Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems

Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand

***Abstract*—Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. Experience with the use of static timing-analysis methods and the tools based on them in the automotive and the aeronautics industries is positive. However, both the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform. In fact, the architecture determines whether a static timing analysis is practically feasible at all and whether the most precise obtainable results are precise enough. Results contained in this paper also show that measurement-based methods still used in industry are not useful for quite commonly used complex processors. This dependence on the architectural development is of growing concern to the developers of timing-analysis tools and their customers, the developers in industry. The problem reaches a new level of severity with the advent of multicore architectures in the embedded domain. This paper describes the architectural influence on static timing analysis and gives recommendations as to profitable and unacceptable architectural features.**

***Index Terms*—Memory hierarchy, pipelines, processor architecture, timing predictability.**

I. Introduction

THIS PAPER is concerned with architectures for embedded systems that are to be used in time-critical applications. These systems are subject to stringent timing constraints which are dictated by the surrounding physical environment. We assume that a real-time system consists of a number of tasks, which realize the required functionality. A schedulability analysis for this set of tasks on a given hardware has to be performed in order to guarantee that the timing constraints of these tasks will be met ("timing validation"). Existing techniques for schedulability analysis require upper (and lower) bounds on the execution times of the tasks to be known. These bounds are called worst-case execution time (WCET) and best-case execution time (BCET), respectively. These bounds have to be *safe*, i.e., they must never underestimate (overestimate) the real execution time. Furthermore, they should be *tight*, i.e., the overestimation (underestimation) should be as small as possible. Guarantees can be given by static-analysis-based methods, even for complex-processor architectures, which exhibit a huge execution-time variability and a strong dependence of the execution time on the initial execution state. These static methods are based on abstract architectural models, which are conservative with respect to timing behavior.

An alternative approach to static timing analysis is based on *measuring* execution times [1]–[3], essentially in two different ways. One way is to measure end-to-end times; the other is to measure execution times of basic blocks or other basic components of tasks and combine them to times or distributions of times for the whole program. However, measurement-based approaches have drawbacks. For complex processors, execution times often cannot be measured for all possible initial states and inputs, neither end-to-end nor piecewise: Only a small subset of initial states and inputs can usually be covered. Therefore, measurement is usually not guaranteed to be sound, i.e., to compute upper bounds on the WCET. It has been observed that, for complex processors, such estimates are also not really precise [4], and the combinators for the so-called execution-time profiles can introduce large pessimism, since they do not exploit context and flow information [5].

Based on experience with static timing analysis in the embedded-systems industry [6], [7] and theoretical insights [8]–[10], we give advice concerning future computer architectures for time-critical systems.

A. Architectural Influence

The efficiency, even the feasibility, and the precision of the results of timing analysis highly depend on the architecture. An essential criterion for future target architectures will be the analyzability of the performance. Architecture without accompanying performance-analysis technology should not be seriously considered for time-critical embedded applications.

The decisive criteria are as follows.

1) Soundness: Without this, no reliable guarantee can be derived.
2) Obtainable precision, which depends on the predictability properties of the architecture.
3) Analysis effort to reach this precision, which depends on the size of the state space to be explored.

Here are some examples concerning precision and efficiency: A cache with a random-replacement strategy does not allow

Manuscript received June 6, 2008; revised August 25, 2008. Current version published June 17, 2009. This work was supported in part by the European Community's Seventh Framework Program FP7/2007-2013 under Grant Agreement 216008 (Predator), by AVACS, Deutsche Forschungsgemeinschaft, and by the European Networks of Excellence ARTIST2 and ARTIST DESIGN. This paper was recommended by Associate Editor M. Di Natale.

R. Wilhelm is with Saarland University, 66123 Saarbrücken, Germany, and also with Leibniz Center for Computer Science, Schloss Dagstuhl, Saarland, Germany (e-mail: wilhelm@cs.uni-sb.de).

D. Grund and J. Reineke are with Saarland University, 66123 Saarbrücken, Germany.

M. Schlickling and M. Pister are with Saarland University, 66123 Saarbrücken, Germany, and also with AbsInt Angewandte Informatik GmbH, 66123 Saarbrücken, Germany.

C. Ferdinand is with AbsInt Angewandte Informatik GmbH.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2013287

for a cache analysis with good precision. An overly complex pipeline leads to an explosion in the size of the state space, which may be too large for any practical application. An out-of-order pipeline supporting a high degree of parallelism allows for many different interleavings of a sequential instruction stream, possibly with different effects on the caches, different collisions on buses, and, in the end, different execution times. The design of the internal and external buses, if done wrongly, leads to hardly analyzable behavior and great loss in precision. Multicore architectures with shared caches, finally, will create a space of interleavings of interactions on these caches that will make sound and precise timing analysis practically infeasible. Quickly, the search space for such analyses becomes too large to be completely explored.

Most of the problems posed to timing analysis are caused by the *interference on shared resources*. Resources are shared for cost, energy, and performance reasons. Different users of a shared resource may often access the resource in a statically unknown way. Different access sequences may result in different states of the resource. The different sequences may already exhibit different execution times, and the resulting resource states may again cause differences in the future timing behavior.

Interferences may be of two kinds, *inherent* and *virtual*. These two types of interferences cause two types of nondeterminism: Inherent interferences cause real nondeterminism and virtual interferences cause artificial nondeterminism. Both are detrimental for predictability.

Inherent interferences on a shared resource observed by one user of the resource happen at unpredictable times through an activity of another user of the resource. Examples of shared resources with inherent interferences are buses and memory: Buses are used by several masters, which may access the buses in unpredictable ways. Memory and caches are shared between several processors or cores. One thread executed on one core does not know when accesses by another thread on another core will happen.

Virtual interferences are introduced by the unavoidable abstraction of the architecture. An out-of-order processor will execute an instruction stream in one (deterministic) way. Exhaustive exploration could, in principle, identify this one way for each input and initial state; in practice, this is infeasible. Thus, this order is assumed to be statically unknown. This forces the analysis to consider all possibilities. Different possibilities may have different effects on the cache contents. Hence, inherent interferences occur due to the nondeterminism in the execution; virtual interferences are caused by the artificial nondeterminism introduced by abstraction. Limiting both types of interferences must be a high-priority design goal. One first principle for architecture design is to strive for a good compromise between cost, performance, and predictability *where the sharing or duplication of resources is concerned.*

B. Structure of this Paper

Section II gives an overview of static timing analysis and presents the main challenges. The remainder of this paper discusses architectural features and their influence on the precision and efficiency of timing analysis. Section III describes adversarial properties of modern pipelines, which make timing analysis inefficient or imprecise. Section IV describes our results about the predictability and the sensitivity of caches and draws consequences for the use of caches in hard real-time systems. Section V considers the influence of the bus properties on timing analysis, in particular, the effects of the competition between several bus masters, of pipelined accesses over the bus, and of mismatching bus frequencies. Section VI gives an outlook into future multicore embedded architectures and makes a proposal of a predictable multicore architecture. Section VIII finally lists recommendations as to which architectural features to prefer and which to avoid.

II. Static Timing Analysis

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. This paper treats timing analysis of tasks with uninterrupted execution. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact WCETs and BCETs. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information and, thus, are in part responsible for the gap between WCETs and upper bounds and between BCETs and lower bounds. How much is lost depends both on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software. The methods used to determine upper and lower bounds are very similar.

In modern microprocessor architectures, caches, pipelines, and all kinds of speculation are key features for improving (average case) performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions and, thus, the contribution to the program's execution time can vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases: 1) The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e., all operands are ready, no resource conflicts with other currently executing instructions exist. 2) "Everything goes wrong," i.e., instruction and/or operand fetches miss the cache, resources needed by the instruction are occupied, etc.We will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases as the *timing penalty* of this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters a timing accident depends on the execution state, e.g., the contents of the cache(s) and the occupancy of other resources, and, thus, on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history.

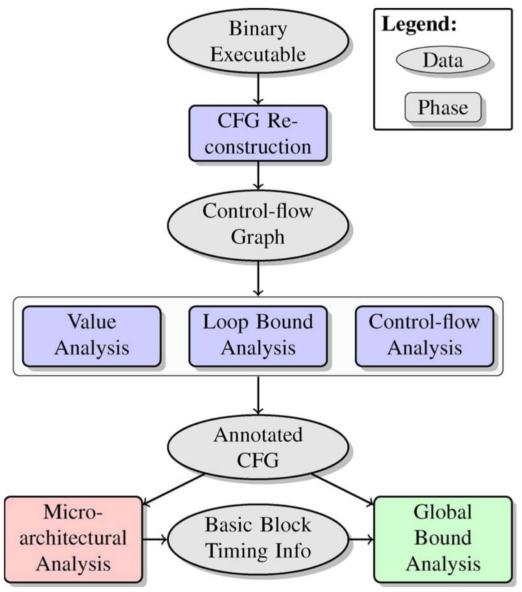


Fig. 1. Main components of a timing-analysis framework and their interaction.

A. Timing-Analysis Framework

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged [11]–[13]. Fig. 1 shows a general view on this architecture. First, one can distinguish three major building blocks:

1) control-flow reconstruction and static analyses for control and data flow;
2) microarchitectural analysis, which computes upper and lower bounds on execution times of basic blocks;
3) global bound analysis, which computes upper and lower bounds for the whole program.

The following list presents the individual phases and describes their objectives and problems. Note that the first four phases are part of the first building block.

1) *Control-flow reconstruction* [14] takes a binary executable to be analyzed, reconstructs the program's control flow, and transforms the program into a suitable intermediate representation. Problems encountered are dynamically computed control-flow successors, e.g., stemming from switch statements, function pointers, etc.
2) *Value analysis* [15], [16] computes an overapproximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. This information is, among others, used for a precise data-cache analysis.
3) *Loop bound analysis* [17], [18] identifies loops in the program and tries to determine bounds on the number of loop iterations, information which is indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop-exit conditions, as well as dependencies in nested loops.
4) *Control-flow analysis* [17], [19] narrows down the set of possible paths through the program by eliminating infeasible paths or to determine correlations between the number of executions of different blocks using the results of value-analysis results. These constraints will tighten the obtained timing bounds.
5) *Microarchitectural analysis* [10], [20], [21] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, taking into account the processor's pipeline, caches, and speculation concepts. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc. Ignoring these average-case-enhancing features would result in imprecise bounds.
6) *Global bound analysis* [22], [23] finally determines bounds on execution time for the whole program. Information about the execution time of basic blocks is combined to compute the shortest and the longest paths through the program. This phase takes into account information provided by the loop bound and control-flow analyses.

The commercially available tool `aiT` by AbsInt, cf. http://www.absint.de/wcet.htm, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [6], [7], [10], [24].

III. Pipelines

For nonpipelined architectures, one can simply add up the execution times of individual instructions to obtain a bound on the execution time of a basic block. Pipelines increase performance by overlapping the executions of different instructions. Hence, a timing analysis cannot consider individual instructions in isolation. Instead, they have to be considered collectively—together with their mutual interactions—to obtain tight timing bounds.

The analysis of a given program for its pipeline behavior is based on an abstract model of the pipeline. All components that contribute to the timing of instructions have to be modeled conservatively. Depending on the employed pipeline features, the number of states the analysis has to consider varies greatly.

A. Contributions to Complexity

Since most parts of the pipeline state influence timing, the abstract model needs to closely resemble the concrete hardware. The more performance-enhancing features a pipeline has, the larger is the search space. Superscalar and out-of-order executions increase the number of possible interleavings. The larger the buffers (e.g., fetch buffers, retirement queues, etc.), the longer the influence of past events lasts. Dynamic branch prediction, cachelike structures, and branch history tables increase history dependence even more.

All these features influence execution time. To compute a precise bound on the execution time of a basic block, the analysis needs to exclude as many *timing accidents* as possible. Such

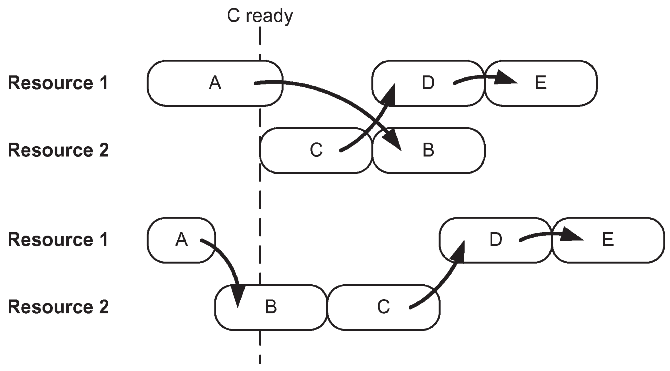


Fig. 2. Scheduling anomaly.

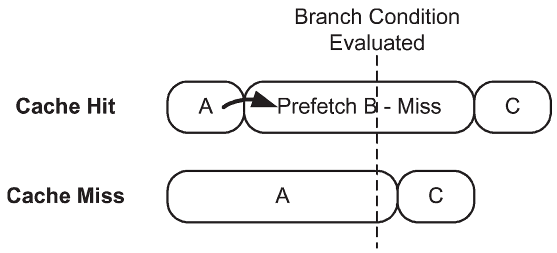


Fig. 3. Speculation anomaly. A and B are prefetches. If A hits, B can also be prefetched and might miss the cache.

accidents are data hazards, branch mispredictions, occupied functional units, full queues, etc.

Abstract states may lack information about the state of some processor components, e.g., caches, queues, or predictors. Transitions of the pipeline may depend on such missing information. This causes the abstract pipeline model to become nondeterministic, although the concrete pipeline is deterministic. When dealing with this nondeterminism, one could be tempted to design the WCET analysis such that only the locally most-expensive pipeline transition is chosen. However, in the presence of *timing anomalies* [8], [25], this approach is unsound. Thus, in general, the analysis has to follow all possible successor states.

B. Timing Anomalies and Domino Effects

The notion of *timing anomalies* was introduced by Lundqvist and Stenström in [25]. In the context of WCET analysis, Reineke *et al.* [8] present a formal definition. Intuitively, a timing anomaly is a situation where the local worst case does not contribute to the global worst case. For instance, a cache miss—the local worst case—may result in a globally shorter execution time than a cache hit because of scheduling effects (see Fig. 2 for an example). Shortening instruction A leads to a longer overall schedule, because instruction B can now block the "more" important instruction C. Analogously, there are cases where a shortening of an instruction leads to an even greater decrease in the overall schedule.

Another example occurs with branch prediction. A mispredicted branch results in unnecessary instruction fetches, which might miss the cache. In case of cache hits, the processor may fetch more instructions. Fig. 3 shows this.

A system exhibits a *domino effect* [25] if there are two hardware states s, t such that the difference in execution time (of the same program starting in s and t, respectively) may be arbitrarily high, i.e., cannot be bounded by a *constant*. For example, given a program loop, the executions never converge to the same hardware state, and the difference in execution time increases in each iteration. The existence of domino effects is undesirable for timing analysis. Otherwise, one could safely discard states during the analysis and make up for it by adding a predetermined constant.

Unfortunately, domino effects show up in real hardware. In [26], Schneider describes a domino effect in the pipeline of the PowerPC 755. Another example is given by Berg [27] who considers the pseudo-least-recently used (PLRU)-replacement policy of caches. In Section IV, we will present sensitivity results of replacement policies, which quantify the maximal extent of domino effects in caches, i.e., by determining the maximal *factor* by which the cache performance may vary.

C. Classification of Architectures

Architectures can be classified into three categories, depending on whether they exhibit timing anomalies or domino effects.

1) **Fully timing compositional architectures**: The (abstract model of) an architecture does not exhibit timing anomalies. Hence, the analysis can safely follow local worst-case paths only. One example for this class is the ARM7. The ARM7 allows for an even simpler timing analysis. On a timing accident, all components of the pipeline are stalled until the accident is resolved. Hence, one could perform analyses for different aspects (e.g., cache, bus occupancy) separately and simply add all timing penalties to the BCET.
2) **Compositional architectures with constant-bounded effects**: These exhibit timing anomalies but no domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local nonworst-case paths by adding a constant number of cycles to the local worst-case path. The Infineon TriCore is assumed, but not formally proven, to belong to this class.
3) **Noncompositional architectures**: These architectures, e.g., the PowerPC 755, exhibit domino effects and timing anomalies. For such architectures, timing analyses always have to follow all paths, since a local effect may influence the future execution arbitrarily.

IV. Caches

Caches are employed to hide the latency gap between memory and CPU by exploiting locality in memory accesses. On current architectures, a cache miss may take several hundred of CPU cycles. Therefore, the cache performance has a strong influence on a system's overall performance.

To obtain tight bounds on the execution time of a task, timing analyses *must* take into account the cache architecture. The precision of a cache analysis is strongly dependent on the predictability of the cache architecture, particularly on its replacement policy. LRU replacement has the best predictability properties of all replacement policies. It yields more precise and more efficient timing analysis. Employing other policies, like PLRU or FIFO, yields less precise WCET bounds, because fewer memory accesses can be classified as hits. Timing analyses having to deal with other policies than LRU are also less efficient: As fewer memory accesses can be classified as hits or misses, more possibilities have to be explored. If the architecture features *timing anomalies*, it is not safe to assume unclassified memory accesses to be misses.

It has been argued before that the execution time cannot be measured for all possible initial states and inputs. Therefore, if worst-case input and initial state are unknown, measurement is not guaranteed to be sound. We investigate the influence of the replacement policy on the soundness of measurement-based timing analysis. Our analysis reveals that measurement-based methods may strongly underestimate the number of misses for FIFO and PLRU and, therefore, yield WCET estimates that are dramatically wrong.

Describing our investigations needs some basic cache notions.

A. Basic Cache Notions

Caches are very fast but small memories that store a subset of the main memory's contents. To reduce traffic and management overhead, main memory is logically partitioned into a set of *memory blocks* B of size b bytes. Memory blocks are cached as a whole in cache lines of equal size. Usually, b is a power of two. This way, the block number is determined by the most significant bits of a memory address.

When the CPU accesses a memory block, the cache logic has to determine whether the memory block is stored in the cache (cache hit) or not (cache miss). To enable an efficient lookup, each memory block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally sized cache sets. The size of a cache set is called the *associativity* k of the cache. The number of such equally sized cache sets s is usually a power of two, such that the set number is determined by the least significant bits of the block number, the *index*. The remaining bits, known as the *tag*, are stored along with the data to finally decide, whether and where a memory block is cached within a set. One can distinguish caches by the type of addresses that are being used to *index* and to *tag* the cache. Addresses could either be physical or virtual. In real-time embedded systems such as the ones we are considering, there is no virtual memory. Virtual memory would introduce many additional challenges for timing analysis, like a TLB analysis.

Since the number of memory blocks that map to a set is far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. To facilitate useful replacement decisions, a number of status bits is maintained that store information about previous accesses. For lack of space, we only consider here replacement policies that have independent status bits per cache set. For example, pseudoround-robin cache-replacement schemes that share the status bits over all sets show very badly predictable behavior [7].

Let us briefly explain the three commonly used families of replacement policies under investigation.

LRU replacement conceptually maintains a queue of length k for each cache set, where k is the associativity of the cache. If an element (a memory block) is accessed that is not in cache (a miss), it is placed at the front of the queue. The last element of the queue, the LRU element, is then removed if the set is full. At a cache hit, the element is moved from its position in the queue to the front, in this respect, treating hits and misses equally. LRU replacement is used in the FREESCALE PPC603E core and the MIPS 24 K/34 K.

FIFO cache sets can also be seen as queues: New elements are inserted at the front, evicting elements at the end of the queue. In contrast to LRU, hits do not change the queue. FIFO is used in the INTEL XSCALE and some ARM9- and ARM11-based processor cells.

PLRU is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k - 1$ "tree bits" pointing to the line to be replaced next (for an in-detail explanation of PLRU, consider [9], [28]). It is used in the POWERPC 75x and the Intel PENTIUM II–IV.

Fig. 4 shows domains and notations used throughout this paper. $m_P(q, s)$ computes the number of misses incurred by replacement policy P conducting access sequence s in state q. Likewise, $h_P(q, s)$ computes the number of hits by replacement policy P conducting access sequence s in state q. $\text{update}_P(q, s)$ computes the cache state after accessing sequence s starting from state q. Given a cache-state q, $CC_P(q)$ returns the set of memory blocks contained in q.

B. Limits on the Precision of Static Cache Analysis

An important part of a static timing analysis is its cache analysis, which tries to classify memory accesses as hits or misses. Due to the presence of timing anomalies, memory accesses that cannot be safely classified as a hit or a miss have to be conservatively accounted for by considering both possibilities. The more of the cache hits that occur during program execution can be statically classified as such, the tighter will be the computed upper bound on the execution time. Conversely, every statically classified cache miss tightens the computed lower bound.

To classify memory accesses, cache analyses compute *may* and *must* information.[1] *May* and *must* caches at a program point are upper and lower approximations, respectively, to the contents (sets of tags) of all concrete caches that occur whenever program execution reaches this program point. *Must* information is used to safely classify memory accesses as cache hits. The complement of the *may* information is used to safely predict cache misses.

[1]Some analyses explicitly represent *may* and *must* information. Other cache analyses do not. Still, they implicitly maintain *may* and *must* information in one way or the other.

a, b, c	$\in$	B	the set of memory blocks
s	$\in$	$S = B^*$	the set of finite access sequences
s	$\in$	$S^{\neq} \subset S$	the set of finite access sequences with pairwise different accesses
p	$\in$	C^P	the set of cache-set states of policy P
$m_P(q, s)$	:	$C^P \times S \to \mathbb{N}$	number of misses incurred by policy P conducting s starting from state q
$h_P(q, s)$	:	$C^P \times S \to \mathbb{N}$	number of hits by policy P conducting s starting from state q
$update_P(q, s)$	:	$C^P \times S \to C^P$	cache-set state after accessing a sequence s starting from state q
$CC_P(q)$	:	$C^P \to 2^B$	the set of memory blocks of a cache-set state of policy P

Fig. 4. Domains and notations.

There are several reasons for uncertainty about cache contents.

1) Static cache analyses usually cannot make any assumptions about the initial cache contents. These depend on previously executed tasks. Even assuming a completely empty cache may not be conservative as shown in [27] and by our sensitivity analyses [29], which will be described in the following section. To be conservative, the analysis has to account for all possible cache states. Therefore, in general, the only safe initial *must* cache is the empty set, whereas the only safe initial *may* cache must contain every memory block that may be mapped to the particular cache set.
2) At control-flow joins, analysis information about different paths needs to be safely combined. Intuitively, one must take the intersection of the incoming *must* information and the union of the incoming *may* information. A memory block can only be in the *must* cache if it is in the *must* caches of all predecessor control-flow nodes, correspondingly, for *may* caches.
3) If the analysis cannot exactly determine the address of a memory access, it must conservatively account for all possible addresses. This particularly deteriorates *may* information. In instruction cache analysis, these addresses are obvious. In data cache analysis, it may be very difficult to precisely determine the accessed memory address.

Since information about the cache state may thus be unknown or lost, it is important that analyses (re)gain information quickly to be able to classify memory accesses safely as cache hits or misses. Fortunately, this is possible for most caches. However, the speed of this (re)gaining process greatly depends on the cache-replacement policy employed and influences uncertainty about cache hits and misses.

We study how quickly *may* and *must* information can be built up from a completely unknown cache state by observing a sequence of *pairwise different* memory accesses. This is sensible because recurring accesses do not contribute additional information about cache contents.

May and *must* information available after observing an access sequence s without knowing the initial set state can be defined as follows, where CC_P and update$_P$ are shown in Fig. 4:

$$May_P(s) := \bigcup_{q \in C^P} CC_P\left(\text{update}_P(q, s)\right)$$

$$Must_P(s) := \bigcap_{q \in C^P} CC_P\left(\text{update}_P(q, s)\right).$$

$May_P(s)$ is the set of cache contents that may still be in the cache set after accessing the sequence s, regardless of the initial cache state. Analogously, $Must_P(s)$ is the set of cache contents that must be in the cache set after accessing the sequence s. Since we take into account every initial state, $Must_P(s)$ is always a subset of the contents of the sequence s.

The following two definitions show *how much may* and *must* information is available after observing any access sequence s of length n if no knowledge of the initial cache state is available

$$may_P(n) := |May_P(s)|, \qquad \text{where } s \in S^{\neq}, |s| = n$$

$$must_P(n) := |Must_P(s)|, \qquad \text{where } s \in S^{\neq}, |s| = n.$$

Note that $may_P(n)$ and $must_P(n)$ are well defined: For all sequences s of length n, $|May_P(s)|$ is equal (the same goes for $|Must_P(s)|$). The sequences contain pairwise different accesses only and are thus equal up to renaming. Thus, $May_P(s_1)$ equals $May_P(s_2)$ also up to renaming.

Results: We have determined $may(n)$ and $must(n)$ for eight-way associative LRU, FIFO, and PLRU, by exhaustively generating all successor states of all possible initial cache-set states, exploiting symmetries. Fig. 5 shows the results.

Under LRU replacement, eight accesses suffice to gain full knowledge of the cache contents of an eight-way cache set: $must_{\text{LRU}(8)}(8) = may_{\text{LRU}(8)}(8) = 8$. This marks a limit for *any* replacement policy, as it is impossible to gain full knowledge of the cache set's contents with less than eight accesses for an eight-way cache set.

Under PLRU or FIFO replacement, it takes considerably longer to gain *may* and *must* information: For PLRU, it takes 13 accesses to obtain any *may* information and another 6 accesses to regain full knowledge of the cache set's contents. In the case of FIFO, the situation is even worse: Only after 15 accesses do we obtain any *may* information and another 8 accesses are necessary for full knowledge. In addition, only one of the eight cached elements can be classified as such, as $must_{\text{FIFO}(8)}(n) = 1$ for $n \leq 16$.

One can identify two milestones in the evolution of *may* and *must* information.

1) The point at which *may* information becomes available. We call this evict$_P(k)$, where P denotes the policy and k the associativity. At this point, all of the unknown previous content must have been *evicted* from the cache.
2) The point at which full *may* and *must* information is obtained, i.e., $may(n) = must(n) = k$, where k is the associativity of the cache. We call this fill$_P(k)$.

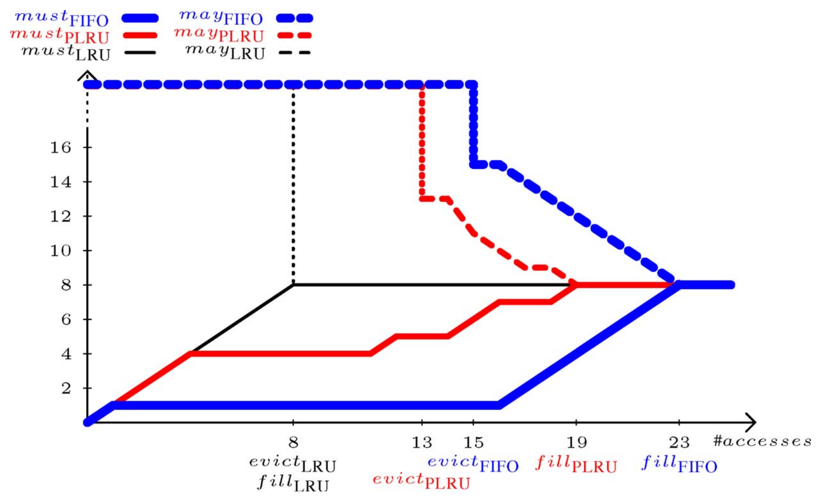


Fig. 5. Evolution of *may* and *must* information of eight-way LRU/PLRU/FIFO cache sets, i.e., $may_{\mathrm{LRU}(8)}(n)$, $must_{\mathrm{LRU}(8)}(n)$, $may_{\mathrm{FIFO}(8)}(n)$, $must_{\mathrm{FIFO}(8)}(n)$, and $may_{\mathrm{PLRU}(8)}(n)$, $must_{\mathrm{PLRU}(8)}(n)$.

TABLE I
EVICT AND FILL FOR LRU, PLRU, AND FIFO

Policy	$evict(k)$	$fill(k)$
LRU	k	k
FIFO	$2k-1$	$3k-1$
PLRU	$\frac{k}{2}\log_2 k + 1$	$\frac{k}{2}\log_2 k + k - 1$

TABLE II
EVICT(4), FILL(4) AND EVICT(8), FILL(8) FOR LRU, PLRU, AND FIFO

Policy	$evict(4)$	$fill(4)$	$evict(8)$	$fill(8)$
LRU	4	4	8	8
FIFO	7	11	15	23
PLRU	5	7	13	19

For $k = 8$, the respective points are shown in Fig. 5. Our tool to generate *may* and *must* curves is limited to fixed associativities. In [9], we analytically obtained evict$_P(k)$ and fill$_P(k)$ formulas in terms of the associativity k for the three policies. They are shown in Table I. Instantiations of these formulas for the typical associativities four and eight are shown in Table II.

It is important to stress that $may(n)$, $must(n)$, evict, and fill limit the precision of *any* cache analysis. In particular, they are independent of whether the analysis is based on approximations of *may* and *must* information or not. For example, no cache miss can be safely predicted after less than evict memory accesses if the analysis begins with no information. LRU stands out, delivering the best possible results under the regarded measures for any possible replacement policy. Cache analyses of PLRU and in particular FIFO will be able to classify considerably less accesses as hits or misses than good analyses of LRU, as [10]. If caches need to be employed, the best pick is definitely LRU, which is often argued to be too expensive. Ackland *et al.* [30] have shown that LRU replacement can be implemented with a one-cycle update up to associativity 16.

In general, alternatives to caches, like scratchpad memory, should be considered in the design of timing-critical systems. They are advantageous in systems with little change, in memory-access patterns, e.g., systems without interrupts and preemptions. Marwedel *et al.* [31] shows that the performance of code using scratchpad memory can be predicted with very high accuracy.

C. Sensitivity to Initial State—Measurement-Based WCET Analysis in the Presence of Caches

The introduction has briefly described measurement-based methods for timing analysis. Relatively simple architectures without any performance-enhancing features like pipelines, caches, etc., exhibit the same timing independently of the initial state. For such architectures, measurement-based timing analysis is sound [3]. Wenzel [3] and Deverge and Puaut [32] propose to lock the cache contents [33], [34] and to flush the pipeline at program points where measurement starts. This is not possible on all architectures, and it also has a detrimental effect on both the average-case execution times and the WCETs of tasks. We study whether measurement-based timing analysis can be performed in the presence of "unlocked" caches. To this end, we introduce the notion of sensitivity of a cache-replacement policy.

Sensitivity: Measurement-based approaches to WCET computation execute fragments of the task on a subset of the possible initial states and inputs. An important part of the initial state is the initial cache state, i.e., the cache's contents and status bits. Depending on the initial cache state, the task will incur different numbers of misses, resulting in different execution times.

We have investigated how sensitive the execution time of a task is toward the initial cache state, i.e., how strongly the initial cache state may influence the number of hits and misses for a given replacement policy.

Let $m_P(q, s)$ be the number of misses incurred by replacement policy P on access sequence s and initial state q as previously shown in Fig. 4. Then, we can define the sensitivity of a replacement policy similarly to competitiveness [35]. In contrast to competitiveness, where a replacement policy competes with the optimal offline policy, a policy competes with itself on different initial states.

Definition 1 (Miss Sensitivity to State): A policy P is k-miss sensitive with additive constant c, short (k, c)-miss sensitive if

$$m_P(q, s) \leq k \cdot m_P(q', s) + c$$

for all access sequences $s \in S$ and all states $q, q' \in C^P$.

If policy P is (k, c)-miss sensitive, it will incur at most k times the number of misses plus a constant c on any access sequence starting in state q instead of some other state q'. Assume P is (3, 2)-miss sensitive. If P incurs eight misses on some access sequence s and state q, it will not incur more than $8 \cdot 3 + 2 = 26$ misses on the same access sequence s and any state q'.

Hit sensitivity can be defined similarly, where $h_P(q, s)$ is the number of hits by replacement policy P on access sequence s and initial state q.

Definition 2 (Hit Sensitivity to State): A policy P is k-hit sensitive with subtractive constant c, short (k, c)-hit sensitive, if

$$h_P(q, s) \geq k \cdot h_P(q', s) - c$$

for all access sequences $s \in S$ and all states $q, q' \in C^P$.

Assume P is (1/2, 1)-hit sensitive. If P results in 12 hits on some access sequence s and state q, it will result in at least $12 \cdot (1/2) - 1 = 5$ hits on the same access sequence s and any state q'.

Notice that the two definitions are not redundant. If policy P is k-miss sensitive with $k > 1$, this does not give us any clue regarding P's hit-sensitivity: Rewriting Definition 1 in terms of hits $(h_P(q, s) = |s| - m_P(q, s))$ yields $|s| - h_P(q, s) \leq k \cdot (|s| - h_P(q', s)) + c$. For $k > 1$, this inequality depends on $|s|$, the length of the access sequence.

We sometimes say that a policy is k sensitive without specifying an appropriate additive (subtractive) constant. In such cases, we implicitly demand that there is a constant c such that the policy is (k, c) sensitive. The following definition is an example of such a case.1

Definition 3 (Sensitive Ratio): The sensitive miss and hit ratios s_P^m and s_P^h of P are defined as

$$s_P^m = inf\ \{k \mid P\ is\ k\text{-}miss\ sensitive\}$$
$$s_P^h = sup\ \{k \mid P\ is\ k\text{-}hit\ sensitive\}.$$

Our focus will be on computing these sensitive ratios and appropriate additive (subtractive) constants. Why are we interested in sensitive ratios? Consider a policy that is k-miss sensitive. It is also l-miss sensitive for $l > k$. However, the former statement is clearly a better characterization of the policy's sensitivity. In this sense, the sensitive ratio is the *best* characterization of the policy's sensitivity. In particular, there are access sequences such that the ratio between the number of misses (hits) in one state and the number of misses (hits) in another state approaches the sensitive ratio in the limit. Every policy is by definition zero-hit sensitive. However, a policy may not be k-miss sensitive for any k. In that case, we will call it ∞-miss sensitive. For a policy that is ∞-miss sensitive, the number of misses starting in one state cannot be bounded by the number of misses starting in another state.

	2	3	4	5	6	7	8
LRU	$1, 2$	$1, 3$	$1, 4$	$1, 5$	$1, 6$	$1, 7$	$1, 8$
FIFO	$2, 2$	$3, 3$	$4, 4$	$5, 5$	$6, 6$	$7, 7$	$8, 8$
PLRU	$1, 2$	$-$	∞	$-$	$-$	$-$	∞

(a)

	2	3	4	5	6	7	8
LRU	$1, 2$	$1, 3$	$1, 4$	$1, 5$	$1, 6$	$1, 7$	$1, 8$
FIFO	0	0	0	0	0	0	0
PLRU	$1, 2$	$-$	$\frac{1}{3}, \frac{5}{3}$	$-$	$-$	$-$	$\frac{1}{11}, \frac{19}{11}$

(b)

Fig. 6. Miss- and hit-sensitivity results. As an example of how this should be read, PLRU(4) is (1/3, 5/3)-hit sensitive. ∞ indicates that a policy is not k-miss sensitive for any k. PLRU is only defined for powers of two. (a) Miss-sensitive ratios k and additive constants c for policies FIFO, PLRU, and LRU. (b) Hit-sensitive ratios k and subtractive constants c for policies FIFO, PLRU, and LRU.

To this end, we have built a tool that takes a concise description of a replacement policy and computes sensitive hit and miss ratios with according additive constants (details about the tool can be found in [29]).

Results: Using our tool, we have obtained sensitivity results for the widely used policies LRU, FIFO, and PLRU at associativities ranging from two to eight. Note that we have computed the precise *sensitive ratios*. For instance, there are arbitrarily long access sequences and pairs of initial states that exhibit the computed hit and miss ratios (for a detailed explanation of how the results have been obtained automatically, see [29]).

Fig. 6(a) shows our results for the miss sensitivity of LRU, FIFO, and PLRU. LRU is very insensitive to its state. The difference in misses is bounded by the associativity k. Hence, there can be no cache domino effects for LRU. No policy can do better, as the initial states q and q' may have completely disjoint contents.

FIFO and PLRU are much more sensitive to their state than LRU. Depending on its state, FIFO (k) may have up to k times as many misses. At associativity 2, PLRU and LRU coincide. For greater associativities, the number of misses incurred by a sequence s starting in state p cannot be bounded the number misses incurred by the same sequence s starting in another state q.

As the number of misses may only differ by a constant for LRU, the number of hits may only differ by the same constant. For FIFO, the situation is different: Starting in one state, the number of hits cannot be bounded by the number of hits starting in another state, on the same access sequence. The results for PLRU are only slightly more encouraging than in the miss-sensitivity case. At associativity eight, a sequence may incur only 1/11 of the number of hits depending on the starting state (see Fig. 6(b) for the analysis results).

Summarizing, both FIFO and PLRU may in the worst-case be heavily influenced by the starting state. LRU is very robust in that the number of hits and misses is affected in the least possible way.

One could argue that it is still safe to assume an empty cache as the starting state, assuming that an empty cache was worse than any nonempty cache. This is *not* true for FIFO and PLRU. We have performed a second analysis that fixed the reference starting state (q' in the definitions) to be empty. The analysis revealed the same sensitive ratios as in the general case with slightly smaller additive (subtractive) constants. It has been observed earlier [27], that the empty cache is not necessarily the worst-case starting state. This paper demonstrates to which extent it may be better than the real worst-case initial state.

Impact of Results on Measurement-Based Timing Analysis: We will try to illustrate on a simplified scenario the impact of the sensitivity results on measured execution times.

To this end, we adopt a simple model of execution time in terms of cache performance of Hennessy and Patterson [36]. In this model, the execution time is the product of the clock cycle time and the sum of the CPU cycles (the pure processing time) and the memory stall cycles

$$\text{Exec. time} = (\text{CPU cycles} + \text{Mem. stall cycles}) \times \text{Clock cycle.}$$

The equation makes the simplifying assumption that the CPU is stalled during a cache miss. Furthermore, it assumes that the CPU clock cycles include the time to handle cache hits.

Let CPI_{hit} be the average number of cycles per instruction if no cache misses occur. Then, the CPU cycles are simply a product of the number of instructions IC and CPI_{hit}

$$\text{CPU cycles} = \text{IC} \times \text{CPI}_{\text{hit}}.$$

The number of memory stall cycles depends on the number of instructions IC, the number of misses per instruction, and the cost per miss, the miss penalty

$$\begin{aligned}\text{Mem. stall cycles} &\\ &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss Pen.}\end{aligned}$$

Now, assume we have measured an execution time of T_{meas} in a system with a four-way set-associative FIFO cache. By which factor may the "real" WCET T_{wc} differ from T_{meas} due to different initial states of the cache? Let the number of memory accesses per instruction be 1.2,[2] and let the miss penalty be 50. Due to pipeline stalls, let the CPI_{hit} be 1.5. Further assume that the miss rate $\text{Miss rate}_{\text{meas}}$ during the measurement was 5%. The sensitive miss ratio of FIFO(4) is four. Neglecting the additive constant, the worst-case miss rate Miss rate_{wc} could thus be as high as 20%. Plugging the earlier assumptions into the equations and simplification yields

$$\begin{aligned}\frac{T_{wc}}{T_{\text{meas}}} &\\ &= \frac{\text{CPI}_{\text{hit}} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{wc} \times \text{Miss Pen.}}{\text{CPI}_{\text{hit}} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{\text{meas}} \times \text{Miss Pen.}}\\ &= \frac{1.5+1.2\times0.20\times50}{1.5+1.2\times0.05\times50} = \frac{13.5}{4.5} = 3.\end{aligned}$$

Therefore, in the example of a four-way set-associative FIFO cache, the WCET may be a factor of three higher than the measured time only due to the influence of the initial cache state. If PLRU were used as a replacement policy, the difference could be even greater. As measurement usually does not allow to determine the miss rate (or simply the number of misses), it is not even possible to add a conservative overhead to the measured execution times to account for the sensitivity to the initial state.

The earlier analysis considers the impact of cache sensitivity on an individual measurement. Measurement-based timing analysis as described in the literature [1]–[3], [32] does not advocate end-to-end measurements. Instead, measurements of program fragments are performed and later combined to obtain an estimate of the WCET of the whole program. The earlier arguments apply to any of the measurements of program fragments. If the measurement of an important fragment like the body of an inner loop is far off, the estimate for the whole program will also, as a consequence, be far off.

Conclusion: Sensitivity analysis revealed that, in the case of PLRU and FIFO, measurement-based WCET approaches *may* be dramatically wrong if measurement is initiated with the wrong initial states. In particular, starting with the empty cache is not safe in general, the notable exception being LRU.

D. Further Cache-Related Issues

There are further properties of caches that influence the precision of timing analysis. Unified caches as opposed to separated data and instruction caches introduce interference between instruction and data accesses. A superscalar out-of-order processor may execute an instruction stream in many different orders resulting in many different access sequences and, consequently, in less precise information about cache contents.

Write-back regimes in combination with replacement schemes that do not allow precise *must* and *may* analyses on noncompositional architectures may offer considerable problems. A so-called dirty cache line is only written back to memory when the line is replaced in the cache. Without *may* information, it is not possible to demonstrate the absence of a replacement of a dirty line. For noncompositional architectures, this means that a potential replacement must be taken into account for each potential data-cache miss. This is usually very pessimistic.

[2]Each instruction causes one instruction fetch and possibly data fetches.

V. Buses

A bus is a subsystem for transferring data between different components inside a computer, between a computer and its peripheral devices, or between different computers. In contrast to point-to-point connections, a bus logically connects several peripherals over the same set of wires using a dedicated protocol. In general, buses can be classified by the involved components, e.g., system buses like the *60x-bus* [37] on the PowerPC, memory buses connecting the memory controller with the memory slots, internal computer buses like Peripheral Component Interconnect (*PCI*), and external computer buses like *CAN* or *FlexRay*.

Subsequently, we describe some properties of buses that have an influence on the system's timing predictability.

In general, buses are clocked with a lower frequency than the CPU. For example, the clock ratio of the PCI bus is specified to 33 MHz in Rev. 2.0 and to 66 MHz in Rev. 2.1 of the PCI standard. The bus controller as the interface for the CPU to the various devices allows the increase of the CPU speed without affecting the bus and the connected resources. Thus, it is possible to develop the peripherals and the processors in a decoupled way.

Analyzing timing behavior of memory accesses is special because these accesses cross the CPU/bus-clock boundary. Thus, the gap between CPU and bus clock must be modeled within a microarchitectural analysis, since the time unit for those analyses is one CPU cycle[3]; the analysis needs to know when the next bus cycle begins. If the analysis does not have this information, it needs to account for all possibilities including the worst case: The bus cycle has just begun, and the CPU needs to wait nearly a full bus cycle to perform a bus action. This pessimism would lead to less-precise WCET bounds.

The number of possible displacements of phase between CPU- and bus-clock signal is bounded, i.e., at the start of a CPU cycle, the bus cycle can only be in a finite number of states. For example, if the CPU operates at $f_{\mathrm{CPU}} = 100$ MHz and the bus at $f_{\mathrm{BUS}} = 25$ MHz, there are four different states. In general, the number of states is determined by

$$\text{bus-clock states} := \frac{f_{\mathrm{CPU}}}{\gcd(f_{\mathrm{CPU}}, f_{\mathrm{BUS}})}.$$

To obtain a more precise WCET bound, the displacement of phase has to be modeled within a microarchitectural analysis. Thus, the search space for the analysis is augmented by the number of different bus-clock states.

The smaller the number of bus-clock states, the more efficient is the microarchitectural analysis. Note that, for integral ratios of CPU to bus frequency, the formula simplifies to $f_{\mathrm{CPU}}/f_{\mathrm{BUS}}$. It might be beneficial to use integral ratios, even if a close-by nonintegral ratio would have a higher average-case performance.

Buses can also be classified as parallel (e.g., *SCSI*) or bit-serial (e.g., *USB*) buses. Parallel buses carry data words in parallel on multiple wires; bit-serial buses carry data in serial form. Because of the separation of addresses and data on parallel buses, the execution of consecutive memory accesses can be overlapped, i.e., for two accesses, the address phase of the second access can be overlapped with the data phase of the first access. This is called *bus pipelining*. The number of accesses that can overlap is called the *pipeline depth* of the bus. Hence, one distinguishes between pipelined and nonpipelined buses. The advantage of bus pipelining is better performance due to reduced idle time. On the other hand, pipelined buses need to arbitrate the incoming bus requests, e.g., if there is an instruction fetch and a data access at the same time, the arbitration logic needs to decide which bus request is issued first.

Instances that can request access to the bus are called *bus masters*. On simple systems, there is only one bus master, since there is typically one CPU that requests the bus. This scenario can be modeled because the timing behavior is deterministic. The more masters a bus has, the more difficult it is to analyze the traffic on the bus and the less precise will be the bounds on latencies that can be guaranteed. There are several methods to handle the arbitration between multiple bus masters; the most prominent ones, namely, *central bus arbiter, time-division multiple access (TDMA)*, and *carrier-sense multiple access/collision avoidance (CSMA/CA)* will be discussed in more detail.

A deterministic method for controlling the access order on the bus is the introduction of a *central bus arbiter*. Every master which would like to access the bus must ask the bus arbiter for admission. For timing analysis, the arbitration logic of this central instance has to be modeled. The *TDMA* is an example for such a deterministic arbitration logic. There are fixed time slots for each bus requester.

However, there are also nondeterministic arbitration logics, like *CSMA/CA*. A device requests the bus and then waits for a random amount of time before it rechecks whether the bus can still be requested by it. This method is designed for collision avoidance, but due to its nondeterministic behavior, its timing behavior cannot be predicted.

Asynchronous mechanisms such as *DMA* or *DRAM* refresh cannot be analyzed with the methods described so far. A DMA transmission and a DRAM refresh and their associated costs cannot be attributed to the execution of an instruction. The costs of a DRAM refresh must be amortized over time. A similar approach can be used if the frequency of DMA is statically known.

In recent research results, Akesson *et al.* [38] have proposed the design of a predictable SDRAM memory controller. They can guarantee a minimum bandwidth and a maximum latency for each memory access.

VI. MultiCore Target Architectures

There is a tendency toward the use of multicore architectures for their good energy/performance ratio. Under the aspect of predictability, some existing and upcoming multicore architectures are unacceptable because of the interference of the different cores on shared resources such as caches and buses.

[3]The microarchitectural analysis described in Section II-A models how instructions pass through the processors' pipeline; thus, the behavior is usually modeled on a cycle-by-cycle basis.

Examples for current automotive multicore architectures are the Infineon TriCore TC1797, the Freescale MC9S12X, and the Freescale MPC5516. They consist of a powerful main processor and less powerful coprocessors. For future automotive multicore architectures, we see a design trend toward the use of identical cores mostly with shared memories.

The shared memories (Flash, RAM) and peripherals are connected to the cores by shared buses or crossbars. Conflicts when accessing shared resources are usually resolved by assigning fixed priorities. Depending on the architecture, conflicts on shared resources can be expected to happen frequently. For example, if the cores have no private RAM, a potential conflict might occur for each access (typically, 20%–30% of all executed instructions).

The execution time of a task running on one core typically depends on the activities on the other cores. Static worst-case execution-time analysis usually assumes the absence of interferences. The additional time (or penalty) caused by interferences must be bounded for a scheduling analysis. For noncompositional architectures (see Section III-C) with domino effects and timing anomalies, determining such a bound can become next to impossible. For a fully timing-compositional architecture, Schliecker *et al.* [39] determine upper bounds of the penalties by computing the number of potential conflicts when accessing shared memory by counting the number of memory accesses possibly generated on different cores.

A. Shared Caches

The unconstrained use of shared caches can make a sound and precise analysis of the cache performance impossible. The set of potential interleavings of the threads running on the different cores result in a huge state space to be explored, resulting in poor precision. There exist first approaches to the analysis of the cache performance of shared caches in multicore systems. All approaches implicitly assume fully timing compositional architectures (see Section III-C). Some approaches stem from the analysis of task interference on caches in preemptive scheduling. They compute the cache footprint of preempted and preempting tasks, determine the intersection, and assume the rest as being eliminated. This approach is neither context sensitive nor flow sensitive and, therefore, overly pessimistic. A recent approach [40] is also flow insensitive but mildly context sensitive as it considers instructions in loops and not in loops differently. The obtained precision is not satisfactory.

B. Elimination of Interferences

The principle to be applied in the design of a multicore architecture with predictable timing behavior is the *systematic elimination of interference on shared resources wherever they are not absolutely needed for performance*.

C. Proposal for a Multicore Architecture

The following multicore architecture offers a good combination of performance and predictability. It would have private L1 (and L2) caches with LRU replacement strategy.

Several design decisions are derived from characteristics of the embedded applications foreseen, mainly from automotive and aeronautics. One observation is that code is rarely shared, the operating system often being the only exception. Allocating the code of the different threads into a shared memory would introduce unnecessary interferences with no gain in performance. Thus, we advocate separate memory hierarchies for code. However, embedded systems interact on the data space and, thus, either need shared memory or communication. Embedded systems used for control often have the characteristic that their tasks read when an activation is triggered and write when the task has terminated. This makes it easier to reduce the collision on shared communication media. In our proposed multicore architecture, crossbars would be used for the communication between cores and the shared cache and memory in such a way that they have deterministic access times despite being global and shared. This requires careful scheduling and allocation of tasks to guarantee exclusive access and noninterference. The approach described in [41] solves this problem by combining system-level scheduling with timing analysis and a TDMA-based bus-access policy.

VII. Future Work

The trend in automotive embedded systems is toward unifying frameworks like AUTOSAR. Such frameworks aim at managing the increasing functional complexity and allowing for better reconfigurability and maintainability. Standardized interfaces allow us to compose components, independently developed by different suppliers, on ECUs. A runtime environment provides basic services, like intra- or inter-ECU communication between components. At a lower level, AUTOSAR abstracts from the underlying hardware, the actually deployed ECUs. From a functional point of view, this framework is appealing because of the gained compositionality. The AUTOSAR timing model currently being developed is concerned mainly with the integration of scheduling requirements. The success of scheduling analysis depends on the predictability of the execution times of the AUTOSAR—"runnables," the basic building blocks of a software component. When multiple components are mapped to a multicore architecture where memory and communication buses are shared, execution times of runnables may vary considerably, and the possibilities to predict safe and precise execution-time bounds can be rather limited. This limits the success of the scheduling analysis and counteracts the idea of composing software components.

The applicability of the AUTOSAR idea depends on availability of architectures on which software composition does not lead to unpredictable timing behavior.

In order to be able to benefit from static-performance analysis, one will need the following elements.

1) Hardware architects making designs that enable an interference free mapping of the application.
2) A good understanding of the architecture by application developers and software integrators who have to develop an interference-free mapping.

3) Tools that support developers in their mapping decisions like compilation tool chains that automatically make good use of scratchpad memories.

VIII. Conclusion and Recommendations

We have presented practically relevant theoretical results and empirical observations about the architectural influence on timing analysis. These concerned the applicability of methods, static versus measurement-based, and the precision and the efficiency of static methods. A list of recommendations is derived from these results and experiences. The most important principle is *reduce the interference on shared resources*.

1) **For the memory hierarchy, we suggest:** Use
   a) caches with LRU replacement policy, and/or scratchpad memories;
   b) separate L1 instruction and data caches;
   c) a flat linear byte-oriented memory model without paging;
   d) nonshared memory where there is little sharing in the application.
2) **For pipelines, we recommend:** Use compositional pipelines, in particular, if the execution time is dominated by memory-access times, anyway.

There are other recommendations such as the reduction of speculative features, which would increase the precision and decrease the complexity of timing analysis and which could not be treated in detail in this paper.

Acknowledgment

The authors would like to thank H. Theiling for suggestions concerning the memory model and the control-flow reconstruction, to S. Schliecker for discussions on penalty computations for response time analysis, and to N. Feiertag for discussions on the AUTOSAR timing model and the TIMMO project.

References

[1] S. M. Petters, "Worst case execution time estimation for advanced processor architectures," Ph.D. dissertation, Technische Universität München, Munich, Germany, Sep. 2002.
[2] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proc. 23rd IEEE RTSS*, 2002, pp. 279–288.
[3] I. Wenzel, "Measurement-based timing analysis of superscalar processors," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2006. Treitlst. 3/3/182-1, 1040.
[4] A. Colin and S. M. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proc. 24th RTSS*, 2003, p. 190.
[5] S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both?" in *Proc. WCET*, C. Rochange, Ed, 2007, pp. 5–12.
[6] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand, "An abstract interpretation-based timing validation of hard real-time avionics software systems," in *Proc. DSN*, Jun. 2003, pp. 625–632.
[7] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Real-Time Syst.*, vol. 91, no. 7, pp. 1038–1054, Jul. 2003.
[8] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proc. 6th Int. Workshop WCET Analysis*, 2006, pp. 23–28.
[9] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, Nov. 2007.
[10] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Syst.*, vol. 17, no. 2/3, pp. 131–181, Nov. 1999.
[11] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the timing analysis of pipelining and instruction caching," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1995, pp. 288–297.
[12] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Syst.*, vol. 18, no. 2/3, pp. 157–179, May 2000.
[13] A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Uppsala Univ., Uppsala, Sweden, 2003.
[14] H. Theiling, "Control flow graphs for real-time systems analysis," Ph.D. dissertation, Universität des Saarlandes, Saarbrücken, Germany, 2002.
[15] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. POPL*, 1977, pp. 238–252.
[16] R. Sen and Y. N. Srikant, "Executable analysis using abstract interpretation with circular linear progressions," in *Proc. MEMOCODE*, 2007, pp. 39–48.
[17] A. Ermedahl and J. Gustafsson, "Deriving annotations for tight calculation of execution time," in *Proc. Euro-Par*, 1997, pp. 1298–1307.
[18] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Syst.*, vol. 18, no. 2/3, pp. 121–148, May 2000.
[19] I. Stein and F. Martin, "Analysis of path exclusion at the machine code level," in *Proc. WCET*, 2007, pp. 71–76.
[20] J. Engblom, "Processor pipelines and static worst-case execution time analysis," Ph.D. dissertation, Dept. Inf. Technol., Uppsala Univ., Uppsala, Sweden, 2002.
[21] S. Thesing, "Safe and precise WCET determinations by abstract interpretation of pipeline models," Ph.D. dissertation, Saarland Univ., Saarbrücken, Germany, 2004.
[22] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. 32nd DAC*, Jun. 1995, pp. 456–461.
[23] H. Theiling, "ILP-based interprocedural path analysis," in *EMSOFT*, vol. 2491. New York: Springer-Verlag, 2002, pp. 349–363.
[24] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *EMSOFT*, vol. 2211. New York: Springer-Verlag, 2001, pp. 469–485.
[25] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. 20th RTSS*, Dec. 1999, pp. 12–21.
[26] J. Schneider, "Combined Schedulability and WCET Analysis for Realtime Operating Systems," Ph.D. dissertation, Saarland Univ., Saarbrücken, Germany, 2003.
[27] C. Berg, "PLRU cache domino effects," in *Proc. 6th Int. Workshop WCET Analysis*, F. Mueller, Ed, 2006, pp. 29–31.
[28] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite," in *Proc. 42nd Annu. ACM-SE*, 2004, pp. 267–272.
[29] J. Reineke and D. Grund, "Sensitivity of cache replacement policies," *SFB/TR 14 AVACS, Reports of SFB/TR 14 AVACS 36*, Mar. 2008. [Online]. Available: http://www.avacs.org
[30] B. Ackland, D. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knoblock, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams, "A single-chip, 1.6 billion, 16-b mac/s multiprocessor DSP," *IEEE J. Solid-State Circuits*, vol. 35, no. 3, pp. 412–423, Mar. 2000.
[31] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, "Fast, predictable and low energy memory references through architecture-aware compilation," in *Proc. ASP-DAC*, 2004, pp. 4–11.
[32] J.-F. Deverge and I. Puaut, "Safe measurement-based WCET estimation," in *Proc. 5th Int. Workshop WCET Analysis*, R. Wilhelm, Ed, Dagstuhl, Germany, 2005.
[33] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23rd RTSS*, 2002, pp. 114–123.
[34] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 272–282, Jun. 2003.
[35] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, Feb. 1985.
[36] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Mateo, CA: Morgan Kaufmann, 2003.

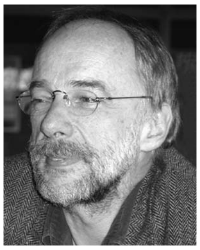
[37] *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*, Freescale Semicond., Inc., Austin, TX, 2004. rev. 0.1.

[38] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *Proc. 5th IEEE/ACM Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2007, pp. 251–256.

[39] S. Schliecker, M. Ivers, and R. Ernst, "Integrated analysis of communicating tasks in MPSoCS," in *Proc. 4th Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2006, pp. 288–293.

[40] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared instruction caches," in *Proc. RTAS*, 2008, pp. 80–89.

[41] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proc. 28th RTSS*, 2007, pp. 49–60.



Reinhard Wilhelm received the Dr. rer. nat. degree from the Technical University of Munich, Munich, Germany, in 1977.

Since 1978, he has been a Professor of computer science with Saarland University, Saarbrücken, Germany, where he is currently the Chair for compiler design and programming languages. Since 1990, he has also been the Scientific Director of the Leibniz Center for Computer Science, Schloss Dagstuhl, Saarland, Germany. In 1998, he and five of his doctoral students founded the spin-off company AbsInt Angewandte Informatik GmbH. His main research activities are in the areas of compiler construction, static-program analysis, and worst-case execution-time analysis.

Dr. Wilhelm became a Fellow of the ACM in 2002. He was the recipient of the Gay–Lussac–Humboldt Award for outstanding scientific contributions and successful French–German research cooperation from the French Minister of Research. Since 2008, he has been a member of Academia Europea. He was the recipient of an honorary doctorate of the RWTH Aachen, Aachen, Germany, in 2008.

Daniel Grund was born in 1980. He received the Diploma degree (with distinction) in computer science from the University of Karlsruhe, Karlsruhe, Germany, in 2005, where he was supported by a scholarship of the German Research Foundation. He is currently working toward the Ph.D. degree at Saarland University under the supervision of Prof. Wilhelm since 2006.

His doctoral research interests include worst-case execution-time analysis, cache analysis, predictable architectures, and compiler analyses and optimizations.

Jan Reineke was born in 1980. He received the B.S. degree in computer science from the University of Oldenburg, Oldenburg, Germany, in 2003 and the M.S. degree in computer science from Saarland University, Saarbrücken, Germany, in 2005, where he has been working toward the Ph.D. degree under the supervision of Prof. Wilhelm and, in the late 2008, defended his Ph.D. thesis on caches in WCET analysis.

His research interests include timing predictability; in particular, the predictability of caches, cache analysis, as well as shape and topology analysis by abstract interpretation.

Marc Schlickling was born in 1980. He received the Diploma degree in computer science from Saarland University, Saarbrücken, Germany, in 2005, where he is currently working toward the Ph.D. degree under the supervision of Prof. Wilhelm.

Since 2005, he has also been a Software Engineer for AbsInt Angewandte Informatik GmbH. His current research interests include worst-case execution-time analysis, predictability, and semi-automatic derivation of abstracted processor models.

Markus Pister was born in 1979. He received the Diploma degree in computer science from Saarland University, Saarbrücken, Germany, in 2005, where he has been working toward the Ph.D. degree under the supervision of Prof. Wilhelm since 2005.

Besides his research position at the university, he is currently also a Software Engineer for AbsInt Angewandte Informatik GmbH. His research focuses on worst-case execution-time analysis and the semi-automatic derivation of abstract timing models out of formal hardware specifications.

Christian Ferdinand received the Ph.D. degree in computer science from Saarland University, Saarbrücken, Germany, in 1997, where his thesis about static cache analysis was supervised by Prof. Wilhelm.

He is a cofounder and, currently, the Managing Director of AbsInt Angewandte Informatik GmbH, which is a company that provides advanced development tools for embedded systems and tools for validation, verification, and certification of safety-critical software.