Building Timing Predictable Embedded Systems

HEIKO FALK, ALAIN GIRAULT, DANIEL GRUND, NAN GUAN, BENGT JONSSON, PETER MARWEDEL, JAN REINEKE, CHRISTINE ROCHANGE, REINHARD VON HANXLEDEN, REINHARD WILHELM, WANG YI, Artist-Design NoE

A large class of embedded systems is distinguished from general purpose computing systems by the need to satisfy strict requirements on timing, often under constraints on available resources. Predictable system design is concerned with the challenge of building systems for which timing requirements can be guaranteed *a priori*. Perhaps paradoxically, this problem has become more difficult by the introduction of performance-enhancing architectural elements, such as caches, pipelines, and multithreading, which introduce a large degree of nondeterminism and make guarantees harder to provide. The intention of this paper is to summarize current state-of-the-art in research concerning how to build predictable yet performant systems. We consider how processor architectures, and programming languages can be devised for predictability. We also consider the integration of compilation and timing analysis, as well as strategies for predictability on multicores.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based systems**]: Real-time and embedded systems

General Terms: Design, Performance, Reliability, Verification

Additional Key Words and Phrases: Embedded systems, predictability, worst-case execution time, resource sharing

ACM Reference Format:
Artist-Design NoE, 2012. Building Timing Predictable Embedded Systems. ACM Trans. Embedd. Comput. Syst. XX, YY, Article ZZ (January 2012), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Embedded systems distinguish themselves from general purpose computing systems by several characteristics, including the limited availability of resources and the requirement to satisfy nonfunctional constraints, e.g., on latencies or throughput. In several application domains, including automotive, avionics, industrial automation, many functionalities are associated with strict requirements on deadlines for delivering results of calculations. In many cases, failure to meet deadlines may cause a catastrophic or at least highly undesirable system failure, associated with risks for human or economical damages.

Predictable system design is concerned with the challenge of building systems in such a way that requirements can be guaranteed from the design. This means that an off-line analysis should demonstrate satisfaction of timing requirements, subject to assumptions made on operating conditions foreseen for the system [Stankovic and Ramamritham 1990]. Devising such an analysis is a challenging problem, since tim-

This work is supported by the ArtistDesign Network of Excellence, supported by the European Commission, grant 214373.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ing requirements propagate down in the system hierarchy, meaning that the analysis must foresee timing properties of all parts of a system: processor and instruction-set architecture, language and compiler support, software design, run-time system and scheduling, communication infrastructure, etc. Perhaps paradoxically, this problem has become more difficult by the trend to make processors more performant, since the introduced architectural elements, such as pipelines, out-of-order execution, on-chip memory systems, etc., lead to a large degree of nondeterminism in system execution, making guarantees harder to provide.

One strategy to the problem of guaranteeing timing requirements, which is sometimes proposed, is to exploit performance-enhancing features that have been developed and over-provision whenever the criticality of the software is high. The drawback is that, often, requirements cannot be completely guaranteed anyway, and that resources are wasted, e.g., when low energy budget is important.

It is therefore important to develop techniques that really guarantee timing requirements that are commensurate with the actual performance of a system. Significant advances have been made in the last decade on analysis of timing properties (see, e.g., [Wilhelm et al. 2008] for an overview). However, these techniques cannot make miracles. They can only make predictions if the analyzed mechanisms are themselves predictable, i.e., if their relevant timing properties can be foreseen with sufficient precision. Fortunately, the understanding of how to design systems that reconcile efficiency and predictability has increased in recent years. Recent research efforts include european projects, such as Predator[1] and MERASA [Ungerer et al. 2010], that have focused on techniques for designing predictable *and* efficiency systems, as well as the PRET project [Edwards and Lee 2007; Lickly et al. 2008a], which aims to equip instruction-set architectures with predictable timing, etc.

The intention of this paper is to summarize some of the recent research advances, concerning how to build predictable yet performant systems. We present techniques, whereby architectural elements that are introduced primarily for efficiency, can also be made timing-predictable. We also discuss how these techniques can be exploited by languages and tools so that a developer can directly control timing properties of a system under development. To limit the exposition, we will not discuss particular analysis methods for deriving timing bounds; this area has progressed significantly (e.g., [Wilhelm et al. 2008]), but to include a meaningful overview would require too much space.

In a first section, we discuss basic concepts, and also make a proposal for how "predictability" of an architectural mechanism could be defined precisely. Our motivation is that a better understanding of "predictability" can preclude futile efforts to develop analyses for inherently unpredictable systems, or to redesign already predictable mechanisms or components. The following Section 3 considers how the instruction-set architecture for a processor can be equipped with predictable timing semantics, i.e., how the execution of machine instructions can be made predictable. Important here is the design and use of processor pipelines and the memory system.

In Sections 4 and 5, we move up one level of abstraction, and consider two different approaches for putting timing under the control of a programmer. In Section 4, we present synchronous programming languages, whose semantics provide timing guarantees, with PRET-C and Synchronous-C as the main examples. We also present how the timing semantics can be supported by specialized processor implementations. In Section 5, we describe how a static timing analysis tool for timing analysis (aiT) can be integrated with a compiler for a widely-used language (C). The integration of these tools can equip program fragments with timing semantics (of course relative to compi-

[1] http://www.predator-project.eu/

Table I. Examples for intuition behind predictability.

	more predictable	less predictable
pipeline	in-order	out-of-order
branch prediction	static	dynamic
cache replacement	LRU	FIFO, PLRU
scheduling	static	dynamic preemptive
arbitration	TDMA	FCFS

lation strategy and target platform). It is also a basis for assessing different compilation strategies when predictability is the main design objective.

In Section 6, we consider techniques for multicores. Such platforms are finding their way into many embedded applications, but introduce difficult challenges for predictability. Major challenges include the arbitration of shared resources such as on-chip memories and buses. Predictability can be achieved only if logically unrelated activites can be isolated from each other, e.g., by partitioning communication and memory resources. We also consider some concerns for the sharing of processors between tasks in scheduling.

2. FUNDAMENTAL PREDICTABILITY CONCEPTS

Predictable system design is made increasingly difficult by past and current developments in system and computer architecture design, where more performant architectural elements are introduced for performance, but make timing guarantees harder to provide. Hence, research on in this area can be divided into two strands: On the one hand there is the development of ever better analyses to keep up with these developments. On the other hand there is the exercise of influence on system design in order to avert the worst problems in future designs. We do *not* want to dispute the value of these two lines of research. Far from it. However, we argue that both are often built on sand: Without a better understanding of "predictability", the first line of research might try to develop analyses for inherently unpredictable systems, and the second line of research might simplify or redesign architectural components that are in fact perfectly predictable. To the best of our knowledge there is no agreement — in the form of a formal definition — what the notion "predictability" should mean. Instead the criteria for predictability are *based on intuition* and arguments are made on a *case-by-case basis*. Table I gives examples for this intuition-based comparison of predictability. In the analysis of worst-case execution times (WCET) for instance, simple in-order pipelines like the ARM7 are deemed more predictable than complex out-of-order pipelines as found in the POWERPC755.

In the following we discuss key aspects of predictability and therefrom derive a template for predictability definitions.

2.1. Key Aspects of Predictability

What does predictability mean? A lookup in the Oxford English Dictionary provides the following definitions:

> predictable: adjective, able to be predicted.
> to predict: say or estimate that (a specified thing) will happen in the future or will be a consequence of something.

Consequently, a system is predictable if one can foretell facts about its future, i.e. determine interesting things about its behavior. In general, the behaviors of such a system can be described by a possibly infinite set of execution traces (sequences of states and transitions). However, a prediction will usually refer to derived properties of such traces, e.g. their length or a number of interesting events on a trace. While

some properties of a system might be predictable, others might not. Hence, the first aspect of predictability is the *property to be predicted*.

Typically, the property to be determined depends on something unknown, e.g. the input of a program, and the prediction to be made should be valid for all possible cases, e.g. all admissible program inputs. Hence, the second aspect of predictability are the *sources of uncertainty* that influence the prediction quality.

Predictability will not be a boolean property in general, but should preferably offer shades of gray and thereby allow for comparing systems. How well can a property be predicted? Is system A more predictable than system B (with respect to a certain property)? The third aspect of predictability thus is a *quality measure* on the predictions.

Furthermore, predictability should be a property *inherent* to the system. Only because *some* analysis cannot predict a property for system A while it can do so for system B does not mean that system B is more predictable than system A. In fact, it might be that the analysis simply lends itself better to system B, yet better analyses do exist for system A.

With the above key aspects we can narrow down the notion of predictability as follows:

THESIS 2.1. *The notion of predictability should capture if, and to what level of precision, a specified property of a system can be predicted by an optimal analysis. It is the sources of uncertainty that limit the precision of any analysis.*

Refinements. A definition of predictability could possibly take into account more aspects and exhibit additional properties.

— For instance, one could refine Proposition 2.1 by taking into account the complexity/cost of the analysis that determines the property. However, the clause "by *any* analysis not more expensive than X" complicates matters: The key aspect of inherence requires a quantification over all analyses of a certain complexity/cost.

— Another refinement would be to consider different sources of uncertainty separately to capture only the influence of one source. We will have an example of this later.

— One could also distinguish the extent of uncertainty. E.g. is the program input completely unknown or is partial information available?

— It is desirable that the predictability of a system can be determined automatically, i.e. computed.

— It is also desirable that predictability of a system is characterized in a compositional way. This way, the predictability of a composed system could be determined by a composition of the predictabilities of its components.

2.2. A Predictability Template

Besides the key aspect of inherence, the other key aspects of predictability depend on the system under consideration. We therefore propose a template for predictability with the goal to enable a concise and uniform description of predictability instances. It consists of the above mentioned key aspects (a) property to be predicted, (b) sources of uncertainty, and (c) quality measure. In the next section we consider one instance of predictability in more detail to illustrate this idea.

2.3. An Illustrative Instance: Timing Predictability

In this section we illustrate the key aspects of predictability at the hand of timing predictability.

— The property to be determined is the execution time of a program assuming uninterrupted execution on a given hardware platform.

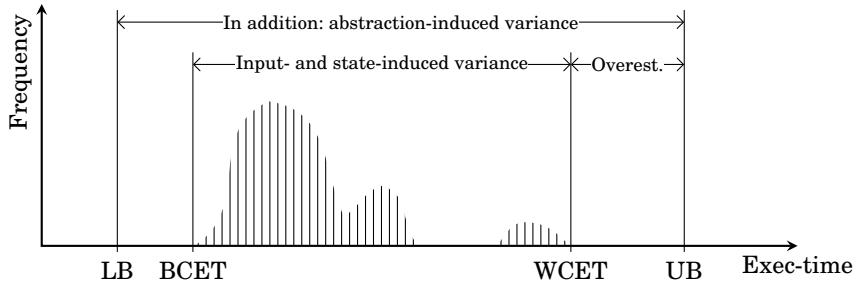


Fig. 1. Distribution of execution times ranging from best-case to worst-case execution time (BCET/WCET). Sound but incomplete analyses can derive lower and upper bounds (LB, UB).

— The sources of uncertainty are the *program input* and the *hardware state* in which execution begins. Figure 1 illustrates the situation and displays important notions. Typically, the initial hardware state is completely unknown, i.e. the prediction should be valid for all possible initial hardware states. Additionally, schedulability analysis cannot handle a characterization of execution times in the form of a function depending on inputs. Hence, the prediction should also hold for all admissible program inputs.

— Usually, schedulability analysis requires a characterization of execution times in the form bounds on the execution time. Hence, a reasonable quality measure is the quotient of BCET over WCET; the smaller the difference the better.

— The inherence property is satisfied as BCET and WCET are inherent to the system.

To formally define timing predictability we need to first introduce some basic definitions.

Definition 2.2. Let $\mathcal{Q}$ denote the set of all *hardware states* and let $\mathcal{I}$ denote the set of all *program inputs*. Furthermore, let $T_p(q, i)$ be the *execution time* of program p starting in hardware state $q \in \mathcal{Q}$ with input $i \in \mathcal{I}$.

Now we are ready to define timing predictability.

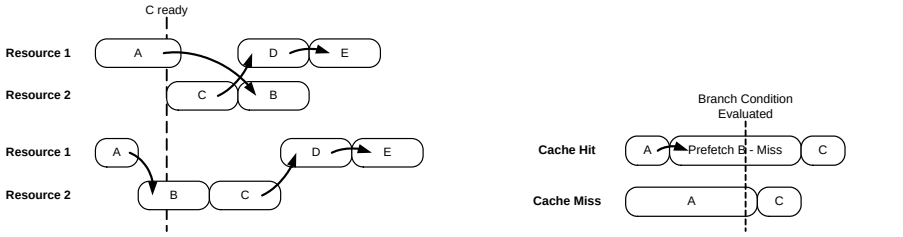
Definition 2.3 (*Timing predictability*). Given uncertainty about the initial hardware state $Q \subseteq \mathcal{Q}$ and uncertainty about the program input $I \subseteq \mathcal{I}$, the timing predictability of a program p is

$$\Pr_p(Q, I) := \min_{q_1, q_2 \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q_1, i_1)}{T_p(q_2, i_2)} \tag{1}$$

The quantification over pairs of states in Q and pairs of inputs in I captures the uncertainty. The property to predict is the execution time T_p. The quotient is the quality measure: $\Pr_p \in [0, 1]$, where 1 means perfectly predictable.

Refinements. The above definitions allow analyses of arbitrary complexity, which might be practically infeasible. Hence, it would be desirable to only consider analyses within a certain complexity class. While it is desirable to include analysis complexity in a predictability definition it might become even more difficult to determine the predictability of a system under this constraint: To adhere to the inherence aspect of predictability however, it is necessary to consider *all* analyses of a certain complexity/cost.

Another refinement is to distinguish hardware- and software-related causes of unpredictability by separately considering the sources of uncertainty:



(a) Scheduling anomaly.

(b) Speculation anomaly. A and B are prefetches. If A hits, B can also be prefetched and might miss the cache.

Fig. 2. Speculation and Scheduling anomalies, taken from [Reineke et al. 2006].

Definition 2.4 (*State-induced timing predictability*).

$$\mathrm{SIPr}_p(Q, I) := \min_{q_1, q_2 \in Q} \min_{i \in I} \frac{T_p(q_1, i)}{T_p(q_2, i)} \tag{2}$$

Here, the quantification expresses the maximal variance in execution time due to different hardware states, q_1 and q_2, for an arbitrary but fixed program input, i. It therefore captures the influence of the hardware, only. The input-induced timing predictability is defined analogously. As a program might perform very different actions for different inputs, this captures the influence of software:

Definition 2.5 (*Input-induced timing predictability*).

$$\mathrm{IIPr}_p(Q, I) := \min_{q \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q, i_1)}{T_p(q, i_2)} \tag{3}$$

Example for state-induced timing unpredictability. A system exhibits a *domino effect* [Lundqvist and Stenström 1999] if there are two hardware states q_1, q_2 such that the difference in execution time of the same program starting in q_1 respectively q_2 may be arbitrarily high, i.e. cannot be bounded by a *constant*. For instance, the iterations of a program loop never converge to the same hardware state and the difference in execution time increases in each iteration.

In [Schneider 2003] Schneider describes a domino effect in the pipeline of the PowerPC 755. It involves the two asymmetrical integer execution units, a greedy instruction dispatcher, and an instruction sequence with read-after-write dependencies.

The dependencies in the instruction sequence are such that the decisions of the dispatcher result in a longer execution time if the initial state of the pipeline is empty than in case it is partially filled. This can be repeated arbitrarily often, as the pipeline states after the execution of the sequence are equivalent to the initial pipeline states. For n subsequent executions of the sequence, execution takes $9n + 1$ cycles when starting in one state, q_1^*, and $12n$ cycles when starting in the other state, q_2^*. Hence, the state-induced predictability can be bounded for such programs p_n:

$$\mathrm{SIPr}_{p_n}(Q, I) = \min_{q_1, q_2 \in Q_n} \min_{i \in I} \frac{T_{p_n}(q_1, i)}{T_{p_n}(q_2, i)} \leq \frac{T_{p_n}(q_1^*, i^*)}{T_{p_n}(q_2^*, i^*)} = \frac{9n + 1}{12n} \tag{4}$$

Another example for a domino effect is given by Berg [Berg 2006] who considers the PLRU replacement policy of caches. In Section 3, we describe results on the state-induced cache predictability of various replacement policies.

Timing Anomalies. The notion of *timing anomalies* was introduced by Lundqvist and Stenström in [Lundqvist and Stenström 1999]. In the context of WCET analysis, [Reineke et al. 2006] presents a formal definition and additional examples of such phenomena. 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 Figure 2(a) 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. Figure 2(b) illustrates this.

3. MICROARCHITECTURE

An *instruction set architecture* (ISA) defines the interface between hardware and software, i.e., the format of software binaries and their semantics in terms of input/output behavior. A *microarchitecture* defines how an ISA is implemented on a processor. A single ISA may have many microarchitectural realizations. For example, there are many implementations of the X86 ISA by INTEL and AMD.

Execution time is not in the scope of the semantics of common ISAs. Different implementations of an ISA, i.e., different microarchitectures, may induce arbitrarily different execution times. This has been a deliberate choice: Microarchitects exploit the resulting implementation freedom introducing a variety of techniques to improve performance. Prominent examples of such techniques include pipelining, superscalar execution, branch prediction, and caching.

As a consequence of abstracting from execution time in ISA semantics, worst-case execution time (WCET) analyses need to consider the microarchitecture a software binary will be executed on. The aforementioned microarchitectural techniques greatly complicate WCET analyses. For simple, non-pipelined microarchitectures without caches one could simply sum up the execution times of individual instructions to obtain the exact execution time of a sequence of instructions. With pipelining, caches, and other features, execution times of successive instructions overlap, and—more importantly—they vary depending on the execution history[2] leading to the execution of an instruction: a read immediately following a write to the same register incurs a pipeline stall; the first fetch of an instruction in a loop results in a cache miss, whereas subsequent accesses may result in cache hits, etc.

3.1. Pipelines

For non-pipelined 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.

[2]In other words: the current state of the microarchitecture.

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 execution 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, cache-like 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 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 between states of the concrete pipeline may depend on such information. This causes the abstract pipeline model to become non-deterministic although the concrete pipeline is deterministic. When dealing with this non-determinism, one could be tempted to design the WCET analysis such that only the "locally worst-case" transition is chosen, e.g., the transition corresponding to a pipeline stall or a cache miss. However, in the presence of *timing anomalies* [Lundqvist and Stenström 1999; Reineke et al. 2006] such an approach is unsound. Thus, in general, the analysis has to follow all possible successor states.

Classification of microarchitectures from [Wilhelm et al. 2009]. Architectures can be classified into three categories depending on whether they exhibit timing anomalies or domino effects [Wilhelm et al. 2009].

— **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. Actually, 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 best-case execution time.
— **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 non-worst-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.
— **Non-compositional 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.

Approaches to Predictable Pipelining. The complexity of WCET analysis can be reduced by regulating the instruction flow of the pipeline at the beginning of each basic block [Rochange and Sainrat 2005]. This removes all timing dependencies within the pipeline between basic blocks. Thus, WCET analysis can be performed on each basic block in isolation. The authors take the stance that efficient analysis techniques are a prerequisite for predictability: "a processor might be declared unpredictable if computation and/or memory requirements to analyse the WCET are prohibitive."

With the advent of multi-core and multi-threaded architectures, new challenges and opportunities arise in the design of timing-predictable systems: Interference between hardware threads on shared resources further complicates analysis. On the other

hand, timing models for individual threads are often simpler in such architectures. Recent work has focussed on providing timing predictability in multithreaded architectures:

One line of research proposes modifications to simultaneous multithreading architectures [Barre et al. 2008; Mische et al. 2008]. These approaches adapt thread-scheduling in such a way that one thread, the real-time thread, is given priority over all other threads, the non-real-time threads. As a consequence, the real-time thread experiences no interference by other threads and can be analyzed without having to consider its context, i.e., the non-real-time threads. This guarantees temporal isolation for the real-time thread, but not for any other thread running on the core. If multiple real-time tasks are needed, then time sharing of the real-time thread is required.

Earlier, a more static approach was proposed by El-Haj-Mahmoud et al. [El-Haj-Mahmoud et al. 2005] called the virtual multiprocessor. The virtual multiprocessor uses static scheduling on a multithreaded superscalar processor to remove temporal interference. The processor is partitioned into different time slices and superscalar ways, which are used by a scheduler to construct the thread execution schedule offline. This approach provides temporal isolation to all threads.

The PTARM [Liu et al. 2010], a precision-timed (PRET) machine [Edwards and Lee 2007] implementing the ARM instruction set, employs a five-stage thread-interleaved pipeline. The thread-interleaved pipeline contains four hardware threads that run in the pipeline. Instead of dynamically scheduling the execution of the threads, a predictable round-robin thread schedule is used to remove temporal interference. The round-robin thread schedule fetches a different thread every cycle, removing data hazard stalls stemming from the pipeline resources. Unlike the virtual multiprocessor, the tasks on each thread need not be determined a priori, as hardware threads cannot affect each other's schedule. Unlike Mische et al.'s [Mische et al. 2008] approach, all the hardware threads in the PTARM can be used for real-time purposes.

3.2. Caches and Scratchpad Memories

There is a large gap between the latency of current processors and that of large memories. Thus, a hierarchy of memories is necessary to provide both low latencies and large capacities. In conventional architectures, caches are part of this hierarchy. In caches, a replacement policy, implemented in hardware, decides which parts of the slow background memory to keep in the small fast memory. Replacement policies are hardwired into the hardware and independent of the applications running on the architecture.

The Influence of the Cache-Replacement Policy. Analogously to the state-induced timing predictability defined in Section 2, one can define the state-induced cache predictability of cache-replacement policy p, $\text{SIPr}_p(n)$, to capture the maximal variance in the number of cache misses due to different cache states, $q_1, q_2 \in Q_p$, for an arbitrary but fixed sequence of memory accesses, s, of length n, i.e. $s \in B_n$, where B_n denotes the set of sequences of memory accesses of length n. Given that $M_p(q, s)$ denotes the number of misses of policy p accessing sequence s starting in cache state q, $\text{SIPr}_p(n)$ is defined as follows:

Definition 3.1 (*State-induced cache predictability*).

$$\text{SIPr}_p(n) := \min_{q_1,q_2 \in Q_p} \min_{s \in B_n} \frac{M_p(q_1, s)}{M_p(q_2, s)} \tag{5}$$

To investigate the influence of the initial cache states in the long run, we have studied $\lim_{n\to\infty} \text{SIPr}_p(n)$. A tool called RELACS[3], described in [Reineke and Grund 2008],

[3]The tool is available at http://rw4.cs.uni-saarland.de/~reineke/relacs

Table II. State-induced cache predictability of $LRU, FIFO$, and $PLRU$ for associativities 2 to 8. $PLRU$ is only defined for powers of two.

	2	3	4	5	6	7	8
LRU	1	1	1	1	1	1	1
FIFO	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$
PLRU	1	–	0	–	–	–	0

is able to compute $\lim_{n\to\infty} \text{SIPr}_p(n)$ automatically for a large class of replacement policies. Using RELACS, we have obtained sensitivity results for the widely-used policies LRU, FIFO, PLRU, and MRU, at associativities ranging from 2 to 8.

Figure II depicts the analysis results. There can be no cache domino effects for LRU. Obviously, 1 is the optimal result and no policy can do better. 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 q_1 cannot be bounded the number misses incurred by the same sequence s starting in another state q_2.

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.

Interference on Shared Caches. Without further adaptation, caches do not provide temporal isolation: the same application, processing the same inputs, may exhibit wildly varying cache performance depending on the state of the cache when the application's execution begins [Wilhelm et al. 2009]. The cache's state is in turn determined by the memory accesses of other applications running earlier. Thus, the temporal behavior of one application depends on the memory accesses performed by other applications. In Section 6, we discuss approaches to eliminate and/or bound interference.

Scratchpad Memories. Scratchpad memories (SPMs) are an alternative to caches in the memory hierarchy. The same memory technology employed to implement caches is also used in SPMs: static random access memory (SRAM), which provides constant low-latency access times. In contrast to caches, however, an SPM's contents are under software control: the SPM is part of the addressable memory space, and software can copy instructions and data back and forth between the SPM and lower levels of the memory hierarchy. Accesses to the SPM will be serviced with low latency, predictably and repeatably. However, similar to the use of the register file, it is the compiler's responsibility to make correct and efficient use of the SPM. This is challenging, in particular when the SPM is to be shared among several applications, but it also presents the opportunity of high efficiency, as the SPM management can be tailored to the specific application, in contrast to the hardwired cache replacement logic. Section 5.3 briefly discusses results on SPM allocation and the related topic of cache locking.

3.3. Dynamic Random Access Memory

At the next lower level of the memory hierarchy, many systems employ Dynamic Random Access Memory (DRAM). DRAM provides much greater capacities than SRAM, at the expense of higher and more variable access latencies.

Conventional DRAM controllers do not provide temporal isolation. As with caches, access latencies depend on the history of previous accesses to the device. In addition, over time, DRAM cells leak charge. As a consequence, each DRAM row needs to be refreshed at least every 64ns, which prevents loads or stores from being issued and modifies the access history, thereby influencing the latency of future loads and stores in an unpredictable fashion.

Modern DRAM controllers reorder accesses to minimize row accesses and thus access latencies. As the data bus and the command bus, which connect the processor with the DRAM device, are shared between all of the banks of the DRAM device, controllers also have to resolve contention for these resource by different competing memory accesses. Furthermore, they dynamically issue refresh commands at—from a client's perspective—unpredictable times.

Recently, several predictable DRAM controllers have been proposed [Akesson et al. 2007; Paolieri et al. 2009b; Reineke et al. 2011]. These controllers provide a guaranteed maximum latency and minimum bandwidth to each client, independently of the execution behavior of other clients. This is achieved by a hybrid between static and dynamic access schemes, which largely eliminate the history dependence of access times to bound the latencies of individual memory requests, and by predictable arbitration mechanisms: CCSP in *Predator* [Akesson et al. 2007] and TDM in *AMC* [Paolieri et al. 2009b], allow to bound the interference between different clients. Refreshes are accounted for conservatively assuming that any transaction might interfere with an ongoing refresh. Reineke et al. [Reineke et al. 2011] partition the physical address space following the internal structure of the DRAM device. This eliminates contention for shared resources within the device, making accesses temporally predictable and temporally isolated. Replacing dedicated refresh commands with lower-latency manual row accesses in single DRAM banks further reduces the impact of refreshes on worst-case latencies.

4. SYNCHRONOUS PROGRAMMING LANGUAGES FOR PREDICTABLE SYSTEMS

4.1. Motivation

Why are new programming languages needed for predictability?. Most predictable systems are at first *real-time systems*, therefore exhibiting concurrency, complex timing reasonings, and strict real-time constraints. Programming languages dedicated to such systems should thus be concurrent and should offer features to reason about the physical time of the system. Over the years, many approaches have been proposed for this, which can be coarsely partitioned into language-based approaches (asynchronous or synchronous concurrent languages) and non-language approaches (typically concurrent threads over an RTOS).

It has been advocated that concurrency managed through RTOS threads is not a good solution for predictability [Lee 2006]. It has also been advocated that asynchronous concurrency is not well suited for programming real-time systems [Benveniste et al. 2003].

It is thus not surprising that almost all the programming languages that have been proposed for predictable systems are *synchronous* languages [Benveniste et al. 2003]. Indeed, the synchronous abstraction makes reasoning about time in a program a lot easier, thanks to the notion of *logical ticks*: a synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous.

To take a concrete exemple, the Esterel [Berry 2000] statement "`every 60 second emit minute`" specifies that the signal `minute` is *exactly synchronous* with the 60^{th} occurrence of the signal `second`. At a more fundamental level, the synchronous abstraction eliminates the non-determinism resulting from the interleaving of concurrent behaviors. This allows deterministic semantics, therefore making synchronous programs amenable to formal analysis and verification, as well as certified code generation. This abstraction is similar to the one made for synchronous circuits at the HDL level, where the time needed for a gate to compute its output is neglected, as if the electrons were flowing infinitely fast.

To make the synchronous abstraction of instantaneous reactions practical, synchronous languages impose restrictions on the control flow possible within a reaction. For example, Esterel forbids *instantaneous loops*, and similarly SyncCharts [André 2003] forbid *immediate self-transitions* or cycles of transitions that can be taken immediately. Furthermore, it is typically required that the compiler can statically verify the absence of such problems; this is not only a conservative measure, but is often also a prerequisite for proving the *causality* or the *constructiveness* of a given program and for computing an execution schedule [Berry 2000].

As it turns out, these control flow restrictions not only make the synchronous abstraction work in practice, but are also a valuable asset for timing analysis, as we will show in this section.

What are the requirements?. As it turns out, time predictability requires more than just the synchronous abstraction. For instance, it is not sufficient to *bound* the number of iteration of a loop, it is also necessary to know *exactly* this number. Another requirement is that, in order to be adopted by industry, time-predictable programming languages should offer the same full power of data manipulations as general purpose programming languages. This is why the two languages we describe (PRET-C and SC) are both predictable synchronous languages based on C. Refer to Section 5 for a coverage of WCET analysis for the regular C language.

Outline. First, we shortly cover the so-called reactive processors (Section 4.2) and the language constructs that should be avoided (Section 4.3). Then, we present two synchronous predictable programming languages in Sections 4.4 (PRET-C) and 4.5 (SC), which are both based on C. We finish with the WCRT analysis (Section 4.6), related work (Section 4.7), and future work directions (Section 4.8).

4.2. ISAs for Synchronous Programming

Synchronous languages can be used to describe both software and hardware, and a variety of synthesis approaches for both domains are covered in the literature [Potop-Butucaru et al. 2007]. The family of *reactive processors* follows an intermediate approach where a synchronous program is compiled into machine code that is then run on a processor with an instruction set architecture (ISA) that directly implements synchronous reactive control flow constructs [von Hanxleden et al. 2006]. The first reactive processor called REFLIX was presented by [Salcic et al. 2002], and this group has since then developed a number of follow-up designs including REPIC, Emperor, and StarPRO [Yuan et al. 2008]. The Kiel Esterel Processor (KEP [Li and von Hanxleden 2010]) pioneered the multi-threaded reactive approach later adopted by StarPRO, which in turn added pipelining. The KEP also includes a *Tick Manager* that minimizes reaction time jitter and can detect timing overruns. This concept is closely related to the deadi instruction of the Berkeley-Columbia PRET language [Lickly et al. 2008b].

W.r.t. predictability, the main advantage of reactive processors is that they offer direct ISA support for crucial features of the languages (e.g., preemption, synchronization, inter-thread communication), therefore allowing a very fine control over the number of machine cycle required to execute each high-level instruction. This idea of jointly addressing the language features and the processor / ISA was at the root of the Berkeley-Columbia PRET solution [Lickly et al. 2008b].

4.3. Language constructs that should be avoided

The language constructs that should be avoided are those commonly excluded by programming guidelines used by the software industry concerned with safety critical systems (at least by the companies that use a general purpose language such as C). The most notable ones are: pointers, recursive data structures, dynamic memory alloca-

tion, assignments with side-effects, recursive functions, and variable length loops. The rationale is that programs should be easy to write, to debug, and to proof-read. The same holds for PRET programming: What is easier to proof-read by humans is also easier to analyze by WCRT analyzers.

4.4. The PRET-C language

PRET-C is a light-weight and concurrent programming language based on C [Roop et al. 2009; Andalam et al. 2010]. A PRET-C program consists of a `main()` function, some regular C functions, and one or more parallel threads. Threads communicate with shared variables, and the synchronous semantics of PRET-C guarantees both a deterministic execution and the absence of race conditions.

PRET-C extends C with a small set of new constructs in order to: (1) declare a reactive input: "`ReactiveInput`"; (2) declare a reactive output: "`ReactiveOutput`"; (3) spawn two or more parallel threads: "`PAR`"; (4) and end the local tick of a thread: "`EOT`", therefore providing a synchronization barrier.

Besides, to make the language usable in practice, a few additional constructs have been introduced in order to: (5) preempt a block `P` of code, weakly or strongly: "`[weak] abort {P} when (C)`"; (6) wait upon a condition: "`await (C)`"; (7) create a thread: "`thread T()`"; (8) and loop over a block `P` of code: "`while (C) #n {P}`".

The main particularity of PRET-C's synchronous semantics compared to, say, Esterel's semantics, is that the threads are not interleaved depending on the causal dependencies of the signals during one instant (dependencies that can vary according to the inputs sampled by the program during that instant). Instead, the PRET-C threads spawned by a given `PAR` statement are interleaved in a *fixed* static order that depends only on the syntactic order in which they appear in this `PAR` statement. For instance, a `PAR(T1,T2)` results in `T1` being scheduled first, up to its first `EOT` or its termination, in `T2` being scheduled next, again up to its first `EOT` or its termination, and so on until both threads are terminated or the `PAR` itself is preempted. This static order guarantees that any variable shared between `T1` and `T2` will always be written and read in a fixed order, therefore making its value deterministic.

Concerning the `while` loops, two variants exist: (1) loops that include an `EOT` in their body (similar to loops in Esterel, which must have a `pause` in their body); and (2) loops that have no `EOT` in their body but for which a fixed bound on the number of iteration is specified by the programmer, thanks to the "`#n`" syntax.

All the new constructs of PRET-C are defined as C macros, so compiling works in two steps: first a macro expansion and then a regular C compiling (both steps can be performed by `gcc`). The resulting assembly code can be either executed as fast as possible (for better average performances), or can be embedded in a periodic execution loop. In both cases, a WCRT analyzer allows precise bounds to be computed (see Section 4.6).

Then, to achieve both predictability and throughput, the ideal is to execute this code on a platform that offers predictable execution. Such a dedicated architecture has been developed, inspired by the reactive processors discussed in Section 4.2. It is based on a customized Microblaze softcore processor (MB) from Xilinx, connected via two fast simplex links to a so-called Functional Predictable Unit (FPU). The FPU maintains the context of each parallel thread and allows thread context switching to be carried out in a constant number of clock cycles, thanks to a linked-lists based scheduler inspired from CEC's scheduler [Edwards and Zeng 2007]. Benchmarking results show that this architecture provides a 26% decrease in the WCRT compared to a stand-alone MB.

Finally, benchmarking results show that PRET-C significantly outperforms Esterel, both in terms of worst case execution time and code size.

4.5. The Synchronous-C language (SC)

Like PRET-C, *Synchronous C* (SC) enriches C with constructs for deterministic concurrency and preemption. Specifically, SC covers all of SyncCharts, hence its original name *SyncCharts in C* [von Hanxleden 2009]. SC was originally motivated by the desire to implement the reactive processing approach and its timing predictability with as little custom, non-standard instructions as possible. As with the KEP and StarPro processors, SC implements reactive control flow with a dynamically scheduled thread interleaving scheme. As we do not have direct access to the program counter at the C language level, SC keeps track of individual threads via state labels, typically implemented as usual C program labels. These labels can also be viewed as continuations, or coroutine re-entry points [Kahn and MacQueen 1977].

SC is a (freely available) library of macros and functions that implements reactive processing fully in software, requiring just a standard C compiler. This might be a bit surprising, in that standard processors have not been developed with reactive processing in mind. However, one may take advantage of certain machine instructions—with predictable timing—to effectively perform reactive processing on a non-reactive COTS processor. For example, SC does thread selection with a single bsr (Bit Scan Reverse) assembler instruction on the active thread vector. This instruction is available on the x86 and not part of the C language, but compilers such as gcc make this instruction available with embedded assembler.

Compared to PRET-C, SC offers a wider range of reactive control and coordination possibilities, such as dynamic priority changes. This makes SC more powerful and allows, for example, a direct synthesis from SyncCharts [Traulsen et al. 2011]. However, this additional power may also be a challenge to the user, in particular when using dynamic priorities, so for the inexperienced programmer it may be advisable to start with an SC subset that corresponds to PRET-C.

4.6. WCRT analysis for synchronous programs

Concerning SC, a compiler including a WCRT analysis was developed for the KEP, to compute safe estimates for the Tick Manager [Boldt et al. 2008]). Compared to typical WCET analysis, the WCRT analysis problem here is more challenging because it includes concurrency and preemption, which in WCET analysis is often delegated to the OS. However, the aforementioned deterministic semantics and guiding principles, such as the absence of instantaneous loops, make it feasible to reach fairly tight estimates.

The flow-graph based approach of [Boldt et al. 2008] was further improved by Mendler et al. with a modular, algebraic approach that also takes signal valuations into account to exclude infeasible paths [Mendler et al. 2009]. Besides, Logothetis et al. used timed Kripke structures to compute tight bounds on synchronous programs [Logothetis et al. 2003].

Concerning PRET-C, Roop et al. proposed a model-checking based WCRT analyzer to compute precisely the tick length of PRET-C programs [Roop et al. 2009]. To further improve the performances of this WCRT analyzer, infeasible execution paths can be discarded, by combining the abstracted state-space of the program with expressive data-flow information [Andalam et al. 2011].

Finally, Ju et al. improved the timing analysis of C code synthesized from Esterel with the CEC compiler by taking advantage of the properties of Esterel [Ju et al. 2008]. They developed an ILP formulation to eliminate redundant paths in the code. This allows more predictable code to be generated.

4.7. Related Work

The seminal paper on PRET languages and architectures was from Edwards and Lee [Edwards and Lee 2007]. They further introduced the so-called Berkeley-Columbia PRET language [Lickly et al. 2008b]. This PRET language is a multi-threaded version of C, extended with a special `deadi` instruction with two arguments, a deadline register `$t` and an immediate value `v`. Placed inside a thread, a "`deadi $t,v`" arms a timer which initializes `$t` with `v`, decrements `$t` every six clock cycles[4], and blocks the thread whenever `$t` is not yet zero. Hence, a `deadi` can only enforce a lower bound on the execution time of code segment. By assigning well chosen values to the `deadi` instructions, it is therefore possible to design predictable multi-threaded systems, where problems such as race conditions will be avoided thanks to the interleaving resulting from the `deadi` instructions.

4.8. Conclusions and Future Work

The synchronous semantics of PRET-C and SC provides correct-by-construction features (i.e., determinism, thread-safe communication, causality, absence of race conditions, and so on), which are essential to design complex predictable systems. For this reason, we argue that these languages are safer than asynchronous (or non concurrent) languages. Numerous examples of reactive systems have been re-implemented with PRET-C or SC, showing that these languages are very easy to use.

Originally developed mainly with functional determinism in mind, the synchronous programming paradigm has also demonstrated its benefits with respect to timing determinism. However, synchronous concepts still have to find their way into mainstream programming of real-time systems. At this point, this seems less a question of the maturity of synchronous languages or the synthesis and analysis procedures developed for them, but rather a question of how to integrate them into programming and architecture paradigms entrenched today. Possibly, this is best done by either enhancing a widely used language such as C with a small set of synchronous/reactive operations, or by moving from the programming level to the modeling level, where concurrency and preemption are already fully integrated.

5. COMPILATION FOR TIMING PREDICTABLE SYSTEMS

Software development for embedded systems uses high-level languages like C, and compilers that include a vast variety of optimizations. However, they mostly aim at reducing *average-case execution times*. The effect of optimizations on worst-case timings has not been studied in-depth up to now. In addition, even modern compilers are often unable to quantify the effect of an optimization since they lack precise timing models.

Currently, software design for real-time systems is tedious: they are often specified graphically using tools like e.g., Matlab/Simulink. These tools automatically generate C code which is compiled in the next step. Since usual compilers have no notion of timing, their optimizations may highly degrade WCETs. Thus, it is common industrial practice to disable most if not all compiler optimizations. The compiler-generated code is then manually fed into a timing analyzer. Only after this very final step in the entire design flow, it can be verified if timing constraints are met. If not, the graphical design is changed in the hope that the resulting C and assembly codes lead to a lower WCET.

Up to now, no tools exist that assist the designer to purposively reduce WCETs of C or assembly code, or to automate the above design flow. In addition, hardware resources are heavily oversized due to the use of unoptimized code. Thus, it is desirable to have a WCET-aware compiler in order to support compilation for timing predictable

[4]Every six clock cycles because the architecture is pipelined with a six-stages pipeline.

systems. Integrating timing analysis into the compiler itself has the following benefits: first, it introduces a formal worst-case timing model such that the compiler has a clear notion of a program's worst-case behavior. Second, this model is exploited by specialized optimizations reducing the WCET. Thus, unoptimized code no longer needs to be used, cheaper hardware platforms tailored towards the real software resource requirements can be used, and the tedious work of manually reducing the WCET of auto-generated C code is eliminated. Third, manual WCET analysis is no more required since this is integrated into and done transparently by the compiler.

5.1. Related Work

A very first approach to integrate WCET techniques into a compiler was presented by [Börjesson 1996]. Flow facts used for timing analysis were annotated manually via source-level pragmas but are not updated during optimization. This turns the entire approach tedious and error-prone. Additionally, the compiler targets the Intel 8051, i.e. an inherently simple and predictable machine without pipeline and caches etc.

While mapping high-level code to object code, compilers apply various optimizations so that the correlation between high-level flow facts and the optimized object code becomes very low. To keep track of the influence of compiler optimizations on high-level flow facts, co-transformation of flow facts is proposed by [Engblom 1997]. However, the co-transformer has never reached a fully working state, and several standard compiler optimizations can not be modeled at all due to insufficient data structures.

Techniques to transform program path information which keep high-level flow facts consistent during GCC's standard optimizations have been presented by [Kirner and Puschner 2001]. Their approach was thoroughly tested and led to precise WCET estimates. However, compilation and timing analysis are done in a decoupled way. The assembly file generated by the compiler is passed to the timing analyzer together with the transformed flow facts. Additionally, the proposed compiler is only able to process a subset of ANSI-C, and the modeled target processor lacks pipelines and caches.

[Zhao et al. 2005a] integrated a proprietarily developed WCET analyzer into a compiler operating on a low-level *intermediate representation (IR)*. Control flow information is passed to the analyzer that computes the worst-case timing of paths, loops and functions and returns this data to the compiler. However, the timing analyzer works with only very coarse granularity since it only computes WCETs of paths, loops and functions. WCETs for basic blocks or single instructions are unavailable. Thus, aggressive optimization of smaller units like single basic blocks is infeasible. Furthermore, important data that is not the WCET itself is unavailable. This excludes e.g., execution frequencies of basic blocks, value ranges of registers, predicted cache behavior etc. Finally, WCET optimization at higher levels of abstraction like e.g., source code level is infeasible since timing-related data is not provided at source code level.

5.2. Structure of the WCET-aware C Compiler WCC

The most advanced compiler for timing predictable systems is the WCET-aware C Compiler [WCC 2012] developed within the ArtistDesign NoE. This section presents WCC in more detail as a case study on how compilers for timing predictable systems could look like. WCC is an ANSI-C compiler for Infineon TriCore processors that are heavily used in the automotive industry. The following subsections describe the key components turning WCC into a unique compiler for real-time systems. A complete description of the compiler's infrastructure is given in [Falk and Lokuciejewski 2010].

Specification of Memory Hierarchies

The performance of many systems is dominated by the memory subsystem. Obviously, timing estimates also heavily depend on the memories. In the WCC environment, it

is up to the compiler to provide the WCET analyzer with detailed information about the underlying memory hierarchy. Thus, the compiler uses an infrastructure to specify memory hierarchies. Furthermore, it exploits this memory hierarchy infrastructure to apply memory-aware optimization by assigning parts of a program to fast memories.

WCC provides a simple interface to specify memory hierarchies. For each physical memory region, attributes like e. g., base address, length, access latency etc. can be defined. For caches, parameters like e. g., size, line size or associativity can be specified. Memory allocation of program parts is now done in the compiler's back-end by allocating functions, basic blocks or data to these memory regions. The compiler provides a convenient programming interface to do such memory allocations of code and data.

Integration of Static WCET Analysis into the Compiler

To obtain a formal worst-case timing model, the compiler's back-end integrates the static WCET analyzer aiT. During timing analysis, aiT stores the program under analysis and its analysis results in an IR called CRL2. Thus, aiT is integrated into WCC by translating the compiler's assembly code IR to CRL2 and vice versa.

Moreover, physical memory addresses provided by WCC's memory hierarchy infrastructure are exploited during CRL2 generation. Using WCC's memory hierarchy API, physical addresses for basic blocks are determined and passed to aiT. Targets of jumps, which are represented by symbolic block labels, are translated into physical addresses.

Using this infrastructure, WCC produces a CRL2 file modeling the program for which worst-case timing data is required. Fully transparent to the compiler user, aiT is called on this CRL2 file. After timing analysis, the results obtained by aiT are imported back into the compiler. Among others, this includes: worst-case execution time of a whole program, or per function or basic block; worst-case execution frequency per function or basic block; approximations of register values; cache misses per basic block.

Flow Fact Specification and Transformation

A program's execution time (on a given hardware) largely depends on its control flow, e. g., on loops or conditionals. Since loop iteration counts are crucial for precise WCETs, and since they can not be computed automatically in general, they must be specified by the user of a timing analyzer. These user-provided control flow annotations are called *flow facts*. WCC fully supports source-level flow facts by means of ANSI-C pragmas.

Loop bound flow facts limit the iteration counts of regular loops. They allow to specify the minimum and maximum iteration counts. For example, the following C code snippet specifies that the shown loop body is executed 50 to 100 times:

```
_Pragma( "loopbound min 50 max 100" )
for ( i = 1; i <= maxIter; i++ )
  Array[ i ] = i * fact * KNOWN_VALUE;
```

A definition of minimum and maximum iteration counts allows to annotate data-dependent loops (see above). For irregular loops or recursions, *flow restrictions* are provided that relate the execution frequency of one C statement with that of others.

However, compiler optimizations potentially restructure the code and invalidate originally specified flow facts. Therefore, WCC's optimizations are fully flow-fact aware. All operations of the compiler's IRs creating, deleting or moving statements or basic blocks now automatically update flow facts. This way, always safe and precise flow facts are maintained, irrespective of how and when optimizations modify the IRs.

5.3. Examples of WCET-aware Optimizations

On top of the compiler infrastructure described above, a large number of novel WCET-aware optimizations are integrated into WCC. The following sections briefly present three of them: scratchpad allocation, code positioning and cache partitioning.

Scratchpad Memory Allocation and Cache Locking

As already motivated in Section 3.2, scratchpad memories (SPMs) or locked caches are ideal for WCET-centric optimizations since their timing is fully predictable. Optimizations allocating parts of a program's code and data onto these memories have been studied intensely in the past [Wehmeyer and Marwedel 2005; Campoy et al. 2005; Suhendra et al. 2005].

WCC exploits scratchpads by placing parts of a program into an SPM [Falk and Kleinsorge 2009] using *integer linear programming (ILP)*. Inequations model the structure of a program's *control flow graph (CFG)*. Constants model the worst-case timing per basic block when being allocated to slow main memory or to the fast SPM. This way, the ILP is always aware of that path in the CFG leading to the longest execution time and can thus optimally minimize the WCET. Besides scratchpads, the compiler also supports cache locking using a similar optimization approach [Plazar et al. 2012].

Experimental results over a total of 73 different benchmarks from e.g. UTDSP, MediaBench and MiBench for the Infineon TriCore TC1796 processor show that already very small scratchpads, where only 10% of a benchmark's code fit into, lead to considerable WCET reductions of 7.4%. Maximum WCET reductions of up to 40% on average over all 73 benchmarks have been observed.

Code Positioning

Code positioning is a well-known compiler optimization improving the I-cache behavior. A contiguous mapping of code fragments in memory avoids overlapping of cache sets and thus decreases the number of cache conflict misses. Code positioning as such was studied in many different contexts in the past, like e. g. to avoid jump-related pipeline delays [Zhao et al. 2005b] or at granularity of entire functions [Lokuciejewski et al. 2008] or tasks [Gebhard and Altmeyer 2007].

WCC's code positioning [Falk and Kotthaus 2011] aims to systematically reduce I-cache conflict misses and thus to reduce the WCET of a program. It uses a *cache conflict graph (CG)* as the underlying model of a cache's behavior. Its nodes represent either functions or basic blocks of a program. An edge is inserted whenever two nodes interfere in the cache, i. e. potentially evict themselves from the cache. Using WCC's integrated timing analysis capabilities, edge weights are computed which approximate the number of possible cache misses that are caused during the execution of a CG node.

On top of the conflict graph, heuristics for contiguous and conflict-free placement of basic blocks and entire functions are applied. They iteratively place those two basic blocks / functions contiguously in memory which are connected by the edge with largest weight in the conflict graph. After this single positioning step, the impact of this change on the whole program's worst-case timing is evaluated by doing a timing analysis. If the WCET is reduced, this last positioning step is kept, otherwise it is undone.

This code positioning decreases cache misses for 18 real-life benchmarks by 15.5% on average for an Infineon TC1797 with a 2-way set-associative cache. These cache miss reductions translate to average WCET reductions by 6.1%. For direct-mapped caches, even larger savings of 18.8% (cache misses) and 9.0% (WCET) were achieved.

Cache Partitioning for Multi-Task Systems

The cache-related optimizations presented so far cannot handle multi-task systems with preemptive scheduling, since it is difficult to predict the cache behavior during context switches. Cache partitioning is a technique for multi-task systems to turn I-caches more predictable. Each task of a system is exclusively assigned a unique cache partition. The tasks in such a system can only evict cache lines residing in the partition they are assigned to. As a consequence, multiple tasks do not interfere with each other

any longer w.r.t. the cache during context switches. This allows to apply static timing analysis to each individual task in isolation. The overall WCET of a multi-task system using partitioned caches is then composed of the worst-case timing of the single tasks given a certain partition size, plus the overhead for scheduling and context switches.

WCET-unaware cache partitioning has already been examined in the past. Cache hardware extensions and associativity- and set-based cache partitioning have been proposed in [Chiou et al. 1999] and [Molnos et al. 2004], resp. [Mueller 1995] presents ideas for compiler support for software-based cache partitioning which serves as basis for WCC's cache partitioning. Software-based cache partitioning scatters the code of each task over the address space such that tasks are solely mapped to only those cache lines belonging to the task's partition. WCC's cache partitioning [Plazar et al. 2009] again relies on ILP to optimally determine the individual tasks' partition sizes.

Cache partitioning has been applied to task sets with 5, 10 and 15 tasks, resp. Compared to a naive code size-based heuristic for cache partitioning, WCC's approach achieves substantial WCET reductions of up to 36%. In general, WCET savings are higher for small caches and lower for larger caches. In most cases, larger task sets exhibit a higher optimization potential as compared to smaller task sets.

5.4. Conclusions and Future Work

This section discussed compiler techniques and concepts for timing predictable systems by exploiting a worst-case timing model. Up till now, not much was known about the WCET savings achievable this way. This section provided a survey over research work exploring the potential of such integrated compilation and timing analysis.

The WCET-aware C Compiler WCC served as case study of a compiler for timing predictable systems. Currently, WCC focuses on code optimization for single-task and single-core systems. Just recently, first steps towards support of multi-task or multi-core systems were made. Therefore, WCET-aware optimizations for multi-task and multi-core systems is the main focus for future work in this area.

6. BUILDING REAL-TIME APPLICATIONS ON MULTICORES

6.1. Background

Multicore processors bring a great opportunity for high-performance and low-power embedded applications. Unfortunately, the current design of multicore architectures is mainly driven by performance, not by considering timing predictability. Typical multicore architectures [Albonesi and Koren 1994] integrate a growing number of cores on a single processor chip, each equipped with one or two levels of private caches. The cores and peripherals usually share a memory hierarchy including L2 or L3 caches and DRAM or Flash memory. An interconnection network offers a communication mechanism between the cores, the I/O peripherals and the shared memory. A shared bus can hold a limited number of components as in the ARM Cortex A9 MPCORE. Larger-scale architectures implement more complex Networks on Chip (NoC), like meshes (e.g. the Tile64 by Tilera) or crossbars (e.g. the P4080 by Freescale), to offer a wider communication bandwidth. In all cases, conflicts among accesses from various cores or DMA peripherals to the shared memory must be arbitrated either in the network or in the memory controller. In the following, we distinguish between *storage resources* (e.g. caches) that keep information for a while, generally for several cycles and *bandwidth resources* (e.g. bus or interconnect) that are typically reallocated at each cycle.

6.2. Timing Interferences and Isolation

The timing behavior of a task running on a multicore architecture depends heavily on the arbitration mechanism of the shared resources and other tasks' usage of the re-

sources. First, due to the conflicts with other requesting tasks on bandwidth resources, the instruction latencies may be increased and can even be unbounded. Furthermore, the contents of storage resources especially caches may be corrupted by other tasks, which results in an increased number of misses. Computing safe WCET estimates requires taking into account the additional delays due to the activity of co-scheduled tasks.

To bound the timing interferences, there are two categories of potential solutions. The first, referred to as *joint analysis*, considers the whole set of tasks competing for shared resources to derive bounds on the delays experienced by each individual task. This usually requires complex computations, and it may provide tighter WCET bounds. However, it is restricted to cases where all the concurrent tasks are statically known. The second approach aims at enforcing *spatial and temporal isolation* so that a task will not suffer from timing interferences by other tasks. Such an isolation can be controlled by software and/or hardware.

Joint Analysis. To estimate the WCETs of concurrent tasks, a joint analysis approach considers all the tasks together to accurately capture the impact of interactions on the execution times. A simple approach to analyzing a shared cache is to statically identify cache lines shared by concurrent tasks and consider them as corrupted [Hardy et al. 2009] at run time. The analysis can be improved by taking task lifetimes into account: tasks that cannot be executed concurrently due to the scheduling algorithm and inter-task dependencies should not be considered as possibly conflicting. Along this line of work, Li *et al.* [Li et al. 2009] propose an iterative approach to estimate the WCET bounds of tasks sharing L2 caches. To further improve the analysis precision, the timing behaviour of cache access may be modeled and analyzed using abstract interpretation and model checking techniques [Lv et al. 2010]. Other approaches aim at determining the extra execution time of a task due to contention on the memory bus [Bjorn Andersson and Lee 2010; Schliecker et al. 2010]. Decoupling the estimation of memory latencies from the analysis of the pipeline behaviour is a way to enhance analysability. However, it is safe for fully timing-compositional systems only.

Spatial and Temporal Isolation. Ensuring that tasks will not interfere in shared resources makes their WCETs analyzable using the same techniques as for single cores. Task isolation can be controlled by software allowing COTS-based multicores or enforced by hardware transparent to the applications.

The PRedictable Execution Model [Pellizzoni et al. 2010] requires programs to be annotated by the programmer and then compiled as a sequence of predictable intervals. Each predictable interval includes a memory phase where caches are prefetched and an execution phase that cannot experience cache misses. A high level schedule of computation phases and I/O operations enables the predictability of accesses to shared resources. TDMA-based resource arbitration allocates statically-computed slots to the cores [Rosen et al. 2007; Andrei et al. 2008]. To predict latencies, the alignment of basic block time-stamps to the allocated bus slots can be analyzed [Chattopadhyay et al. 2010]. However, TDMA-based arbitration is not so common in multicore processors on the market due to performance reasons.

To make the latencies to shared bandwidth resources predictable (boundable), hardware solutions rely on bandwidth partitioning techniques, e.g. round-robin arbitration [Paolieri et al. 2009a]. Software-controlled cache partitioning schemes allocate private partitions to tasks. For example, *Page-coloring* [Guan et al. 2009a] allocates the cache content of each task to certain areas in the shared cache by mapping the virtual memory addresses of that task to proper physical memory regions. Then the avoidance of cache interference does not come for free, as the explicit management of cache space adds another dimension to the scheduling and complicates the analysis.

6.3. System-Level Scheduling and Analysis

For single-processor platforms, there are well-established techniques (e.g. rate-monotonic scheduling) for system-level scheduling and schedulability analysis. The designer may rely on the WCET bounds of tasks and allocate computing resources accordingly to ensure system-level timing guarantees. For multicore platforms, one may take a similar approach. However the *multiprocessor scheduling* problem to map tasks onto parallel architectures is a much harder challenge. No well-established techniques exist but various scheduling strategies with mostly sufficient conditions for schedulability have been proposed.

Global Scheduling. One may allow all tasks to compete for execution on all cores. Global scheduling is a realistic option for multcore systems, on which the task migration overhead is much less significant compared with traditional loosly-coupled multiprocessor systems thanks to the hardware mechanisms like on-chip shared cache. So a rapidly increasing interest rises in the study of global scheduling since the late 1990s, around the same time as the major silicon vendors such as IBM and AMD started the development of multicore processors. Global multiprocessor scheduling is a much more difficult problem than uniprocessor scheduling, as first pointed out by Liu in 1969 [Liu 1969]: *The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.*

One may simply adopt a global task queue and map the released tasks onto the parallel processing cores using single-processor scheduling algorithms such as RM and EDF. Unfortunately these algorithms suffer from the so-called *Dhall effect* [Dhall and Liu 1978], namely some system with utilization arbitrarily close to 1 can be infeasible no matter how many processors are added to the system. This result leads to the negative view that global scheduling is widely considered unsuitable for real-time systems. One way to overcome the Dhall effect is fairness scheduling [Baruah et al. 1996], which splits up the task's execution into small pieces and interleaves them with other tasks, to keep the execution of a task to progress in the speed proportional to its workload. Fairness scheduling and its variants [Anderson and Srinivasan 2001] can achieve optimality, but is usually considered impracticable to implement due to the run-time overheads.

The major obstacle in precisely analyzing global scheduling and thereby fully exploring its potential is that global scheduling suffers from *timing anomalies*, i.e., a schedulable system can become unschedulable by a parameter change that appears to be harmless. In uniprocessor fixed-priority scheduling the critical instant is the situation where all the interfering tasks release their first instance simultaneously and all the following instances are released as soon as possible. Unfortunately, the critical instant in global scheduling is in general unknown. The critical instant in uniprocessor scheduling, with a strong intuition of resulting in the maximal system workload, does not necessarily lead to the worst-case situation in global fixed-priority scheduling [Lauzac et al. 1998]. Therefore, the analysis of global scheduling requires to explore all the possible system behavior.

A large body of works has been done on the efficient analysis of global scheduling by over-approximation. The common approach is to derive an upper bound on the total workload of a task system. Much work has been done on tightening the workload estimation by excluding impossible system behavior from the calculation (e.g. [Bertogna and Cirinei 2007; Guan et al. 2009b]). The work in [Guan et al. 2009b] established the concept of *abstract critical instant* for global fixed-priority scheduling, namely the worst-case response time of a task occurs under the situation that all higher-priority tasks, except at most $M-1$ of them (M is the number of processors), are released in

the same way as the critical instant in uniprocessor fixed-priority scheduling. Although the *abstract critical instant* does not provide an accurate worst-case release pattern, it restricts the analysis to a significantly smaller subset of the overall state space.

The uniprocessor scheduling algorithms like RM and EDF lose their optimality on multicores, which gives rise to the question of what are actually the *good* global scheduling strategies? The fundamental work on global scheduling [Devi and Anderson 2005] showed that global EDF, although it can not guarantee deadlines under full workload (100% utilization) any longer, still maintains a weaker concept of optimality in the sense of guaranteeing bounded tardiness (response time) under full workload. In contrast, global fixed-priority scheduling is proved to be able to guarantee bounded tardiness (response time) under a more restricted condition [Guan et al. 2009b].

Partitioned Scheduling. For a long time, the common wisdom in multiprocessor scheduling is to partition the system into subsets each of which is scheduled on a single processor [Carpenter et al. 2004]. The design and analysis of partitioned scheduling is relatively simple: as soon as the system has been partitioned into subsystems that will be executed on individual processors each, the traditional uniprocessor real-time scheduling and analysis techniques can be applied to each individual subsystem/processor. The system partitioning is similar to the bin-packing problem [Coffman et al. 1997], for which efficient heuristics are known although it is in general intractable. Similar to the bin-packing problem, partitioned scheduling suffers from resource waste due to fragmentation. Such a waste will be more significant, as the multi core evolves in the direction to integrate a larger number of less powerful cores and the workload of each task becomes relatively heavier comparing with the processing capacity of each individual core. Theoretically, the worst-case utilization bound of partitioned scheduling can not exceed 50% regardless of the local scheduling algorithm on each processor [Carpenter et al. 2004].

To overcome this theoretical bound, one may take a hybrid approach where most tasks may be allocated to a fixed core, while only a small number of tasks are allowed to run on different cores, which is similar to task migration but in a controlled and predictable mannor as the migrating tasks are mapped to dedicated cores statically. This is sometimes called *semi-partitioned scheduling*. Similar to splitting the items into small pieces in the bin-packing problem, semi-partitioned scheduling can very well solve the resource waste problem in partitioned scheduling and exceed the 50% utilization bound limit. On the other hand, the context-switch overhead of semi-partitioned scheduling is smaller than global scheduling as it involves less task migration between different cores.

Several different partitioning and splitting strategies have been applied to both fixed-priority and EDF scheduling (e.g. [Lakshmanan et al. 2009; Guan et al. 2010]). Recently, a notable result is obtained in [Guan et al. 2010], which generalizes the famous Liu and Layland's utilization bound $N \times (2^{\frac{1}{N}} - 1)$ [Liu and Layland 1973] for uniprocessor fixed priority scheduling to multicores by a semi-partitioned scheduling algorithm using RM [Liu and Layland 1973] on each core. This result is further extended to generalize various parametric utilization bounds (for example the 100% utilization bound for *harmonic* task systems) to multi cores [Guan et al. 2012]. Another hybrid approach combining global and partitioned scheduling is *clustered scheduling* [Bastoni et al. 2010b], which partitions the processor cores into subsets (called a cluster each), and uses global scheduling to schedule the subset of tasks assigned to each cluster. Clustered scheduling suffers less resource waste than partitioned scheduling, and may reduce the context switch penalty than global scheduling on hardware architectures where cores are actually grouped into clusters with closer resource sharing.

Implementation and Evaluation. To evaluate the performance and applicability of different scheduling paradigms in RTOS supporting multicore architectures, LITMUS^{RT} [Calandrino et al. 2006], a Linux-based testbed for real-time multiprocessor scheduling has been developed. Much research has been done using the testbed to account for the (measured) run-time overheads of various multiprocessor scheduling algorithms in the respective theoretical analysis (e.g. [Bastoni et al. 2010b]). The run-time overheads include mainly the scheduler latency (typically several tens μs in Linux [Zhang et al. 2011]) and cache-related costs, which depends on the application work space characterization, and can vary between several μs and tens of ms [Bastoni et al. 2010a; Zhang et al. 2011]. Their studies indicate that partitioned scheduling and global scheduling have both pros and cons, but partitioned scheduling performs better for hard real-time applications [Bastoni et al. 2010b]. Clustered scheduling exhibits competitive performance on cluster-based multi-core architectures as it mitigates both the high run-time overhead in global scheduling and the resource waste of fragmentation in partitioned scheduling. Recently, evaluations have also been done with semi-partitioned scheduling algorithms [Bastoni et al. 2011], together with the work in [Zhang et al. 2011; Bletsas and Andersson 2011], indicating that semi-partitioned scheduling is indeed a promising scheduling paradigm for multicore real-time system. The work of [Zhang et al. 2011] shows that on multicore processors equipped with shared caches and high-speed inter-connections, task migration overhead is typically with the same order of magnitude as intra-core context-switch overhead; for example, on an Intel Core-i7 4-cores machine running LINUX, the typical costs for task migration are in the scale of one to two hundred μs for a task with one MB working size.

6.4. Conclusion and Challenges

On multicore platforms, to predict the timing behaviour of an individual task, one must consider the global behaviour of all tasks on all cores and also the resource arbitration mechanisms. To trade timing composability and predictability with performance decreases, one may partition the shared resource with performance decreases. For storage resource, page-coloring may be used to avoid conflicts and ensure bounded delays. Unfortunately, it is not clear how to partition a bandwidth resource unless a TDMA-like arbitration protocol is used. To map real-time tasks onto the processor cores for system-level resource management and integration, a large number of scheduling techniques has been developed in the area of multiprocessor scheduling. However, the known techniques all rely on safe WCET bounds of tasks. Without proper spatial and temporal isolation, it seems impossible to achieve such bounds. To the best of our knowledge, there is no work on bridging WCET analysis and multiprocessor scheduling. Future challenges include also integrating different types of real-time applications with different levels of criticality on the same platform to fully utilize the computation resources for low-criticality applications and to provide timing guarantees for high-criticality applications.

7. CONCLUSIONS

In this paper, we have surveyed some recent advances regarding techniques for building timing predictable embedded systems. A previous survey [Thiele and Wilhelm 2004] examined the then state-of-the-art regarding techniques for building predictable systems, and outlined some directionss ahead. We can now see that interesting developments have occurred along several of them.

In [Thiele and Wilhelm 2004], one suggested path was to integrate timing analysis across several design layers. The development of the WCC compiler, and of timing-predictable synchronous languages, are offering a solution to this problem, at least on task-level. Another suggestion was to develop better coordination of shared resources:

this is becoming critical with the advent of multicores. Although good solutions for predictable systems on multicore are not yet available, the understanding of the necessary elements towards this goal has increased significantly. But perhaps the main obstacle for building truly predictable systems is that although it is in many respects understood how to build predictable systems, the building blocks for actually realizing them are not available in today's processor platforms.

References

1991. Can specification 2.0.

2004. Iso 11898-4:2004 road vehicles – controller area network (can) – part 4: Time-triggered communication.

2008. Functional safety of electrical/electronic/programmable electronic safety-related systems (iec 61508).

AKESSON, B., GOOSSENS, K., AND RINGHOFER, M. 2007. Predator: A predictable SDRAM memory controller. In *CODES+ISSS '07*. 251–256.

ALBONESI, D. H. AND KOREN, I. 1994. Tradeoffs in the design of single chip multiprocessors. In *2nd International Conference on Parallel Architectures and Compilation Techniques*.

ANDALAM, S., ROOP, P., AND GIRAULT, A. 2010. Predictable multithreading of embedded applications using PRET-C. In *International Conference on Formal Methods and Models for Codesign, MEMOCODE'10*. Grenoble, France.

ANDALAM, S., ROOP, P., AND GIRAULT, A. 2011. Pruning infeasible paths for tight WCRT analysis of synchronous programs. In *Design Automation and Test in Europe Conference, DATE'11*. Grenoble, France.

ANDERSON, J. H. AND SRINIVASAN, A. 2001. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *ECRTS*.

ANDRÉ, C. 2003. Semantics of SyncCharts. Tech. Rep. ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France. April.

ANDREI, A., ELES, P., PENG, Z., AND ROSEN, J. 2008. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSID*.

AUSSAGUÈS, C., CHABROL, D., DAVID, V., ROUX, D., WILLEY, N., TOURNADRE, A., AND GRANIOU, M. 2010. Pharos, a multicore os ready for safety-related automotive systems: results and future prospects. In *Embedded Real Time Software and Systems*.

AVIZIENIS, A., LAPRIE, J., AND RANDELL, B. 2000. Fundamental concepts of dependability. In *3rd IEEE Information Survivability Workshop (ISW)*. 7 – 12.

AXER, P., SEBASTIAN, M., AND ERNST, R. 2011. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Taiwan.

BARRE, J., ROCHANGE, C., AND SAINRAT, P. 2008. A predictable simultaneous multithreading scheme for hard real-time. In *Architecture of computing systems '08*. 161–172.

BARUAH, S. K., COHEN, N. K., PLAXTON, C. G., AND VARVEL, D. A. 1996. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*.

BASTONI, A., BRANDENBURG, B., AND ANDERSON, J. 2011. Is semi-partitioned scheduling practical. *ECRTS*.

BASTONI, A., BRANDENBURG, B. B., AND ANDERSON, J. H. 2010a. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *OSPERT*.

BASTONI, A., BRANDENBURG, B. B., AND ANDERSON, J. H. 2010b. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. *RTSS*.

BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. 2003. The synchronous languages twelve years later. *Proceedings of the IEEE 91,* 1, 64–83. Special issue on embedded systems.

BERG, C. 2006. PLRU cache domino effects. In *WCET '06*. IBFI, Schloss Dagstuhl, Germany.

BERRY, G. 2000. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 425–454.

BERTOGNA, M. AND CIRINEI, M. 2007. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*.

BJORN ANDERSSON, A. E. AND LEE, J. 2010. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. In *ACM SIGBED Review*.

BLETSAS, K. AND ANDERSSON, B. 2011. Implementing slot-based task-splitting multiprocessor scheduling. In *SIES*.

BOLDT, M., TRAULSEN, C., AND VON HANXLEDEN, R. 2008. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems 2008*, 1–21.

BÖRJESSON, H. 1996. Incorporating worst case execution time in a commercial c-compiler. M.S. thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden.

BROSTER, I., BERNAT, G., AND BURNS, A. 2002a. Weakly hard real-time constraints on controller area network. In *14th Euromicro Conference on Real-Time Systems, 2002. Proceedings*. 134–141.

BROSTER, I., BURNS, A., AND RODRÍGUEZ-NAVAS, G. 2002b. Probabilistic analysis of can with faults. In *Proceedings of the 23rd Real-Time Systems Symposium*. 269–278.

BROSTER, I., BURNS, A., AND RODRIGUEZ-NAVAS, G. 2004. Comparing real-time communication under electromagnetic interference. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. IEEE Computer Society.

BURNS, A., PUNNEKKAT, S., STRIGINI, L., AND WRIGHT, D. 1999. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications*.

CALANDRINO, J. M., LEONTYEV, H., BLOCK, A., DEVI, U. C., AND ANDERSON, J. H. 2006. Litmus^{r}t : A testbed for empirically comparing real-time multiprocessor schedulers. *RTSS*.

CAMPOY, A. M., PUAUT, I., IVARS, A. P., ET AL. 2005. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. Palma de Mallorca, Spain, 49–56.

CARPENTER, J., FUNK, S., HOLMAN, P., SRINIVASAN, A., ANDERSON, J., AND BARUAH, S. 2004. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*.

CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND MITRA, T. 2010. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*.

CHIOU, D., RUDOLPH, L., DEVADAS, S., AND ANG, B. S. 1999. Dynamic cache partitioning via columnization. Tech. Rep. 430, Massachusetts Institute of Technology, Cambridge, United States. Nov.

COFFMAN, E. G., GAREY, M. R., AND JOHNSON, D. S. 1997. *Approximation algorithms for bin packing: a survey*.

DEVI, U. AND ANDERSON, J. 2005. Tardiness bounds for global EDF scheduling on a multiprocessor. In *RTSS*.

DHALL, S. K. AND LIU, C. L. 1978. On a real-time scheduling problem. In *Operations Research, Vol. 26, No. 1, Scheduling*.

EDWARDS, S. AND LEE, E. 2007. A case for precision timed (PRET) machine. In *Design Automation Conference, DAC'07*. IEEE, Los Alamitos, San Diego (CA), USA, 264–265.

EDWARDS, S. AND ZENG, J. 2007. Code generation in the Columbia Esterel Compiler. *EURASIP J. on Embedded Systems*. Article ID 52651.

EL-HAJ-MAHMOUD, A., AL-ZAWAWI, A. S., ANANTARAMAN, A., AND ROTENBERG, E. 2005. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proc. of CASES*. ACM, New York, NY, USA, 213–224.

ENGBLOM, J. 1997. Worst-case execution time analysis for optimized code. M.S. thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden.

FALK, H. AND KLEINSORGE, J. C. 2009. Optimal static wcet-aware scratchpad allocation of program code. In *Proceedings of the 46th Design Automation Conference (DAC)*. San Francisco, United States, 732–737.

FALK, H. AND KOTTHAUS, H. 2011. Wcet-driven cache-aware code positioning. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. Taipei, Taiwan, 145–154.

FALK, H. AND LOKUCIEJEWSKI, P. 2010. A compiler framework for the reduction of worst-case execution times. *The International Journal of Time-Critical Computing Systems (Real-Time Systems) 46,* 2, 251–300.

FERREIRA, J., OLIVEIRA, A., FONSECA, P., AND FONSECA, J. 2004. An experiment to assess bit error rate in can. In *Proceedings of 3rd International Workshop of Real-Time Networks (RTN2004*. 15–18.

GEBHARD, G. AND ALTMEYER, S. 2007. Optimal task placement to improve cache performance. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. Salzburg, Austria, 259–268.

GUAN, N., STIGGE, M., YI, W., AND YU, G. 2009a. Cache-aware scheduling and analysis for multicores. In *EMSOFT*.

GUAN, N., STIGGE, M., YI, W., AND YU, G. 2009b. New response time bounds of fixed priority multiprocessor scheduling. In *RTSS*.

GUAN, N., STIGGE, M., YI, W., AND YU, G. 2010. Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In *RTAS*.

GUAN, N., STIGGE, M., YI, W., AND YU, G. 2012. Parametric utilization bounds for fixed-priority multiprocessor scheduling. In *IPDPS*.

HARDY, D., PIQUET, T., AND PUAUT, I. 2009. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *RTSS*.

IZOSIMOV, V., POP, P., ELES, P., AND PENG, Z. 2005. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*. IEEE Computer Society, 864–869.

JU, L., HUYNH, B. K., ROYCHOUDHURY, A., AND CHAKRABORTY, S. 2008. Performance debugging of Esterel specifications. In *CODES+ISSS*. 173–178.

KAHN, G. AND MACQUEEN, D. B. 1977. Coroutines and networks of parallel processes. In *IFIP Congress*. 993–998.

KIM, K., DIAZ, J., BELLO, L., LOPEZ, J., LEE, C.-G., AND MIN, S. L. 2005. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *Computers, IEEE Transactions on 54,* 11, 1460 – 1466.

KIRNER, R. AND PUSCHNER, P. 2001. Transformation of path information for wcet analysis during compilation. In *Proceedings of ECRTS*. Delft, Netherlands.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA.

LAKSHMANAN, K., RAJKUMAR, R., AND LEHOCZKY, J. 2009. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*.

LAUZAC, S., MELHEM, R., AND MOSSE, D. 1998. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *ECRTS*.

LEE, E. 2006. The problem with threads. *IEEE Comput. 5*, 33–42.

LI, X. AND VON HANXLEDEN, R. 2010. Multi-threaded reactive programming—the Kiel Esterel Processor. *IEEE Transactions on Computers*.

LI, Y., SUHENDRA, V., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*.

LICKLY, B., LIU, I., KIM, S., PATEL, H., EDWARDS, S., AND LEE, E. 2008a. Predictable programming on a precision timed architecture. In *CASES '08*. 137–146.

LICKLY, B., LIU, I., KIM, S., PATEL, H. D., EDWARDS, S. A., AND LEE, E. A. 2008b. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*. Atlanta, USA.

LIU, C. L. 1969. Scheduling algorithms for multiprocessors in a hard real-time environment. In *JPL Space Programs Summary*.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*.

LIU, I., REINEKE, J., AND LEE, E. A. 2010. A pret architecture supporting concurrent programs with composable timing properties. In *44th Asilomar Conference on Signals, Systems, and Computers*. 2111–2115.

LOGOTHETIS, G., SCHNEIDER, K., AND METZLER, C. 2003. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, Cancun, Mexico, 256–264.

LOKUCIEJEWSKI, P., FALK, H., AND MARWEDEL, P. 2008. Wcet-driven cache-based procedure positioning optimizations. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS 08)*. Prague, Czech Republic, 321–330.

LUNDQVIST, T. AND STENSTRÖM, P. 1999. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '09*. 12–21.

LUNDQVIST, T. AND STENSTRÖM, P. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*. 12–21.

LV, M., YI, W., GUAN, N., AND YU, G. 2010. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*.

MENDLER, M., VON HANXLEDEN, R., AND TRAULSEN, C. 2009. Wcrt algebra and interfaces for esterel-style synchronous processing. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'09)*. Nice, France.

MISCHE, J., UHRIG, S., KLUGE, F., AND UNGERER, T. 2008. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *ICCD '08*. 371–376.

MOLNOS, A. M., HEIJLIGERS, M. J. M., COTOFANA, S. D., AND VAN EIJNDHOVEN, J. T. J. 2004. Cache partitioning options for compositional multimedia applications. In *Proceedings of the 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*. Veldhoven, the Netherlands, 86–90.

MUELLER, F. 1995. Compiler support for software-based cache partitioning. In *Proceedings of the Workshop on Languages, Compilers and Tools for Real-Time Systems*. La Jolla, United States, 125–133.

NAVET, N., SONG, Y., AND SIMONOT, F. 2000. Worst-case deadline failure probability in real-time applications distributed over can (controller area network). *Journal of System Architectures 46*, 606–617.

PAOLIERI, M., NONES, E. Q., CAZORLA, F. J., BERNAT, G., AND VALERO, M. 2009a. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*.

PAOLIERI, M., QUINONES, E., CAZORLA, F., AND VALERO, M. 2009b. An analyzable memory controller for hard real-time CMPs. *Embedded Syst. Letters 1,* 4, 86–90.

PELLIZZONI, R., SCHRANZHOFER, A., CHEN, J.-J., CACCAMO, M., AND THIELE, L. 2010. Worst case delay analysis for memory interference in multicore systems. In *DATE*.

PLAZAR, S., FALK, H., KLEINSORGE, J. C., AND MARWEDEL, P. 2012. Wcet-aware static locking of instruction caches. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. San Jose, United States.

PLAZAR, S., LOKUCIEJEWSKI, P., AND MARWEDEL, P. 2009. Wcet-aware software based cache partitioning for multi-task real-time systems. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Dublin, Ireland.

POTOP-BUTUCARU, D., EDWARDS, S. A., AND BERRY, G. 2007. *Compiling Esterel*. Springer.

REINEKE, J. AND GRUND, D. 2008. Sensitivity of cache replacement policies. Reports of SFB/TR 14 AVACS 36, SFB/TR 14 AVACS. March. ISSN: 1860-9821, http://www.avacs.org.

REINEKE, J., LIU, I., PATEL, H. D., KIM, S., AND LEE, E. A. 2011. Pret dram controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*. ACM.

REINEKE, J., WACHTER, B., THESING, S., WILHELM, R., POLIAN, I., EISINGER, J., AND BECKER, B. 2006. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*.

ROCHANGE, C. AND SAINRAT, P. 2005. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Computing Frontiers '05*. 307–314.

ROOP, P., ANDALAM, S., VON HANXLEDEN, R., YUAN, S., AND TRAULSEN, C. 2009. Tight WCRT analysis of synchronous C programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES'09*. ACM, Grenoble, France.

ROSEN, J., ANDREI, A., ELES, P., AND PENG, Z. 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*.

SALCIC, Z. A., ROOP, P. S., BIGLARI-ABHARI, M., AND BIGDELI, A. 2002. REFLIX: A processor core for reactive embedded applications. In *Proceedings of the 12th International Conference on Filed Programmable Logic and Applications (FPL-02)*, M. Glesner, P. Zipf, and M. Renovell, Eds. LNCS Series, vol. 2438. Springer, Montpellier, France, 945–945.

SCHLIECKER, S., NEGREAN, M., AND ERNST, R. 2010. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*.

SCHNEIDER, J. 2003. Combined schedulability and WCET analysis for real-time operating systems. Ph.D. thesis, Saarland University.

SEBASTIAN, M., AXER, P., AND ERNST, R. 2011. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *17th IEEE Pacific Rim International Symposium on Dependable Computing*.

SEBASTIAN, M. AND ERNST, R. 2009. Reliability analysis of single bus communication with real-time requirements. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, 3–10.

SMOLENS, J., GOLD, B., KIM, J., FALSAFI, B., HOE, J., AND NOWATZYK, A. 2004. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ACM SIGARCH Computer Architecture News*. Vol. 32. ACM, 224–234.

STANKOVIC, J. AND RAMAMRITHAM, K. 1990. What is predictability for real-time systems? *Real-Time Syst. 2*, 247–254.

SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., ET AL. 2005. Wcet centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-time Systems Symposium (RTSS)*. Miami, Florida, USA, 223–232.

THIELE, L. AND WILHELM, R. 2004. Design for timing predictability. *Real-Time Syst. 28,* 2-3, 157–177.

TRAULSEN, C., AMENDE, T., AND VON HANXLEDEN, R. 2011. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*. IEEE, Grenoble, France, 563–566.

UNGERER, T. ET AL. 2010. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro 99*.

VON HANXLEDEN, R. 2009. Syncchartsin c—a proposal for light-weight, deterministic concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*. ACM, Grenoble, France, 225–234.

VON HANXLEDEN, R., LI, X., ROOP, P., SALCIC, Z., AND YOONG, L. H. 2006. Reactive processing for reactive systems. *ERCIM News 67*, 28–29.

WCC 2012. Wcet-aware compilation. http://ls12-www.cs.tu-dortmund.de/research/activities/wcc.

WEHMEYER, L. AND MARWEDEL, P. 2005. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design Automation and Test in Europe (DATE)*. Munich, Germany, 600–605.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transaction on Embedded Computing Systems*.

WILHELM, R., GRUND, D., REINEKE, J., SCHLICKLING, M., PISTER, M., AND FERDINAND, C. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. on CAD of Integrated Circuits and Syst. 28,* 7, 966–978.

YUAN, S., ANDALAM, S., YOONG, L. H., ROOP, P. S., AND SALCIC, Z. 2008. STARPro—a new multithreaded direct execution platform for Esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'08)*. Budapest, Hungary.

ZHANG, Y., GUAN, N., YI, W., AND XIAO, Y. 2011. Implementation and empirical comparison of partitioning-based multi-core scheduling. In *SIES*.

ZHAO, W., KREAHLING, W., WHALLEY, D., ET AL. 2005a. Improving wcet by optimizing worst-case paths. In *Proceedings of RTAS*. San Francisco, California.

ZHAO, W., WHALLEY, D., HEALY, C., ET AL. 2005b. Improving wcet by applying a wc code-positioning optimization. *ACM Transactions on Architecture and Code Optimization 2,* 4, 335–365.