# Building Timing Predictable Embedded Systems

PHILIP AXER and ROLF ERNST, TU Braunschweig
HEIKO FALK, Ulm University
ALAIN GIRAULT, INRIA and University of Grenoble
DANIEL GRUND, Saarland University
NAN GUAN and BENGT JONSSON, Uppsala University
PETER MARWEDEL, TU Dortmund
JAN REINEKE, Saarland University
CHRISTINE ROCHANGE, University of Toulouse
MAURICE SEBASTIAN, TU Braunschweig
REINHARD VON HANXLEDEN, CAU Kiel
REINHARD WILHELM, Saarland University
WANG YI, Uppsala University

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 uncertainty and make guarantees harder to provide. The intention of this article is to summarize the current state of the art in research concerning how to build predictable yet performant systems. We suggest precise definitions for the concept of "predictability", and present predictability concerns at different abstraction levels in embedded system design. First, we consider timing predictability of processor instruction sets. Thereafter, we consider how programming languages can be equipped with predictable timing semantics, covering both a language-based approach using the synchronous programming paradigm, as well as an environment that provides timing semantics for a mainstream programming language (in this case C). We present techniques for achieving timing predictability on multicores. Finally, we discuss how to handle predictability at the level of networked embedded systems where randomly occurring errors must be considered.

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, safety-critical systems, predictability, timing analysis, resource sharing

This work was supported by the ArtistDesign Network of Excellence (European Commission, grant no. IST-214373) as part of the transversal activity "Design for Predictability and Performance."
Author's address: B. Jonsson, bengt@it.uu.se.

DOI: http://dx.doi.org/10.1145/2560033

**ACM Reference Format:**
Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building timing predictable embedded systems. ACM Trans. Embedd. Comput. Syst. 13, 4, Article 82 (February 2014), 37 pages.
DOI: http://dx.doi.org/10.1145/2560033

## 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, for instance, on latencies or throughput. In several application domains, including automotive, avionics, or 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 timing 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, runtime 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 uncertainty 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, for instance, when a 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, that is, 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. An earlier tutorial paper by Thiele and Wilhelm [2004] examined the then state of the art regarding techniques for building predictable systems, with the purpose to propose design principles and outline directions for further work. 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* efficient systems, as well as the PRET project [Edwards and Lee 2007; Liu et al. 2012], which aims to equip instruction set architectures with control over timing.

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

The intention of this article is to survey some recent advances in research on building predictable yet performant systems. Thiele and Wilhelm [2004] listed performance-enhancing features of modern processor architectures, including processor pipelines and memory hierarchies, and suggested design principles for handling them when building predictable systems. In this article, we show how the understanding of predictability properties of these features has increased, and survey techniques that have emerged. Since 2004, multicore processors have become mainstream, and we survey techniques for using them in predictable system design. Thiele and Wilhelm [2004] also discussed the influence of the software structure on predictability, and suggested disciplined software design, for instance, based on some predictability-supporting computation paradigm, as well as the integration of development techniques and tools across several layers. In this article, we describe how compilation and timing analysis can be integrated with the goal to make the timing properties of a program visible directly to the developer at design-time, enabling control over the timing properties of a system under development. We also describe a language-based approach to predictable system design based on the synchronous programming paradigm. To keep our scope limited, we will not discuss particular analysis methods for deriving timing bounds (again, see Wilhelm et al. [2008]).

In a first section, we discuss basic concepts, including how "predictability" of an architectural mechanism could be defined precisely. The motivation is that a better understanding of "predictability" can preclude efforts to develop analyses for inherently unpredictable systems, or to redesign already predictable mechanisms or components. In the sections thereafter, we present techniques to increase predictability of architectural elements that have been introduced for efficiency.

In Section 3, we consider the predictability of various microarchitectural components. Important here is the design of processor pipelines and the memory system.

In Sections 4 and 5, we move up one level of abstraction, to the programming language, and consider two different approaches for putting timing under the control of a programmer. Section 4 contains a presentation of synchronous programming languages, PRET-C and Synchronous-C, in which constructs for concurrency have a deterministic semantics. We explain how they can be equipped with predictable timing semantics, and how this timing semantics can be supported by specialized processor implementations. In Section 5, we describe how a static timing analysis tool (aiT) can be integrated with a compiler for a widely-used language (C). The integration of these tools can equip program fragments with timing information (given a compilation strategy and target platform). It also serves as a basis for assessing different compilation strategies when predictability is a 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 activities can be isolated from each other, for instance, by partitioning communication and memory resources. We also discuss concerns for the sharing of processors between tasks in scheduling.

In Section 7, we discuss how to achieve predictability when considering randomly occurring errors that, for instance, may corrupt messages transmitted over a bus between different components of an embedded system. Without bounding assumptions on the occurrence of errors (which often cannot be given for actual systems), predictability guarantees can only be given in a probabilistic sense. We present mechanisms for achieving such guarantees, for instance, in order to comply with various standards for safety-critical systems. Finally, Section 8 presents conclusions and challenges for the future.

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 | priority-based |

## 2. FUNDAMENTAL PREDICTABILITY CONCEPTS

Predictable system design is made increasingly difficult by past and current developments in system and computer architecture design, where more powerful architectural elements are introduced for performance, but make timing guarantees harder to provide [Cullmann et al. 2010; Wilhelm et al. 2009]. Hence, research 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 effort to influence future system design in order to avert the worst problems for predictability in future designs. Both these lines of research are very important. However, we argue that they need to be based on a better and more precise understanding of the concept of "predictability." Without such a better understanding, 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, 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 of different architectural elements, for the case of analyzing timing predictability. For instance, simple in-order pipelines like the ARM7 are deemed more predictable than complex out-of-order pipelines as found in the PowerPC 755.

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, that is, 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. However, a prediction will usually refer to derived properties of such traces, for instance, their length or whether some interesting event(s) occurred. 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, for instance, the input of a program, and the prediction to be made should be valid for all possible cases, for instance, 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 these 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 a system-specific optimal analysis.[2] 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 Thesis 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. For instance, is the program input completely unknown or is partial information available?
—It is also desirable that predictability of a system is characterized in a compositional fashion. 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 [Grund et al. 2011] with the goal of enabling a concise and uniform description of predictability instances. It consists of the abovementioned key aspects (a) property to be predicted, (b) sources of uncertainty, and (c) quality measure.

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.
—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, that is, 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.
—In multicore systems (cf. Section 6), execution time is also influenced by contention on shared resources [Fernandez et al. 2012; Nowotsch and Paulitsch 2012; Radojković et al. 2012] induced by resource accesses of co-running threads. It is possible to consider the state and inputs of the corunning threads as part of the initial hardware

[2]Due to the undecidability of all nontrivial properties, no system-independent optimal analysis exists.

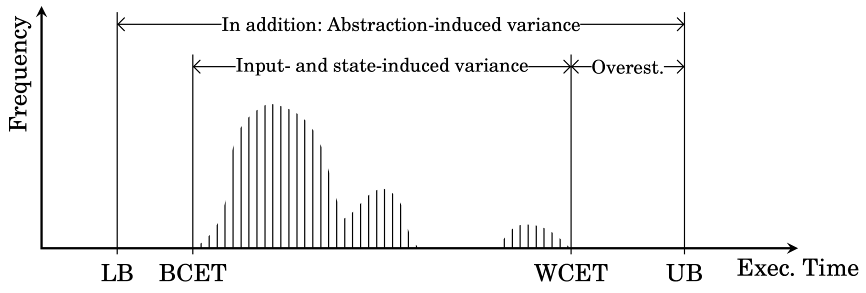


Fig. 1. Distribution of execution times ranging from best-case to worst-case execution time (BCET/WCET). Sound but incomplete analyses can derive lower and upper bounds (LB, UB).

state and program inputs, respectively. This is what we do in the following. It may, however, be interesting to separate the uncertainty induced by contention on shared resources in the future.
—Usually, schedulability analysis requires a characterization of execution times in the form of bounds on the execution time. Hence, a reasonable quality measure is the quotient of *Best-Case Execution Time* (BCET) over *Worst-Case Execution Time* (WCET); the closer to 1, the better.
—The inherence property is satisfied, as BCET and WCET are inherent to the system.

Let us introduce some basic definitions. 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* 2.2 (*Timing Predictability*). Given uncertainty about the initial hardware states $Q \subseteq \mathcal{Q}$ and uncertainty about the program inputs $I \subseteq \mathcal{I}$, the timing predictability of a program $p$ is

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

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: $\mathrm{Pr}_p \in [0, 1]$, where 1 means perfectly predictable.

Timing predictability as defined in Equation (1) is incomputable for most systems. So, it is not possible to construct a general procedure that, given a system, computes its predictability exactly. However, it is possible to develop procedures that compute approximations, that is, upper and/or lower bounds on a system's predictability. As in the study of the computational complexity of mathematical problems, the determination of the predictability of some systems will always require human participation.

*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.

A refinement of this definition is to distinguish hardware- and software-related causes of unpredictability by separately considering the sources of uncertainty.

*Definition* 2.3 (*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* 2.4 (*Input-Induced Timing Predictability*).

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

Clearly, by definition, $\mathrm{Pr}_p(Q, I) \leq \mathrm{IIPr}_p(Q, I)$ and $\mathrm{Pr}_p(Q, I) \leq \mathrm{SIPr}_p(Q, I)$ for all $Q$ and $I$. Somewhat less obviously, it can be shown that $\mathrm{IIPr}_p(Q, I) * \mathrm{SIPr}_p(Q, I) \leq \mathrm{Pr}_p(Q, I)$ for all $Q$ and $I$. Together, this implies that if either of $\mathrm{IIPr}_p$ or $\mathrm{SIPr}_p$ equals 1, then $\mathrm{Pr}_p$ equals the respective other one.

*Example* 2.5 (*Predictable Software*). Consider a program that executes the same sequence of instructions regardless of the program inputs. For such a program, one would possibly expect $\mathrm{IIPr}_p(Q, I)$ to be 1. However, this need not be true. One example where $\mathrm{IIPr}_p(Q, I) < 1$ is a system that features variable-latency instructions (e.g., division) and whose operands depend on the program input.

*Example* 2.6 (*Unpredictable Software*). Consider a program containing a loop whose iteration count is determined by an input value. For such a program, $\mathrm{IIPr}_p(Q, I)$ will be close to 0, given that different inputs, $i_1$ and $i_2$, that trigger vastly different iteration counts are contained in $I$.

*Example* 2.7 (*Predictable Hardware*). Consider a micro-architecture where execution times of instructions do not depend on the hardware state, for instance, PTARM [Liu et al. 2012]. For such a system, $\mathrm{SIPr}_p(Q, I) = 1$ holds.

*Example* 2.8 (*Unpredictable Hardware*). Consider a program that transmits a single message over Ethernet. Ethernet employs a binary exponential backoff mechanism to retransmit messages after collisions on the channel: After $n$ collisions, retransmission of data is delayed for a random number of slots taken from $[0, 2^n - 1)$. If one initial state, $q_1$, triggers a series of collisions, while another one, $q_2$, does not, and both are contained in $Q$, then $\mathrm{SIPr}_p(Q, I)$ will be low.

### 2.3. Related Work

At this point, we discuss related work that tries to capture the essence of predictability or aims at a formal definition.

The question about the meaning of predictability was already posed in Stankovic and Ramamritham [1990]. The main answers given in this editorial is that "it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made." Hence, it is rather seen as the existence of successful analysis methods than an inherent system property.

Bernardes Jr. [2001] considers a discrete dynamical system $(X, f)$, where $X$ is a metric space and $f$ describes the behavior of the system. Such a system is considered predictable at a point $a$, if a predicted behavior is sufficiently close to the actual behavior. The actual behavior at $a$ is the sequence $(f^i(a))_{i \in \mathbb{N}}$ and the predicted behavior is a sequence of points in $\delta$-environments, $(a_i)_{i\in\mathbb{N}}$, where $a_i \in B(f(a_{i-1}), \delta)$, and the sequence starts at $a_0 \in B(a, \delta)$.

Thiele and Wilhelm [2004] measure timing predictability as difference between the worst- (best-) case execution time and the upper (lower) bound as determined by an analysis. This emphasizes the qualities of particular analyses rather than inherent system properties.

Henzinger [2008] describes predictability as a form of determinism. Several forms of nondeterminism are discussed. Only one of them influences observable system behavior, and thereby qualifies as a source of uncertainty in our sense.

The work presented in this section was first introduced in a presentation[3] at a workshop during ESWEEK 2009. The main point, as opposed to almost all prior attempts, is that predictability should be an inherent system property. In Grund et al. [2011], we extend that discussion, introduce the herein repeated predictability template, and cast prior work in terms of that template.

## 3. MICROARCHITECTURE

In this and the following sections, we consider predictability of architectural elements at different levels in the system hierarchy. This section discusses microarchitectural features, focusing primarily on pipelines (Section 3.1), predictable multithreading mechanisms (Section 3.2), caches and scratchpads (Section 3.3), and dynamic RAM (Section 3.4).

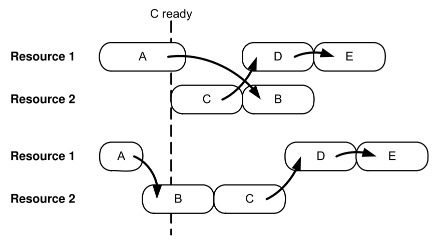
An *instruction set architecture* (ISA) defines the interface between hardware and software, that is, the format of software binaries and their semantics in terms of input/output behavior. A *microarchitecture* defines how an ISA is implemented on a processor. A single ISA may have many microarchitectural 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, that is, 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.

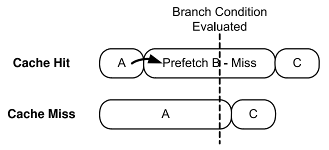
As a consequence of abstracting from execution time in ISA semantics, WCET analyses need to consider the microarchitecture a software binary will be executed on. The aforementioned microarchitectural techniques greatly complicate WCET analyses. For simple, nonpipelined microarchitectures without caches, 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[4] 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.

*Classification of Microarchitectures.* In previous work [Wilhelm et al. 2009], the following classification of microarchitectures into three categories has been provided. It classifies microarchitectures based on the presence of timing anomalies and domino effects, which will be discussed in the following text.

[3]See http://rw4.cs.uni-saarland.de/~grund/talks/repp09-preddef.pdf.
[4]In other words: The current state of the microarchitecture.



(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].

—*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.
—*Noncompositional architectures.* These architectures, for instance, 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.

*Timing Anomalies.* The notion of *timing anomalies* was introduced by Lundqvist and Stenström [1999]. In the context of WCET analysis, Reineke et al. [2006] present 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, cf. 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 shortening of 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.

*Domino Effects.* 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 path starting in $q_1$ respectively $q_2$ is proportional to the path's length, that is, there is no constant bounding the difference for all possible program paths. 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.

Let $p$ be a program that may execute arbitrarily long instruction sequences, depending on its inputs. Then, let $I_n$ denote the subset of program inputs $I$ that yield executions of instruction sequences of length exactly $n$. A system exhibits a domino effect if such a program exists and $\lim_{n\to\infty} \mathrm{SIPr}_p(Q, I_n) < 1$.

*Example of Domino Effects.* Schneider [2003] 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 pipeline state is empty, and in a shorter execution time if the initial state 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 instruction sequence considered in Schneider [2003], execution takes $9n + 1$ cycles when starting in one state, $q_1^*$, and $12n$ cycles when starting in the other state, $q_2^*$.

An application of Definition 2.3 is the quantitative characterization of domino effects. Let $p$ be a program that, depending on its inputs, executes the instruction sequence described above arbitrarily often. Then, let $I_n$ denote the inputs to $p$ that result in executing the instruction sequence exactly $n$ times. For this program $p$, the state-induced predictability can be bounded as follows:

$$\mathrm{SIPr}_p(Q, I_n) = \min_{q_1,q_2\in Q_n} \min_{i\in I_n} \frac{T_p(q_1, i)}{T_p(q_2, i)} \leq \frac{T_p\left(q_1^*, i^*\right)}{T_p\left(q_2^*, i^*\right)} = \frac{9n+1}{12n}, \tag{4}$$

with $\lim_{n\to\infty} \mathrm{SIPr}_p(Q, I_n) \leq \frac{3}{4} < 1$.

Another example for a domino effect is given by Berg [2006], who considers the PLRU replacement policy of caches. In Section 3.3, we describe results on the state-induced cache predictability of various replacement policies.

## 3.1. Pipelines

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

The analysis of a given program for its pipeline behavior is based on an abstract model of the pipeline. A transition in the model of the pipeline corresponds to the execution of a single machine cycle in the processor. 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.

*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 may result from data hazards, branch mispredictions, occupied functional units, full queues, etc.

Abstract states may lack information about the state of some processor components, for instance, caches, queues, or predictors. Transitions between states of the concrete pipeline may depend on such information. This causes the abstract pipeline model to become nondeterministic, although a more concrete model of the pipeline would be deterministic. When dealing with this nondeterminism, one could be tempted to design the WCET analysis such that only the “locally worst-case” transition is chosen, for instance, the transition corresponding to a pipeline stall or a cache miss. However, such an approach is unsound in the presence of timing anomalies [Lundqvist and Stenström 1999; Reineke et al. 2006]. Thus, in general, the analysis has to follow all possible successor states.

In particular if an abstract pipeline model may exhibit timing anomalies, the size of its state space strongly correlates with analysis time. Initial findings of a study into the tradeoffs between microarchitectural complexity and analysis efficiency are provided by Maksoud and Reineke [2012]. Surprisingly, reducing the sizes of buffers in the load-store unit may sometimes result in both improved performance as well as reduced analysis times.

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 for 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 analyze the WCET are prohibitive.”

### 3.2. Multithreading

With the advent of multicore and multithreaded 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 focused 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, that is, 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. [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. 2012], which is a precision-timed (PRET) machine [Edwards and Lee 2007] that implements 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 an instruction from a different thread in every cycle, removing data hazard stalls stemming from the pipeline resources. While this scheme achieves perfect utilization of the pipeline, it limits the performance of

Table II.
State-induced cache predictability, more precisely $\lim_{n\to\infty}$ SICPr$_p(n)$, for different replacement policies at associativities 2 to 8. PLRU is only defined for powers of two. For example, row 2, column 4 denotes $\lim_{n\to\infty}$ SICPr$_{\text{FIFO}(4)}(n)$.

| | 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 |
| RANDOM | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

each individual hardware thread. Unlike the virtual multiprocessor, the tasks on each thread need not be determined a priori, as hardware threads cannot affect each other's schedule. As opposed to Mische et al. [2008], all the hardware threads in the PTARM can be used for real-time purposes.

## 3.3. 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 $p$, SICPr$_p(n)$, 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 $n$, that is, $s \in B_n$, 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 $q$, SICPr$_p(n)$ is defined as follows:

*Definition* 3.1 (*State-Induced Cache Predictability*).

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

To investigate the influence of the initial cache states in the long run, we have studied $\lim_{n\to\infty}$ SICPr$_p(n)$. A tool called Relacs[5], described in Reineke and Grund [2012], is able to compute $\lim_{n\to\infty}$ SICPr$_p(n)$ automatically for a large class of replacement policies. Using Relacs, we have obtained sensitivity results for the widely used policies LRU, FIFO, and PLRU at associativities ranging from 2 to 8. For truly random replacement, the state-induced cache predictability is 0 for all associativities.

Table 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 by the number of misses incurred by the same sequence $s$ starting in another state $q_2$.

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

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 5.2 briefly discusses results on SPM allocation and the related topic of cache locking.

## 3.4. 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 resources 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. [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.

### 3.5. Conclusions and Challenges

Considerable efforts have been undertaken to construct safe and precise analyses of the execution time of programs on complex microarchitectures. What makes a microarchitecture "predictable" and even what that is supposed to mean is understood to a lesser extent. Classes of microarchitectures have been identified that admit efficient analyses, for instance, fully timing compositional architectures. Microarchitectures are currently classified into these classes based on the beliefs of experienced engineers. So far, only microarchitectures following very simple timing models such as the PTARM [Liu et al. 2012] can be classified with very high certainty. A precise, formal definition of timing compositionality and effective mechanisms to determine whether a given microarchitecture is timing compositional, though, are yet lacking.

The situation is a little less dire when it comes to individual microarchitectural components such as private caches or memory controllers. However, guidelines for the construction of timing-compositional yet truly high-performance microarchitectures, possibly from predictable components, are so far elusive.

## 4. SYNCHRONOUS PROGRAMMING LANGUAGES FOR PREDICTABLE SYSTEMS

Embedded systems typically perform a significant number of different activities that must be coordinated and that must satisfy strict timing constraints. A prerequisite for achieving predictability is to use a processor platform with a timing predictable ISA, as discussed in the previous section. However, the timing semantics should also be exposed to the programmer. Coarsely, there are two approaches to this challenge. One approach, described in Section 5, retains traditional techniques for constructing real-time systems, in which tasks are programmed individually (e.g., in C), and equips program fragments with timing information supplied by a static timing analysis tool. This relieves the programmer from the expensive procedure of assigning WCETs to program segments, but does not free him from designing suitable scheduling and coordination mechanisms to meet timing constraints, avoid critical races and deadlocks, etc. Another approach, described in this section, is based on *synchronous programming languages*, in which explicit constructs express the coordination of concurrent activities, communication between them, and the interaction with the environment. These languages are equipped with formal semantics that guarantee deterministic execution and the absence of critical races and deadlocks.

### 4.1. Context: The Synchronous Language Approach to Predictability

*The Essence of Synchronous Programming Languages.* In programming languages, 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 discrete reactions (called ticks), and computations within a tick are performed as if they were instantaneous and synchronous with each other [Benveniste et al. 2003]. Thus, a synchronous program behaves as if the processor executing it was infinitely fast. This abstraction is similar to the one made when designing synchronous circuits at the HDL level: At this abstraction level, a synchronous circuit reacts in a sequence of discrete reaction and its logical gates behave as if the electrons were flowing infinitely fast.

In contrast to asynchronous concurrency, synchronous languages avoid the introduction of nondeterminism by interleaving. On a sequential processor, with the asynchronous concurrency paradigm, two independent, atomic parallel tasks must be executed in some nondeterministically chosen sequential order. The drawback is that this interleaving intrinsically forbids deterministic semantics, which limits formal reasoning such as analysis and verification. On the other hand, in the semantics of synchronous languages, the execution of two independent, atomic parallel tasks is simultaneous. The concept of logical execution time, as exemplified in Giotto [Henzinger et al. 2003] or PTIDES [Zou et al. 2009], also provides a concurrent semantics that is independent from concrete execution times, but does not have the concept of a logical tick with instantaneous interthread communication within one tick. Another characteristic of synchronous languages is that they are finite state, for instance, they do not allow arbitrary looping or recursion, another prerequisite for predictability.

To take a concrete example, the Esterel [Berry 2000] statement "`every 60 second emit minute`" specifies that the signal `minute` is *exactly synchronous* with the 60th occurrence of the signal `second`. At a more fundamental level, the synchronous abstraction eliminates the nondeterminism resulting from the interleaving of concurrent behaviors. This allows deterministic semantics, thereby making synchronous programs amenable to formal analysis and verification, as well as certified code generation. This crucial advantage has made possible the successes of synchronous languages in the design of safety-critical systems; for instance, Scade (the industrial version of Lustre [Halbwachs et al. 1991]) is widely used both in the civil airplane industry [Brière et al. 1995] and in the railway industry [LeGoff 1996].

The recently proposed synchronous time-predictable programming languages that we present in this section take also advantage of this deterministic semantics.

*Validating the Synchronous Abstraction.* Of course, no processor is infinitely fast, but it does not need to be so, it just needs to be *faster than the environment*. Indeed, a synchronous program is embedded in a periodic execution loop of the form: "`loop {read inputs; react; write outputs} each tick`." Hence, when programming a reactive system using a synchronous language, the designer must check the validity of the synchronous abstraction. This is done by (a) computing the *Worst-Case Response Time* (WCRT) of the program, defined as the WCET of the body of the periodic execution loop; and (b) checking that this WCRT is less than the real-time constraint imposed by the system's requirement. The WCRT of the synchronous program is also known as its *tick length*.

To make the synchronous abstraction practical, synchronous languages impose restrictions on the control flow within a reaction. For instance, loops within a reaction are forbidden, that is, each loop must have a tick barrier inside its body (e.g., a `pause` statement in Esterel or an `EOT` statement in PRET-C). 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 that a given program is *causal*, meaning that different evaluation orders cannot lead to different results (see Berry [2000] for a more detailed explanation), and for compiling the program into deterministic sequential code executable in bounded time and bounded memory.

Finally, 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.

*Requirements for Timing Predictability.* Maximizing timing predictability, as defined in Definition 2.2, requires more than just the synchronous abstraction. For instance, it is not sufficient to *bound* the number of iterations of a loop; it is also necessary to know *exactly* this number to compute the exact execution time. Another requirement is that, in order to be adopted by industry, synchronous 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 (Section 4.2).

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

## 4.2. Language Constructs to Express Synchrony and Timing

We now illustrate how synchronous programming and timing predictability interact in concrete languages. As space does not permit a full introduction to synchronous programming, we will restrict our treatment to a few representative concepts. Readers unfamiliar with synchronous programming are referred to the excellent introductions given by Benveniste et al. [2003] and Berry [2000]. We here consider languages that incorporate synchronous concepts into the C language, to illustrate how synchronous concepts can be incorporated into a widely used sequential programming language. However, one must then avoid programming constructs that break analyzability again, such as unbounded loops or recursion. Our overview is based on a simple *producer/consumer/observer example* (PCO). This program starts three threads that then run forever (i.e., until they are terminated externally) and share an integer variable `buf` (cf. Figure 3). This is a typical pattern for reactive real-time systems.

*The Berkeley-Columbia PRET Language.* The original version of PCO (cf. Figure 3(a)) was introduced to illustrate the programming of the Berkeley-Columbia PRET architecture [Lickly et al. 2008]. The programming language is a multithreaded version of C, extended by a special deadline instruction, called `DEAD(`$t$`)`, which behaves as follows: The first `DEAD(`$t$`)` instruction executed by a thread terminates as soon as at least $t$ instruction cycles have passed since the start of the thread; subsequent `DEAD(`$t$`)` instructions terminate as soon as at least $t$ instruction cycles have passed since the previous `DEAD(`$t$`)` instruction has terminated.[6] Hence, a `DEAD` instruction can only enforce a lower bound on the execution time of code segment. However, by assigning values to the `DEAD` instructions that are conservative with respect to the WCET, it is therefore possible to design predictable multithreaded systems, where problems such as race conditions will be avoided thanks to the interleaving resulting from the `DEAD` instructions. Assigning the values of the `DEAD` instructions requires to know the exact number of cycles taken by each instruction. Fortunately, the Berkeley-Columbia PRET architecture [Lickly et al. 2008] guarantees that.

In Figure 3(a), the first `DEAD` instructions of each thread enforce that the Producer thread runs ahead of the Consumer and Observer threads. The subsequent `DEAD` instructions enforce that the threads iterate through the for-loops in lockstep, one iteration every 26 instruction cycles. This approach to synchronization exploits the predictable timing of the PRET architecture and alleviates the need for explicit scheduling or synchronization facilities of the language or the *operating system* (OS). However, this comes at the price of a brittle, low-level, nonportable scheduling style.

[6]The `DEAD()` operator is actually a slight abstraction from the underlying processor instruction, which also specifies a timing register. This register is decremented every six clock cycles, corresponding to the six-stage pipeline of the PRET [Lickly et al. 2008].

```
int producer() {
  DEAD(28);
  volatile unsigned int* buf =
      (unsigned int*)
      (0x3F800200);
  unsigend int i = 0;
  for (i = 0; ; i++) {
      DEAD(26);
      *buf = i;
  }
  return 0;
}
```

```
int consumer() {
  DEAD(41);
  volatile unsigned int* buf =
      (unsigned int*)
      (0x3F800200);
  unsigend int i = 0;
  int arr[8];
  for (i = 0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

```
int observer() {
  DEAD(41);
  volatile unsigned int* buf =
      (unsigned int*)
      (0x3F800200);
  volatile unsigned int* fd =
      (unsigned int*)
      (0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

(a) Berkeley-Columbia PRET version of PCO according to Lickly et al. [2008]. Threads are scheduled via the `DEAD()` instruction which also specifies physical timing.

```
#include "sc.h"

int main()
{
 int notDone,
   init = 1;

 RESET();
 do {
  notDone = tick();
  sleep(1);
   init = 0;
 } while (notDone);
 return 0;
}
```

```
int tick ()
{
 static int buf, fd, i,
    j, k=0, tmp, arr [8];

 MainThread (1) {
  State (PCO) {
   FORK3(
     Producer, 4,
     Consumer, 3,
     Observer, 2);

   while (1) {
    if (k == 20)
     TRANS(Done);
    if (buf == 10)
     TRANS(PCO);
    PAUSE; }
  }

  State (Done) {
   TERM; }
 }
```

```
Thread (Producer) {
 for (i=0; ; i++) {
  buf = i;
  PAUSE; }
}

Thread (Consumer) {
 for (j=0; j < 8; j++)
  arr[j] = 0;
 for (j=0; ; j++) {
  tmp = buf;
  arr[j % 8] = tmp;
  PAUSE; }
}

Thread (Observer) {
 for ( ; ; ) {
  fd = buf;
  k++;
  PAUSE; }
}

TICKEND;
}
```

(b) SC version of PCO. Scheduling requirements are specified with explicit thread priorities (1 - 4).

Fig. 3. Two variants of the Producer Consumer Observer example, extended by preemptions.

As it turns out, this lockstep operation of concurrent threads directly corresponds to the logical tick concept used in synchronous programming. Hence it is fairly straightforward to program the PCO in a synchronous language, without the need for low-level, explicit synchronization, as illustrated in the following.

*Synchronous C and PRET-C.* Synchronous C (originally introduced as SyncCharts in C [von Hanxleden 2009]) and PRET-C [Andalam et al. 2010] are both lightweight, concurrent programming languages based on C. A Synchronous C (SC) program consists of a `main()` function, some regular C functions, and one or more parallel threads. Threads communicate via shared variables, and the synchronous semantics guarantees both a deterministic execution and the absence of race conditions. The thread management is done fully at the application level, implemented with plain C `goto` or `switch` statements and C labels/cases hidden in the SC macros defined in the `sc.h` file. PRET-C programs are analogous.

Figure 3(b) shows the SC variant of an extended PCO example. The extended PCO variant includes additional behavior that restarts the threads when `buf` has reached the value 10, and that terminates the threads when the loop index `k` has reached the value 20. A loop in `main()` repeatedly calls a `tick()` function which implements the reactive behavior of one logical tick. This behavior consists of a `MainThread`, running at priority 1, which contains the states `PCO` and `Done`. The state `PCO` forks the three other threads specified in `tick()`. The reactive control flow is managed with the SC operators `FORK`*n* (which forks *n* threads with specific priorities), `TRANS` (which aborts its child threads, transfer control), `TERM` (which terminates its thread), and `PAUSE` (which pauses its thread until the next tick). Moreover, the execution states of the threads are stored statically in global variables declared in `sc.h`. This behavior is similar to the `tick()` function synthesized by an Esterel compiler. Finally, the return value of the `tick()` function is computed and returned by the `TICKEND` macro.

Hence, an SC program is a plain, sequential C program, fully deterministic, without any race conditions or OS dependencies. The same is true for PRET-C programs.

Compared again to the original PCO example in Figure 3(a), the SC variant illustrates additional preemption functionality. Also, physical timing and functionality are separated, using `PAUSE` instructions that refer to logical ticks rather than `DEAD` instructions that refer to instruction cycles. However, with both SC and PRET-C, it is the programmer who specifies the execution order of the threads within a tick. This order is the priority order specified in the `FORK3` instruction: The priority of the `Producer` thread is 4, and so on.

Unlike SC, PRET-C specifies that loops must either contain an `EOT` (the equivalent to a `PAUSE`), or must specify a maximal number of iterations (e.g., "`while (1) #n {...}`", where `n` is the maximal number of iterations of the loop); this ensures the timing predictability of programs with loops. Conversely, SC offers a wider range of reactive control and coordination possibilities than PRET-C, such as dynamic priority changes.

## 4.3. Instruction Set Architectures 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 ISA that directly implements synchronous reactive control flow constructs. With respect to predictability, the main advantage of reactive processors is that they offer direct ISA support for crucial features of the languages (e.g., preemption, synchronization, interthread communication), therefore allowing a very fine control over the number of machine cycles 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 [Edwards and Lee 2007; Lickly et al. 2008].

In summary, ISAs for synchronous programming are the dual to synchronous language constructs, in that the former provide predictability at the execution platform level, whereas the latter provide predictability at the language level.

The first reactive processor, called REFLIX, was designed by Salcic et al. [2002], followed by a number of follow-up designs [Yuan et al. 2009]. This concept of reactive processors was then adapted to PRET-C with the ARPRET platform (Auckland Reactive PRET). It is built around a customized Microblaze softcore processor (MB), connected via two fast simplex links to a so-called Functional Predictable Unit that 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.

Similarly, the *Kiel Esterel Processor* (KEP) includes a Tick Manager that minimizes reaction time jitter and can detect timing overruns [Li and von Hanxleden 2012]. The

ISA of reactive processors has strongly inspired the language elements introduced by both PRET-C and SC.

### 4.4. WCRT Analysis for Synchronous Programs

Compared to typical WCET analysis, the WCRT analysis problem here is more challenging because it includes concurrency and preemption; in classical WCET computation, concurrency and preemption analysis is often delegated to the OS. However, the synchronous deterministic semantics on one hand, and the coding rules on the other hand (e.g., absence of loops without a tick barrier), make it feasible to reach tight estimates.

Concerning SC, a compiler including a WCRT analysis was developed for KEP to compute safe estimates for the Tick Manager [Boldt et al. 2008]), further improved with a modular, algebraic approach that also takes signal valuations into account to exclude infeasible paths.

Similarly, a WCRT analyzer was developed for PRET-C programs running on ARPRET, where the *Control Flow Graph* (CFG) is decorated with the number of machine cycles required to execute it on ARPRET, and then is analyzed with UPPAAL to compute the WCRT. Combining the abstracted state space of the program with expressive data flow information allows infeasible execution paths to be discarded [Andalam et al. 2011].

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

### 4.5. Conclusions and Challenges

The synchronous semantics of PRET-C and SC directly provides several features that are essential for the design of complex predictable systems, including determinism, thread-safe communication, causality, absence of race conditions, and so on. These features relieve the designer from concerns that are problematic in languages with asynchronous timing and asynchronous concurrency. Numerous examples of reactive systems have been reimplemented with PRET-C or SC, showing that these languages are easy to use [Andalam et al. 2010].

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 firmly established 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 typically uses high-level languages like C, often using tools like, for instance, Matlab/Simulink, which automatically generate C code. Compilers for C include a vast variety of optimizations. However, they mostly aim at reducing *Average-Case Execution Times* (ACETs) and have no timing model. In fact, 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 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. Fundamentals of WCET-aware Compilation

In order to obtain a compiler performing code generation and optimization for timing predictable systems, it is not enough to simply develop novel aggressive optimizations. Instead, such novel WCET-aware optimizations rely on massive support by an infrastructure providing formal timing, control flow and hardware models. The following subsections describe key components of such a WCET-aware compiler infrastructure.

*Integration of Static WCET Analysis into the Compiler.* For a systematic consideration of worst-case execution times by a compiler, it is mandatory to provide a formal and safe WCET timing model. The easiest way to achieve this goal is to integrate static WCET analysis tools into the compiler.

A very first approach was proposed by Zhao et al. [2005a] where a proprietarily developed WCET analyzer was integrated 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 preventing the optimization of smaller units like basic blocks. Furthermore, important data beyond the WCET itself is unavailable, for instance, execution frequencies of basic blocks, value ranges of registers, predicted cache behavior, etc. Finally, WCET optimization at higher levels of abstraction like, for instance, source code level is infeasible since timing-related data is not provided at source code level.

These issues were cured within the WCET-aware C Compiler [WCC 2014] where 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. aiT is integrated into WCC by translating the compiler's assembly code IR to CRL2 and vice versa. This way, the compiler 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.

*Specification of Memory Hierarchies.* The performance of many systems is dominated by the memory subsystem. Obviously, timing estimates also heavily depend on the memories so that a WCET-aware compiler must provide the timing analyzer with detailed information about the underlying memory hierarchy. Thus, such a compiler must be aware of a processor's memories which is usually delegated to the linker in a classical compilation flow. Furthermore, the compiler exploits this memory hierarchy infrastructure to apply memory-aware optimization by assigning parts of a program to fast memories.

As an example, WCC allows to simply specify memory hierarchies. For each physical memory, attributes like, for instance, base address, length, access latency, etc. can be defined. Cache parameters like, for instance, size, line size or associativity can be specified. Memory allocation of program parts is now done by the compiler instead of the linker by allocating functions, basic blocks or data to these memory regions. Moreover, physical memory addresses provided by compiler's memory hierarchy infrastructure are exploited during WCET analysis such that 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 for a highly accurate WCET analysis.

*Flow Fact Specification and Transformation.* A program's execution time (on a given hardware) largely depends on its control flow, for instance, on loops or conditionals. Since loop iteration counts are crucial for precise WCETs, and since they cannot be computed in general, they must be specified by the user of a timing analyzer. These user-provided annotations are called *flow facts*. In an environment where the timing analyzer is tightly integrated into the compilation flow, it is critical that the compiler provides highly accurate flow facts to the WCET analyzer.

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 pragmas within the source code, but are not updated during optimization. This turns the entire approach tedious and error-prone, since compiler optimizations potentially restructure the code and invalidate originally specified flow facts.

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 cannot 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 work fully supports source-level flow facts by means of ANSI-C pragmas and was thoroughly tested and led to precise WCET estimates.

Inspired by Kirner and Puschner [2001], WCC's flow facts are specified similarly in ANSI-C [Falk and Lokuciejewski 2010]. *Loop bound* flow facts limit the iteration counts of regular loops. In contrast to previous work, they allow to specify minimum and maximum iteration counts allowing to annotate data-dependent loops. For irregular loops or recursions, *flow restrictions* can be used to relate the execution frequency of one C statement with that of others. Furthermore, WCC's optimizations are fully flow-fact aware. All operations of the compiler's IRs creating, deleting or moving statements or basic blocks now inherently update flow facts. Thus, always safe and precise flow facts are maintained, irrespective of how and when optimizations modify the IRs.

### 5.2. Examples of WCET-aware Optimizations

On top of a compiler infrastructure sketched above, a large number of novel WCET-aware optimizations has been proposed recently. 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 3.3, scratchpad memories 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 [Liu et al. 2009; Wan et al. 2012].

A first approach for WCET-aware SPM allocation was proposed by Suhendra et al. [2005]. In an integer linear program, inequations model the structure of a function's control flow graph. Constants model the worst-case timing per basic block when being allocated to slow flash memory or to the fast SPM. This way, the ILP is always aware of that path in a function's CFG having the longest execution time. Unfortunately, the ILP of Suhendra et al. [2005] is unable to allocate code onto an SPM and suffers from several limitations preventing it from being applied to real-life code.

Falk and Kleinsorge [2009] resolved these drawbacks by adding support for SPM allocation of code, jump penalties, and global control flow to the ILP. As a consequence, this ILP now is aware of that path of a whole program leading to the longest execution time and can thus optimally minimize a program's WCET. A similar optimization approach can be used to also support cache locking.

Experimental results over a total of 73 different benchmarks from, for instance, 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, for instance, to avoid jump-related pipeline delays [Zhao et al. 2005b] or at granularity of entire functions or even 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, that is, potentially evict each other 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 Multitask Systems.* The cache-related optimizations presented so far cannot handle multitask systems with preemptive scheduling, since it is difficult to predict the cache behavior during context switches. Cache partitioning is a technique for multitask 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 with respect to the cache during context switches. This allows to apply static timing analysis to each individual task in isolation. The overall WCET of a multitask 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 by Chiou et al. [1999] and Molnos et al. [2004], resp. A very recent work on WCET-aware cache partitioning by Liu et al. [2011] proposes heuristics to assign tasks to cores and to partition a shared L2 cache, but relies on hardware support for cache locking. Mueller [1995] presents ideas for purely software-based cache partitioning. Here, 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. However, an implementation or evaluation of these ideas is not given.

The cache partitioning of WCC by Plazar et al. [2009] picks up these ideas and uses an 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, this 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.

### 5.3. Conclusions and Challenges

This section discussed compiler techniques and concepts for timing predictable systems by exploiting a WCET timing model. Until recently, not much was known about the WCET savings achievable in this way. This section provided a survey over work exploring the potential of such integrated compilation and timing analysis. All the presented optimizations improve the state-induced timing predictability (cf. Definition 2.3) since they heavily minimize the uncertainty about hardware states of caches (cache locking and partitioning, code positioning) and flash memories (SPM allocation).

While the works briefly presented in this section are able to reduce the WCET of single programs, most of them fail if multitask or multicore systems come into play. In such systems, shared resources like, for instance, pipelines, caches, memories or buses lead to the situation that the timing of one task can vary, depending on activities of other tasks running potentially on other cores. These interferences between tasks are not yet thoroughly handled during code generation and optimization, only very first works deal with timing analysis and code optimization for such systems with shared resources. As a consequence, compilation for timing predictable systems has to address the challenges imposed by multitask and multicore systems in the near future.

## 6. BUILDING REAL-TIME APPLICATIONS ON MULTICORES

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.

## 6.1. Timing Interferences and Isolation

The timing behavior of a task running on a multicore architecture depends heavily on the arbitration mechanisms of the shared resources and other tasks' usage of the resources. 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. Bypassing the L2 cache for single-usage cache lines is a way to reduce the number of conflicts and improve the accuracy [Hardy et al. 2009]. The analysis can also 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. [2009] propose an iterative approach to estimate the WCET bounds of tasks sharing L2 caches. To further improve the analysis precision, the timing behavior 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 [Andersson et al. 2010; Schliecker et al. 2010]. Decoupling the estimation of memory latencies from the analysis of the pipeline behavior is a way to enhance analyzability. 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.

To make the latencies to shared bandwidth resources predictable (boundable), hardware solutions rely on bandwidth partitioning techniques, for instance, round-robin arbitration [Paolieri et al. 2009a]. To limit the overestimation of worst-case latencies, long-latency transactions, for instance, atomic synchronization operations, may be executed in split-phase mode [Gerdes et al. 2012].

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 timestamps 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. An extended instruction set architecture with temporal semantics combined with low-level mechanisms that enforce temporal isolation enhances timing composability and predictability [Bui et al. 2011].

Cache partitioning schemes allocate private partitions to tasks. Paolieri et al. [2011] consider software-controlled hardware mechanisms: Columnization (a partition is a set of cache ways) and bankization (a partition is a set of cache banks). The configuration of partitions is set by software. Their interference-aware allocation algorithm determines a configuration that makes a given task set schedulable while minimizing the cache usage. Page-coloring [Guan et al. 2009a] is a software-controlled scheme that 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.

## 6.2. System-Level Scheduling and Analysis

The system predictability heavily depends on how the workload is scheduled at the system level. For single-processor platforms, well-established techniques (e.g., rate-monotonic scheduling) for system-level scheduling and schedulability analysis can be found in both textbooks [Liu 2000; Buttazzo 2011] and industry standards, such as POSIX. However, the *multiprocessor scheduling* problem to map tasks onto parallel architectures is a much harder challenge and lacks well-established techniques, which brings unique challenges to building timing predictable embedded systems on multicore processors.

*Global Scheduling.* One may allow all tasks to compete for execution on all cores. Global scheduling is a realistic option for multicore systems, on which the task migration overhead is much less significant compared with traditional loosely-coupled multiprocessor systems thanks to hardware mechanisms like on-chip shared caches. Global multiprocessor scheduling is a much more difficult problem than uniprocessor scheduling, as first pointed out by Liu and Layland [1973]: “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.”

The major obstacle in precisely analyzing global scheduling is the lack of a known *critical instant*. 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 effective approximate techniques. Much work has been done on tightening the workload estimation by excluding impossible system behavior from the calculation (e.g., [Baker 2003; Baruah 2007]). Guan et al. [2009b] established the concept of the 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.

*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. Similar to the bin-packing problem, partitioned scheduling suffers from resource waste due to fragmentation. Such a waste will be more significant, as multicores evolve in the direction to integrate a larger number of less powerful cores and the workload of each task becomes relatively heavier compared to the processing capacity of each individual core. Theoretically, the worst-case utilization bound of partitioned scheduling cannot 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 manner 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 that of 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., [Kato and Yamasaki 2008; Lakshmanan et al. 2009]). Recently, a notable result is obtained in Guan et al. [2010], which generalizes the famous Liu and Layland's utilization bound $N * (2^{\frac{1}{N}} - 1)$ [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.

*Implementation and Evaluation.* To evaluate the performance and applicability of different scheduling paradigms in *Real-Time Operating Systems* (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 runtime overheads include mainly the scheduler latency (typically several tens $\mu$s in Linux) and cache-related costs, which depends on the application work space characterization, and can vary between several $\mu$s and tens of $m$s [Bastoni et al. 2010a]. 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]. Recently, evaluations have also been done with semi-partitioned scheduling algorithms [Bastoni et al. 2011] indicating that semi-partitioned scheduling is indeed a promising scheduling paradigm for multicore real-time systems.

## 6.3. Conclusions and Challenges

On multicore platforms, to predict the timing behavior of an individual task, one must consider the global behavior 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 resources with performance decreases. For storage resources, 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. RELIABILITY ISSUES IN PREDICTABLE SYSTEMS

In the previous sections, predictability was always achieved under the assumption that the hardware works without errors. Behavior under errors has been considered as an exception requiring specific error handling mechanisms that require redundancy in space and/or time. At the level of integrated circuits, this is still common practice while at the level of distributed systems, handling of errors, for instance, due to noise, is usually part of the regular system behavior, such as the extra time needed for retransmission of a distorted message. This approach was justified by the enormous physical reliability of digital semiconductor system operation. Only at very high levels of safety requirements, redundancy to increase reliability was needed, which was typically provided by redundancy in space, masking errors without changing the system timing. However, the ongoing trend of semiconductor downscaling leads to an increased sensitivity towards radiation, electromagnetic interference or transistor variation. As a result, the rate of transient errors is expected to increase with every technology generation [Borkar 2005]. Transient errors are caused by physical effects that are described by statistical fault models. These statistical fault models have an infinite range, such that there is always a nonzero probability of an arbitrary number of errors. This is in fundamental conflict with the usual perception of predictability which aims at bounding system behavior without any uncertainty.

To predict system behavior under these circumstances, quality standards define probability thresholds for correct system behavior. Safety standards (predictable systems are often required in the context of safety requirements) are most rigorous, defining maximum allowed failure probabilities for different safety classes, such as the *Safety Integrity Level* (SIL) classification of the IEC 61508 [2010]. Using redundancy in space, these reliability requirements can directly be mapped to extra hardware resources and mechanisms that mask errors with sufficiently high probability. However, redundancy in space is expensive in terms of chip cost and power consumption, such that redundancy in time, typically in the form of error detection and repetition in case of error, is preferred in system design. Both approaches are not mutually exclusive, Izosimov et al. [2006] presented a design synthesis methodology to construct a robust schedule by applying a combination of redundancy in time and space which sustains a given number of errors at any time.

Unfortunately, error correction by repetition increases execution time which invalidates the predicted worst-case execution time. A straightforward idea would be to just increase the predicted WCET by the time to correct an error. Given the unbounded statistic error models, however, the time for repetitions cannot be bounded with a guaranteed WCET. This dilemma can, however, be solved in the same way as in the case of redundancy in space, that is, by introducing a probabilistic threshold. This way, predictability can be reestablished in a form that is appropriate to design and verify safety- and timing-critical systems, even in the presence of hardware errors.

## 7.1. An Example: The Controller Area Network

To explain the approach, we will start with an example from distributed systems design. The most important automotive bus standard is the *Controller Area Network* (CAN) [CAN 1991]. CAN connects distributed systems, consisting of an arbitrary number of electronic control units in a car. Being used in a noisy electrical environment, CAN messages might be corrupted by errors, with average error rates strongly depending on the current environment [Ferreira et al. 2004].

The CAN protocol applies state-of-the-art *Cyclic Redundancy Checks* (CRC) to detect the occurrence of transmission errors. Subsequently, a fully automated error signaling mechanism is used to notify the sender about the error such that the original message can be retransmitted. This kind of error handling mechanism affects predictability in different ways. For the case that the message is transmitted correctly, transmission latency can be bounded using well-known response time calculation methods [Tindell and Burns 1994; Tindell et al. 1995; Davis et al. 2007]. If errors occur, two different cases must be distinguished.

(1) The error is detected and a retransmission is initiated. The latency of the corrupted message increases due to the necessity of a retransmission. Nonaffected messages might also be delayed due to scheduling effects. In this case, the error affects the overall timing on the CAN bus.
(2) The error is not detected and the message is considered as being received correctly. This might happen in rare cases as the error detection of CAN does not provide full error coverage. In this case, the error directly affects the logical correctness of the system.

In both cases, the random occurrence of errors might cause a system failure, either a timing failure due to a missed timing constraint or a logical failure due to invalid data which are considered to be correct. To predict the probability that the CAN bus transmits data without logical or timing failures, a statistical error model must be given that specifies probability distributions of errors and correlations between them. This model must be included in timing prediction for critical systems as well as in error coverage analysis. This way, it is possible to compute the probability of failure-free operation, normally measured as a time-dependent function $R$ with $R(t) = P(\textit{no failure in } [0;t])$. This basic thinking is also reflected by current safety standards and can therefore be adapted for new directions in predictability-driven development. Safety standards prescribe the consideration of different types of errors which might threat the system's safety and recommend different countermeasures. They also define probabilistic measures for the maximum failure rate, depending on severity on the affected functions. Examples are the SIL target failure rate in IEC 61508 or the maximum incident rate in ISO 26262 [2011].

The issue of logical failures for CRC-protected data transmission, usually referred to as residual error probability, has initially been addressed by a couple of research work during the 1980's, where theory of linear block codes has been applied to derive the residual error probability for CRCs of different length [Wolf et al. 1982; Wolf and Blakeney 1988]. In Charzinski [1994], similar research has been carried out explicitly for the CAN bus. It has been shown that the residual error probability on CAN is less than $10^{-16}$ even for a high bit error rate of $10^{-5}$.

Initial work on timing effects of errors on CAN has been presented by Tindell and Burns [1994]. There, the traditional timing analysis has been extended by an error term, and error thresholds have been derived. Even though this approach presented a first step towards the inclusion of transmission errors into traditional CAN bus timing analysis, the issue of errors as random events has been neglected. Subsequently, numerous extensions of this general approach have been presented, assuming probabilistic error models to derive statistical measures for CAN real-time capabilities. In Broster et al. [2002b], exact distribution functions for worst-case response times of messages on a CAN bus have been calculated. A more general error model that allows the consideration of a simple burst error model has been proposed in Navet et al. [2000]. Weakly hard real-time constraints for CAN, that is, constraints that are allowed to be missed from time to time, have been considered under the aspect of errors in Broster et al. [2002a]. This approach is not restricted to the worst case anymore. However, it does not take a stochastic error model into account but relies on a given minimum interarrival time between errors that is assumed not to be underrun. A more general model that overcomes the worst-case assumption and considers probabilistic error models has been presented by Sebastian and Ernst [2009]. Based on the simplified assumption of bit errors occurring independently from each other, a calculation method for the overall CAN bus reliability and related measures such as *Mean-Time-to-Failure* (MTTF) has been introduced. The approach focuses on timing failures, but would be combinable with the occurrence probability for logical failures as well. In addition, it has been shown that reliability analyses for messages with different criticality can be decoupled, and each criticality level can be verified according to its own safety requirements. In Sebastian and Ernst [2009], this technique has been applied to an exemplary CAN bus setup with an overall failure rate of only a couple of hours, which is normally not acceptable for any safety-related function. Anyway, by decoupling analyses from each other, highly critical messages could be verified up to SIL 3, while only a subset of all messages (e.g., those ones related to best-effort applications) missed any safety constraint given by IEC 61508. The simplifying assumption of independent bit errors has been relaxed in Sebastian et al. [2011], where hidden Markov models have been utilized for modeling and analysis purposes to include arbitrary bit error correlations. The authors point out the importance of appropriate error models by showing that the independence assumption is neither optimistic nor pessimistic and can therefore hardly be applied for a formal verification in the context of safety-critical design.

### 7.2. Predictability for Fault-Tolerant Architectures

The CAN bus is just an example of a fault-tolerant architecture that provides predictability in form of probabilistic thresholds even in the presence of random errors. In general, fault tolerance must be handled with care concerning timing impacts and predictability because of two main reasons. First, fault tolerance adds extra information or calculations, causing a certain temporal overhead even during error-free operation. This overhead can normally be statically bounded, so that it does not affect predictability, but might delay calculations or data transfers, that is, it might affect feasibility of schedules. The second issue is temporal overhead that occurs randomly because of measures to be performed explicitly in case of (random) hardware errors. As explained above, predictability based on worst-case assumptions is not given in this case anymore but has to be replaced by the previously introduced concept of probability thresholds. There is a wide variety of fault tolerance mechanisms protecting networks, CPU or memories, differing in efficiency, complexity, costs and effects on timing and predictability. Following above discussion on redundancy concepts, these mechanisms can basically be categorized in two classes.

The first class aims at realizing error masking without random overhead using hardware redundancy. A well-established representative of this class is *Triple Modular Redundancy* (TMR) [Kuehn 1969]. Three identical hardware units are executing the same software in lockstep mode, such that a single component error can be corrected using a voter. The only effect on timing is given by the voting delay which is normally constant and does not change its latency in case of errors. Thus, timing of a TMR architecture is fully predictable. However, as mentioned earlier in this section, it has several disadvantages, mainly the immense resource and power waste due to oversizing the system by a factor of 3. Another issue is that the voter is a single point of failure [Wakerly 1976], thus the reliability of the voter must be at least one order of magnitude above the reliability of the devices to vote on.

Another solution that realizes error correction without impact on predictability is the appliance of *Forward Error Correction* (FEC) using *Error Correcting Codes* (ECC) [Van Lint 1999]. It is mainly applicable to memory and communication systems to protect data against distortion, but it can also be used to harden registers in hardware state machines [Rokas et al. 2003]. FEC exploits the concepts of information redundancy. It encodes individual blocks of data by inserting additional bits according to the applied ECC such that decoding is possible even if errors occurred. While FEC is normally less hardware- and power-demanding compared to TMR, it often provides only limited error coverage and is therefore more susceptible to logical failures. Using a Hamming code with a Hamming distance of 3 for example, only one bit error per block is recoverable. It is therefore mainly applied for memory hardening and bus communication where the assumption of single bit errors is reasonable. For this purpose, memory scrubbing can additionally be used to correct errors periodically before they accumulate over time [Saleh et al. 1990]. Communication systems which might suffer from burst errors must use more powerful ECCs such as Reed-Solomon-Codes [Wicker and Bhargava 1999], which in turn significantly increase encoding and decoding complexity as well as the static transmission overhead.

The second class of fault tolerance mechanisms mainly focuses on error detection with subsequent recovery. In contrast to error correcting techniques, no or only little additional hardware is necessary. Instead, the concept of time redundancy is exploited by initiating recovery measures after an error has been detected. The resulting temporal overhead occurs randomly according to the component's error model, that is, only probabilistic thresholds for the timing behavior can be given anymore. One example is the previously mentioned retransmission mechanism of CAN. Whenever an error is detected using CRC, an error frame is sent and the original sender can schedule the distorted message for retransmission. In this case, the temporal overhead is quite large: Apart from the error frame and the retransmission, additional queuing delays might arise due to higher priority traffic on the CAN bus. Analysis approaches have to take all these issues into account and combine them with the corresponding error model. This leads to probabilistic predictions of real-time capabilities. FlexRay is another popular transmission protocol that uses CRC. In contrast to CAN, the FlexRay standard only prescribes the use of CRC for error detection but leaves it to the designer how to react on errors [Paret and Riesco 2007].

Similar approaches exist for CPUs. Rather than masking errors with TMR, *Double Modular Redundancy* (DMR) is used to only detect errors. It is implemented using two identical hardware units running in lockstep mode. A comparator connected to the output of both units continuously compares their results. In case of any inconsistencies, an error is indicated such that the components can initiate (usually time-consuming) recovery. DMR is a pure hardware solution that protects the overall processing unit with nearly zero error detection latency (because results are compared continuously and errors can be signaled immediately). However, it is quite expensive due to large hardware overhead such that a couple of alternative solutions has been proposed. The N-version programming approach [Avizienis 1985] executes multiple independent implementations of the same function in parallel and compares their results after each version has been terminated. This approach covers random hardware errors as well as systematic design errors (software bugs). It poses new challenges on result comparison, for example when results consist of floating point values. In this case, results might be unequal not because of errors but due to the inherent loss of precision in floating point arithmetic which depends on the order of operations. A solution would be to use inexact voting mechanisms [Parhami 1994], which in turn raise new issues concerning their applicability for systems with high reliability requirements [Lala and Harper 1994]. A simplified variant of N-version programming is to execute the same implementation of a function multiple times [Pullum 2001]. This can be realized in a time-multiplexed mode on the same CPU (reexecution) or by exploiting space redundancy (replication). In contrast to DMR, these techniques can be adopted in a more fine-grained way by protecting only selected tasks, potentially leading to substantial cost savings, because spare hardware can now be utilized by best-effort applications. While DMR has nearly no error detection latency, N-version programming, reexecution and replication require the designer to annotate the code at points where data is to be compared. This can be a tedious task and is not very flexible. In most cases, a designer will probably decide that only the final result of the task is subject to voting, thus the error detection latency can be high. Additionally, dormant errors may stay in the state for arbitrary long time, since only a subset of the application state is compared.

A hardware solution that addresses these problems has been presented by Smolens et al. [2004]. Here, the processor pipeline is extended by a fingerprint register which hashes all instructions and operands on the fly. This hash can then be used as basis for regular voting, for instance, after a predetermined number of retired instructions. The key idea is that the hash value for all redundant executions must be the same, unless errors appear. Since the fingerprint is calculated by dedicated hardware, nearly no additional time-overhead is introduced in the error-free case. The *Fine-Grained Task Redundancy* (FGTR) method [Axer et al. 2011] replicates only selected tasks and performs regular error checks during execution. Checking is realized in hardware using the fingerprint approach. In the error-free case and under a predictable scheduling policy, this method also behaves predictably since no additional uncertainty is added. However, the analysis of such tasks under the presence of errors is not straightforward due to the mutual dependencies introduced by the comparison and the additional recovery overhead (similar to a retransmissions in CAN). Every time a comparison is successful, a checkpoint is created. If an error is detected due to inconsistent fingerprints, the last checkpoint must be restored. FGTR provides a tradeoff between static overhead due to regular checkpointing and random overhead in case of errors. In general, the method to analyze error-induced timing effects on the processor under FGTR is very similar to the analysis of erroneous frames on the CAN bus, besides the difference in protocol and overhead parameters.

## 7.3. Conclusions and Challenges

By applying well-known fault-tolerance mechanisms such as DMR, TMR and coding schemes, it is possible to harden individual components against transient as well as permanent errors. By system-wide application of these methods (e.g., computation and on-chip as well as off-chip communication), it is still possible to design a predictable system which is now annotated by a conservatively bounded safety metric, such as MTTF. This is sufficient to meet the requirements of safety standards.

One of the major remaining challenges is the expressiveness of the error model. To get a conservative bound, it is especially important to have an accurate error model which reflects reality sufficiently. Assuming the standard single-bit error model can be an optimistic assumption in aggressive environments. On the other hand, if the error model is of complex nature (i.e., a hidden Markov model with many states), it is likely that this leads to a state-space explosion with today's system analysis and synthesis approaches.

Due to functional and timing dependencies among components, it is not easily possible to decompose an error-aware system analysis into independent component analyses. Thus, system analyses are usually holistic and don't scale with the size of a complex system.

## 8. CONCLUSIONS AND CHALLENGES

In this article, we have surveyed some of the recent advances regarding techniques for building timing predictable embedded systems. In particular, we have covered techniques whereby architectural elements that are introduced primarily for efficiency, can also be made timing predictable. Compared to the situation described, for instance, in the earlier paper by Thiele and Wilhelm [2004], significant advances have been made.

Concerning processor architectural elements, we now understand the predictability properties of a range of pipelines, memory system designs, etc., as a basis for design principles for predictability. Also, processor designs with timing as part of their instruction set semantics have been developed. Concerning multicore platforms, we have obtained a good understanding of, and some solutions to, the difficult problem of providing predictability guarantees for program execution: The solutions involve partitioning the resources and isolating as much as possible each task from interference. Unfortunately, current multicore processors provide rather limited support for solutions.

Thiele and Wilhelm [2004] proposed the integration of development techniques and tools across several layers as an important path forward. We have described the integration of compilation and WCET analysis as an important instance of such an integration. A corresponding tool provides a platform that allows to systematically investigate the impact on predictability of various common compiler optimizations, and the trade-off between average- and worst-case execution time. As another such integration, we described the incorporation of execution time analysis into synchronous C dialects which provide deterministic coordination and communication constructs for concurrent threads, resulting in timing predictable synchronous programming languages.

Sections 2 to 6 considered predictability assuming the absence of hardware errors. In contrast, Section 7 described how the definition of predictability can accommodate the unreliability inherent in networked systems, and how this relates to safety standards.

In conclusion, research on predictability of hardware and software features, and how to analyze them, has produced results that allow predictable systems to be built, at least on uniprocessor platforms. Since predictability cuts across all levels in system design, a design flow for predictable system design must carefully integrate solutions at all these levels. Thiele and Wilhelm [2004] pointed to model-based design as a promising approach for increasing predictability, since code generators can be tailored to generate disciplined code. Code generators in existing model-based design tools do not fully realize this potential since they are typically designed with other goals in mind.

Concerning multicores, there are still a number of unsolved challenges for truly predictable system design, including how to strictly isolate tasks, how to share bandwidth and other resources in a predictable manner. We expect that better solutions for these challenges must appear before industry-strength timing analyzers can be applied to multicore systems. Also, processor designers and manufacturers must produce multicore platforms that prioritize support for predictability in addition to performance.

Another important future challenge is to provide techniques to integrate different types of applications with different predictability requirements on the same platform.

This will allow engineers to fully utilize the computation resources for low-criticality applications and to provide predictability guarantees for high-criticality applications.

## REFERENCES

B. Akesson, K. Goossens, and M. Ringhofer. 2007. Predator: a predictable SDRAM memory controller. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 251–256.

D. H. Albonesi and I. Koren. 1994. Tradeoffs in the design of single chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 25–34.

S. Andalam, P. Roop, and A. Girault. 2010. Predictable multithreading of embedded applications using PRET-C. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*.159–168.

S. Andalam, P. Roop, and A. Girault. 2011. Pruning infeasible paths for tight WCRT analysis of synchronous programs. In *Proceedings of the Conference on Design Automation and Test in Europe*. 204–209.

B. Andersson, A. Easwaran, and J. Lee. 2010. Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. *ACM SIGBED Review* 7, 1, 4:1–4:4.

A. Andrei, P. Eles, Z. Peng, and J. Rosen. 2008. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the International Conference on VLSI Design*. 103–110.

A. Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Trans. Software Eng.* 11, 12, 1491–1501.

P. Axer, M. Sebastian, and R. Ernst. 2011. Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 149–158.

T. P. Baker. 2003. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the International Real-Time Systems Symposium*. 120–129.

J. Barre, C. Rochange, and P. Sainrat. 2008. A predictable simultaneous multithreading scheme for hard real-time. In *Proceedings of the International Conference on Architecture of Computing Systems*. 161–172.

S. K. Baruah. 2007. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the International Real-Time Systems Symposium*. 119–128.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2010a. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 33–44.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2010b. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proceedings of the International Real-Time Systems Symposium*. 14–24.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2011. Is semi-partitioned scheduling practical? In *Proceedings of the Euromicro Conference on Real-Time Systems*. 125–135.

A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages twelve years later. *Proc. IEEE* 91, 1, 64–83.

C. Berg. 2006. PLRU cache domino effects. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*.

N. C. Bernardes Jr. 2001. On the predictability of discrete dynamical systems. *Proc. Amer. Math. Soc.* 130, 7, 1983–1992.

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

M. Boldt, C. Traulsen, and R. Von Hanxleden. 2008. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP J. Embed. Syst.* 4.

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

S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6, 10–16.

D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. 1995. Methods and specification tools for Airbus on-board systems. *Microprocess. Microsyst.* 19, 9, 511–515.

I. Broster, G. Bernat, and A. Burns. 2002a. Weakly hard real-time constraints on controller area network. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 134–141.

I. Broster, A. Burns, and G. Rodríguez-Navas. 2002b. Probabilistic analysis of CAN with faults. In *Proceedings of the International Real-Time Systems Symposium*. 269–278.

D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke. 2011. Temporal isolation on multiprocessing architectures. In *Proceedings of the Design Automation Conference*. 274–279.

G. Buttazzo. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* 3rd Ed. Springer.

J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. Anderson. 2006. LITMUS/RT: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the International Real-Time Systems Symposium*. 111–126.

CAN. 1991. CAN specification 2.0. Robert Bosch GmbH.

J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. 2004. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods and Models*, Chapman Hall/CRC, Boca Raton, FL.

J. Charzinski. 1994. Performance of the error detection mechanisms in CAN. In *Proceedings of the International CAN Conference*. 1–20.

S. Chattopadhyay, A. Roychoudhury, and T. Mitra. 2010. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the International Workshop on Software & Compilers for Embedded Systems*. 6:1–6:10.

D. Chiou, L. Rudolph, S. Devadas, L. Randolph, and B. S. Ang. 1999. Dynamic cache partitioning via columnization. Tech. Rep. 430, Massachusetts Institute of Technology.

C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. 2010. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*. 36–42.

R. Davis, A. Burns, R. Bril, and J. J. Lukkien. 2007. Controller area network CAN schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.* 35, 3, 239–272.

S. Edwards and E. Lee. 2007. The case for the precision timed (PRET) machine. In *Proceedings of the Design Automation Conference*. 264–265.

S. Edwards and J. Zeng. 2007. Code generation in the Columbia Esterel Compiler. *EURASIP J. Embed. Syst. 2007*.

A. El-Haj-Mahmoud, A. S. Al-Zawawi, A. Anantaraman, and E. Rotenberg. 2005. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 213–224.

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

H. Falk and J. C. Kleinsorge. 2009. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the Design Automation Conference*. 732–737.

H. Falk and H. Kotthaus. 2011.WCET-driven cache-aware code positioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 145–154.

H. Falk and P. Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Syst.* 46, 2, 251–300.

M. Fernandez, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. 2012. Assessing the suitability of the NGMP multi-core processor in the space domain. In *Proceedings of the International Conference on Embedded Software*. 175–184.

J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca. 2004. An experiment to assess bit error rate in CAN. In *Proceedings of the International Workshop of Real-Time Networks*. 15–18.

G. Gebhard and S. Altmeyer. 2007. Optimal task placement to improve cache performance. In *Proceedings of the International Conference on Embedded Software*. 259–268.

M. Gerdes, F. Kluge, T. Ungerer, and C. Rochange. 2012. The split-phase synchronisation technique: Reducing the pessimism in the WCET analysis of parallelised hard real-time programs. In *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications*. 88–97.

D. Grund, J. Reineke, and R. Wilhelm. 2011. A template for predictability definitions with supporting evidence. In *Proceedings of the Workshop on Predictability and Performance in Embedded Systems*. 22–31.

N. Guan, M. Stigge, W. Yi, and G. Yu. 2009a. Cache-aware scheduling and analysis for multicores. In *Proceedings of the International Conference on Embedded Software*. 245–254.

N. Guan, M. Stigge, W. Yi, and G. Yu. 2009b. New response time bounds for fixed priority multiprocessor scheduling. In *Proceedings of the International Real-Time Systems Symposium*. 387–397.

N. Guan, M. Stigge, W. Yi, and G. Yu. 2010. Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 165–174.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data-flow programming language Lustre. *Proc. IEEE* 79, 9, 1305–1320.

D. Hardy, T. Piquet, and I. Puaut. 2009. Using bypass to tightenWCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the International Real-Time Systems Symposium*. 68–77.

T. Henzinger. 2008. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Trans. Royal Soc. A* 366, 1881, 3727–3736.

T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2003. Giotto: A time-triggered language for embedded programming. *Proc. IEEE* 91, 1, 84–99.

IEC 61508. 2010. IEC 61508 Edition 2.0: Functional safety of electrical/electronic/programmable electronic safety-related systems. http://www.iec.ch/functionalsafety/standards/page2.htm.

ISO 26262. 2011. ISO 26262: Road vehicles – functional safety. International Organization for Standardization.

V. Izosimov, P. Pop, P. Eles, and Z. Peng. 2006. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *Proceedings of the International Workshop on Electronic Design, Test and Applications*. 440–447.

L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. 2008. Performance debugging of Esterel specifications. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 173–178.

S. Kato and N. Yamasaki. 2008. Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the International Conference on Embedded Software*. 139–148.

R. Kirner and P. Puschner. 2001. Transformation of path information for WCET analysis during compilation. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 29–36.

R. Kuehn. 1969. Computer redundancy: design, performance, and future. *IEEE Trans. Reliab.* 18, 1, 3–11.

K. Lakshmanan, R. Rajkumar, and J. Lehoczky. 2009. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 239–248.

J. Lala and R. Harper. 1994. Architectural principles for safety-critical real-time applications. *Proc. IEEE* 82, 1, 25–40.

S. Lauzac, R. G. Melhem, and D. Mossé. 1998. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 188–195.

G. Legoff. 1996. Using synchronous languages for interlocking. In *Proceedings of the International Conference on Computer Application in Transportation Systems*.

X. Li and R. Von Hanxleden. 2012. Multithreaded reactive programming – the Kiel Esterel Processor. *IEEE Trans. Computers* 61, 3, 337–349.

Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proceedings of the International Real-Time Systems Symposium*. 57–67.

B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. 2008. Predictable programming on a precision timed architecture. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 137–146.

C. L. Liu and J. W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1, 46–61.

I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. 2012. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of the International Conference on Computer Design*.

J. W. S. Liu. 2000. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ.

T. Liu, M. Li, and C. Xue. 2009. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 35–44.

T. Liu, Y. Zhao, M. Li, and C. J. Xue. 2011. Joint task assignment and cache partitioning with cache locking for WCET minimization on MPSoC. *J. Parallel Distrib. Comput.* 71, 11, 1473–1483.

T. Lundqvist and P. Stenström. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the International Real-Time Systems Symposium*. 12–21.

M. Lv, W. Yi, N. Guan, and G. Yu. 2010. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the International Real-Time Systems Symposium*. 339–349.

M. A. Maksoud and J. Reineke. 2012. An empirical evaluation of the influence of the load-store unit on WCET analysis. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*. 13–24.

J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. 2008. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *Proceedings of the International Conference on Computer Design*. 371–376.

A. M. Molnos, M. J. M. Heijligers, S. D. Cotofana, and J. T. J. van Eijndhoven. 2004. Cache partitioning options for compositional multimedia applications. In *Proceedings of the Annual Workshop on Circuits, Systems and Signal Processing*. 86–90.

F. Mueller. 1995. Compiler support for software-based cache partitioning. In *Proceedings of the Workshop on Languages, Compilers, & Tools for Real-Time Systems*. 125–133.

N. Navet, Y. Song, and F. Simonot. 2000. Worst-case deadline failure probability in real-time applications distributed over controller area network. *J. Syst. Archit.* 46, 7, 607–617.

J. Nowotsch and M. Paulitsch. 2012. Leveraging multi-core computing architectures in avionics. In *Proceedings of the European Dependable Computing Conference*. 132–143.

M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. 2009a. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the International Symposium on Computer Architecture*. 57–68.

M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero. 2009b. An analyzable memory controller for hard real-time CMPs. *IEEE Embed. Sys. Lett.* 1, 4, 86–90.

M. Paolieri, E. Quiñones, F. J. Cazorla, R. I. Davis, and M. Valero. 2011. IA3: An interference aware allocation algorithm for multicore hard real-time systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 280–290.

D. Paret and R. Riesco. 2007. *Multiplexed Networks for Embedded Systems: CAN, LIN, Flexray, Safe-by-Wire. . . .* John Wiley & Sons, Hoboken, NJ.

B. Parhami. 1994. Voting algorithms. *IEEE Trans. Reliab.* 43, 4, 617–629.

R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. 2010. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design Automation and Test in Europe*. 741–746.

S. Plazar, P. Lokuciejewski, and P. Marwedel. 2009. WCET-aware software based cache partitioning for multi-task real-time systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*. 78–88.

D. Potop-Butucaru, S. A. Edwards, and G. Berry. 2007. *Compiling Esterel*. Springer.

L. Pullum. 2001. *Software Fault Tolerance—Techniques and Implementation*. Artech House.

P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. 2012. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.* 8, 4, Article 34.

J. Reineke and D. Grund. 2013. Sensitivity of cache replacement policies. *ACM Trans. Embed. Comput. Syst.* 12, 1.

J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. 2011. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 99–108.

J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. 2006. A definition and classification of timing anomalies. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*.

C. Rochange and P. Sainrat. 2005. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proceedings of the Conference on Computing Frontiers*. 307–314.

K. Rokas, Y. Makris, and D. Gizopoulos. 2003. Low cost convolutional code based concurrent error detection in FSMs. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems*. 344–351.

J. Rosen, A. Andrei, P. Eles, and Z. Peng. 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the International Real-Time Systems Symposium*. 49–60.

Z. A. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli. 2002. REFLIX: A processor core for reactive embedded applications. In *Proceedings of the International Conference on Field Programmable Logic and Application*. 945–954.

A. Saleh, J. Serrano, and J. Patel. 1990. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. Reliab.* 39, 1, 114–122.

S. Schliecker, M. Negrean, and R. Ernst. 2010. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design Automation and Test in Europe*. 759–764.

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

M. Sebastian, P. Axer, and R. Ernst. 2011. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*. 79–88.

M. Sebastian and R. Ernst. 2009. Reliability analysis of single bus communication with real-time requirements. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*. 3–10.

J. Smolens, B. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. 2004. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 224–234.

J. Stankovic and K. Ramamritham. 1990. What is predictability for real-time systems? *Real-Time Syst.* 2, 4, 247–254.

V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. 2005. WCET centric data allocation to scratchpad memory. In *Proceedings of the International Real-Time Systems Symposium*. 223–232.

L. Thiele and R. Wilhelm. 2004. Design for timing predictability. *Real-Time Syst.* 28, 2–3, 157–177.

K. Tindell and A. Burns. 1994. Guaranteed message latencies for distributed safety-critical hard real-time control networks. Tech. Rep. YCS229, Department of Computer Science, University of York.

K. Tindell, A. Burns, and A. Wellings. 1995. Calculating controller area network (CAN) message response times. *Control Eng. Pract.* 3, 8, 1163–1169.

T. Ungerer, F. J. Cazorla, P. Sainrat, et al. 2010. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro* 30, 5, 66–75.

J. Van Lint. 1999. *Introduction to Coding Theory* 3rd Ed. Springer.

R. Von Hanxleden. 2009. SyncCharts in C–a proposal for light-weight, deterministic concurrency. In *Proceedings of the International Conference on Embedded Software*. 225–234.

J. Wakerly. 1976. Microcomputer reliability improvement using triple-modular redundancy. *Proc. of the IEEE* 64, 6, 889–895.

Q. Wan, H. Wu, and J. Xue. 2012. WCET-aware data selection and allocation for scratchpad memory. In *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. 41–50.

WCC. 2014. WCET-aware Compilation. http://ls12-www.cs.tu-dortmund.de/research/activities/wcc.

S. Wicker and V. Bhargava. 1999. *Reed-Solomon Codes and Their Applications*. IEEE.

R. Wilhelm, J. Engblom, A. Ermedahl, et al. 2008. The worst-case execution-time problem—Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36.

R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. Comput.-Aid. Desi. Integrat. Circuits Syst.* 28, 7, 966–978.

J. Wolf and R. Blakeney. 1988. An exact evaluation of the probability of undetected error for certain shortened binary CRC codes. In *Proceedings of the Military Communications Conference*. 287–292.

J. K. Wolf, A. Michelson, and A. Levesque. 1982. On the probability of undetected error for linear block codes. *IEEE Trans. Commun.* 30, 2, 317–325.

S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and C. Salcic. 2009. STARPro — a new multithreaded direct execution platform for Esterel. *Electron. Notes Theoret. Computer Science* 238, 1, 37–55.

W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. 2005a. Improving WCET by optimizing worst-case paths. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 138–147.

W. Zhao, D. Whalley, C. Healy, and F. Mueller. 2005b. Improving WCET by applying a WC code-positioning optimization. *ACM Trans. Archit. Code Optim.* 2, 4, 335–365.

J. Zou, S. Matic, E. Lee, T. H. Feng, and P. Derler. 2009. Execution strategies for PTIDES, a programming model for distributed embedded systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 77–86.

Received June 2012; revised October 2012; accepted November 2012