Towards Compositionality in Execution Time Analysis – Definition and Challenges

Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm
Saarland University, Saarbrücken, Germany

***Abstract*—For hard real-time systems, timeliness of operations has to be guaranteed. Static timing analysis is therefore employed to compute upper bounds on the execution times of a program. Analysis results at high precision are required to avoid over-provisioning of resources. For current processors, timing analysis is a complex task mainly due to interdependencies of the processors' features that affect the overall timing behaviour. To still obtain tight bounds, state-of-the-art approaches collect detailed information about these interdependencies by exploring the state space of the system as a whole.**

Modern systems, such as multi-core processors, introduce even more timing dependencies – e.g. due to interference on shared resources between functionally independent programs running on different cores. This will eventually render the above, non-compositional, approach infeasible in terms of analysis runtime and memory consumption. Therefore, recent analysis approaches often assume a certain independence of system components – referred to as *timing compositionality*.

We aim at a formal definition of timing compositionality as it was previously only introduced informally. How to achieve timing compositionality *in general* is an unsolved question. We highlight challenges and summarise open problems that arise in the context of compositional analyses.

I. Introduction

In general-purpose computing, the fast execution of programs in *most cases* is a *desirable* property. In safety-critical, hard real-time (embedded) systems, program execution on time in *all cases* is strictly *required* [1]. Therefore, static analysis methods are employed to derive guarantees on 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 its underlying hardware platform. A memory access during program execution, e.g., can be served in 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 processors' features are also highly interdependent [2] – 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 content and thereby its 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 system 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 system states, abstractions are employed. Efficient and sufficiently precise abstractions have been found for some isolated features, e.g. caches [3], 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[1] by AbsInt GmbH, is successfully applied for programs that execute uninterruptedly and in isolation – even on complex processors [4]. 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 content 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 originally independent programs are introduced [5]. 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 on (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 [6] [7] as well as

[1] http://www.absint.com/ait/

analyses for preemptively scheduled systems [8]. The inherent, underlying assumption is that the system allows for such a decoupling of analyses. This assumption is referred to as *timing compositionality*.

Up to now, timing compositionality is a term whose meaning is solely based on intuition without a rigorous, formal definition. A first attempt towards a definition has been made in [9] and is discussed in Section III-D.

A. Our Contributions

Our contributions are threefold. First, we examine existing approaches that assume "timing compositionality" with respect to their, often intuitive, understanding of compositionality. Second, based on our findings, we present a new, unified, and thorough formal definition of timing compositionality. Thereby, we want to soundly replace the analysis of a whole system by a combination of individual analyses that focus only on selected features. We discuss *timing compositional architectures* as introduced by [9] and highlight the differences to our definition of compositionality.

How to achieve timing compositionality in general is an unsolved question. Is it possible to construct compositional analyses even for complex processors, as well as to design hardware (features) that allow for precise and compositional analyses by construction? We identify challenges and pose open problems that are subject to future work.

B. Overview

We start with a discussion of compositionality assumptions in the literature in Section II and identify the main ingredients for our formal definition. Our definition is then given in Section III and discussed in detail. In particular, we distinguish our definition from the previous definition of timing compositional architectures [9]. Further related work is described in Section V. In Section IV, we present a summary of challenges and open problems. We conclude in Section VI.

II. 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 sound analysis results for the rest of the system. We give several examples where timing compositionality is assumed or required.

A. Resource-Sharing Systems

Schranzhofer et al. [6] [7] 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 [6] or adaptive arbitration [7]).

Timing compositionality enables the decoupled analysis of the timing contributions of selected features and allows

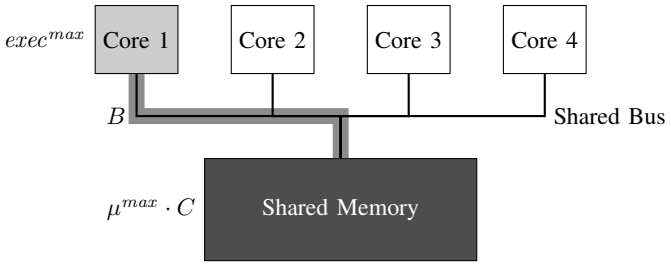

Figure 1. Multi-core system with shared bus and memory. Compositional view of the worst-case response time of a task running on Core 1.

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 maximize the blocking time B – e.g. given the static TDMA schedule and the initial offset (starting time). For other arbitration policies such as Round-Robin, the computation of B might also depend 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

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

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

The authors explicitly assume a *fully timing compositional architecture* in the sense of [9], i.e. an architecture without timing anomalies. 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. For an in-depth discussion, see Section III-D. 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 (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.

B. Preemptively Scheduled Systems

Altmeyer et al. [8] 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 [8] and [10], 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

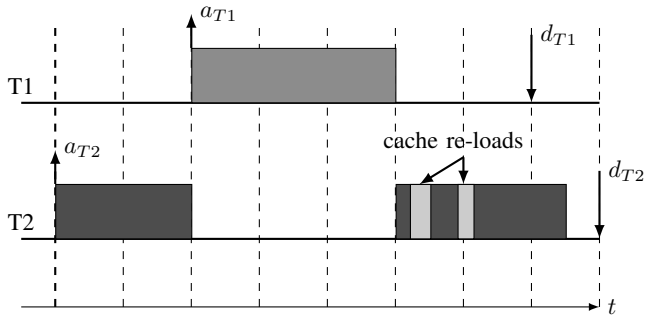

Figure 2. Compositional view on preemptively scheduled systems: Task 2 is preempted by task 1. Task 1 evicted useful cache content resulting in additional cache re-loads. a_{Ti} arrival time of task i, d_{Ti} deadline of task i.

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 might contribute 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 size of the program (so-called domino effects). Note, that chain reactions 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 some cases.

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 [8]. 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.

C. Dynamic Random Access Memory (DRAM)

The use of DRAM (compared to static SRAM) complicates timing analysis as the behaviour of the DRAM controller has to be additionally modeled. One complication arises from DRAM refreshes that prolong ongoing memory accesses and appear (in general) asynchronously of program execution. A possibility to account for these DRAM refreshes in timing analysis is described by Atanassov et al. [11]. Let t be a bound on the worst-case execution time (WCET) assuming no DRAM refreshes, n the maximum number of refreshes occurring during program execution and p the maximal delay caused by a refresh. They claim that the WCET with enabled refreshes is bounded by $t+n \cdot p$. Thus, they implicitly assume timing compositionality because the penalty due to DRAM refreshes and the WCET are independently computed and then summed up to obtain a valid bound.

D. 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. Next, the timing of the system is *decomposed* into several *timing contributions* that capture a part of the system's timing behaviour – e.g. the execution time without bus accesses, the resource access time and the bus blocking time. The individual timing results are then *combined* to a *sound* upper bound on the system's timing.

Each timing contribution is associated with a *component* of the system – e.g. a processor core, a shared resource such as a bus (Section II-A). The term component (and system) might refer to a hardware component, but also to a software component such as an individual task (Section II-B). It is general and can thus capture timing decompositions at different levels, such as the actual hardware level as well as the software level.

III. Timing Compositionality

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

A. Formal Definition – The Basics

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 timings at a finer level (e.g. in case of multiple clock domains).

System and Components: As already stated in the previous sections, we consider a system together with a set of components where the system's timing will later be decomposed into timing contributions of components. Note, the generality of the term system and component. Depending on the intended application, it might correspond to hardware units in a processor (e.g. caches, pipelines or buses) or to tasks on the software level. A component can again be seen as a system whose timing could be further decomposed – vice versa, a system can be seen as a component of a larger system. We denote the component under consideration, the *system*, by C, and its i-th component by C_i.

The behaviour of a system/component is dependent on its current *state*. E.g. the execution of a program depends on the

program itself, the current program counter, the values of its input variables, the microarchitectural state and the state of the environment (such as co-running programs on a multicore). The current state thereby determines the subsequent behaviour of a system/component. Associated to system C, S denotes the set comprising the states C can be in (analogously S_i for component C_i).

The states of individual components are not necessarily independent, they can *overlap*, i.e. the can share common information. As an example consider a concrete processor system with a cache and a pipeline component: both timing contributions depend on the instructions to execute. Necessarily, they share common information about the current state of the program, and so their states overlap.

Timing Behaviour: Next, we need a notion of timing behaviour of a system/component. The timing behaviour of a system/component is *state-dependent*, e.g. the execution time of a program varies depending on whether a memory access hits or misses the cache. The timing behaviour of a system C captures the time needed to reach a *final* state starting in a given state s. Final means e.g. the termination of the program implicitly given by s, or the point in time where all tasks were successfully scheduled and executed. For a component i, we are interested in the *timing contribution* of component i to the overall system's timing. As an example, consider the timing contribution of the shared bus in terms of additional blocking time (Section II-A) or the contribution of additional cache reloads due to preemption (Section II-B).

To capture these timing contributions, we employ state-dependent functions as described below.

Definition 1. *Let C be a component (or a system) with associated state space S. Furthermore let T denote the set of possible timings. We call a function $tc : S \to T$* timing contribution *of C.*

The definition is quite general and the precise meaning of a timing contribution function depends on its intended application. How to derive these state-dependent timing contribution functions systematically is challenging (Section IV). For one system/component, there exist different functions that conservatively capture, i.e. over-approximate, the respective timing contribution.

The timing contribution of a system reflects the system's overall timing.

Decomposition of Timing Behaviours: Before defining timing compositionality, we need a notion of decomposition of a system's timing (contribution) into the timing contributions of its components. As the timing contributions are state-dependent, the decomposition also relates states of the system with the corresponding states of each component. Furthermore, the decomposition provides a combination function that combines the timings of the individual components. As examples for a decomposition, refer to Section II-A and Section II-B.

Definition 2. *Let C be a system with state space S, and $(C_i)_{i=1..n}$ components with associated component state spaces $(S_i)_{i=1..n}$. Furthermore, let $tc : S \to T$ and $(tc_i : S_i \to T)_{i=1..n}$ be the timing contributions of the system and its components, respectively. We call $(tc_i)_{i=1..n}$ together with a family of functions $(a_i : S \to S_i)_{i=1..n}$ and a combination function $\bigoplus : T^n \to T$, a* decomposition *of tc.*

Each function $a_i : S \to S_i$ maps a system state to a corresponding component state. It can be seen as an abstraction function as it only keeps the state relevant for component C_i, based on the state of the system. In some cases, the abstraction might simplify to a projection (if the system state is a tuple of component states), but it can be more complex. We write s_i instead of $a_i(s)$ for short to denote the state relevant for component C_i.

The combination function captures the *type of composition*. In the examples presented in Section II, the combination is given by the addition operator – the individual timing contributions are added up to obtain an overall timing bound. The combination function might e.g. compute the maximum of the components' individual timings in case the components execute in parallel and independently from each other. In general, the combination function can be more complex and is solely determined by the chosen decomposition.

For one given system, there might exist multiple decompositions of the system's timing into different components' timing contribution functions – associated with different abstraction functions and a different combination operator, respectively.

B. Definition

Now, we give the definition of timing compositionality.

Definition 3. *Let C be a system and $(C_i)_{i=1..n}$ its components with associated state spaces S and $(S_i)_{i=1..n}$. Furthermore, let the timing contributions $(tc_i : S_i \to T)_{i=1..n}$ together with state abstraction functions $(a_i : S \to S_i)_{i=1..n}$ and combination operator $\bigoplus : T^n \to T$ be a decomposition of the system's timing $tc : S \to T$. We call the decomposition* timing compositional *if and only if*

$$\forall s \in S.\ tc(s) \leq \bigoplus_{i=1}^{n} tc_i(a_i(s)).$$

Timing compositionality is not a property of the system under consideration but of the *specific decomposition*. It states that the contribution of individual components to the overall system's timing can be considered separately. Which components can be considered separately depends on the chosen decomposition.

Besides the aspect of *separating* the overall system's timing into timing contributions of components, Definition 3 comprises a view on the *"complexity"* of the individual timing contributions. In case the state abstraction function a_i is the identity function ($a_i(s) = s$), the timing contribution tc_i uses whole system states to capture the timing of C_i. It has the same information need as the timing contribution tc of the original system C and is thus similarly complex. But the goal of timing compositionality is to reduce the complexity of analysing the whole system by employing less complex component analyses. Therefore, the abstraction functions capture the information need of the timing contribution functions and thereby of the component analyses. In general, the smaller the component states $a_i(s)$ and the smaller the overlap between

component states $a_i(s)$ and $a_j(s)$, the better for the complexity of individual component analyses.

Note that the definition also captures nesting, i.e. that the definition is again applicable to any constituent component C_i (if considered as system itself) and so on. Thus compositionality can be employed at different levels within a system.

As already stated, timing compositionality is a property that *always* depends on a chosen decomposition. There exist trivial decompositions (e.g. $n = 1$, $S_1 = C$ 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. Whether a decomposition is significant, depends inherently on the intended application.

So far, timing compositionality (Definition 3) 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 component 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 overestimated by a lot for other decompositions. Therefore, we now refine our definition of timing compositionality by introducing a notion of precision.

Definition 4. *Let C be a system and $(C_i)_{i=1..n}$ its components with associated state spaces S and $(S_i)_{i=1..n}$. Furthermore, let the timing contributions $(tc_i : S_i \to T)_{i=1..n}$ together with state abstraction functions $(a_i : S \to S_i)_{i=1..n}$ and combination operator $\bigoplus : T^n \to T$ be a decomposition of the system's timing $tc : S \to T$. We call the decomposition* (μ, α)-timing compositional *where $\mu \in \mathbb{R}_{\geq 1}$, $\alpha \in \mathbb{R}_0^+$ if and only if*

$$\forall s \in S.\ tc(s) \leq \bigoplus_{i=1}^{n} tc_i(a_i(s)) \leq \mu \cdot tc(s) + \alpha.$$

The additional inequality restricts the Definition 3 of timing compositionality because μ and α are finite constants. While the original definition is a boolean property, this refined definition offers several shades of timing compositionality by introducing the concept of precision. If a decomposition is (μ, α)-timing compositional, we have an upper bound on the overestimation of the combined components' timings compared to the system's timing. The values μ and α are thus a measure of *how precise* the results for a specific decomposition are at least – 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 generally interested in the minimal μ and α such that compositionality still holds. Furthermore, a decomposition (together with timing contribution functions) that permits small constants μ and α is preferable.

Next, we introduce some specific notions of timing compositionality based on Definition 4. Consider a (μ, α)-timing compositional decomposition of a component. We call the decomposition

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

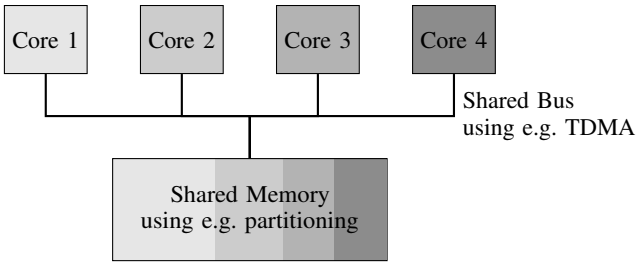

Figure 3. Multi-core system with shared resources: Composability by temporal isolation.

In case of a fully timing compositional decomposition, the two inequalities in Definition 4 imply equality,

$$\forall s \in S.\ tc(s) = \bigoplus_{i=1}^{n} tc_i(a_i(s)).$$

C. Compositionality and Composability

Timing compositionality and timing composability are two orthogonal properties; both with applications in timing analysis and often mixed up.

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 the timing contributions of its constituent components and the type of composition. This enables a modular view on the system's timing behaviour as described by examples in Section II.

In contrast, timing composability is a property of an *individual component*'s timing behaviour within a larger system. It states that the timing behaviour of the component is independent of the behaviour of the other components, and can thus be analysed in isolation.

Timing compositionality of a decomposition does not imply timing composability of its constituent components, and vice versa. Next, we present two examples to highlight that compositionality and composability are *orthogonal*.

Compositionality: As an example, consider the multi-core setting in Figure 1 together with the analysis presented in Section II-A. The response time of the program running on core 1 is decomposed into its computation time, the memory access time and the bus blocking time. Given correct timing contribution functions, this decomposition is *compositional*. However, the calculation of the bus blocking time needs information about the computation time and the number of accesses (the other components) – thus it is not *composable*.

Composability: Consider the multi-core system in Figure 3 with the cores as components. 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: 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 possible interferences in multicores as well as techniques to achieve temporal isolation is given in [5]. Akesson et al. [12] and Goossens et al. [13] provide an overview on 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 other components. 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 interfering accesses.

Interplay between Compositionality and Composability: However, there is an interplay between compositionality and composability. Consider the two above 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 a multi-core system (Figure 3). In case the cores operate in a composable fashion, the bus blocking time only depends on the behaviour of the core under consideration, not on the behaviour of the other three cores. Thus, the computation of the bus blocking time can be (tremendously) simplified.

D. Timing Compositional Architectures

Previously, there have been attempts towards defining a notion of compositionality. In [9], 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 wrong local choice 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 [14]. Fully timing compositional architectures are then defined as architectures whose abstract model does not exhibit timing anomalies (and 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 component timing contributions. In contrast to the definition of timing compositional architectures [9], our definition does not forbid arbitrarily complex timing behaviours within one component. 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 pipeline and cache component. The analysis of the pipeline component might have to take timing anomalies into account due to dynamic scheduling effects caused by the out-of-order execution – so-called scheduling anomalies (see Figure 4, [14]). However, the decomposition pipeline-cache can be timing compositional, e.g. in case the pipeline stops execution while servicing a cache miss from main memory. Timing compositionality is thus *unaffected* by the anomalous behaviour *within* one component.

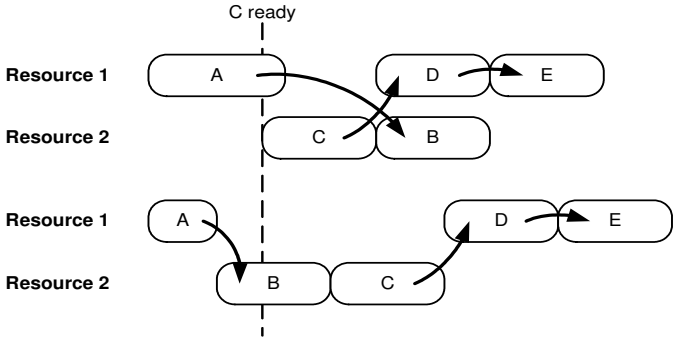

Figure 4. Scheduling anomaly [15]: Shorter execution of instruction A leads to longer overall execution. Timing compositionality is not necessarily affected by anomalous behaviour within the components.

Furthermore, *the* abstract model of an architecture is not uniquely defined. As already stated in Section III-A and Section III-B, there might exist several decompositions of a system's timing; and multiple different timing contribution functions are possible for each system/component. 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.

E. Summary

We started by introducing basic concepts such as 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 II. A decomposition is called *timing compositional* if and only if the combination of the timings of individual components 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 of a system's decomposition and we described their respective applications in timing analysis.

Finally, we discussed a previous definition of timing compositional architectures [9]. We sketched the issues that arise from this definition and we highlighted the differences with respect to our definition of timing compositionality.

IV. Challenges and Open Problems

In the remainder of the paper, we discuss challenges and open problems in the context of compositionality applied to execution time analysis.

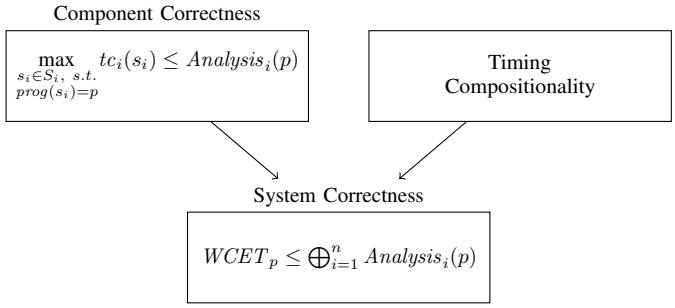

Figure 5. Overall structure of compositional analyses.

A. Achieving Timing Compositionality

Having timing compositionality formally defined, the challenge to *achieve* compositionality remains. We identify the following two tasks:

1) finding compositional analyses for a *given* system, e.g. a complex processor, and
2) designing *new* (hardware) systems that "support" timing compositionality.

Analyses: Consider the compositional analysis framework in Figure 5 that relates analyses and compositionality. Component analyses are always based on the timing contribution functions that describe the timing behaviour of the respective component. First, these functions have to be derived – potentially automatically – from a formal system description. Second, analyses have to be designed that soundly approximate these timing contribution functions.

For *existing* analyses such as presented in Section II, the challenge is to check whether the (compositional) timing contribution functions for a given platform are correctly approximated. This might include the computation of sound penalties, such as the block reload time (BRT), for specific processor systems. In the ideal case, this computation is done automatically using a formal model of the processor, e.g. provided as Verilog/VHDL code.

Another challenge is to design *new* analyses that make use of compositionality to precisely and efficiently analyse complex processor systems. Potential applications of compositionality include the following scenarios:

- *Write-back 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. Uncertainty as to when the memory operation happens, renders the integrated, non-compositional analysis approach infeasible in terms of complexity. A 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.
- *Shared Caches in Multicores*. The execution of a program p using a shared cache experiences interference: Co-running programs might evict cache lines that were still useful for p. If at all, bounding the amount of interference in a cumulative way rather than locally classifying memory accesses as hit/miss seems more likely feasible in terms of precision and efficiency.

Hardware Design: Finding a suitable timing compositional decomposition can, in general, be difficult. 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 how components are connected and how they interact with each other. As seen in Section III-D, the behaviour within one component is not a concern as long as it does not influence the interaction with other components. Therefore, hardware design that supports compositionality has to focus on how hardware components are connected and interact.

B. Compositional vs. Non-Compositional Analyses

There is a trade-off concerning when to use compositional or non-compositional analyses. Tightly-coupled systems (e.g. out-of-order pipeline with several functional units acting in parallel) favour non-compositional analyses in order to precisely capture timing effects at a fine-grained level. However, with increasing complexity of the systems (e.g. multi-core processors), the non-compositional approach becomes infeasible in terms of computational effort and memory consumption.

In such situations, compositional analyses are inevitable due to efficiency. But compositional analyses can also be profitable in terms of precision, e.g. when only cumulative information is precise (for write-back caches or shared caches in multicores). Therefore, it is worth to investigate the gain/loss in precision/efficiency when compositional methods are employed, e.g. in the scenarios mentioned above.

V. Related Work

In Section II, 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 [6] [7],
- the cache-related preemption delay in preemptively scheduled systems [10] [8], and
- the refreshes in a DRAM system [11].

In [16], 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.

Wilhelm et al. [9] introduce the notion of timing compositional architecture. We discussed problems and limitations of this definition in detail in Section III-D.

In [17], 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. [13] as well as Akesson et al. [12] give an overview on how to achieve timing composability in a system-on-chip setting. In [13], 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 [12] 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 III-C.

The increasing complexity in real-time software makes composable and compositional methods necessary to efficiently reason about its timing [18]. 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). These notions are more restrictive than the definitions we give in this paper.

In [19], 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 by using regressions on a set of training data.

VI. Conclusions and Future Work

With the increasing complexity of processors, state-of-the-art approaches to execution time analysis become more and more problematic 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 [6] for resource-sharing systems or [8] for preemptively scheduled systems. This paper contributes a formal definition of *timing compositionality* that is based on the previous, intuitive understandings. The definition may 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* [9]. Furthermore, we have presented challenges and open problems in the context of compositional execution time analysis, which we consider future work.

Acknowledgements

We thank the anonymous reviewers 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.

References

[1] R. Wilhelm *et al.*, "The worst-case execution time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[2] R. Heckmann *et al.*, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, July 2003.

[3] M. Alt *et al.*, "Cache behavior prediction by abstract interpretation," in *Proceedings of SAS'96, Static Analysis Symposium*, ser. LNCS, vol. 1145. Springer Verlag, 1996.

[4] L. Tan, "The worst case execution time tool challenge 2006: The external test," in *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods)*, November 2006.

[5] A. Abel *et al.*, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR*, August 2013.

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

[7] ——, "Timing analysis for resource access interference on adaptive resource arbiters," in *RTAS*, April 2011, pp. 213–222.

[8] S. Altmeyer *et al.*, "Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, December 2011, pp. 261–271.

[9] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, July 2009.

[10] S. Altmeyer *et al.*, "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*. ACM, April 2010, pp. 153–162.

[11] P. Atanassov and P. 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*, December 2001, pp. 29–34.

[12] B. Akesson *et al.*, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer, November 2010, ch. 2, pp. 25–56.

[13] K. Goossens *et al.*, "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*, December 2012, pp. 23–30.

[14] J. Reineke and R. 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.

[15] J. Reineke *et al.*, "A definition and classification of timing anomalies," in *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[16] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 22:1–22:27, January 2011.

[17] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *ASILOMAR*. IEEE, 2010, pp. 2111–2115.

[18] P. Puschner, R. Kirner, and R. G. Pettit, "Towards composable timing for real-time programs," in *Software Technologies for Future Dependable Distributed Systems*. IEEE, 2009, pp. 1–5.

[19] B. C. Lee *et al.*, "CPR: Composable performance regression for scalable multiprocessor models," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. IEEE Computer Society, 2008, pp. 270–281.