Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core

Sebastian Hahn and Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{sebastian.hahn, reineke}@cs.uni-saarland.de

***Abstract*—We introduce the strictly in-order core (SIC), a timing-predictable pipelined processor core. SIC is provably timing compositional and free of timing anomalies. This enables precise and efficient worst-case execution time (WCET) and multi-core timing analysis.**

SIC's key underlying property is the monotonicity of its transition relation w.r.t. a natural partial order on its microarchitectural states. This monotonicity is achieved by carefully eliminating some of the dependencies between consecutive instructions from a standard in-order pipeline design.

SIC preserves most of the benefits of pipelining: it is only about 6-7% slower than a conventional pipelined processor. Its timing predictability enables orders-of-magnitude faster WCET and multi-core timing analysis than conventional designs.

I. Introduction

One of the main challenges for timing analysis is the dependence of the execution time on the state of the underlying hardware platform. Even simple single-core processors feature stateful performance-enhancing mechanisms such as pipelining and caches. In the presence of such stateful mechanisms an individual instruction's execution time may vary widely depending on the state of the hardware when the instruction is executed. For example, a cache miss usually takes significantly longer than a cache hit.

Simply assuming the worst-case latency of instructions throughout a program's execution would result in a dramatic overestimation of its worst-case execution time (WCET). To achieve accurate results, WCET analysis thus needs to precisely take into account in which hardware states a program's instructions are executed. State-of-the-art static WCET analysis tools [1] explore a program's possible executions on a given hardware platform by a combination of explicit and implicit techniques. It is desirable to employ implicit techniques, such as abstractions [2], to efficiently explore large sets of states. While precise and efficient abstractions are known for caches [3], the efficient implicit analysis of pipelining is impeded by the presence of *timing anomalies* [4], [5]. Due to timing anomalies it is not safe for WCET analysis to only explore "local worst-case" successors of pipeline states; nor is it possible to devise efficient abstractions in which sets of concrete pipeline states are represented by individual abstract pipeline states.

Timing analysis for applications deployed on multi-core processors is even more challenging. Multi-core processors share resources such as buses, caches, or memory channels among multiple cores. As a consequence, the execution time of a task depends on the interference on shared resources that it experiences due to co-running tasks on other cores.

As in single-core WCET analysis, assuming the worst-case latency upon every shared-resource access is not a viable option as it would result in highly pessimistic execution time bounds. The greatest analysis precision would be achieved by *fully-integrated timing analyses* [6]–[8]: such analyses simultaneously analyze the tasks running on different cores of a multi core, precisely capturing all possible interleavings of resource accesses from different cores. Unfortunately, this approach appears to be practically infeasible for realistic systems due to the astronomical number of system states to explore. The most promising approach to multi-core timing analysis to date is *compositional timing analysis* [9]–[18], which can be seen as a natural extension of the classical two-step approach to timing analysis: low-level analysis, corresponding to classical WCET analysis, computes the "resource demand" of each task for each shared resource. Given such task characterizations, schedulability analysis then determines, whether each task can be guaranteed to meet its deadlines, accounting for the interference it may experience on each of the shared resources. Compositional timing analysis relies on the assumption that the response time of a task may be decomposed into contributions from different resources, which can each be efficiently analyzed separately.

We have shown in previous work that, unfortunately, even simple in-order pipelined cores feature timing anomalies and do not admit compositional timing analysis [19].

Based on preliminary ideas presented in [20], in this paper, we introduce the *strictly in-order core* (SIC), a pipelined processor core that is provably free of timing anomalies and that admits compositional timing analysis.

The starting point of this work has been the observation that the presence of timing anomalies in conventional in-order pipelines can be traced back to the non-monotonicity of their timing behavior. This non-monotonicity is due to dependencies between consecutive instructions, where progress of one instruction may be detrimental to the progress of another instruction. The key property of SIC is the monotonicity of its timing behavior, which is enforced by carefully eliminating some of the dependencies between instructions in the pipeline.

The experimental evaluation demonstrates that SIC achieves timing predictability without significantly sacrificing average-case performance: SIC is about 6-7% slower than a conventional pipelined core across a large set of benchmarks. On the other hand, due to its timing predictability, WCET analysis for SIC is about 4x faster, and low-level analysis suitable for multi-core timing analysis is about 30x faster than for a conventional design.

We also compare SIC experimentally with the PTARM [21], an instance of a precision-timed (PRET) machine [22], which exhibits similar predictability properties. In terms of single-thread performance SIC is preferable to the PTARM: a single thread is about 2x faster on a SIC core than in a PTARM hardware thread. This is because due to thread interleaving, individual threads do not profit from pipelining on the PTARM. On the other hand, considering all available hardware threads, the overall instruction throughput of the PTARM is about twice as high as SIC's.

Outline: We introduce the necessary background concerning in-order pipelines and timing analysis in Section II. In Section III we discuss how monotonic timing behavior is related to timing anomalies and timing compositionality. Then, in Section IV we formally define SIC and prove its timing predictability. After discussing the related work in Section V, we present SIC's experimental evaluation in Section VI. We conclude the paper with a discussion of potential future work in Section VII.

II. Background

A. In-order Pipelined Core

During its execution an instruction passes through multiple phases, such as fetching the instruction from memory, fetching required operands from registers, and performing an arithmetic computation or a data memory access. Each of these phases needs only a subset of the processor's datapath to execute. To increase instruction throughput, *pipelining* overlaps the execution of multiple instructions by performing different phases of consecutive instructions in parallel. A traditional textbook-style pipeline [23] consists of five stages as depicted in Figure 1 and executes instructions in-(program)-order, i.e. each stage encounters the instructions in the order specified in the program.

Data and control dependencies among instructions can cause hazards that prevent the pipeline from being perfectly filled. Data hazards, cases in which an instruction requires a value computed by a preceding instruction, can be reduced by forwarding results from intermediate pipeline stages to their use. The effect of control hazards, i.e. the next instruction to fetch depends on an unknown branch outcome, can be mitigated by branch prediction and speculative instruction fetches to keep the pipeline filled. If a prediction turns out to be wrong, the speculatively fetched instructions have to be removed from the pipeline. For details, we refer to [23].

As shown in Figure 1, pipelined cores are typically connected to a memory hierarchy with a common main memory accessed via separate instruction and data caches. Accesses to

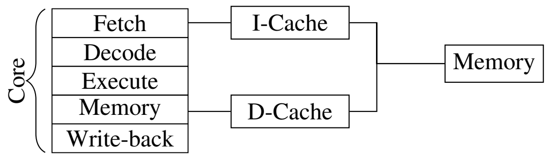


Fig. 1. Overview of system with an in-order pipelined core.

the memory are arbitrated in a greedy first-come first-serve manner. If both caches want to access the main memory at the same time, priority is commonly given to the data cache.

Unlike load instructions, store instructions do not produce results that have to be written to a register. Consequently, a store instruction can leave the pipeline once the store memory access has started. The pipeline can thus advance to the subsequent instructions while the memory hierarchy asynchronously completes the store by writing to main memory.

Modern processors feature multiple cores that share a common memory. Each core is connected to a shared interconnect to access the memory—potentially through private caches. Due to these shared resources, independent programs that run on different cores can interfere and thus impact each other's execution time.

B. Timing Analysis

Low-level analysis is the first step in the timing verification process. It computes characteristics of the behavior of a single task when executed in isolation, e.g. the task's worst-case execution time (WCET) or its worst-case number of accesses to the bus. These characteristics depend not only on the inputs to the program but also on the microarchitectural state (e.g. pipeline and cache state) of the underlying hardware platform.

To achieve soundness, low-level analysis has to take all possible inputs and hardware states into account. The de facto standard approach to this problem is to construct a graph in which the system's reachable states are represented by nodes and where edges correspond to state transitions. This graph captures all possible executions of the program at the granularity of individual processor cycles. Given this graph, the program's WCET, or similarly its worst-case number of bus accesses, can then be calculated by implicit path enumeration [24], which is typically encoded as an integer linear program (ILP).

As the space of concrete system states is too large to explicitly explore, abstractions [2] are employed. Common abstractions are value abstraction, e.g. intervals [2] to compactly represent the possible values of registers; and cache abstraction, e.g. must- and may-caches [25] that over-/underapproximate the cache contents to predict cache hits and misses.

While such abstractions are essential for efficient analysis, they introduce uncertainty: e.g. the must-cache analysis is not always able to determine whether or not a memory access results in a cache hit. If the processor's state transition depends on such uncertain information, the abstract analysis state is split, following all cases permitted by the abstraction. As an example, if it is uncertain whether an access hits the cache, both possibilities—cache hit and cache miss—are explored. As

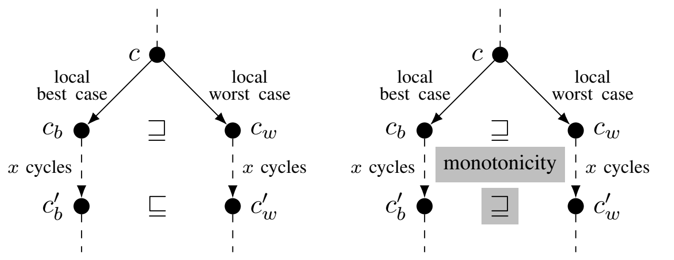


Fig. 2. Timing anomaly on the left. Anomaly Freedom on the right. The dashed arrow describe an arbitrary but fixed number of cycle transitions.

a consequence, abstractions generally result in nondeterminism in the analysis that explores the reachable system states.

The per-task characteristics computed by low-level analysis are inputs to the next step in timing verification: *schedulability analysis*. Schedulability analyses determine whether all tasks meet their respective deadline under a given scheduling policy. To this end, they account for any interference due to the execution of other tasks, e.g. due to preemptions or due to shared-resource competition on a multi core.

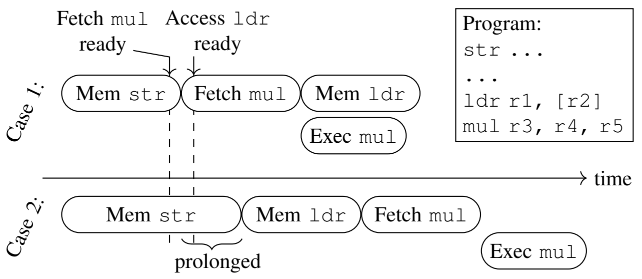
III. Anomaly Freedom and Compositionality

A. Timing Anomalies

As described in the Section II-B, abstractions induce nondeterminism in the low-level timing analysis. A *timing anomaly* occurs when the locally better case of a nondeterministic split, e.g. a cache hit instead of a cache miss, results in the global worst case—namely a longer overall execution time [4]. Due to timing anomalies, low-level (WCET) analysis has to explore all alternatives upon a nondeterministic choice, which makes the analysis expensive.

Consider the left part of Figure 2 for an illustration of timing-anomalous behavior. After the split, the program execution has made more "progress" in the *local best* case state c_b than in the *local worst* case state c_w, e.g. in case of a cache hit, the corresponding memory access is finished in c_b, while it is still ongoing in c_w. However, after a certain number of cycle transitions the situation reverses: the program's progress in c'_b falls behind c'_w and finally leads to a longer execution time. We conclude that the presence of anomalies requires the cycle behavior of the underlying hardware to be *non-monotonic* w.r.t. the progress order $\sqsubseteq$, which orders pipeline states according to their progress in executing a program. The progress order $\sqsubseteq$ is defined in more detail in Section IV.

Timing anomalies are known to be present in complex systems with out-of-order execution or speculation [4]. In [20], we have shown that even simple in-order pipelines feature timing anomalies. Indeed, conventional in-order pipelines as described in Section II-A behave non-monotonically. As an example, a memory instruction i in the memory stage can be delayed by a *subsequent* instruction k if the instruction fetch of k has already started when i reaches its memory stage. In this case, *more* progress of k leads to *less* progress of i in the future.



```
Program:
str ...
...
ldr r1, [r2]
mul r3, r4, r5
```

Fig. 3. Indirect effect upon uncertainty of the length of store access `str`, e.g. due to shared-bus blocking [19].

Using the contraposition, a *monotonic* cycle behavior is sufficient to guarantee the absence of timing anomalies. This idea is illustrated in the right part of Figure 2.

Anomaly freedom enables significantly more efficient low-level analysis, because it is then sound to only follow the local worst case successors in case of nondeterminism.

In Section IV, we present an in-order pipeline design called SIC whose cycle behavior is provably monotonic w.r.t. the progress order $\sqsubseteq$.

B. Timing Compositionality

Most approaches to schedulability analysis, in particular in the context of multi-core systems, assume *timing compositionality*. Compositionality as formally defined in [26] allows to compose the individual characteristics of tasks to a *sound* response time bound. The composition relies on penalties that quantify the impact of an interfering event, e.g. an access blocking the shared bus, on the response time. A sound composition has to not only account for the *direct effect* of an interfering event, e.g. the time in which the bus has been blocked, but also for potential *indirect effects*, i.e. an additional prolongation caused by a change in the microarchitectural state as a consequence of the interfering event.

In [19], we have shown that even simple systems with conventional in-order pipelined cores exhibit indirect effects. We provide an example in Figure 3. The indirect effect is triggered by uncertainty about concurrent bus accesses resulting in different latencies of the store instruction. Due to the greedy scheduling of main-memory accesses, a prolongation of the `str` instruction causes a different access order of the later data load and the instruction fetch on the common bus to the main memory. The access order arising in the prolonged case does not suit the execution of remainder of the program (i.e. the `mul` instruction) as the multiplication cannot be executed in parallel to the data load anymore. A more detailed explanation can be found in [19].

It is an open question how to calculate a bound on potential indirect effects for a given *arbitrary* microarchitecture.

However, assume we have a microarchitecture that behaves *monotonically* w.r.t. to the progress of a program's execution. Furthermore, assume that following the local worst case, p cycles after the split, the program's execution is guaranteed to have progressed at least as far as it would have progressed in the state reached immediately after following the local best

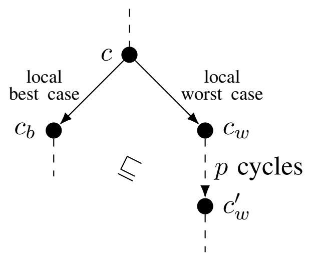


Fig. 4. General proof technique for timing compositionality with penalty p.

case. Then, due to monotonicity of the cycle behavior, we can conclude that a penalty of p cycles is an upper bound on the effect of a single occurrence of the local worst case. Figure 4 illustrates this reasoning. As we will show in Section IV the monotonicity of SIC enables such a formal reasoning on the possible indirect effects.

Timing compositionality enables an efficient and sound separation of concerns into low-level analysis and schedulability analysis. The low-level analysis can follow the local best cases only, e.g. the absence of a preemption or shared bus blocking. Schedulability analysis then accounts for the maximal number of local worst cases that can occur, e.g. additional cache misses due to preemptions, weighted by the appropriate penalty p.

IV. SIC: A Timing-Predictable Pipelined Core

In this section, we present and discuss SIC, a provably timing-predictable pipelined processor core. By timing-predictable, we mean that the core is *provably* free of anomalies and that it enables compositional timing analysis. As motivated in Section III, *monotonicity* w.r.t. to the program's execution progress is SIC's key underlying property.

A. Intuitive Design Decisions

Branch prediction and consequently speculative fetching and execution have been known to exhibit timing anomalies [5]. In our design, we conservatively refrain from any branch prediction and speculative actions.

More recently in [20], we identified the reordering of accesses on the memory bus that is caused by the greedy first-come first-serve arbiter, as the reason behind timing anomalies in simple in-order pipelines. Just as every pipeline stage encounters all instructions in program order, the bus arbiter should ensure that all bus accesses are performed in program order. In other words, all accesses (instruction fetch and data access) of an instruction are performed before any access of a subsequent instruction. To enforce this ordering, the pipeline controller of SIC *delays memory accesses caused by instruction cache misses as long as preceding instructions might still access the data memory*. This strict access ordering inspires the name of our design: the *strictly* in-order pipelined core (SIC).

B. Cycle Behavior

To be able to formally prove properties of SIC, we formally define its set of states as well as its cycle behavior, i.e. how a state evolves during a processor clock cycle.

The concrete execution of a program with a given input generates a sequence of machine instructions that are processed within the processor. In the following, we consider an arbitrary, but fixed sequence $i_0 i_1 i_2 \ldots$ of machine instructions. We denote the set of machine instructions in this sequence by $\mathcal{I} = \{i_0, i_1, i_2, \ldots\}$. Machine instructions are totally ordered by their position within the sequence, i.e. $i_n < i_m$ iff $n < m$. In the following, we use i,j, and k as metavariables for machine instructions within $\mathcal{I}$.

We define the state of the strictly in-order pipeline by mapping each instruction to its *progress* within the pipeline. The set of possible pipeline states is given by the subset

$$C \subseteq \mathcal{I} \to \mathcal{P},$$

such that no two instructions occupy the same pipeline stage (except *pre* and *post*) and only one instruction accesses the main memory at a time.

The progress of an individual instruction is determined by the pipeline stage it resides in and the number of cycles remaining to complete the current stage. The set of possible progress is thus defined as

$$\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$$

where $\mathcal{S} := \{\mathit{pre}, \mathit{IF}, \mathit{ID}, \mathit{EX}, \mathit{MEM}, \mathit{ST}, \mathit{WB}, \mathit{post}\}$ denotes the set of pipeline stages. The set $\mathcal{S}$ features artificial stages to model that an instruction has not yet entered (pre) or has already left (post) the pipeline. Just before leaving the pipeline, store instructions reach the ST stage in which the corresponding store is performed in memory, asynchronously to the rest of the pipeline. The other stages correspond to the classic five pipeline stages.

We formally define the concrete cycle behavior of the strictly in-order pipeline by the function $\mathit{cycle} : C \to C$ specified in Figure 5 relating a state to its successor state. In order to focus on the control behavior of the pipeline itself, we use external functions to model data-dependent effects, as well as the memory hierarchy including main memory and caches. The predicate $\mathit{ichit}(i)$ ($\mathit{dchit}(i)$) holds iff the instruction fetch (data access) of instruction i results in a cache hit; $\mathit{exlat}(i)$ returns the execution latency of instruction i; $\mathit{memlat}_f(i)$ ($\mathit{memlat}_d(i)$) returns the memory latency that the fetch (potential data access) of instruction i experiences. The auxiliary functions $\mathit{opc}(i)$, $\mathit{ops}(i)$, and $\mathit{target}(i)$ provide the type of instruction (opcode), the source operand registers, and the target register of instruction i.

The changes compared to a conventional in-order pipeline are highlighted in Figure 5. As long as a conditional branch—whose outcome is determined in the execute stage—is pending in the upper part of the pipeline, no new instructions are issued into the pipeline. Furthermore, as long as a data-memory-accessing instruction is pending in the pipeline, no accesses caused by an instruction cache miss are started. Note, however, that as long as instruction fetches hit the cache, the enforcement of a strict access order has no impact on the pipeline's behavior.

$$c' := \lambda i \in \mathcal{I}. \begin{cases} (\mathit{stage}'(i), \mathit{latency}(i)) & : \mathit{ready}(i) \wedge \mathit{willbefree}(\mathit{stage}'(i)) \\ (\mathit{stage}(i), \mathit{cnt}'(i)) & : \text{otherwise} \end{cases}$$

$$\mathit{cnt}'(i) := \begin{cases} \mathit{cnt}(i) - 1 & : \mathit{cnt}(i) > 0 \\ 0 & : \mathit{cnt}(i) = 0 \end{cases}$$

$$\mathit{stage}'(i) := \begin{cases} post & : \mathit{stage}(i) \in \{ST, WB\} \vee (\mathit{stage}(i) = ID \wedge opc(i) = nop) \\ WB & : \mathit{stage}(i) = MEM \wedge opc(i) \neq store \\ ST & : \mathit{stage}(i) = MEM \wedge opc(i) = store \\ MEM & : \mathit{stage}(i) = EX \\ EX & : \mathit{stage}(i) = ID \wedge opc(i) \neq nop \\ ID & : \mathit{stage}(i) = IF \\ IF & : \mathit{stage}(i) = pre \end{cases}$$

$$\begin{aligned} \mathit{ready}(i) := {} & (\mathit{stage}(i) = MEM \wedge opc(i) = store) \\ & \vee (\mathit{cnt}(i) = 0 \\ & \quad \wedge (\mathit{stage}(i) = EX \Rightarrow opc(i) \notin \{load, store\} \vee (dchit(i) \vee \neg stpending(i)) \\ & \quad \wedge (\mathit{stage}(i) = ID \Rightarrow \neg ophaz(i)) \\ & \quad \wedge (\mathit{stage}(i) = pre \Rightarrow \neg brpending(i) \wedge \mathit{next}(i) \wedge (ichit(i) \vee \neg mempending(i)))) \end{aligned}$$

$$\begin{aligned} \mathit{willbefree}(s) := {} & s = post \\ & \vee (\neg \exists i.\mathit{stage}(i) = s) \\ & \vee (\exists i.\mathit{stage}(i) = s \wedge \mathit{ready}(i) \wedge \mathit{willbefree}(\mathit{stage}'(i))) \end{aligned}$$

$$\mathit{latency}(i) := \begin{cases} memlat_f(i) & : \mathit{stage}'(i) = IF \wedge \neg ichit(i) \\ memlat_d(i) & : \mathit{stage}'(i) = MEM \wedge \neg dchit(i) \\ \mathit{cnt}(i) - 1 & : \mathit{stage}'(i) = ST \\ exlat(i) & : \mathit{stage}'(i) = EX \\ 0 & : \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathit{next}(i) &:= \mathit{stage}(i) = pre \wedge \forall j < i.\mathit{stage}(j) \neq pre \\ brpending(i) &:= \exists j < i.opc(j) = cond.branch \wedge c(j) \sqsubset_{\mathcal{P}} (EX, 0) \\ mempending(i) &:= \exists j < i.opc(j) = load \wedge c(j) \sqsubset_{\mathcal{P}} (MEM, 0) \\ & \quad \vee stpending(i) \\ stpending(i) &:= \exists j < i.opc(j) = store \wedge c(j) \sqsubset_{\mathcal{P}} (ST, 0) \\ ophaz(i) &:= \exists o \in ops(i).\exists j < i.c(j) \sqsubset_{\mathcal{P}} (MEM, 0) \\ & \quad \wedge opc(j) = load \wedge o \in target(j) \end{aligned}$$

Fig. 5. Definition of the cycle behavior of the strictly in-order pipelined core. Differences to in-order pipeline are highlighted.

C. Progress and Monotonicity

As discussed in Section III, the key ingredient to provable timing predictability is a partial order on pipeline states such that the cycle behavior is *monotonic*. Here, we formalize the notions of progress and monotonicity. We define a partial order on pipeline states based on the notion of progress of instructions' execution within the pipeline.

First, we define the order $\sqsubset_{\mathcal{S}}$ on pipeline stages, where later pipeline stages represent more progress:

$$\begin{tikzcd} & & & & & WB \arrow[dr, phantom, "\sqsubset_{\mathcal{S}}"] & \\ pre \arrow[r, phantom, "\sqsubset_{\mathcal{S}}"] & IF \arrow[r, phantom, "\sqsubset_{\mathcal{S}}"] & ID \arrow[r, phantom, "\sqsubset_{\mathcal{S}}"] & EX \arrow[r, phantom, "\sqsubset_{\mathcal{S}}"] & MEM \arrow[ur, phantom, "\sqsubset_{\mathcal{S}}"] \arrow[dr, phantom, "\sqsubset_{\mathcal{S}}"] & & post \\ & & & & & ST \arrow[ur, phantom, "\sqsubset_{\mathcal{S}}"] & \end{tikzcd}$$

Within the same pipeline stage, fewer remaining cycles indicate more progress. Consequently, we define the order on progress $(s, n), (s', n') \in \mathcal{P}$ as follows:

$$(s, n) \sqsubseteq_{\mathcal{P}} (s', n') :\Leftrightarrow s \sqsubset_{\mathcal{S}} s' \vee (s = s' \wedge n \geq n').$$

Finally, pipeline state c' has at least the progress of pipeline state c if each instruction has the same or more progress in c':

$$c \sqsubseteq c' :\Leftrightarrow \forall i \in \mathcal{I}.c(i) \sqsubseteq_{\mathcal{P}} c'(i).$$

Our goal is to show that the cycle behavior is monotonic in the partial order $\sqsubseteq$ on pipeline states defined above. To this end, we introduce the following lemma, which captures a key property of SIC that ensures monotonicity:

Lemma IV.1 (Progress Dependence)**.** *For the* SIC *cycle behavior, the progress of an instruction i depends* solely *on the progress of previous instructions j, $j < i$, and thus* never *on the progress of subsequent instructions k, $k > i$:*

$$\forall c, c' \in C : [\forall i : (\forall j \leq i : c(j) = c'(j)) \Rightarrow cycle(c)(i) = cycle(c')(i)].$$

In a conventional in-order pipeline, Lemma IV.1 does not hold. There, the progress of a memory instruction i in the memory stage depends on the progress of the fetch of a *subsequent* instruction k, $k > i$: if the fetch of k has already started a memory access, i has to delay its memory access until the fetch completes. SIC eliminates such dependencies by enforcing *all* instruction and data memory accesses in program order.

Every sensible core design, and in particular SIC, fulfills the following lemma, as otherwise the core might livelock:

Lemma IV.2 (Positive Progress)**.** *The* SIC *cycle behavior applied to a configuration $c \in C$ yields a successor configuration with more progress than c:*

$$\forall c \in C : c \sqsubset cycle(c),$$

where $c \sqsubset c' \Leftrightarrow c \sqsubseteq c' \wedge c' \not\sqsubseteq c$.

Note that by definition of the progress order $\sqsubseteq_{\mathcal{P}}$, even if no instruction advances to the next pipeline stage, a decrease in the number of cycles that an instructions has to remain in its current pipeline stage—e.g. to account for a cache miss or shared bus blocking—is considered as progress.

The two previous lemmas are important ingredients in the proof of the monotonicity of the cycle behavior:

Theorem IV.3 (Monotonicity)**.** *The* SIC *cycle behavior is* monotonic*:*

$$\forall c, d \in C : c \sqsubseteq d \Rightarrow cycle(c) \sqsubseteq cycle(d).$$

The strictly in-order pipeline behaves monotonically because the execution of an instruction is *never* delayed by other instructions making more progress. Unlike in SIC (Lemma IV.1), in a conventional in-order pipeline, a load instruction might for example get delayed by a subsequent instruction reaching the fetch stage earlier.

The formal proofs of the above statements involve lengthy case distinctions of the cycle behavior but do not offer any additional insights. Thus we only make them available in the appendix as supplementary material.

Ultimately, we are interested in the worst-case execution time of a program p starting from pipeline state c. The execution of a program finishes if its last instruction i leaves the pipeline, i.e. reaches *post*, during the next cycle transition.

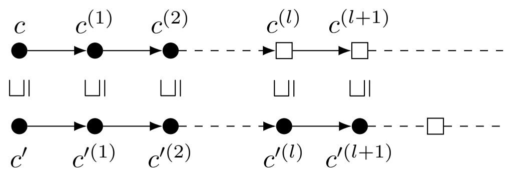


Fig. 6. Illustration of Theorem IV.4. The squares denote states where instruction i finished execution.

We define the finishing time $f(c,i)$ of instruction i starting from pipeline state c recursively as

$$f(c,i) := \begin{cases} 0 & : c(i) = (post, 0) \\ 1 + f(cycle(c), i) & : \text{otherwise} \end{cases}$$

We conclude this section by showing that a pipeline state c with at least the progress of another state c', i.e, $c \sqsupseteq c'$, finishes program execution no later than c'. Thus, to cover the worst-case timing behavior it is sufficient to only consider state c' during WCET analysis.

Theorem IV.4. *Let $i \in \mathcal{I}$ be an arbitrary machine instruction. Furthermore, let $c, c' \in C$ be such that $c \sqsupseteq c'$. Then*

$$f(c,i) \leq f(c',i).$$

Proof. Consider the sequence of pipeline configurations $c^{(0)}c^{(1)}\ldots$ such that $c^{(0)} = c$ and $c^{(n)} = cycle(c^{(n-1)})$. Due to Lemma IV.2, there exists an l such that $c^{(l)}$ is the first state with $c^{(l)}(i) = (post, 0)$, i.e. $f(c,i) = l$.
We prove the above claim by induction on l. The base case $l = 0$ is trivial, as by definition, any finishing time is non-negative.

In the induction step, we use monotonicity (Theorem IV.3) to derive that $cycle(c) \sqsupseteq cycle(c')$. Using the induction hypothesis for $cycle(c)$ and $cycle(c')$, we conclude:

$$\begin{aligned} f(c,i) &\overset{\text{Def. of } f}{=} 1 + f(cycle(c), i) \\ &\overset{\text{I.H.}}{\leq} 1 + f(cycle(c'), i) \\ &\overset{\text{Def. of } f}{=} f(c', i). \end{aligned}$$ □

D. Anomaly Freedom and Compositionality Proofs

Here, we carry out the proofs of anomaly freedom and timing compositionality following the proof strategies developed in Section III. These proofs depend on the kind of nondeterminism that might cause an anomaly or an indirect effect. In particular, we will show that cache uncertainty will not lead to anomalies, which allows for an efficient low-level timing analysis that follows only the cache-miss case upon a split. Furthermore, we will prove sound penalties for cache misses and prolongations of the memory accesses. This allows the schedulability analysis to account for cache misses induced by preemption [18], [27] as well as for shared bus blocking [18] in a *sound* and *compositional* manner.

Formally, we model uncertainty by different valuations of the external functions $ichit$, $dchit$, $memlat_f(i)$, and $memlat_d(i)$. In the proofs, we consider without loss of generality valuations that differ in only one point, e.g. whether a particular instruction hits or misses the cache. For valuations that differ in multiple points, the respective argument can be applied inductively in a backward manner starting from the uncertain information used last.

Theorem IV.5 (Anomaly freedom w.r.t. cache uncertainty)**.** *Let two valuations of $dchit$ (or $ichit$) be given that differ for an arbitrary instruction $i \in \mathcal{I}$. The valuation that predicts a cache miss, i.e. the local worst case, will lead to a finishing time at least as high as the valuation that predicts a cache hit, i.e. the local best case.*

Proof. Let c be the state that splits upon the cache uncertainty of instruction i, leading to the hit-case successor state c_b and miss-case successor c_w. Without loss of generality, we consider a data cache miss. We need to prove that $c_b \sqsupseteq c_w$ which proves the overall claim by Theorem IV.4. Due to Lemma IV.1, the progress of instructions j, $j < i$, does not depend on this uncertainty and thus even $c_b(j) = c_w(j)$. For i, we know $c_b(i) = (MEM, 0) \sqsupseteq_{\mathcal{P}} c_w(i)$ as $c_w(i)$ is either $(MEM, memlat_d(i))$ or $(EX, 0)$ depending on $stpending(i)$. For instruction k, $k > i$, $c_b(k) \sqsupseteq_{\mathcal{P}} c_w(k)$ follows from the fact that $ready(k)$ holds under a data cache hit if $ready(k)$ holds under a data cache miss. Thus, if k progressed in c_w it has progressed in c_b as well. □

Theorem IV.6 (Compositionality w.r.t. latency prolongation)**.** *Let two valuations of $memlat_d$ (or $memlat_f$) be given that differ by p cycles for an arbitrary instruction $i \in \mathcal{I}$, e.g. due to shared bus blocking. The valuation that predicts a longer latency leads to a finishing time at most p cycles higher than the valuation that predicts the shorter latency.*

Proof. Without loss of generality, we consider the latency $memlat_d$ of data accesses. Let c be the state that splits upon the latency uncertainty of instruction i, leading to the fast-case successor state c_b and slow-case successor c_w. As the latency value does not directly influence the progress of instructions other than i, c_b and c_w have identical progress for all instructions but i. By definition of $cycle$, after p cycles, c'_w reaches the same progress for i as c_b. According to Lemma IV.2, instructions in c'_w exhibit at least the progress as in c_w. Consequently, $c'_w \sqsupseteq c_b$ and the claim follows by Theorem IV.4. □

Theorem IV.6 characterizes the *impact* of prolonging a memory access by p cycles, e.g. due to shared bus blocking. It does not provide means to bound the actual amount of shared bus blocking, which is an orthogonal problem. As an example, Altmeyer et al. [18], [28] bound the amount of shared bus contention for a variety of bus protocols.

Theorem IV.7 (Compositionality w.r.t. cache uncertainty)**.** *Let two valuations of $dchit$ (or $ichit$) be given that differ for an arbitrary instruction $i \in \mathcal{I}$. The valuation that predicts a cache miss will lead to a finishing time at most p cycles higher than the valuation that predicts a cache hit. For the data cache with* *write-through policy, p is twice the memory latency. For the instruction cache, p is five times the memory latency.*

Proof. Let c be the state that splits upon the cache uncertainty of instruction i, leading to the hit-case successor state c_b and miss-case successor c_w. First, we consider a data cache miss. As long as $stpending(i)$, i is stalled in the execute stage in the states following c_w. Let ml denote the memory latency. After at most ml cycles, the store finishes and $\neg stpending(i)$. Now, the memory access caused by the data cache miss can start. After ml additional cycles, i reaches progress $(MEM, 0)$ in state c'_w—the same as $c_b(i)$. It remains to show that $c'_w \sqsupseteq c_b$. By Lemma IV.1, instructions j, $j < i$, are not affected by this uncertainty and thus progressed further in c'_w than in c_b. By the definition of *ready* and *willbefree*, it follows that instructions k, $k > i$, that progressed in c_b could also progress at latest during the cycle transition leading to c'_w. Consequently, $c'_w \sqsupseteq c_b$ and the claim follows by Theorem IV.4.

For the instruction cache miss, i is stalled in the pre stage as long as $mempending(i)$. In the worst case, a store is pending in ST and three load instructions occupy the stages IF, ID, EX. It takes four times the memory latency to execute these such that $\neg mempending(i)$ and the instruction cache miss can be served. The remainder of the proof is analogous to the data cache case. □

Similarly to Theorem IV.6, Theorem IV.7 characterizes the *impact* of an additional cache miss, e.g. due to multi-core cache contention, on the execution time. It does not provide means to bound the amount of actual cache contention. The sound approximation of the external functions $ichit$ and $dchit$, possibly under multi-core contention, is an orthogonal problem.

Furthermore, Theorem IV.7 provides cache miss penalties in case of a write-through data cache. For write-back caches, the penalties have to be increased respectively in order to account for the number of potentially required write backs.

V. Related Work on Timing-Predictable Microarchitectures

Design for timing predictability has been an active research topic since at least 2003, when a Dagstuhl Perspectives Workshop on "Design of Systems with Predictable Behavior" was organized [29]. The outcome of this workshop was summarized in a paper by Thiele and Wilhelm [30]. One important insight from [30] is that there are two orthogonal ways to improve timing predictability:

1) "Reducing the sensitivity to interference from non-available information," and
2) "to match implementation concepts with analysis techniques to improve analyzability."

While SIC may also slightly reduce the sensitivity of execution times to interference from non-available information compared with a conventional in-order pipeline, we view the strictly in-order pipeline primarily as an instance of the second kind: Due to the absence of timing anomalies and the ability to carry out compositional timing analysis, a program's worst-case timing may be analyzed efficiently even if the program's execution time may vary widely.

Berg et al. [31] discussed several "principles of a predictable processor", partly based on insights from prior work by Engblom and Jonsson [32]. In particular, they suggest to construct a pipeline with no hardware interlocks between instructions, which means that the compiler needs to ensure that instruction dependencies are respected. They argue that such a pipeline will not exhibit "long timing effects" (LTEs) [32] of length greater than 3, *provided that* no data cache is employed. On the other hand, they realize that introducing a data cache immediately results in a pipeline exhibiting infinite LTEs, a property related to the presence of timing anomalies. The strictly in-order pipeline design presented in this paper is one way of resolving this issue.

A number of European projects, including PREDATOR [33], MERASA [34], parMERASA [35], and T-CREST [36] have tackled the challenge of designing timing-predictability multi-core architectures and are discussed below:

Wilhelm et al. [33] introduce the notion of "fully timing-compositional architectures" as microarchitectures which do not feature timing anomalies. They conjecture but do not provide a formal proof that the ARM7, which stalls all components of the pipeline upon a "timing accident", is indeed fully timing-compositional.

Ungerer et al. [34] propose the MERASA multicore architecture, which features a multi-threaded core, comprising one hard real-time thread (HRT) and a number of non-HRT threads. The HRT is temporally isolated from the non-HRT threads, so that it can be analyzed like a single-threaded core. The HRT is equipped with local instruction (D-ISP) and data scratchpad (DSP) memories. The D-ISP loads the complete code of an activated function dynamically on call and return. It is conceivable, but has not been proven, that this construction eliminates timing anomalies, as there can be no direct interference between instruction and data memory accesses. The parMERASA project [35] focused on multi-core execution of parallelized hard real-time applications. To our knowledge it did not introduce new processor core designs.

As part of the T-CREST project, Schoeberl et al. [36] propose the time-predictable PATMOS processor. PATMOS is statically scheduled, which means that the compiler needs to ensure that instruction dependencies are respected. Similarly to MERASA, the Patmos processor contains a method cache [37], a cache that stores whole functions, which are loaded upon calls and returns. Similarly to MERASA, it is conceivable, but has not been proven, that this construction eliminates timing anomalies.

De Dinechin et al. [38] describe the KALRAY MPPA®-256, a many-core processor suitable for time-critical computing. Its cores are claimed to be "fully timing-compositional" [33]. We note that due to the presence of a store buffer the cores likely feature timing anomalies as discussed in our prior work [19].

Lee and Edwards [22] make the case for precision-timed (PRET) machines, processors that exhibit predictable and repeatable timing and "whose temporal behavior is as easily controlled as their logical function." Liu et al. [21] present the Precision-Timed ARM (PTARM), a PRET machine that employs a thread-interleaved pipeline with an exposed memory hierarchy. By interleaving four hardware threads in a standard five-stage in-order pipeline, the PTARM eliminates all pipeline hazards. It uses a scratchpad memory instead of caches as a fast local memory, and a DRAM controller [39] with repeatable timing. As a consequence of the PTARM's design, each instruction has a fixed latency that is independent of its execution context. This greatly simplifies WCET analysis, and can be seen as an instance of the first of the two orthogonal ways to improve timing predictability. Furthermore, by eliminating pipeline hazards, its overall instruction throughput is higher than the instruction throughput achieved by a conventional pipelined processor. On the other hand, each of its hardware thread is slower than a conventional pipelined processor as it does not profit from pipelining. By construction, the PTARM does not exhibit timing anomalies and it naturally admits compositional timing analysis, even if its thread-interleaved pipeline is connected to a regular memory hierarchy. We evaluate its performance relative to SIC in the experimental evaluation.

Zimmer et al. [40] introduce FlexPRET, a more flexible multi-threaded architecture, in which each hardware thread can be configured to use between one and all of the processor's pipeline stages. Thus, depending on its configuration, from the perspective of a single hardware thread, FlexPRET looks like a regular in-order pipelined processor, like a hardware thread in the PTARM, or like something in between the two. Hardware threads configured to use several pipeline stages may exhibit the same timing anomalies as a regular pipelined processor.

VI. Experimental Evaluation

In this section, we want to shed light on the performance degradation of SIC compared with a conventional in-order pipeline and on the gain in analysis efficiency due to its timing predictability. Besides the in-order pipeline, we also conduct a comparison of SIC with the PTARM [21] implementation of a PRET machine, as it exhibits similar predictability properties.

We evaluate SIC in terms of actual performance as well as in terms of timing bounds. To this end, we have implemented a conventional in-order pipelined core as described in Section II-A, the proposed strictly in-order pipelined core, and a PTARM-like core in the hardware description language Verilog to target an FPGA. Our PTARM-like core implements the behavior of a single thread in the original PTARM core proposed in [21]. All three core designs support the same subset of the ARM instruction set architecture. To obtain timing (WCET) bounds and to assess the timing analysis efficiency, we implemented the timing model of all three core designs in our low-level timing analysis tool LLVMTA as also used in [19]. In order to focus the comparison just on the core design, for the experiments, we assume all cores to be connected to the same memory hierarchy with separate instruction and data caches and a common background memory.

TABLE I
Comparison of FPGA-related characteristics.

Design	Max. Frequency	Logical Elements
in-order	60 MHz	5037
strictly in-order	60 MHz	5046 (+0.2%)
PTARM-like	65 MHz (+8.3%)	4294 (-14.8%)

We present results for different cache sizes and memory latencies. By memory latency, we denote the time between the start of the access until the last word of the cache line has been put into the cache. All caches are direct-mapped with a cache line size of four 32-bit words. The data cache employs the write-through policy to handle stores. Taking the memory footprint of our benchmarks into account, we evaluate caches with 64, 256, and 1024 cache sets, amounting to cache sizes of 1 KiB, 4 KiB, and 16 KiB. We consider three different memory latencies: 4, 12, and 100 clock cycles. The latency of 12 cycles corresponds to a realistic value for actual main memory (Micron MT46V16M16 [41] Automotive DDR SDRAM run at 100MHz). We consider a latency of four and 100 cycles to analyze the effects of a very fast and a very slow memory. Due to a memory bus width of 32 bits, four cycles is the minimum memory latency to transfer a whole cache line.

As benchmarks, we use the TACLeBench suite [42]. If not stated otherwise, the benchmarks have been compiled using CLANG with optimization level `-O2`.

A. FPGA Resources

We have synthesized the three core variants using Intel Quartus Prime Lite targeting an Altera Cyclone IV E (EP4CE115) FPGA on a Terasic DE2-115 development board. We have configured the synthesis tool to optimize for high clock frequency. The FPGA-related characteristics of the different designs are depicted in Table I. The logical element count only covers the processor core module and excludes the logic needed to implement the memory hierarchy. A logical element of the Altera FPGA contains a one-bit flip-flop and a lookup table to implement a 4-to-1 boolean function. Each core designs uses 8 DSP slices to implement multiplication. No core design makes use of the FPGA's block RAM.

The additional logic needed to enforce the strict access ordering on the bus has a negligible effect on the clock frequency and the number of required logical elements as can be seen in Table I. This shows that a strictly in-order pipelined core can be implemented with low overhead.

The PTARM-like core improves upon the conventional pipelined core w.r.t. the maximal clock frequency as well as the number of required logical elements. Due to the nature of the thread-interleaved pipeline of the PTARM, a single thread cannot encounter hazards—neither control nor data hazards. This obviates the need for result forwarding and interlocking of pipeline stages which in turn leads to shorter circuit paths (higher clock frequency). However, we have only implemented a single thread of the PTARM. Extending our prototype to support three additional threads will increase the number of logical elements by at least 1536 to implement the three

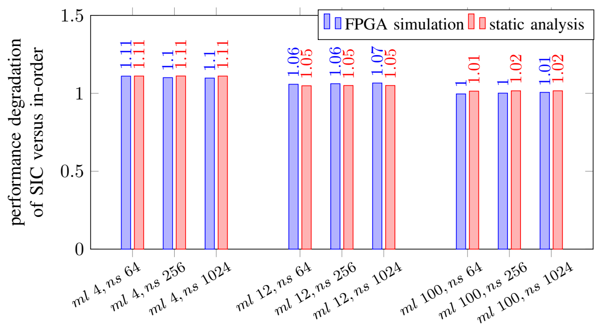


Fig. 7. Performance degradation of SIC compared to a conventional in-order pipeline. Geometric mean over all benchmarks. Results shown for different memory latencies (ml) and cache sizes (ns = number of cache sets).

additional register files (each with sixteen 32-bit registers). This amounts to at least 5830 logical elements for a full-fledged PTARM core.

B. Actual and Guaranteed Performance

Here, we compare the actual and the guaranteed performance of the strictly in-order pipelined core (SIC)—first against the conventional in-order pipelined core, and then against a single thread of the PTARM core.

1) SIC versus in-order: Here, we want to assess the impact of the modifications towards SIC on the performance of programs. For each program we determine its execution time in cycles by executing it on both core designs using the FPGA. Each program is run on a fixed input and an empty initial hardware state, i.e. empty cache and empty pipeline. Then we calculate the geometric mean of the ratios of the clock cycles needed—across all benchmarks. We repeated the experiments using our static timing analyzer to compute WCET bounds of programs run on both core design respectively. Figure 7 depicts the geometric means for different memory latencies and cache sizes.

The performance degradation induced by SIC is around 6-7% for the realistic memory latency. With increasing memory latencies the performance degradation induced by SIC decreases. This is expected because with increasing memory latencies, the time spent waiting for memory more and more dominates the total execution time. We observe that the cache size has almost no impact on the performance penalty induced by SIC.

2) SIC versus PTARM: Due to the noticeable difference in maximal clock frequency between SIC and PTARM, we consider the ratio of real execution time instead of number of clock cycles. According to Table I, the PTARM-like core can clock around 8% faster. As the memory latency in nanoseconds is constant, the memory latency *in clock cycles* is consequently 8% higher for the PTARM-like core compared to SIC. Due to the nature of the thread-interleaved PTARM core, a single thread can use the memory stage only in every fourth cycle. As a consequence, an access experiences a memory latency that is a multiple of four cycles—even though the actual memory latency might not be a multiple of four. The memory latency of 12 cycles@f_{SIC} thus corresponds to 13 cycles@f_{PTARM} and an experienced latency of 16 cycles@f_{PTARM} (rounded to the next multiple of four). Due to the significant gap between the actual and the experienced memory latency, we additionally consider a latency of 11 cycles@f_{SIC}, which corresponds to an experienced latency of 12 cycles@f_{PTARM}.

In a PTARM-like core, four threads share the local fast memory in a spatial manner. To account for this, we compare SIC to PTARM with a quarter of the cache size. As reference, we also provide the numbers for an equally-sized cache.

We depict the results of the FPGA simulation in Figure 8 and the results of the LLVMTA static analysis in Figure 9. They y-axis corresponds to the slowdown a program experiences when executed on a single PTARM thread relative to SIC. We provide the geometric mean across all benchmarks. The first (second) bar compares SIC to PTARM using caches with quarter (equal) size.

The third bar per block depicts the results for the benchmarks compiled without optimizations. Those have a significantly larger share of memory instructions on the total number of instructions (50% instead of 28% for optimized programs). Therefore, the influence of the memory hierarchy's behavior is higher which leads to smaller ratios.

A full-fledged PTARM core implementation with four threads requires more chip resources than SIC. The fourth bar of each block shows the ratio of the performance PTARM versus SIC *relative to the implementation cost*, i.e. the number of logical elements used in the FPGA.

As before, we observe that the performance differences caused by the different core designs vanish with higher memory latency and increasing influence of the memory performance. The greater the cache size, the smaller the effect of the reduced cache size of PTARM compared to SIC.

The lower ratios in static analysis compared to the FPGA simulation are explained by the imperfection of the cache analysis. Not every access that hits the cache during the actual execution is guaranteed to hit the cache by the static analysis. Thus, the static analysis considers more cache misses, i.e. more main memory accesses, which increases the influence of the memory performance.

For realistic memory latencies, the single-thread performance of the SIC core is roughly twice the single-thread performance of the PTARM core. However, the PTARM core can execute four threads in parallel. Thus, the total *instruction throughput* of SIC is about half the throughput of the PTARM. Note, however, that this analysis is potentially optimistic w.r.t. the PTARM, as it assumes that each of its four threads would experience the same worst-case main-memory latency as the single SIC thread it competes with.

Comparing the performance of SIC versus PTARM, which can be seen as the performance of an unpipelined processor, with the performance degradation of SIC versus a conventional in-order pipeline, we conclude that SIC preserves most of the performance benefits of pipelining.

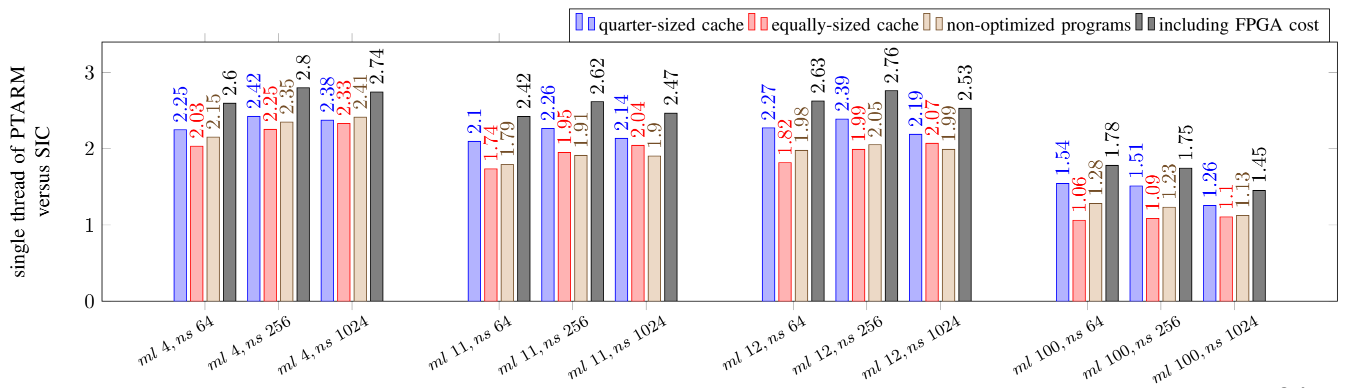


Fig. 8. Results obtained by simulation on the FPGA. Different memory latencies and cache sizes. The memory latencies are given in cycles@f_{SIC}.

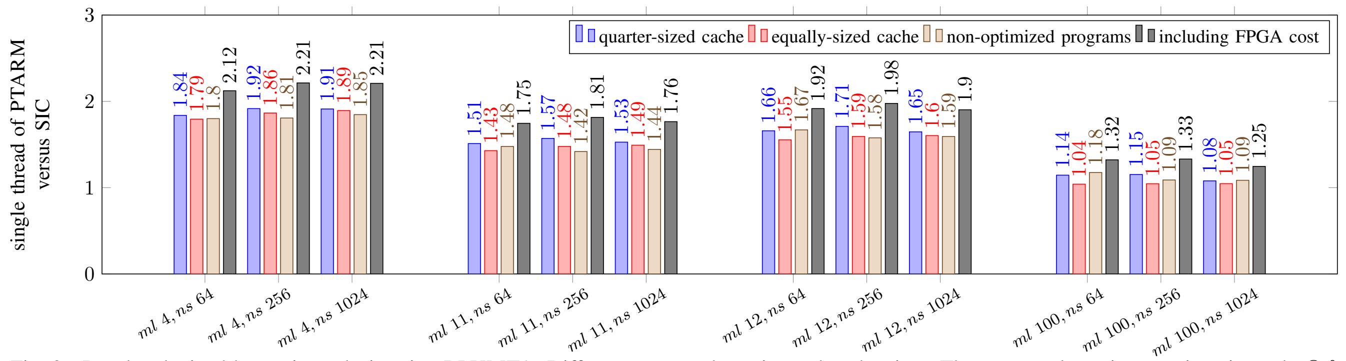


Fig. 9. Results obtained by static analysis using LLVMTA. Different memory latencies and cache sizes. The memory latencies are given in cycles@f_{SIC}.

C. Analysis Performance

The anomaly freedom of SIC enables the low-level timing analysis to soundly follow the local worst case only. In a multi-core setting, the compositionality of SIC enables the low-level analysis to assume the absence of interference on shared resources, such as a shared bus. The subsequent schedulability analysis may use the penalties calculated in Section IV-D to soundly account for shared-resource interference.

In contrast, the analysis of the conventional in-order pipeline has to follow all alternatives upon uncertainty. In a multi-core setting, timing analysis has to additionally account for potential indirect effects caused by shared resource interference. The only feasible approach currently known is the computation of a *compositional base bound* [19]. There, the low-level analysis explores the impact of potential interference on the microarchitectural states and thus captures the worst-case execution time including any potential indirect effects. In the following experiments, we consider a bus with round-robin event-driven arbitration as the only shared resource.

For each benchmark we obtain the analysis runtime of the following four analyses: 1) the PTARM analysis, 2) the analysis for the conventional in-order core, 3) the compositional base bound analysis for the conventional in-order core, and 4) the analysis for SIC. To assess the analysis speedup afforded by SIC, we determine the ratios of the first three analysis runtimes to the analysis runtime for SIC. Figure 10 depicts the geometric mean of these ratios across all benchmarks for the three different memory configurations. To assess the per-benchmark runtimes, Figure 11 depicts scatter plots for the memory configuration ml 12, ns 256.

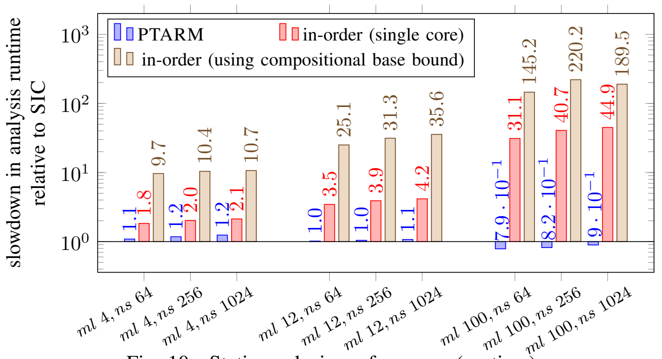


Fig. 10. Static analysis performance (runtime)

As expected, we observe significant differences in the analysis runtimes of SIC and the conventional in-order pipeline. For the realistic memory latency, following all alternatives upon uncertainty causes the analysis to slowdown by a factor of roughly 4. If indirect effects due to shared-bus interference are taken into account in addition, the slowdown amounts to a factor of roughly 30.

The scatter plots show that the slowdown of individual benchmarks is generally close to the geometric mean. The regression indicates a slight trend that bigger benchmarks tend to profit more from SIC than smaller ones.

Furthermore, in Figure 10, we observe that the analysis-runtime ratios increase with higher memory latencies, i.e. following all alternatives becomes more expensive.

The analysis runtimes of SIC and PTARM are very similar.

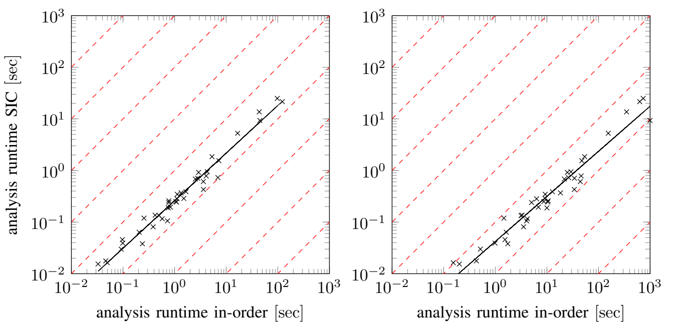


Fig. 11. Scatter plot of analysis runtime of SIC versus the in-order pipeline. Single core on the left. Multi core using the compositional base bound approach on the right. Memory with latency of 12 cycles and cache with 256 sets. Solid black line represents the regression curve.

In some cases, the PTARM analysis may take slightly longer, as the execution time bounds are generally higher and thus the analysis needs to explore more states. However, due to its deterministic behavior, a slightly more efficient PTARM analysis is conceivable that operates at the granularity of instructions instead of individual processor cycles.

VII. Conclusions and Future Work

We propose the use of the strictly in-order pipelined core as a *provably* timing-predictable hardware design. Its anomaly freedom enables more *efficient* low-level (WCET) analyses. Its timing compositionality allows for an *efficient and sound* separation of concerns into low-level analysis and schedulability analysis.

To the best of our knowledge, we present the first formal proofs of timing predictability for a non-trivial pipelined processor core.

The key innovation is to establish a cycle behavior that is *monotonic* w.r.t. the progress of instructions within the core. To this end, we refrain from speculation and we ensure that all bus accesses are performed strictly in program order.

We have quantified the performance degradation showing that most of the performance benefits of pipelining are preserved. Due to SIC's minimal implementation overhead, it is conceivable to equip future processors with a *monotonic mode*. The mode could be dynamically turned on and off to guarantee timing predictability only when required.

The principle of monotonic timing behavior is more general. In future work, we plan to examine how to employ performance-enhancing hardware features without sacrificing monotonicity and thus timing predictability. In particular, we will consider restricted forms of speculation as well as more dynamic instruction scheduling, i.e. restricted forms of out-of-order execution.

VIII. Appendix

Here, we provide the proofs for the lemmas stated in the main section that can all be proven by case distinction of the cycle behavior relation.

In the process of proving monotonicity of the strictly in-order pipeline, we use the following, rather technical lemma.

Lemma VIII.1 (Update enable)**.** *Let a, b be two configurations. Furthermore, let $i \in \mathcal{I}$ be an instruction with equal progress in a and b ($a(i) = b(i)$) and all previous instruction $j < i$ have progressed more in a than b ($a(j) \sqsupseteq_{\mathcal{P}} b(j)$). For any given valuation of the free variables in ready, if b advances to the next pipeline stage, a advances as well:*

$$b.\mathit{ready}(i) \Rightarrow a.\mathit{ready}(i)$$
$$b.\mathit{willbefree}(b.\mathit{stage}'(i)) \Rightarrow a.\mathit{willbefree}(b.\mathit{stage}'(i))$$

Proof. $\underline{\mathit{ready}}$
Let $b.\mathit{ready}(i)$. There are two cases. In the first case, $b.\mathit{stage}(i) = \mathit{MEM}$ and $\mathit{opc}(i) = \mathit{store}$. As $a(i) = b(i)$, it follows that $a.\mathit{stage}(i) = \mathit{MEM}$ and thus $a.\mathit{ready}(i)$. In the second case, we have $b.\mathit{cnt}(i) = 0$ and by $a(i) = b(i)$ also $a.\mathit{cnt}(i) = 0$. For all pipeline stages except *pre*, *ID* and *EX* this is sufficient to conclude $a.\mathit{ready}(i)$. We prove the claim for *EX*, *ID* and *pre* each by contradiction.

- For stage EX, $\neg a.\mathit{ready}(i)$ implies that i is a store instruction or a load instruction that misses the data cache, while a store is pending. If $a.\mathit{stpending}(i)$, there is a store instruction $j < i$ with $a(j) \sqsubset_{\mathcal{P}} (ST, 0)$. By assumption, we know that $a(j) \sqsupseteq_{\mathcal{P}} b(j)$. By transitivity, $b(j) \sqsubset_{\mathcal{P}} (ST, 0)$ and thus $b.\mathit{stpending}(i)$. It follows, that $\neg b.\mathit{ready}(i)$ which is a contradiction.
- For stage ID, $\neg a.\mathit{ready}(i)$ requires an operand hazard $\mathit{ophaz}(i)$. This means, there is a load instruction $j < i$ with $a(j) \sqsubset_{\mathcal{P}} (\mathit{MEM}, 0)$ that writes our operand. By $a(j) \sqsupseteq_{\mathcal{P}} b(j)$, we conclude that $b(j) \sqsubset_{\mathcal{P}} (\mathit{MEM}, 0)$, i.e. there is an operand hazard in b. This contradicts $b.\mathit{ready}(i)$.
- For stage *pre*, $\neg a.\mathit{ready}(i)$ requires either $\mathit{brpending}(i)$, $\mathit{mempending}(i) \wedge \neg \mathit{ichit}(i)$, or $\neg \mathit{next}(i)$. In all three cases an argument analogous to ophaz applies. If there is a branch j pending in a, j is also a pending branch in b as $a(j) \sqsupseteq_{\mathcal{P}} b(j)$. If there is an older instruction $j < i$ to be fetched next, j is also in the *pre* stage in b and is to be fetched next in b. If a memory operation j is pending in a, j is also a pending memory operation in b as $a(j) \sqsupseteq_{\mathcal{P}} b(j)$. Thus, if any of the three expressions above evaluates to true for a, it also evaluates to true for b resulting in $\neg b.\mathit{ready}(i)$. This is a contradiction.

This concludes the proof for *ready*.

$\underline{\mathit{willbefree}}$
Let $s = b.\mathit{stage}'(i)$ and $b.\mathit{willbefree}(s)$. If $s = \mathit{post}$, $a.\mathit{willbefree}(s)$ follows by definition of *willbefree*. If s is empty in b, i.e. no $j < i$ is in s, s must also be empty in a since $a(j) \sqsupseteq_{\mathcal{P}} b(j)$.

Otherwise a $j < i$ is in s such that $b.\mathit{ready}(j)$ and $b.\mathit{willbefree}(b.\mathit{stage}'(j))$. Since $a(j) \sqsupseteq_{\mathcal{P}} b(j)$, either $a(j) = b(j)$ or $a(j) \sqsupset_{\mathcal{P}} b(j)$. In the latter case, j in a is already in a stage further than s and consequently $a.\mathit{willbefree}(s)$ since stage s must be free in a. In the other case, $a(j) = b(j)$, we can inductively use this Lemma VIII.1 for j which is in a later stage than i. We repeat this argument until we hit either a free stage or stage *post*. By applying the induction hypothesis, i.e. Lemma VIII.1 for j, we get $a.\textit{ready}(j)$ and $a.\textit{willbefree}(b.\textit{stage}'(j))$. This results in $a.\textit{willbefree}(s)$. □

Proof of Lemma IV.1.

Proof. The progress of an instruction depends on the progress of other instructions exclusively via *ready* and *willbefree*. By the proof of Lemma VIII.1, we know that *ready* and *willbefree* solely depend on the progress of previous instructions. □

Proof of Lemma IV.2.

Proof. First, we prove $c \sqsubseteq \mathit{cycle}(c)$ by case distinction of cycle. We denote $\mathit{cycle}(c)$ by c' for short. Let an instruction $i \in \mathcal{I}$ be given. If $c'(i)$ is stalled, $c'(i) = c(i)$ and thus $c(i) \sqsubseteq_{\mathcal{P}} c'(i)$. If $c'(i)$ reduces the number of remaining cycles, $c'(i) = (c.\textit{stage}(i), c.\textit{cnt}(i){-}1)$ and thus $c(i) \sqsubset_{\mathcal{P}} c'(i)$. If $c'(i)$ advances to the next pipeline stage, $c.\textit{stage}(i) \sqsubset_{\mathcal{S}} c'.\textit{stage}(i)$ and thus $c(i) \sqsubset_{\mathcal{P}} c'(i)$.

To prove the strictness of $c \sqsubset c'$, it is sufficient to show that not every instruction is stalled in the pipeline. We will show that the instruction farthest down the pipeline is not stalled. Let instruction i be the farthest instruction, i.e. all instructions $j < i$ already left the pipeline. All stages below $c.\textit{stage}(i)$ are empty, which results in $\textit{willbefree}(c.\textit{stage}'(i))$. If $c.\textit{cnt}(i) > 0$, i is not stalled as the number of remaining cycles is reduced. If $c.\textit{cnt}(i) = 0$, $c.\textit{ready}(i)$ and thus i would progress to the next stage. Even if the current stage of i is EX, ID, or *pre*, the readiness of i cannot be prevented from operand hazard or pending branches/memory operations as the pipeline in front of i is empty. □

Proof of Theorem IV.3.

Proof. Let c, d be given such that $c \sqsubseteq d$. We need to prove that $\mathit{cycle}(c) \sqsubseteq \mathit{cycle}(d)$. We denote $\mathit{cycle}(c)$ by c' and $\mathit{cycle}(d)$ by d' for short.

For every $i \in \mathcal{I}$, we need to show that $c'(i) \sqsubseteq_{\mathcal{P}} d'(i)$. We distinguish three possible cases of cycle applied to c: (1) i is stalled, (2) i counts down its remaining cycles, or (3) i advances to the next pipeline stage.

Pipeline stall
If instruction i is stalled in configuration c, we obtain $c'(i) = c(i)$. By assumption, we know $c'(i) = c(i) \sqsubseteq_{\mathcal{P}} d(i)$. By Lemma IV.2, we conclude that $c'(i) \sqsubseteq_{\mathcal{P}} d'(i)$.

Remaining cycles countdown
If instruction i reduces its remaining cycles in c, we obtain $c.\textit{stage}(i) = c'.\textit{stage}(i)$ and $c'.\textit{cnt}(i) = c.\textit{cnt}(i) - 1$. If $c(i) = d(i)$, by definition of cycle we obtain $c'(i) = d'(i)$. Otherwise, $c(i) \sqsubset_{\mathcal{P}} d(i)$:

- If $c.\textit{stage}(i) \sqsubset_{\mathcal{S}} d.\textit{stage}(i)$, we conclude by Lemma IV.2 that $c'.\textit{stage}(i) \sqsubset_{\mathcal{S}} d'.\textit{stage}(i)$ and so $c'(i) \sqsubset_{\mathcal{P}} d'(i)$.
- Otherwise $c.\textit{cnt}(i) > d.\textit{cnt}(i)$.
  - If $d.\textit{cnt}(i) = 0$, either i advances in the pipeline resulting in $c'.\textit{stage}(i) \sqsubset_{\mathcal{S}} d'.\textit{stage}(i)$ or i is stalled in d resulting in $d'.\textit{cnt}(i) = d.\textit{cnt}(i) = 0 \leq c.\textit{cnt}(i) - 1 = c'.\textit{cnt}(i)$.
  - If $d.\textit{cnt}(i) \neq 0$, by definition of cycle we conclude $c'.\textit{cnt}(i) = c.\textit{cnt}(i) - 1 < d.\textit{cnt}(i) - 1 = d'.\textit{cnt}(i)$.

Pipeline stage advance
We know that $c(i) \sqsubseteq_{\mathcal{P}} d(i)$, i.e. either $c(i) = d(i)$ or $c(i) \sqsubset_{\mathcal{P}} d(i)$. We consider the case $c(i) = d(i)$ first. As we are in the *pipeline stage advance* case, we know $c.\textit{ready}(i)$ and $c.\textit{willbefree}(c.\textit{stage}'(i))$. By using Lemma VIII.1, we get $d.\textit{ready}(i)$ and $d.\textit{willbefree}(c.\textit{stage}'(i))$. Consequently, we know that i will also advance its pipeline stage in d which results in $c'.\textit{stage}(i) = d'.\textit{stage}(i)$. In the second case, $c(i) \sqsubset_{\mathcal{P}} d(i)$, we know $c.\textit{stage}(i) \sqsubset_{\mathcal{S}} d.\textit{stage}(i)$ since we are in the *pipeline stage advance* case. By definition of $\textit{stage}'$, i can at most move to the consecutive stage and thus either $c'.\textit{stage}(i) \sqsubset_{\mathcal{S}} d'.\textit{stage}(i)$ or $c'\textit{stage}(i) = d'.\textit{stage}(i)$.

To conclude $c'(i) \sqsubseteq_{\mathcal{P}} d'(i)$, it remains to be proven that $c'.\textit{cnt}(i) \geq d'.\textit{cnt}(i)$ if $c'.\textit{stage}(i) = d'.\textit{stage}(i)$. Immediately after the pipeline advance, $c'.\textit{cnt}(i)$ is the latency determined by $\mathit{latency}(i)$ and thus $d'.\textit{cnt}(i)$ cannot be higher because the number of remaining cycles is never increased during cycle. □

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[2] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, 1977, pp. 238–252.

[3] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1–05:48, 2016.

[4] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, 1999, pp. 12–21.

[5] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[6] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL," in *WCET*, B. Lisper, Ed., vol. 15, Dagstuhl, Germany, 2010, pp. 101–112.

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

[8] T. Kelter, "WCET analysis and optimization for multi-core real-time systems," Ph.D. dissertation, TU Dortmund University, 2015.

[9] A. Schranzhofer, J.-J. Chen, and L. 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*, April 2010, pp. 215–224.

[10] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 339–349.

[11] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March 2010, pp. 741–746.

[12] D. Dasari, B. Andersson, V. Nelis, S. Petters, A. Easwaran, and J. Lee, "Response time analysis of COTS-based multicores considering the contention on the shared memory bus," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 1068–1075.

[13] 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.

[14] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. 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*, April 2011, pp. 213–222.

[15] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *EMSOFT*. ACM, 2012, pp. 63–72.

[16] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov, "A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets," *Real-Time Systems*, vol. 50, no. 5, pp. 736–773, 2014.

[17] W. Huang, J. Chen, and J. Reineke, "MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, pp. 158:1–158:6.

[18] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, Jul 2018.

[19] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, 2016, pp. 299–308.

[20] S. Hahn, J. Reineke, and R. Wilhelm, "Toward compact abstractions for processor pipelines," in *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, 2015, pp. 205–220.

[21] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *30th International IEEE Conference on Computer Design, ICCD 2012, Montreal, QC, Canada, September 30 - Oct. 3, 2012*. IEEE Computer Society, 2012, pp. 87–93.

[22] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*. IEEE, 2007, pp. 264–265.

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

[24] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995). La Jolla, California, June 21-22, 1995*, 1995, pp. 88–98.

[25] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, 1996, pp. 52–66.

[26] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

[27] S. Altmeyer, R. I. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, 2011, pp. 261–271.

[28] S. Altmeyer, R. I. Davis, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, 2015, pp. 129–138.

[29] L. Thiele and R. Wilhelm, "03471 abstracts collection – design of systems with predictable behaviour," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, ser. Dagstuhl Seminar Proceedings, L. Thiele and R. Wilhelm, Eds., no. 03471. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[30] ——, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[31] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, ser. Dagstuhl Seminar Proceedings, L. Thiele and R. Wilhelm, Eds., no. 03471. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[32] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. L. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Springer, 2002, pp. 334–348.

[33] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "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.

[34] T. Ungerer, F. J. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[35] T. Ungerer, C. Bradatsch, M. Frieb, F. Kluge, J. Mische, A. Stegmeier, R. Jahr, M. Gerdes, P. G. Zaykov, L. Matusova, Z. J. J. Li, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, N. Lay, D. George, I. Broster, E. Quiñones, M. Panic, J. Abella, C. Hernández, F. J. Cazorla, S. Uhrig, M. Rohde, and A. Pyka, "Parallelizing industrial hard real-time applications for the parMERASA multicore," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 3, pp. 53:1–53:27, 2016.

[36] M. Schoeberl, S. Abbaspour, B. Akesson, N. C. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture - Embedded Systems Design*, vol. 61, no. 9, pp. 449–471, 2015.

[37] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for patmos," in *17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014*. IEEE Computer Society, 2014, pp. 100–108.

[38] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *DATE*, 2014, pp. 1–6.

[39] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: bank privatization for predictability and temporal isolation," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, R. P. Dick and J. Madsen, Eds. ACM, 2011, pp. 99–108.

[40] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*. IEEE Computer Society, 2014, pp. 101–110.

[41] Micron Technology, Inc., *Automotive DDR SDRAM MT46V32M8, MT46V16M16, Available at* https://www.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf.

[42] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016, pp. 2:1–2:10.