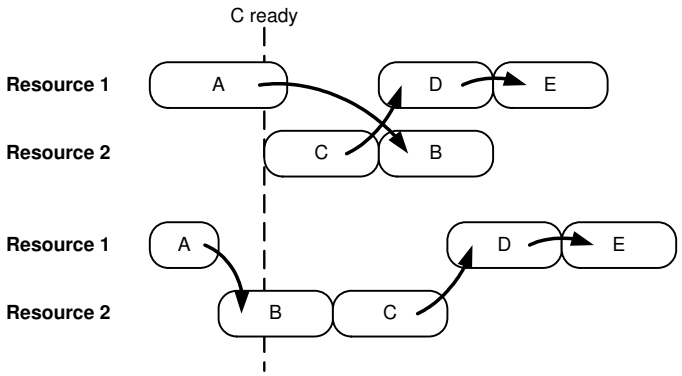


Figure 4: Scheduling anomaly [28]: Shorter execution of instruction A leads to longer overall execution. Timing compositionality is not necessarily affected by anomalous behaviour.

compositional architectures [37], our definition does not forbid arbitrarily complex timing behaviours within parts of the system. In particular timing-anomalous behaviour is not forbidden in general. As an example, consider the decomposition of a processor with out-of-order execution into timing contributions of the pipeline and the cache. The analysis of the pipeline's timing contribution might have to take timing anomalies into account due to dynamic scheduling effects caused by out-of-order execution – so-called scheduling anomalies (see Figure 4, [28]). However, the decomposition into timing contributions of the pipeline and the cache can be timing compositional, e.g. in case the pipeline stalls execution while servicing a cache miss from main memory. We present a hardware-design guideline by means of a stalling mechanism in Section 6 that enables compositionality of such decompositions by construction. The property to be timing compositional is then *unaffected* by the anomalous behaviour *within* parts of the system (such as a pipeline).

Furthermore, *the* abstract model of an architecture is not uniquely defined. As already stated in Section 3.1 and Section 3.2, there might exist several decompositions of a system's timing; and multiple different functions are possible to describe the timing of a system. Similarly, several different abstract models exist for one architecture – some of which may exhibit timing anomalies while others do not. Therefore, relating the above definition to architectures rather than to specific formal models is problematic.

3.5 Summary

We started by introducing basic concepts such as configurations, timing contributions and the decomposition of a system's timing behaviour. Next, we presented

our definition of timing compositionality, following the intuition given in Section 2. A decomposition is called *timing compositional* if and only if the combination of individual timing contributions is always an upper bound on the system's timing. We later refined this notion to (μ, α)-timing compositionality to incorporate a notion of precision.

In the following, we distinguished timing compositionality from timing composability and we described their respective applications in timing analysis.

Finally, we discussed the previous definition of timing compositional architectures [37]. We sketched the issues that arise from this definition and we highlighted the differences with respect to our definition of timing compositionality.

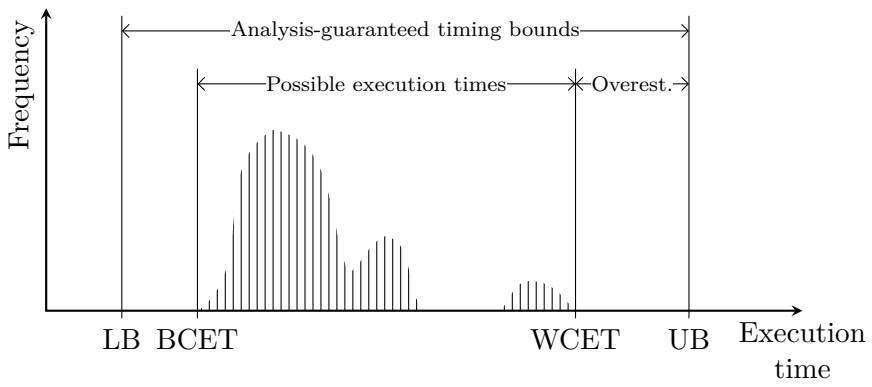


Figure 5: The worst-case execution time problem. Introducing basic notions.

4 Worst-Case Execution Time Analysis

As mentioned earlier, timing compositionality has applications in the context of timing analysis (Worst-Case Execution Time Analysis). After a short introduction, we give an overview of the integrated and compositional analysis approach and we discuss their respective advantages and shortcomings. Furthermore, we present potential applications of timing compositionality in execution time analysis. This is complemented by a summary of challenges in compositional timing analysis.

4.1 The Worst-Case Execution Time Problem

In a hard real-time setting, the possible execution times of a given program p running on a hardware platform are of major interest for proving timeliness of the system. The execution time of a program varies due to e.g.:

- different inputs leading to different execution times of individual instructions (variable-latency instructions) or even to different paths through the program p,
- different initial microarchitectural states such as the varying cache contents leading to different memory access latencies, and
- different execution environments such as co-running programs on a multi-core system causing interference on shared resources.

Therefore, we have a distribution of possible execution times as depicted in Figure 5.

We are interested in *safe approximations* of the execution times – especially upper bounds on the execution times including the execution time in the worst case. As a sufficient criterion that such upper bounds exist, the program p is

assumed to terminate, i.e. the number of loop iterations and the depth of recursive function calls are known. The bounds should be as tight as possible in order to avoid over-provisioning of system resources.

One technique to compute such bounds is the exploration of the space of system configurations within a single (integrated) analysis – possibly employing abstractions (e.g. value or cache abstractions) to speed up computation. In contrast, timing compositionality enables the decoupling into multiple, independent analyses. Before going into detail about this, we define some basic notions in the context of WCET analysis.

Definition of the WCET The execution time of a program $p \in \mathit{Prog}$ depends on the input (initial values of registers and memory) $i \in \mathit{MemReg}$, the initial microarchitectural state $\mu a \in \mu\mathit{Arch}$, and the state of the execution environment $e \in \mathit{Env}$. *Prog* denotes the set of program states (the program itself and the program counter), *MemReg* the set of register and memory values, $\mu\mathit{Arch}$ the set of microarchitectural states (e.g. pipeline state and cache contents), and *Env* the set of environmental states (e.g. state of co-running cores, shared resources). The whole system's configuration space C is then described by $\mathit{Prog} \times \mathit{MemReg} \times \mu\mathit{Arch} \times \mathit{Env}$. We call $\mathit{Prog} \times \mathit{MemReg}$ the instruction-set-architectural (ISA) configuration as it is sufficient to determine the semantics of the program, e.g. which values are computed during execution.

The timing contribution function of the concrete system $tc : C \to T$ maps a configuration $c \in C$ to the number of cycles needed to execute the system from c to a *final configuration*, i.e. a configuration in which the initial program terminates. In case of non-determinism, the timing function gives the maximal possible number of cycles to reach a final configuration from c. The worst-case execution time of a program p is then defined as

$$\mathit{WCET}_p := \max_{i \in \mathit{MemReg}} \max_{\mu a \in \mu\mathit{Arch}} \max_{e \in \mathit{Env}} tc(\langle p, p_{\mathit{init}} \rangle, i, \mu a, e), \tag{1}$$

where p_{init} denotes the entry point of program p.

In case the possible inputs are restricted to $I \subseteq \mathit{MemReg}$ (analogously for restrictions on $\mu\mathit{Arch}$ or *Env*), we can refine the above Equation 1 to

$$\mathit{WCET}_{p,I} := \max_{i \in I} \max_{\mu a \in \mu\mathit{Arch}} \max_{e \in \mathit{Env}} tc(\langle p, p_{\mathit{init}} \rangle, i, \mu a, e).$$

Microarchitectural Analysis and Compositionality Microarchitectural analysis, i.e. the analysis concerned with the microarchitectural influence on execution time, is a complex task due to the interdependencies in modern processors as described in the next section. Timing compositionality aims at splitting the microarchitectural

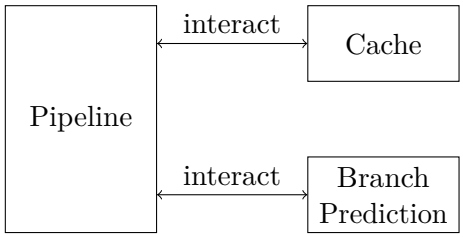


Figure 6: Modern microprocessors feature several performance-enhancing components that interact closely. These interactions and possibly occurring interferences complicate WCET analyses.

analysis into less complex subanalyses that focus on specific features of the hardware separately. In this setting, individual timing contributions are often associated with physical, microarchitectural *components* such as a cache, a pipeline, or a branch prediction unit. The structural organisation of a microarchitecture into components guides the decomposition of the microarchitecture's timing behaviour. In the following, we focus on compositionality in the context of microarchitectural analysis.

4.2 Interdependencies in Modern Microprocessors

Modern microprocessors have several performance-enhancing features such as (out-of-order/ superscalar) pipelines, caches and branch prediction units. They aim at increasing the number of completed instructions per cycle, e.g. by exploiting instruction-level parallelism. As an example, pipelining allows to overlap computation and memory accesses, thereby partially hiding latencies such as main memory access latency. These performance-enhancing features introduce non-trivial interdependencies that make timing analysis a complex task [20]. Consider a microprocessor with pipelining, caching and branch prediction as depicted in Figure 6.

Pipelining and Caching The position of an instruction in the pipeline at a certain point in time depends on whether the fetch of the instruction itself and the fetch of its operands hit the cache or not. In the first case, the instruction can be dispatched and executed early while it has to wait for main memory otherwise. The contents of the cache are determined by its replacement policy and the memory access history. The access history is, among others, influenced by out-of-order scheduling (pipeline) and the interleaving of instruction and data accesses in case of a unified cache.

Pipelining and Branch Prediction Whether the branch prediction unit speculates on the outcome of a branch depends on the state of the pipeline, e.g. how far the branch instruction advanced in the pipeline and when the condition can be evaluated. The condition evaluation can depend on ongoing computations or data fetches from main memory. Vice versa, the pipeline state is influenced by the decision of the branch prediction. In case of correct speculation, the latency introduced by condition evaluation is hidden. Otherwise, a misprediction causes the pipeline to roll back the incorrect execution of instructions and to redirect the fetching to the correct target, which is overall expensive.

Transitively, caching and branch prediction are also interdependent, e.g. speculative memory accesses can have detrimental effects on the cache in case of branch misprediction.

In general, these interdependencies make compositional (also called modular) analyses more complicated and affect the obtainable precision negatively. Heckmann et al. [20] conclude that modular analyses are prohibitive, and only an integrated approach is feasible. Next, we describe this integrated approach and outline a compositional approach to timing analysis. Finally, we trade off the integrated against the compositional approach and discuss the issues of analysis precision and efficiency in Section 4.5.

4.3 The Integrated Analysis Approach

Overview As already mentioned earlier, the integrated approach to timing analysis considers configurations of the whole system at once. It starts with an initial set of possible system configurations and explores the configurations that are reachable by executing a given program. Thereby, the analysis closely follows the real execution on a cycle level. Later on, a path analysis is employed to find the longest path from an initial configuration to a final configuration, taking loop bounds, recursion depth, etc. into account.

To allow for a more compact representation, similar concrete configurations are condensed to one abstract configuration. This abstraction step potentially introduces uncertainty to the analysis w.r.t. the succeeding (abstract) configuration. In case of analysis uncertainty, e.g. whether a memory access hits the cache or not, the current abstract system configuration is split into several succeeding configurations – one configuration per possibility. In the context of finite state machines, the integrated analysis approach corresponds to the search of a *longest sequence of consecutive configurations* from an initial to a final configuration. Uncertainty during analysis corresponds to non-determinism in the finite state machine [2]: all possible successor configurations have to be considered.

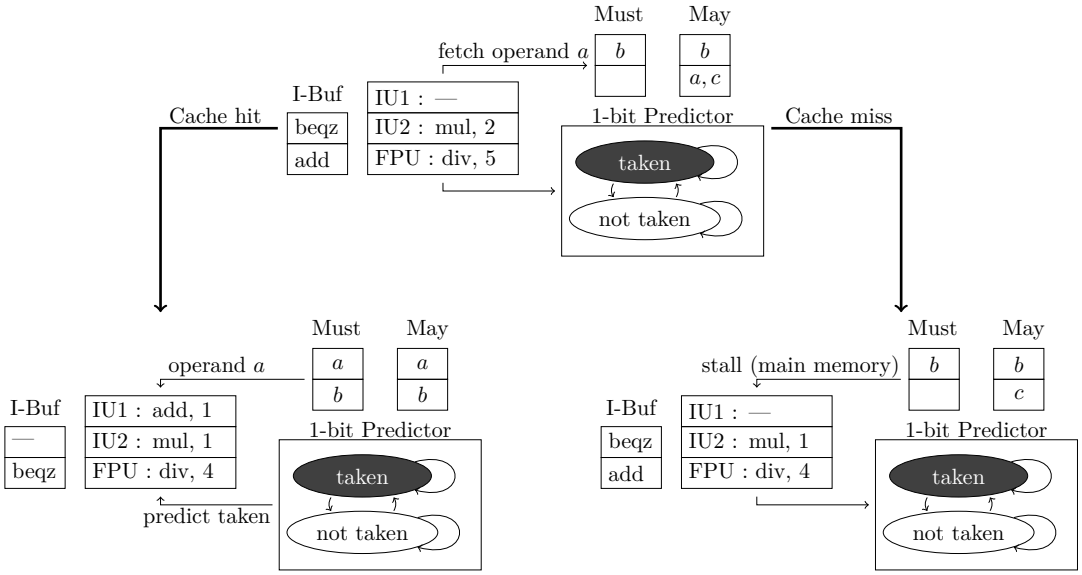


Figure 7: Example step during the integrated analysis.

In practice, the program to analyse is given in form of a control-flow graph. The integrated approach as described by Thesing [35] explicitly explores the space of microarchitectural states that arise during program execution – it does not explicitly explore the space of whole system configurations. The following discussion therefore focuses on the exploration of the microarchitectural state space, but it can be generalised to the exploration of the system's configuration space.

Example Consider the analysis of a microarchitecture as depicted in Figure 6. Three possible abstract microarchitectural states that arise during the microarchitectural state space exploration are depicted in Figure 7.

One microarchitectural state comprises the state of the pipeline, branch prediction unit, and cache. The pipeline has an instruction buffer that holds fetched instructions waiting for dispatch and several functional units that might be occupied by the execution of an instruction. The branch prediction unit (1-bit dynamic prediction) can either be in "predict taken" or "predict not taken" mode – depending on the behaviour of the last branch. The concrete cache state records all memory blocks that are currently cached. Even for caches of moderate size, enumerating all concrete cache states arising during the state space exploration is impracticable w.r.t. memory consumption. Fortunately, efficient and sufficiently precise abstractions have been found for certain cache architectures, e.g. for LRU caches in terms of must and may analysis [4]. The must cache under-approximates

the concrete cache contents and can be used to predict cache hits. The may cache over-approximates the concrete cache contents, and its complement can be used to predict cache misses. For details about the analysis of caches, see [15].

Next, we discuss one example exploration step. Consider the top abstract microarchitectural state in Figure 7. In the next cycle, the `add` instruction is about to be dispatched. Prior, one of its operands, a, has to be fetched from memory. It is uncertain whether this access hits the cache – it *might* (may cache) but it *does not have to* (must cache). Therefore, the current abstract state is split into the cache hit case and the cache miss case.

In case of a cache miss, fetching and dispatching (in-order dispatch) is stalled until the main memory request has been served. In the meantime, the execution of the `mul` and `div` instructions advances and thereby hides the main memory latency. Note, that the cache information can be refined. The state of the branch predictor stays unchanged.

In case of a cache hit, the memory request is served within one cycle and the `add` instruction can be dispatched. The `mul` and `div` instructions also advance in the execution pipeline in parallel. There is one empty slot in the instruction buffer, so a new instruction can be fetched. The address of the next instruction depends on the outcome of the branch `beqz`. The branch prediction unit speculates that the branch will be taken and thus determines the next instruction to fetch (speculatively). As soon as the branch condition is evaluated, the state of the predictor is adjusted accordingly.

4.4 Towards Compositional Analyses

Instead of analysing the timing behaviour of a microarchitecture in an integrated fashion, it can also be analysed in a compositional way. We first decompose the timing behaviour into timing contributions of microarchitectural components. Second, we *analyse* (i.e. we compute sound bounds on) the timing behaviour of each component individually. Third, we combine the obtained timing contribution bounds to an overall bound on the microarchitectural timing. Timing compositionality of this decomposition guarantees the soundness of this result.

In Section 2, we already presented examples for compositional analyses. Now, we focus more on the analysis *framework* rather than the individual analyses of the components. These individual analyses are in general specific to the decomposition and should therefore be discussed elsewhere.

Consider Figure 8 that provides an overview of the overall structure of the compositional analysis framework. For a given program p, we want to approximate the worst-case execution time $WCET_p$ (Equation 1). First, a timing-compositional decomposition of the microarchitectural timing into components' timing contribu-

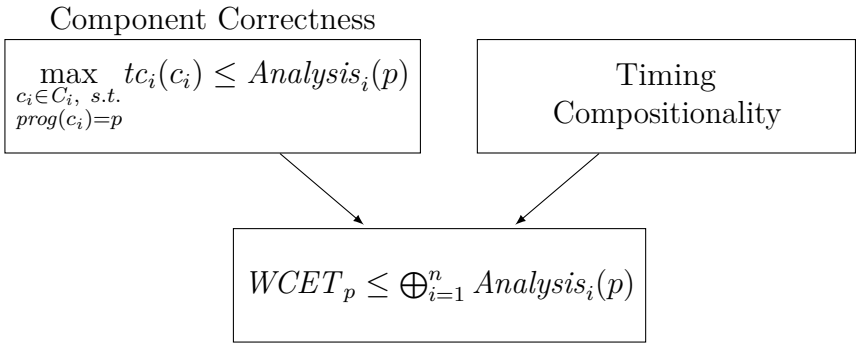


Figure 8: Overall structure of compositional analyses.

tions is required. Second, we need to come up with an analysis per component that soundly approximates the timing contribution of the respective component when program p is executed (component correctness). Each analysis has knowledge about the program p under consideration – possibly also about the inputs, initial microarchitectural states, or the environment if such additional knowledge is available (e.g. when approximating $WCET_{p,I}$). Finally, if the combination operator of the decomposition is monotonic, we can conclude that the worst-case execution time $WCET_p$ as defined in Section 4.1 is soundly approximated by the combination of the individual analysis results. We go into more detail about possible analysis frameworks in Section 5.

4.5 Integrated versus Compositional Approach

We described two possible approaches to timing analysis in the previous sections. We advocate neither the integrated nor the compositional approach *in general* – rather a careful trade-off between the two *from case to case*.

Before we go into details about this trade-off, we want to correlate the two analysis approaches to methods known in program analysis based on abstract interpretation. Component-wise combination methods as described in [24] allow to construct an analysis for a composite structure based on analyses for individual components of this structure. The *independent-attribute method* resembles our compositional approach: Individual analyses are kept separate and their results are combined by means of a cartesian product. While being efficient, the missing correlation between individual analyses can lead to precision loss. The *relational method* resembles the integrated approach: The overall analysis tracks the correlations between individual analysis results by employing an expensive power set domain. In general, the increase in precision comes at the price of decreased efficiency.

Now, we want to discuss the advantages and shortcomings of the integrated as

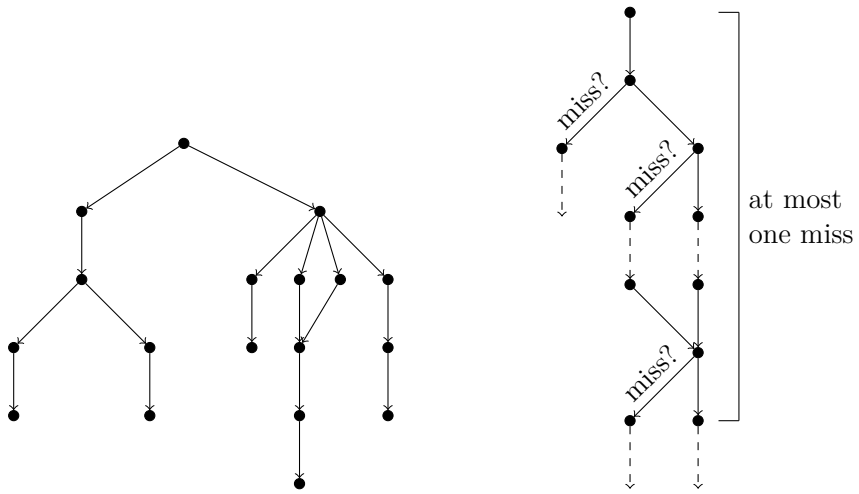
well as the compositional approach. The integrated approach captures the relations between components at a fine-grained level and is thus able to detect overlapping effects, such as the hiding of memory latency. This comes at the price of high analysis time and high memory consumption. Figure 9a depicts an example graph that might arise during the microarchitectural state space exploration. The circles denote abstract microarchitectural states and each edge denotes a possible state evolution during one processor cycle. We start with one initial abstract state and we split in case of uncertain information e.g. cache hit or miss, or access to an unknown address. If two abstract states are sufficiently similar (e.g. the states of pipeline and branch predictor are identical), they can be joined, which helps efficiency. Nevertheless, the resulting state space is typically still very large. Each additional processor feature is a further source of uncertainty and might cause additional splits during analysis.

Theoretically, the integrated approach has the potential to yield the most precise results possible. However in practice, the obtainable precision is limited by efficiency concerns. As an example, consider the use of so-called *cumulative* information in the integrated analysis. We distinguish *local* information, i.e. information valid at a particular point during analysis, and *cumulative* information, i.e. information valid for a group of analysis points. An example for local information is "*this access* will hit the cache", whereas "at most one of *these accesses* will miss the cache" is an example for cumulative information. Such cumulative information can be more precise than purely local information. In our example depicted in Figure 9b, we know that at most one cache miss occurs during execution (cumulative information). In case we do not know exactly which access misses the cache, we have to (locally) assume that every access might miss the cache. This leads to a growth of the state space. The joining of microarchitectural states as described in the previous paragraph has the detrimental effect, that the subsequent path analysis identifies the (infeasible) path with two misses as worst case. Integrating cumulative information as constraints into the path analysis is possible but makes the analysis less efficient.

At this point compositionality comes into play. As the components are analysed individually, uncertainty of the individual components does not multiply. Thus we can avoid the microarchitectural state space explosion problem – even in case that each component is analysed using state space exploration techniques.

Cumulative information could also be used in an efficient manner. In case of a timing-compositional decomposition with respect to the cache, the cache timing contribution in the above example (Figure 9b) is the penalty of one miss. This is a precision improvement regarding the result of the integrated approach.

Thus, timing compositionality might lead to a more efficient overall analysis, it enables new types of possibly more precise analyses (those that yield cumulative



(a) Complexity. Uncertainty causes splitting of states leading to state space explosion.

(b) Imprecision. How to incorporate cumulative information?

Figure 9: Shortcomings of the Integrated Analysis Approach. Circles denote abstract microarchitectural states. Edges denote one cycle execution.

statements) and allows for the analysis of currently unsupported features (e.g. write-back caches or multi-core systems, see Section 4.6 for a list of example applications). However, achieving timing compositionality – either by analysis or by design – is not trivial and comes, in general, at the price of overestimation or performance degradation. Such overestimation can emerge if the compositional analyses are not able to capture overlapping effects in tightly-coupled systems, e.g. when components act in parallel. Analyses that capture such overlapping effects have to analyse the interacting components together, which might be as expensive as an integrated analysis.

4.6 Example Applications for Compositional Timing Analysis

There are several systems for whom compositional analyses could be beneficial in terms of increased analysis efficiency, but also increased analysis precision (e.g. if only cumulative information is available). We already described an analysis of preemptively scheduled systems [5, 6] and an analysis of shared buses in multi-core systems [31, 32] in Section 2. In the following, we list some more examples.

- **Shared Caches in Multicores**. The execution of a program p running on one core and using a shared cache experiences interference (additional to bus interference): Co-running programs might evict cache lines that are useful for

p. Due to this interference, it is hard to obtain a local classification of an access as either hit or miss. Assuming the worst case for each local classification leads to severe overestimation. If at all, bounding the amount of interference in a cumulative way rather than locally classifying memory accesses as hit/miss is more likely feasible in terms of precision and efficiency. The overall system's timing is decomposed into execution time without accesses to the shared cache and the time needed to serve accesses to the shared cache.

- **Asynchronous events such as DRAM refreshes**. Events that happen asynchronously w.r.t. program execution introduce uncertainty as to when an event occurs during program execution. Therefore, it is hard to predict how an event affects a program's timing behaviour. DRAM refreshes are an example for such events: they happen periodically and asynchronously. Whether an individual DRAM access is affected by a refresh, thereby prolonging the overall access time, is hard to predict statically. In case the system is compositional with respect to such events (this can e.g. be achieved by stalling), their overall effect on the execution time can be considered separately – possibly in a cumulative way.

- **Write Backs in Caches**. In contrast to write-through caches, a store only modifies a cache line, marks it as dirty and delays the main memory operation until the dirty cache line is evicted – potentially reducing the overall number of memory accesses. While the memory operation is performed at a statically known point in time for write-through caches, the write-back policy introduces uncertainty as to when the memory operation happens. This is similar to the effects of asynchronous events. This uncertainty renders the integrated, non-compositional analysis approach infeasible in terms of complexity. Consider a decomposition into execution time in the absence of write backs and the overall time needed to perform occurring write backs. This compositional view enables the use of cumulative information about the number of performed write backs – independent of the point in time they happen. The number of potentially performed write backs could be approximated efficiently by tracking the dirty bit of a cache line during a separate cache analysis.

- **Cache Analysis** Cache analyses based on local classifications of accesses as either hit or miss are well-known and are used in the integrated approach to avoid splits. However, there are analyses that provide (more precise) cumulative information about the cache behaviour, e.g. cache persistence analyses [9, 23]. These are of use in the compositional analysis setting. There are further examples for analyses computing sound cumulative cache

information. Cumulative information for non-LRU cache replacement policies can e.g. be derived using sound LRU cache analyses and corresponding relative miss-competitiveness ratios [27]. In [16, 17], Guan et al. present analyses that compute cumulative information for caches with the FIFO and MRU replacement policies. Motivated by cache-aware compiler optimisations, Ghosh et al. [12] and Chatterjee et al. [8] present techniques to cumulatively predict cache performance in loops. Ghosh et al. use Cache Miss Equations to model the cache behaviour within a loop while Chatterjee et al. employ Presburger formulas for the modelling.

4.7 Challenges in Compositional Timing Analysis

Besides the presented opportunities, there are several challenges in the context of timing compositionality and its use for timing analysis.

Existing Analyses For existing analyses such as described in Section 2, we need to check the validity of the compositionality assumption. The analyses already determine an underlying decomposition of the system's timing into timing contribution functions. However, a formal proof of timing compositionality of this decomposition w.r.t. the concrete system has yet to be done – as well as correctness proofs for the individual analyses w.r.t. the timing contribution functions. In many cases, this requires the computation of sound penalties for individual events, e.g. the block reload time in case of an additional cache miss. The penalty depends on the underlying microarchitecture and its behaviour in case such an event (e.g. a cache miss) occurs. Ideally, the penalty would be computed by a sound static analysis on a Verilog/VHDL model of the microarchitecture whose timing is preserved during synthesis.

Designing new Analyses First, obtaining a compositional decomposition of a system's timing behaviour into timing contributions is crucial – especially such that μ and α are small. Typically, we already have a decomposition in mind when talking about timing compositionality, e.g. into cache timing and pipeline timing, or into computation and communication time. The timing contribution functions specify what has to be approximated by sound analyses. The obtained timing contribution functions enable, in the ideal case, an easy design of sound analyses and an easy proof of timing compositionality. In the following section, we present a possible approach to tackle the systematic derivation of compositional timing contributions for microarchitectural analysis.

Instead of a single integrated analysis, compositionality allows for multiple individual analyses whose combination upper bounds the execution times of a given

program. Ideally, these analyses are more efficient in terms of analysis time and memory consumption than an integrated analysis. An analysis framework comprises conditions that the respective analyses have to fulfil ("What is approximated?") and methods to compute analysis results. We present two possible analysis frameworks in Section 5.3.

4.8 Summary

In general, neither the integrated nor the compositional approach to execution time analysis is best. Which type of analysis to choose depends on the behaviour of the system, the available amount of computational power and memory, and the degree of overestimation/performance degradation that is tolerable. We advocate the use of integrated analyses in tightly-coupled systems such as out-of-order processors for reasons of precision. We advocate the use of compositional analyses on a larger scale such as multi-core systems or systems with "external" components such as caches for efficiency reasons – but also for reasons of precision in case only precise cumulative information is available.

We have presented example applications for the compositional approach to timing analysis, such as the analysis of write-back caches or shared caches in multicores. In the next section, we focus on the challenges that have been outlined previously.

5 Compositional Timing Analysis

As outlined in Section 4.7, finding a suitable decomposition of a system's timing behaviour into timing contributions is crucial for the design of new compositional analyses. In the following, we present one possible approach to the derivation of timing contribution functions in the context of microarchitectural analysis. This approach makes use of the underlying structural organisation of the microarchitecture into components, e.g. caches and pipeline.

The timing contribution functions form the conceptual basis for the design of individual compositional analyses by determining what to approximate. Consequently, we present a generic, compositional analysis that is based on accident counting.

Third, we describe two possible *analysis frameworks* – i.e. how individual analyses can be combined to obtain overall sound results. We conclude this section by discussing practical aspects of a new, compositional WCET analysis tool chain.

5.1 Accountability, or: Deriving Timing Contribution Functions

The main idea is to associate each execution cycle with those microarchitectural components that are considered "accountable" for this cycle. The approach relies on a description of the system's (timing) behaviour on a cycle-level, e.g. given as transition system. We explain it with an example at hand. We have a processor system with configuration space C together with two microarchitectural components A and B with respective configuration spaces C_A and C_B and projection functions $p_i : C \to C_i$. We assume $C = P \times M$, where P represents the instruction-set-architectural configuration and M the micro-architectural configuration. Furthermore, we assume that our system can be modelled as a transition system with the processor system's configuration space as set of states and a transition relation that models one execution cycle of the system. We make no assumptions on the determinism of the transition system.

Consider the snippet of an example transition system in Figure 10. Some transitions are annotated with names of components, possibly several names per transition. A transition is labelled with A if component A is *accountable for/causes* this cycle. For example, the cache is considered accountable for the cycles during which the pipeline is stalled due to a cache miss. These cycles are necessary for the main memory to serve the requested data. In case of a cache hit, these cycles would not occur. Hence the according transitions are labelled with the name of the cache component. Pipelining effects may cause the execution of a functional unit (e.g. component B) to overlap with the cache miss. In such cases, a transition is labelled with the names of multiple components.

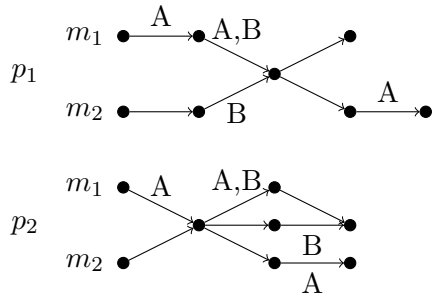


Figure 10: Deriving timing contribution functions in the accountability approach by labelled transition systems. p_i denote different programs, m_j different microarchitectural states. (p_i, m_j) form the system states. The labels A and B refer to microarchitectural components.

Labelling – An example The labelling could be inferred dynamically during execution or simulation as follows. A processor pipeline comes with a stall engine that regulates the instruction flow through the pipeline by stalling specific pipeline stages in case of hazards (e.g. by inserting pipeline bubbles). In case of hazards, some input signals to the stall engine are active indicating the reason for a stall. These hazard signals could be used to annotate the transition system. In case no hazard signal is active, the transition is not labelled as it is caused by the normal program execution – and would occur even during an *ideal* execution. These transitions are taken into account by a special timing contribution function tc_0.

Timing Contributions The derivation of timing contribution functions is based on the length of paths through the transition system. Let π be a path through the transition system that starts in π_0 and finally ends in π_{l-1} (every program is assumed to terminate for each input and initial hardware state). We call $|\pi| := l$ the length of this path. By $|\pi|_A$, we denote the number of transition labelled with A on path π. The timing contribution $tc : C \to T$ of our system is then given as

$$tc(c) := \max_{\pi, \pi_0 = c} |\pi|.$$

The function $tc_0 : C \to T$ accounts for the ideal execution and is defined by the maximal number of unlabelled transitions on any path.

$$tc_0(c) := \max_{\pi, \pi_0 = c} |\pi|_0 = \max_{\pi, \pi_0 = c} |\pi|_{\text{unlabelled}}$$

We define the timing contribution function $tc_A : C_A \to T$ for component A once – the function for B can be derived analogously.

$$tc_A(c_A) := \max_{\pi, p_A(\pi_0)=c_A} |\pi|_A$$

The results of the timing contribution functions are then combined in an additive way.

In case a transition is labelled with multiple components, this transition is also accounted for multiple times leading to overestimation. This overestimation could be alleviated by assigning unique *priorities* to the components. If a transition is labelled multiple times, it is only accounted for by the component with the highest priority. As an example, consider a multi-level cache hierarchy where each cache-level is considered a component and cache lookups are done in parallel. Each cache-level accounts for the parallel lookup separately, which leads to overestimation. When assigning higher priorities to higher cache levels, only the highest cache-level will account for the parallel lookup. Finding analyses that profit from these tighter timing contribution functions might nevertheless be complicated.

We conclude by proving timing compositionality for decompositions that have been derived as described above.

Theorem 5.1. *Let a system with n components be given together with configuration spaces C, $(C_i)_{i=1..n}$ and projection functions $(p_i : C \to C_i)_{i=1..n}$. Furthermore, let the timing contribution functions be as described above with the addition in the time domain as combination operator. The obtained decomposition of the system's timing tc is timing compositional:*

$$\forall \langle p, m\rangle \in C.\ tc(\langle p, m\rangle) \leq \sum_{i=0}^{n} tc_i(p_i(\langle p, m\rangle)).$$

Proof. Let a configuration $\langle p, m\rangle$ be given. There exists a path π through the transition system such that $tc(\langle p, m\rangle) = |\pi|$. By construction, each transition on that path is either unlabelled or labelled at least once. Thus, there is a covering of the path $|\pi| \leq \sum_{i=0}^{n} |\pi|_i$. Each timing contribution function takes the maximum over all paths, so $|\pi|_i \leq tc_i(p_i(\langle p, m\rangle)$ for all $0 \leq i < n$. Thus, the above inequality holds and the decomposition is timing compositional. □

Summary – Derivation of Timing Contributions In the first place, this approach provides a conceptual view on the derivation of timing contribution functions for individual components. The timing contribution of a component should capture the amount of execution time it is "accountable" for. The definition of "accountability"

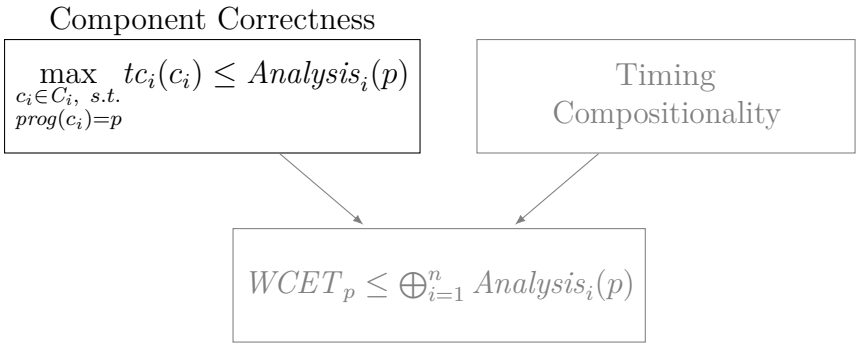


Figure 11: Compositional analysis framework. How to obtain individual analyses?

and the identification of accountable components are thereby the key parts. However, the automatic derivation of timing contributions, given a transition system, a set of components, and an "accountability metric", might not be practically feasible – especially for systems with large configuration spaces. The accountability approach forms a conceptual basis for the individual analyses in a timing compositional setting by determining what to approximate.

5.2 Compositional Analysis based on Accident Counting

Besides the compositionality property, we require analyses that individually capture timing contributions (Figure 11). Now, we want to illustrate one generic type of compositional analysis that is based on counting so-called *timing accidents*. A timing accident is an event that has a detrimental effect on the overall system's timing. Possible timing accidents are

- **write backs** when analysing write-back effects in caches,
- **cache misses** when analysing the additional cache misses due to preemption,
- **accesses to shared resources** when analysing communication time separately from computation time in a multi-core setting.

First, we have a *baseline* analysis that captures the timing behaviour B under the assumption that no timing accidents occur during execution. Second, we have an analysis that counts the number of accidents n in the worst-case. In a pre-processing step, we have to compute a sound accident penalty a that captures the timing effect of one timing accident on the overall system's timing. Ideally, this penalty is obtained by a sound (static) analysis of a formal system model (e.g. given in Verilog/VHDL).

Thus, we have a timing-compositional decomposition into *baseline* execution and *timing accidents* such that $B + n \cdot a$ is an upper bound on the overall system's timing. The example analyses in Section 2 also follow this form, e.g. the computation of the preemption cost $\gamma_{i,j}$. Note, that the penalty might not be bounded by a constant for some systems. This type of analysis based on accident counting is thus not feasible in such cases.

5.3 Analysis Frameworks

In this section, we present possible analysis frameworks applicable in a timing-compositional setting, i.e. how individual timing contributions can be safely combined (Figure 12). We give necessary conditions that the analysis of each individual component has to fulfil, to obtain overall sound results. We show that the $WCET_p$ (Equation 1) is safely approximated in these cases.

While a timing contribution function makes a statement about the execution behaviours starting from one initial configuration (namely the one passed as argument), an analysis has to compute bounds on the execution behaviours starting from all possible initial configurations. There are different kinds of analysis uncertainty, e.g. uncertainty inherent to varying input or unknown initial microarchitectural state but also introduced by analysis abstractions. We discuss sources of uncertainty and how to reduce potential overestimation.

We present two possible analysis frameworks. The first one approximates the maximum of each timing contribution separately. While being a simple and computationally inexpensive analysis framework (ignoring the cost of the individual analyses), the drawback of this approach is that it allows for infeasible combinations of configurations. The component configurations c_i^{wc} that lead to the individual worst cases are not necessarily compatible, i.e. there is no system configuration c such that all components behave worst $p_i(c) = c_i^{wc}$. The partitioning approach refines this separate maxima approach by ruling out infeasible paths.

Separate Maxima Approach

Let the decomposition of a system's timing behaviour $tc : C \to T$ into timing contributions $(tc_i : C_i \to T)_{i=1..n}$ with an additive combination operator be timing compositional. Let C, $(C_i)_{i=1..n}$ denote the respective configuration spaces as described in Section 4.1 with projection functions $(p_i : C \to C_i)_{i=1..n}$. We consider the execution times of a program p with entry point p_{init}. In the Separate Maxima Approach, an analysis A_i of the timing of component i is *sound* if

$$A_i(\langle p, p_{init}\rangle) \geq \max_{\substack{m,\mu a,e \text{ s.t.} \\ \langle p,p_{init},m,\mu a,e\rangle \in C}} tc_i(p_i(\langle p, p_{init}, m, \mu a, e\rangle)) \tag{2}$$

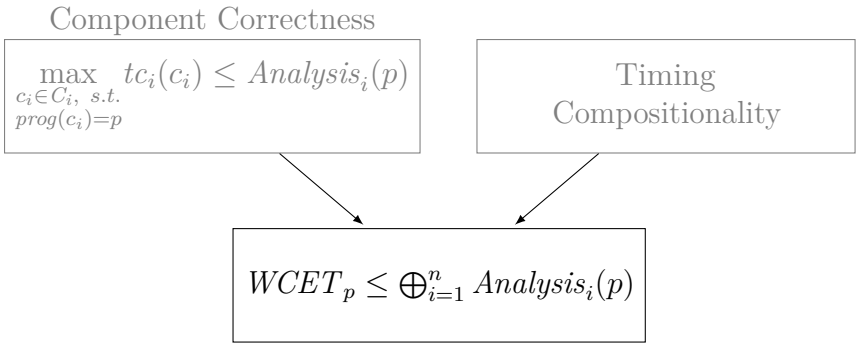


Figure 12: Compositional analysis framework. How to obtain globally sound results?

holds. Then, the sum of the analysis results soundly approximates the execution times of program p:

$$
\begin{aligned}
WCET_p &= \max_{\substack{m,\mu a,e \text{ s.t.} \\ \langle p,p_{init},m,\mu a,e\rangle \in C}} tc(\langle p, p_{init}, m, \mu a, e\rangle) && \text{Definition, Equation 1} \\
&= tc(\langle p, p_{init}, m', \mu a', e'\rangle) && \text{Definition max} \\
&\overset{(a)}{\leq} \sum_{i=1}^{n} tc_i(p_i(\langle p, p_{init}, m', \mu a', e'\rangle)) && \text{Compositionality} \\
&\overset{(b)}{\leq} \sum_{i=1}^{n} \max_{\langle p,p_{init},m,\mu a,e\rangle \in C} tc_i(p_i(\langle p, p_{init}, m, \mu a, e\rangle)) && \text{Definition max} \\
&\overset{(c)}{\leq} \sum_{i=1}^{n} A_i(\langle p, p_{init}\rangle) && \text{Analysis, Equation 2}
\end{aligned}
$$

We call the tuple $\langle p, p_{init}, m', \mu a', e'\rangle$ also worst-case configuration for the program p as this is the configuration that leads to the worst system's timing.

We have three inequalities that possibly lead to overestimation of the WCET by the analysis results. Inequality (a) is related to an overestimation introduced by timing compositionality. This overestimation might be bounded by $\langle \mu, \alpha \rangle$. It could be reduced by either choosing a different decomposition or by methods to obtain more precise timing contribution functions. We have equality in case of a fully-timing-compositional decomposition.

Inequality (b) is related to the analysis framework. The simple approach considers all timing contribution functions in isolation and thus allows for infeasible combinations of configurations. This overestimation can be reduced by different analysis frameworks, e.g. by eliminating infeasible combinations. This is tackled in the next

section by the Partitioning Approach. Note, that full timing compositionality does not forbid infeasible combinations.

Last but not least, inequality (c) models the pessimism of the individual analyses. In order to efficiently compute the required approximation (see Equation 2), analyses often employ abstractions that lead to a loss of precision. This overestimation is not bounded in general and could be reduced by the use of more sophisticated abstractions if available.

Refinement by Partitioning

In the Separate Maxima Approach, an analysis approximates the maximum timing contribution of an individual component independently of the other analyses. This allows for infeasible configuration combinations: There might be no system configuration c such that the projection $p_i(c)$ is the worst initial configuration for all tc_i.

In order to eliminate such infeasible configuration combinations, we use a partitioning approach. A partitioning of C is a set of disjoint sets (partitions) $\{\Pi_1, \ldots, \Pi_k\}$ such that $C = \bigcup_{i=1}^{k} \Pi_i$. The system configuration space is partitioned and the Separate Maxima Approach is used per partition to obtain a partition-specific timing bound. The overall result is the maximum partition-specific timing bound. This rules out infeasible configuration combinations *across* partition boundaries. Infeasible combinations are still possible within one partition.

As an example, consider two components A and B such that A/B behaves worst exclusively in c_A^{wc}/c_B^{wc} and there is no system configuration $c \in C$ with $p_A(c) = c_A^{wc}$ and $p_B(c) = c_B^{wc}$. Using the above approach with partitions $\Pi_1 = \{c \mid p_A(c) = c_A^{wc}\}$ and $\Pi_2 = C \setminus \Pi_1$ rules out the possibility that A and B behave worst together and thus improves the computed bound.

Now, we present the Partitioning Approach in more detail. Similar to the Separate Maxima Approach, let the decomposition of a system's timing behaviour $tc : C \to T$ into timing contributions $(tc_i : C_i \to T)_{i=1..n}$ with an additive combination operator be timing compositional. Let C, $(C_i)_{i=1..n}$ denote the respective configuration spaces as described in Section 4.1 with projection functions $(p_i : C \to C_i)_{i=1..n}$.

An individual analysis now computes a *partition-specific* timing bound. I.e., the analysis has to soundly approximate the execution times only when starting from a system configuration contained in the given partition. Therefore, the analyses require, besides knowledge about the executed program, additional knowledge about the partition. We call this additional knowledge the *characterisation* $\chi(\Pi_i)$ of a partition Π_i. The partition characterisation $\chi(\Pi_i)$ is passed as argument to the respective analyses. In the simplest case χ is the identity function – $\chi(\Pi_i) = \Pi_i$ – explicitly enumerating all system configurations contained in this partition. For

efficiency reasons, χ may provide abstract information about a partition instead of an explicit enumeration, e.g. "all system configurations such that memory block a is initially cached".

A (partition-specific) analysis A_i of the timing of component i is *sound*, if for each partition Π_l in a partitioning of C

$$A_i(\langle p, p_{init}, \chi(\Pi_l)\rangle) \geq \max_{\substack{m,\mu a,e \text{ s.t.} \\ \langle p,p_{init},m,\mu a,e\rangle \in \Pi_l}} tc_i(p_i(\langle p, p_{init}, m, \mu a, e\rangle))$$

holds. Plugging it all together, we soundly approximate the execution times of program p for a partitioning $\{\Pi_1, \ldots, \Pi_k\}$ of C.

$$\begin{aligned}
WCET_p &= \max_{\substack{m,\mu a,e \text{ s.t.} \\ \langle p,p_{init},m,\mu a,e\rangle \in C}} tc(\langle p, p_{init}, m, \mu a, e\rangle) && \text{Definition} \\
&= tc(\langle p, p_{init}, m', \mu a', e'\rangle) && \text{Definition max} \\
&\overset{(a)}{\leq} \sum_{i=1}^{n} tc_i(p_i(\langle p, p_{init}, m', \mu a', e'\rangle)) && \text{Compositionality} \\
&\overset{(b)}{\leq} \max_{\Pi_l \in \{\Pi_1 \ldots \Pi_k\}} \sum_{i=1}^{n} \max_{\langle p,p_{init},c\rangle \in \Pi_l} tc_i(p_i(\langle p, p_{init}, c\rangle)) && \text{Definition max} \\
&\overset{(c)}{\leq} \max_{\Pi_l \in \{\Pi_1 \ldots \Pi_k\}} \sum_{i=1}^{n} A_i(\langle p, p_{init}, \chi(\Pi_l)\rangle) && \text{Analysis} \\
&\leq \sum_{i=1}^{n} A_i(\langle p, p_{init}, \chi(C)\rangle) && \text{Separate Maxima}
\end{aligned}$$

In case there is only one partition $\Pi_1 = C$, we obtain the Separate Maxima Approach.

Still, we have three sources of uncertainty as already described in the previous section. The advantage of this approach is that the overestimation associated with Inequality (b) can be reduced compared to the Separate Maxima Approach by eliminating infeasible combinations. Inequality (b) holds, as $\langle p, p_{init}, m', \mu a', e'\rangle \in C$ and thus, according to the definition of the partitioning, $\exists l.\langle p, p_{init}, m', \mu a', e'\rangle \in \Pi_l$. The Partitioning Approach provably never yields worse results than the Separate Maxima Approach.

The partitioning allows to trade off precision of the results (maximum of sums) against efficiency in computing (sum of maxima). The more partitions, the more analysis runs are required (one analysis run per partition and component) while the precision potentially increases as infeasible configuration combinations are ruled out.

Trace Partitioning [29] is one possibility to choose an appropriate partitioning. Trace partitioning is an analysis technique that distinguishes different execution traces during analysis to prevent a loss of information at control-flow

joins in the program. E.g. it can be used to distinguish different paths following a conditional branch by partitioning the outcome of the branch condition, i.e. $\Pi_1 = \{c \mid \text{condition true in } c\}$ and $\Pi_2 = \{c \mid \text{condition false in } c\}$.

5.4 Practical Considerations

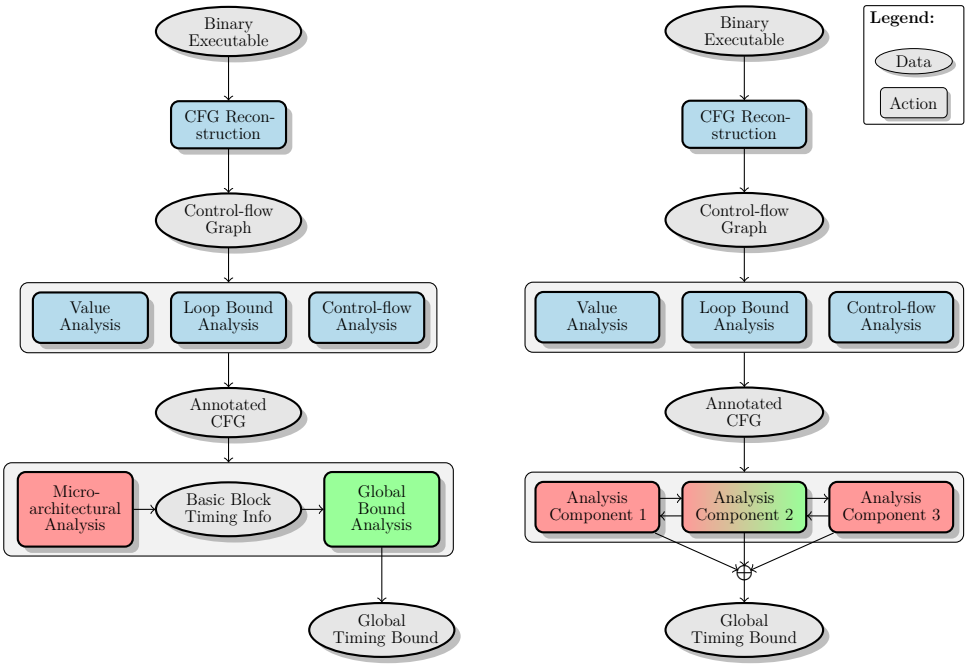
We conclude the section on timing compositionality in WCET analysis with a few practical considerations and the presentation of a new "compositional" WCET tool chain.

Information Sharing In our view, individual timing contributions share information by having configuration spaces that overlap. For example, most timing contributions that we considered throughout this article require knowledge about the executed instructions. Thus, all their configuration spaces contain information about the ISA configuration that can be used to reconstruct the sequence of instructions to execute. From a practical point of view, computing such information several times (once per timing contribution) is inefficient. Instead, *pre-processing* analyses can be employed to derive this information (e.g. the possible sequences of instructions) only once. Subsequently, their results are provided to each individual analysis that requires it. Such pre-processing analyses have already been used in the "traditional" WCET tool chain that has emerged over the years [11] [34] [19].

The individual analyses can also *cooperate* in order to avoid re-computation of the same information. Reconsider the multi-core example in Section 2.1. The first two analyses approximate the computation time and the number of accesses individually. The third analysis computes the worst-case blocking time that depends on the computation time and the number of accesses. Instead of re-computing this information on its own, the third analysis can use the results of the first two analyses.

Thus, using pre-processing analyses and cooperation between the individual analyses avoids unnecessary re-computation of information.

A new WCET Tool Chain Taking the ideas of the previous paragraph into account leads to a new, compositional WCET tool chain as depicted in Figure 13b. The frontend resembles the one of the "traditional" WCET tool chain that has developed over the years [11] [34] [19] and is depicted in Figure 13a. First, a control-flow-graph (CFG) representation is reconstructed from a given program binary. The control-flow analysis determines infeasible paths, thus restricting the possible sequences of instructions. The loop bound analysis determines upper bounds on the number of loop iterations, thus also restricting possible sequences of



(a) Traditional, Integrated Approach (b) Compositional Approach

Figure 13: Tool Chains for WCET Analysis. Blue indicates the pre-processing analyses. Red indicates the microarchitectural analyses. Green indicates the path analysis.

instructions. The value analysis approximates the contents of registers and memory cells for each program point.

After this pre-processing steps, we have the actual timing analysis – either integrated or compositional. In the traditional tool chain, the global bound analysis obtains a global timing bound given information about possible paths and the timing of individual basic blocks. The timing information of the individual basic blocks is computed using an integrated microarchitectural analysis. In the compositional tool chain, the global bound is obtained by combining the timing contributions from the individual compositional analyses. Note that part of the traditional global bound analysis, also known as path analysis, is now done within the individual analyses if necessary (cf. Analysis of Component 2 in Figure 13b).

6 Timing Compositionality by Design

So far, we were concerned with obtaining timing-compositional analyses for a given, fixed system. Depending on the behaviour of the system, it can be difficult to obtain compositional timing contribution functions (especially for small μ and α) or to find precise and efficient analyses. Designing hardware that *supports* timing compositionality, i.e. that allows for a compositional decomposition by construction, might be a solution. The challenge is to find hardware designs that support compositionality with low or even without performance degradation.

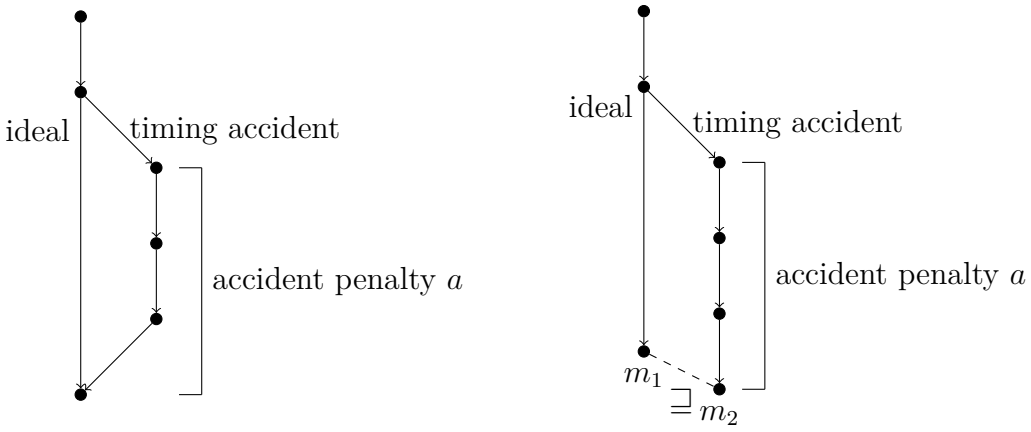
Whether a decomposition is compositional depends on the way hardware components are connected and interact with each other. As seen in Section 3.4, the behaviour within one component is not a concern as long as it does not influence the interaction with other components. Therefore, hardware designs that support compositionality has to focus on how hardware components are connected and interact. In the following, we present one microarchitectural design that supports timing compositionality.

6.1 Stall on Timing Accidents

We consider a decomposition into an *idealised execution* and *timing accidents*. A timing accident is an event that has a detrimental effect on the overall system's timing, e.g. a cache miss, a write back to main memory or a DRAM refresh. The idealised execution is the execution without any timing accidents.

A simple solution to achieve timing compositionality by design is to *stall* the execution until the timing accident is rectified, e.g. until a requested cache line is fetched from main memory. Stalling means that all parts of the system stop execution and thus keep their microarchitectural states – except those necessary for rectifying the timing accident. We get the execution behaviour as depicted in Figure 14a. In case of a timing accident, its rectification takes a cycles and afterwards the execution reaches the same microarchitectural state as in the ideal case – due to stalling. Therefore, the penalty a only needs to capture the *direct* effect of the timing accident on the overall execution time, e.g. the time needed to transfer a cache line from main memory into the cache. In contrast to modern complex processors, there are no *indirect* timing effects, e.g. a cache miss additionally triggering an expensive misspeculation. We can employ the analysis presented in Section 5.2.

This approach can be seen as a co-design of microarchitecture and timing contribution functions. First, we want to add a new (performance-enhancing) feature that speeds up execution in some cases and possibly has detrimental effects in other cases (timing accident). Second, we choose appropriate timing



(a) Stalling supports timing compositionality.

(b) Relaxation: Overlapping effects are allowed if microarchitectural state m_1 leads to an at least as high execution time ($\sqsupseteq$) than m_2.

Figure 14: Achieving timing compositionality by design. The circles denote microarchitectural states, the edges denote one processor cycle of execution.

contributions: the speed-up effects are captured by the idealised execution; the direct effects of the timing accidents are captured separately by counting "accident" events. Third, the stalling mechanism ensures timing compositionality of the new decomposition.

The stalling mechanism is not appropriate for all kinds of timing accidents, e.g. misspeculation. In case misspeculation is detected, the microarchitectural state has already changed (e.g. cache contents changed due to speculative prefetching). Reaching the same state as in case of no speculation/correct speculation is more difficult and cannot be achieved by stalling.

6.2 Relaxation: Monotonicity

While the "simple" stalling mechanism presented above supports timing compositionality, it has the disadvantage that actual system performance is degraded. E.g. in case of a cache miss, no arithmetic instruction executes in parallel and thus cannot hide the main memory latency. Such overlapping effects are not problematic for timing compositionality in general. Consider the situation depicted in Figure 14b. As long as the microarchitectural state m_1 in the ideal case always leads to a higher remaining execution time compared to state m_2 arising in the "accident" case, timing compositionality is not affected. We call this property *monotonicity*.

Sound analyses designed for the stalling case (Figure 14a) remain sound in the monotonic setting (Figure 14b) and give the same provable bounds. However, the actual system performance might profit from this relaxation as overlapping effects are allowed. Therefore, in the monotonic case, the decomposition might lead to overestimation in terms of $\mu \geq 1$.

6.3 Summary

We presented a stalling mechanism as an example of designs that support timing compositionality. However, this requires hardware changes that might be impractical due to technical/economical constraints. If used extensively, the stalling mechanism degrades the actual system performance. Similar to the trade-off between integrated and compositional analyses in Section 4.5, application of the stalling mechanism has to be carefully chosen.

7 Related Work

In Section 2, we described approaches that use timing compositionality in order to decouple analyses of different parts of a system. The topics include the analysis of

- the bus blocking time in resource-sharing systems with shared bus [31, 32],
- the cache-related preemption delay in preemptively scheduled systems [6, 5], and
- the refreshes in a DRAM system [7].

Wilhelm et al. [37] introduce the notion of timing compositional architecture. We discussed problems and limitations of this definition in detail in Section 3.4.

In [30], Schliecker et al. present their approach to performance analysis of real-time multiprocessor systems with resource sharing. They assume timing compositionality "in the sense that any shared resource delays are additive to the execution times". Such additive behaviour may be achieved by the processor stalling execution on accesses to shared resources (see also Section 6).

In [22], Liu et al. present a precision timed (PRET) architecture for timing predictability and timing composability. The timing composability enables a modular verification of systems with concurrent programs. The microarchitectural design includes a thread-interleaved pipeline, scratchpad memories, and a specialised, predictable DRAM controller. The design also allows for *compositional* decompositions e.g. of a thread's execution time into computation time and memory access time. On memory access, the thread continues execution only after the memory access has completed, thereby clearly separating computation and memory access time of the thread. Furthermore, their predictable DRAM controller allows for precise bounds on the latency of a memory access independently of the execution context.

Goossens et al. [14] as well as Akesson et al. [3] give an overview of how to achieve timing composability in a system-on-chip setting. In [14], the authors survey their previous work on the CompSOC architecture that provides temporal isolation between applications by, e.g., employing time-division multiplexing (TDM) techniques. Besides techniques to achieve composability, the authors of [3] additionally discuss that composability and predictability (i.e. the ability to determine precise performance bounds) are orthogonal properties. This discussion partially resembles our discussion in Section 3.3.

The increasing complexity in real-time software makes composable and compositional methods beneficial to efficiently reason about its timing [26]. Puschner et al. introduce composability of execution times and I/O-compositionality of worst-case execution times (WCETs) and discuss ways to achieve these properties. The timing of a task executed on a processor must not be affected by co-running tasks

(composability). The WCET of sequentially executed tasks should be the sum of the WCETs of each task (I/O-compositionality). Their notion of compositionality is more restrictive than the definition we give in this paper.

In [21], Lee et al. tackle the scalability problems of multiprocessor simulation for performance estimation. They propose the so-called composable performance regression that splits multiprocessor simulation as follows. First, a uniprocessor model estimates the baseline performance assuming no interference from other cores. Second, a contention model is used to capture the interference effects (e.g. due to memory accesses) caused by co-running cores. Third, a penalty model combines the result of the two previous models to estimate the multiprocessor performance. The models are obtained using regressions on a set of training data.

Compositionality as a property can be found in diverse contexts. In linguistics, the principle of compositionality (or Frege's Principle) as described by Partee et al. in [25] is characterised as follows:

> *The meaning of a complex expression is determined by the meanings of its constituent expressions and rules to combine meanings.*

In (functional) programming languages, the evaluation of expressions follows this compositionality principle. First, the respective subexpressions are evaluated and the results are combined according to the employed operator to obtain the overall result.

In the context of the verification of concurrent programs, de Roever et al. [10] discuss compositional and non-compositional proof methods. Compositionality in this context of program correctness is defined as follows:

> *"That a program meets its specification should be verified on the basis of specifications of its constituent components only, without additional need for information about the interior construction of those components."*[10]

The authors discuss the respective advantages and shortcomings of compositional as well as non-compositional proof techniques. They indicate possible scalability problems of non-compositional verification approaches, but they also point out that such approaches can be more successful, e.g. in "tightly-coupled [...] systems" ([10, page 60]) such as mutual exclusion algorithms based on shared-variable communication. Note, that we encountered a similar trade-off between compositional and non-compositional methods in the context of execution time analysis in Section 4.5.

8 Conclusions and Future Work

Modern hard real-time systems tend to become more complex thereby complicating the derivation of guarantees on the systems' timeliness. E.g. the increasing microarchitectural complexity of the employed processors will render current state-of-the-art, non-compositional approaches to execution time analysis infeasible in terms of computational effort and memory consumption.

This trend makes it necessary to move from these non-compositional approaches towards compositional methods as done e.g. in [31] for resource-sharing systems or [5] for preemptively scheduled systems. This paper presents a formal definition of *timing compositionality* that is based on the previous, intuitive understandings. The definition will serve as a foundation for correctness proofs of compositional analyses. We have discussed the definition in detail and contrasted it with the definition of *timing compositional architectures* [37].

The second part of the paper addresses compositionality in the context of execution time analysis. We have discussed the advantages and shortcomings of both the integrated and the compositional approach to timing analysis. Possible applications of timing compositionality in timing analysis have been highlighted, e.g. the analysis of write-back caches. A conceptual approach to the derivation of timing contribution functions based on the accountability of components has been presented. We have described a generic compositional analysis based on accident counting as well as methods to soundly combine the individual results to an overall timing bound. Furthermore, we have presented a simple microarchitectural design to achieve timing compositionality by design in specific scenarios.

Future Work One line of future research is to come up with sufficient and practically testable conditions for timing compositionality of a decomposition. This involves concrete methods to compute sound penalties of an individual timing accident based on a formal specification of the underlying system, e.g. given as VHDL/Verilog models.

The proposed stalling mechanism to achieve compositionality by design will, in general, degrade the system's performance as overlapping effects are eliminated. More sophisticated design guidelines are required that achieve compositionality and show good performance. The estimation of the respective degradation of provable as well as actual system performance is considered future work.

References

[1] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, August 2013. On pages 2 and 13.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. On pages 8 and 19.

[3] Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, chapter 2, pages 25–56. Springer, November 2010. On pages 13 and 42.

[4] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, volume 1145 of *LNCS*. Springer Verlag, 1996. On pages 1 and 20.

[5] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In Robert I. Davis and Nathan Fisher, editors, *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 261–271, December 2011. On pages 2, 5, 6, 24, 42, and 44.

[6] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: Tightening the CRPD bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 153–162, New York, NY, USA, April 2010. ACM. On pages 5, 24, and 42.

[7] Pavel Atanassov and Peter Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *Proceedings of the IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, December 2001. On pages 7 and 42.

[8] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and*

Implementation, PLDI '01, pages 286–297, New York, NY, USA, 2001. ACM. On page 26.

[9] Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013. On page 25.

[10] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, USA, 2001. On page 43.

[11] Andreas Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, Uppsala University, 2003. On page 36.

[12] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *In Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, 1997. On page 26.

[13] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers `http://www.absint.com/ait/`, October 2014. On page 1.

[14] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. Virtual execution platforms for mixed-time-criticality applications: the CompSOC architecture and design flow. In *Proceedings of the 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pages 23–30, December 2012. On pages 13 and 42.

[15] Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012. On page 21.

[16] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *Proceedings of the 2012 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64, April 2012. On page 26.

[17] 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*, pages 296–301, March 2013. On page 26.

[18] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis – definition and challenges. In *Proceedings of the 6th Workshop on Compositional Theory and Technology for Real-Time Embedded System*, December 2013. On pages iv, v, and 2.

[19] Christopher A. Healy, , David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, 1995. On page 36.

[20] 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, July 2003. On pages 1, 18, and 19.

[21] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 270–281. IEEE Computer Society, 2008. On page 43.

[22] Isaac Liu, Jan Reineke, and Edward A. Lee. A PRET architecture supporting concurrent programs with composable timing properties. In *Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 2111–2115. IEEE, 2010. On page 42.

[23] Kartik Nagar. Cache analysis for multi-level data caches. Master's thesis, Indian Institute of Science, June 2012. On page 25.

[24] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005. On page 22.

[25] Barbara H. Partee, Alice ter Meulen, and Robert E. Wall. *Mathematical Methods in Linguistics*. Kluwer, Dordrecht, 1990. On page 43.

[26] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time programs. In *Software Technologies for Future Dependable Distributed Systems*, pages 1–5. IEEE, 2009. On page 42.

[27] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 51–60, New York, NY, USA, June 2008. ACM. On page 26.

[28] Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009. On pages 13 and 14.

[29] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007. On page 35.

[30] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, January 2011. On page 42.

[31] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, April 2010. On pages 2, 4, 24, 42, and 44.

[32] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–222, April 2011. On pages 2, 4, 24, and 42.

[33] Lili Tan. The worst case execution time tool challenge 2006: The external test. In Tiziana Margaris, Anna Philippou, and Bernhard Steffen, editors, *2nd International Symposium on Leveraging Applications of Formal Methods)*, Paphos, Cyprus, November 2006. On page 1.

[34] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3), May 2000. On page 36.

[35] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004. On page 20.

[36] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter 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 (TECS)*, 7(3), 2008. On page 1.

[37] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009. On pages 2, 4, 13, 14, 15, 42, and 44.