Building Timing Predictable Embedded Systems*

PHILIP AXER[1], ROLF ERNST[1], HEIKO FALK[2], ALAIN GIRAULT[3], DANIEL GRUND[4], NAN GUAN[5], BENGT JONSSON[5], PETER MARWEDEL[6], JAN REINEKE[4], CHRISTINE ROCHANGE[7], MAURICE SEBASTIAN[1], REINHARD VON HANXLEDEN[8], REINHARD WILHELM[4], WANG YI[5]
1: TU Braunschweig, 2: Ulm University, 3: INRIA Grenoble Rhône-Alpes,
4: Saarland University, 5: Uppsala University, 6: TU Dortmund,
7: University of Toulouse, 8: Christian-Albrechts-Universität, Kiel

Abstract

A large class of embedded systems is distinguished from general purpose computing systems by the need to satisfy strict requirements on timing, often under constraints on available resources. Predictable system design is concerned with the challenge of building systems for which timing requirements can be guaranteed *a priori*. Perhaps paradoxically, this problem has become more difficult by the introduction of performance-enhancing architectural elements, such as caches, pipelines, and multithreading, which introduce a large degree of nondeterminism and make guarantees harder to provide. The intention of this paper is to summarize current state-of-the-art in research concerning how to build predictable yet performant systems. We suggest precise definitions for the concept of "predictability", and present predictability concerns at different abstractions levels in embedded software 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 based on the synchronous 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.

Keywords: Embedded systems, safety-critical systems, predictability, timing analysis, resource sharing

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 non-functional constraints, e.g., on latencies or throughput. In several application domains, including automotive, avionics, industrial automation, many functionalities are associated with strict requirements on deadlines for delivering results of calculations. In many cases, failure to meet deadlines may cause a catastrophic or at least highly undesirable system failure, associated with risks for human or economical damages.

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

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 [99]. 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, run-time system and scheduling, communication infrastructure, etc. Perhaps paradoxically, this problem has become more difficult by the trend to make processors more performant, since the introduced architectural elements, such as pipelines, out-of-order execution, on-chip memory systems, etc., lead to a large degree of nondeterminism in system execution, making guarantees harder to provide.

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

It is therefore important to develop techniques that really guarantee timing requirements that are commensurate with the actual performance of a system. Significant advances have been made in the last decade on analysis of timing properties (see, e.g., [114] for an overview). However, these techniques cannot make miracles. They can only make predictions if the analyzed mechanisms are themselves predictable, i.e., if their relevant timing properties can be foreseen with sufficient precision. Fortunately, the understanding of how to design systems that reconcile efficiency and predictability has increased in recent years. Recent research efforts include European projects, such as Predator[1] and MERASA [105], that have focused on techniques for designing predictable *and* efficiency systems, as well as the PRET project [37, 63], which aims to equip instruction-set architectures with predictable timing.

The intention of this paper is to summarize some recent advances in research on building predictable yet performant systems. In particular, it will cover techniques, whereby architectural elements that are introduced primarily for efficiency, can also be made timing-predictable. Such elements include processor pipelines, memory hierarchies, and multiple processors. It will also show how such techniques can be exploited to make the timing properties of a program directly visible to the developer at design-time, thus giving him direct control over the timing properties of a system under development. We will not discuss particular analysis methods for deriving timing bounds; this area has progressed significantly (e.g., [114]), but a meaningful overview would require too much space.

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 how the instruction-set architecture for a processor can be equipped with predictable timing semantics, so that the timing of program execution can be made predictable. Important here is the design and use of processor pipelines and the memory system. In Sections 4 and 5, we move up one level of abstraction, from the instruction-set architecture 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 ex-

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

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

Table 1: Examples for intuition behind predictability.

plain 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 for timing analysis (aiT) can be integrated with a compiler for a widely-used language (C). The integration of these tools can equip program fragments with timing semantics (given a compilation strategy and target platform). It also serves as a basis for assessing different compilation strategies when predictability is the main design objective.

In Section 6, we consider techniques for multicores. Such platforms are finding their way into many embedded applications, but introduce difficult challenges for predictability. Major challenges include the arbitration of shared resources such as on-chip memories and buses. Predictability can be achieved only if logically unrelated activities can be isolated from each other, e.g., 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, e.g., may corrupt messages transmitted over a bus between different components of an embedded system. Without bounding assumptions on the occurrence of errors (which often can not be given for actual systems), predictability guarantees can only be given in a probabilistic sense. We present mechanisms for achieving such guarantees, e.g., in order to comply with various standards for safety-critical systems. Finally, in Section 8, we present some brief conclusions.

2 Fundamental Predictability Concepts

Predictable system design is made increasingly difficult by past and current developments in system and computer architecture design, where more performant architectural elements are introduced for performance, but make timing guarantees harder to provide [34, 115, 113]. Hence, research on in this area can be divided into two strands: On the one hand there is the development of ever better analyses to keep up with these developments. On the other hand there is the 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 1 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 PowerPC755.

In the following we discuss key aspects of predictability and therefrom derive a template for pre-

dictability definitions.

2.1 Key Aspects of Predictability

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

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

Consequently, a system is predictable if one can foretell facts about its future, i.e. determine interesting things about its behavior. In general, the behaviors of such a system can be described by a possibly infinite set of execution traces. However, a prediction will usually refer to derived properties of such traces, e.g. 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, e.g. the input of a program, and the prediction to be made should be valid for all possible cases, e.g. all admissible program inputs. Hence, the second aspect of predictability are the *sources of uncertainty* that influence the prediction quality.

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

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

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

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

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

- For instance, one could refine Proposition 2.1 by taking into account the complexity/cost of the analysis that determines the property. However, the clause "by *any* analysis not more expensive than X" complicates matters: The key aspect of inherence requires a quantification over all analyses of a certain complexity/cost.
- Another refinement would be to consider different sources of uncertainty separately to capture only the influence of one source. We will have an example of this later.
- One could also distinguish the extent of uncertainty. E.g. is the program input completely unknown or is partial information available?
- It is desirable that the predictability of a system can be determined automatically, i.e. computed.

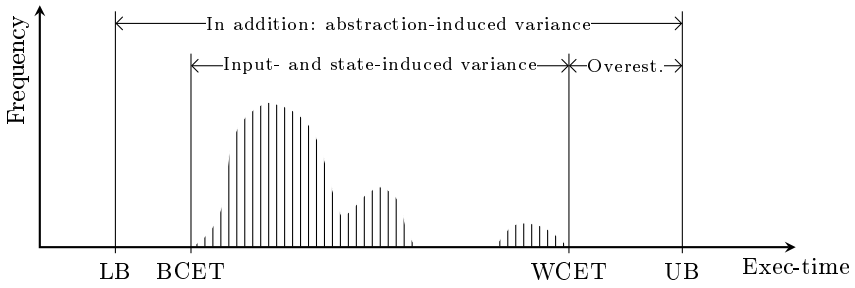


Figure 1: Distribution of execution times ranging from best-case to worst-case execution time (BCET/WCET). Sound but incomplete analyses can derive lower and upper bounds (LB, UB).

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

2.2 A Predictability Template

Besides the key aspect of inherence, the other key aspects of predictability depend on the system under consideration. We therefore propose a template for predictability with the goal to enable a concise and uniform description of predictability instances. It consists of the above mentioned key aspects (a) property to be predicted, (b) sources of uncertainty, and (c) quality measure.
In 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, i.e. the prediction should be valid for all possible initial hardware states. Additionally, schedulability analysis cannot handle a characterization of execution times in the form of a function depending on inputs. Hence, the prediction should also hold for all admissible program inputs.
- Usually, schedulability analysis requires a characterization of execution times in the form bounds on the execution time. Hence, a reasonable quality measure is the quotient of BCET over WCET; the smaller the difference the better.
- The inherence property is satisfied as BCET and WCET are inherent to the system.

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 state $Q \subseteq \mathcal{Q}$ and uncertainty about the program input $I \subseteq \mathcal{I}$, the timing predictability of a program p is*

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

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

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

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)

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

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)

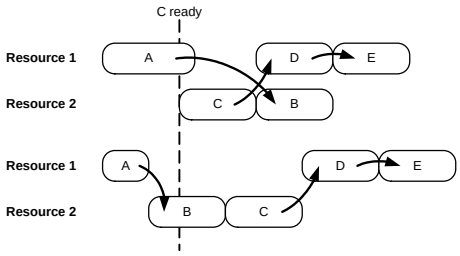
$$IIPr_p(Q,I) := \min_{q \in Q} \min_{i_1,i_2 \in I} \frac{T_p(q,i_1)}{T_p(q,i_2)} \tag{3}$$

Example of state-induced timing unpredictability As an application of Definition 2.3, we show how it can be used to give a quantitative characterization of domino effects. A system exhibits a *domino effect* [68] if there are two hardware states q_1, q_2 such that the difference in execution time of the same program starting in q_1 respectively q_2 is proportional to its length, i.e. cannot be bounded by a constant. For instance, the iterations of a program loop never converge to the same hardware state and the difference in execution time increases in each iteration. [95] describes a domino effect in the pipeline of the PowerPC 755. It involves the two asymmetrical integer execution units, a greedy instruction dispatcher, and an instruction sequence with read-after-write dependencies.

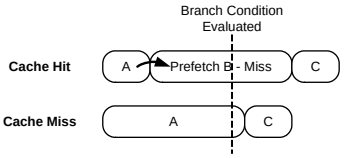
The dependencies in the instruction sequence are such that the decisions of the dispatcher result in a longer execution time if the initial state of the pipeline is empty than in case it is partially filled. This can be repeated arbitrarily often, as the pipeline states after the execution of the sequence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

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

Another example for a domino effect is given by [16] who considers the PLRU replacement policy of caches. In Section 3, we describe results on the state-induced cache predictability of various replacement policies.



(a) Scheduling anomaly.



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

Figure 2: Speculation and Scheduling anomalies, taken from [87].

Timing Anomalies The notion of *timing anomalies* was introduced by Lundqvist and Stenström in [68]. In the context of WCET analysis, [87] presents a formal definition and additional examples of such phenomena. Intuitively, a timing anomaly is a situation where the local worst-case does not contribute to the global worst-case. For instance, a cache miss–the local worst-case–may result in a globally shorter execution time than a cache hit because of scheduling effects. See Figure 2(a) for an example. Shortening instruction A leads to a longer overall schedule, because instruction B can now block the "more" important instruction C. Analogously, there are cases where a shortening of an instruction leads to an even greater decrease in the overall schedule.

Another example occurs with branch prediction. A mispredicted branch results in unnecessary instruction fetches, which might miss the cache. In case of cache hits the processor may fetch more instructions. Figure 2(b) illustrates this.

3 Microarchitecture

In this and the following sections, we consider predictability of architectural elements at different levels in the system hierarchy. This section discusses microarchitectural features at the uniprocessor level, focussing primarily on pipelines (Section 3.1), caches (Section 3.2), and memories (Section 3.3).

An *instruction set architecture* (ISA) defines the interface between hardware and software, i.e., the format of software binaries and their semantics in terms of input/output behavior. A *microarchitecture* defines how an ISA is implemented on a processor. A single ISA may have many microarchitectural realizations. For example, there are many implementations of the x86 ISA by Intel and AMD.

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

As a consequence of abstracting from execution time in ISA semantics, worst-case execution time (WCET) analyses need to consider the microarchitecture a software binary will be executed on. The aforementioned microarchitectural techniques greatly complicate WCET analyses. For simple, non-pipelined microarchitectures without caches one could simply sum up the execution times of individual instructions to obtain the exact execution time of a sequence of instructions. With

pipelining, caches, and other features, execution times of successive instructions overlap, and—more importantly—they vary depending on the execution history[2] leading to the execution of an instruction: a read immediately following a write to the same register incurs a pipeline stall; the first fetch of an instruction in a loop results in a cache miss, whereas subsequent accesses may result in cache hits, etc.

3.1 Pipelines

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

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

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

All these features influence execution time. To compute a precise bound on the execution time of a basic block, the analysis needs to exclude as many *timing accidents* as possible. Such accidents are data hazards, branch mispredictions, occupied functional units, full queues, etc.

Abstract states may lack information about the state of some processor components, e.g., caches, queues, or predictors. Transitions between states of the concrete pipeline may depend on such information. This causes the abstract pipeline model to become non-deterministic although the concrete pipeline is deterministic. When dealing with this non-determinism, one could be tempted to design the WCET analysis such that only the "locally worst-case" transition is chosen, e.g., the transition corresponding to a pipeline stall or a cache miss. However, in the presence of *timing anomalies* [69, 87] such an approach is unsound. Thus, in general, the analysis has to follow all possible successor states.

Classification of microarchitectures from [113] Architectures can be classified into three categories depending on whether they exhibit timing anomalies or domino effects [113].

- **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

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

precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant number of cycles to the local worst-case path. The Infineon TriCore is assumed, but not formally proven, to belong to this class.

- **Non-compositional architectures:** These architectures, e.g., the PowerPC 755 exhibit domino effects and timing anomalies. For such architectures timing analyses always have to follow all paths since a local effect may influence the future execution arbitrarily.

Approaches to Predictable Pipelining The complexity of WCET analysis can be reduced by regulating the instruction flow of the pipeline at the beginning of each basic block [88]. This removes all timing dependencies within the pipeline between basic blocks. Thus, WCET analysis can be performed on each basic block in isolation. The authors take the stance that efficient analysis techniques are a prerequisite for predictability: "a processor might be declared unpredictable if computation and/or memory requirements to analyse the WCET are prohibitive."

With the advent of multi-core and multi-threaded architectures, new challenges and opportunities arise in the design of timing-predictable systems: Interference between hardware threads on shared resources further complicates analysis. On the other hand, timing models for individual threads are often simpler in such architectures. Recent work has focussed on providing timing predictability in multithreaded architectures:

One line of research proposes modifications to simultaneous multithreading architectures [10, 72]. These approaches adapt thread-scheduling in such a way that one thread, the real-time thread, is given priority over all other threads, the non-real-time threads. As a consequence, the real-time thread experiences no interference by other threads and can be analyzed without having to consider its context, i.e., the non-real-time threads. This guarantees temporal isolation for the real-time thread, but not for any other thread running on the core. If multiple real-time tasks are needed, then time sharing of the real-time thread is required.

Earlier, a more static approach was proposed by El-Haj-Mahmoud et al. [39] 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 [65], a precision-timed (PRET) machine [37] implementing the ARM instruction set, employs a five-stage thread-interleaved pipeline. The thread-interleaved pipeline contains four hardware threads that run in the pipeline. Instead of dynamically scheduling the execution of the threads, a predictable round-robin thread schedule is used to remove temporal interference. The round-robin thread schedule fetches a different thread every cycle, removing data hazard stalls stemming from the pipeline resources. Unlike the virtual multiprocessor, the tasks on each thread need not be determined a priori, as hardware threads cannot affect each other's schedule. Unlike Mische et al.'s [72] approach, all the hardware threads in the PTARM can be used for real-time purposes.

3.2 Caches and Scratchpad Memories

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

	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

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

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, $\mathrm{SIPr}_p(n)$, to capture the maximal variance in the number of cache misses due to different cache states, $q_1, q_2 \in Q_p$, for an arbitrary but fixed sequence of memory accesses, s, of length n, i.e. $s \in B_n$, where B_n denotes the set of sequences of memory accesses of length n. Given that $M_p(q, s)$ denotes the number of misses of policy p accessing sequence s starting in cache state q, $\mathrm{SIPr}_p(n)$ is defined as follows:

Definition 3.1 (State-induced cache predictability)

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

To investigate the influence of the initial cache states in the long run, we have studied $\lim_{n\to\infty} \mathrm{SIPr}_p(n)$. A tool called RELACS[3], described in [85], is able to compute $\lim_{n\to\infty} \mathrm{SIPr}_p(n)$ automatically for a large class of replacement policies. Using RELACS, we have obtained sensitivity results for the widely-used policies LRU, FIFO, PLRU, and MRU, at associativities ranging from 2 to 8.

Figure 2 depicts the analysis results. There can be no cache domino effects for LRU. Obviously, 1 is the optimal result and no policy can do better. FIFO and PLRU are much more sensitive to their state than LRU. Depending on its state, FIFO(k) may have up to k times as many misses. At associativity 2, PLRU and LRU coincide. For greater associativities, the number of misses incurred by a sequence s starting in state q_1 cannot be bounded the number misses incurred by the same sequence s starting in another state q_2.

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

Interference on Shared Caches Without further adaptation, caches do not provide temporal isolation: the same application, processing the same inputs, may exhibit wildly varying cache performance depending on the state of the cache when the application's execution begins [113]. 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

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

the SPM and lower levels of the memory hierarchy. Accesses to the SPM will be serviced with low latency, predictably and repeatably. However, similar to the use of the register file, it is the compiler's responsibility to make correct and efficient use of the SPM. This is challenging, in particular when the SPM is to be shared among several applications, but it also presents the opportunity of high efficiency, as the SPM management can be tailored to the specific application, in contrast to the hardwired cache replacement logic. Section 5.3 briefly discusses results on SPM allocation and the related topic of cache locking.

3.3 Dynamic Random Access Memory

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

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

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

Recently, several predictable DRAM controllers have been proposed [1, 76, 86]. 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* [1] and TDM in *AMC* [76], 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. [86] partition the physical address space following the internal structure of the DRAM device. This eliminates contention for shared resources within the device, making accesses temporally predictable and temporally isolated. Replacing dedicated refresh commands with lower-latency manual row accesses in single DRAM banks further reduces the impact of refreshes on worst-case latencies.

4 Synchronous programming languages for predictable systems

Embedded systems typically perform a significant number of different activities that must be coordinated and 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 coordinated by a suitable RTOS, and augments them by giving compile-time semantics to programs and program segments. 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 The synchronous language approach to predictability

4.1.1 The essence of synchronous programming languages

Many programming languages that have been proposed for predictable systems are *synchronous* 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 [15]. 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 with asynchronous concurrency, synchronous programs avoid introducing non-determinism by interleaving. On a sequential processor, with the asynchronous concurrency paradigm, two independent, atomic parallel tasks must be executed in some non-deterministically chosen sequential order. The drawback is that 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*.

To take a concrete example, the Esterel [17] statement "`every 60 second emit minute`" specifies that the signal `minute` is *exactly synchronous* with the 60^{th} occurrence of the signal `second`. At a more fundamental level, the synchronous abstraction eliminates the non-determinism resulting from the interleaving of concurrent behaviors. This allows deterministic semantics, 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 [50]) is widely used both in the civil airplane industry [24] and in the railway industry [60].

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

4.1.2 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: (i) computing the worst case reaction time (WCRT) of the program, defined as the WCET of the body of the periodic execution loop; and (ii) 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, i.e., 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 [17] 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.

4.1.3 Requirements for time 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 (as opposed to just computing the WCET). 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 (Sec. 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 proof-read by humans is also easier to analyze by WCRT analyzers.

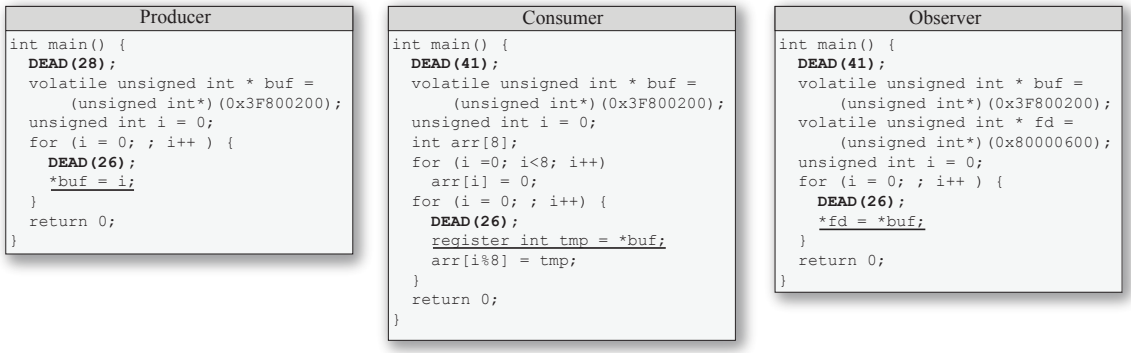
4.2 Language constructs for expressing synchrony and timing

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 [15] and [17]. 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 `buf` (see Fig. 3). This is a typical pattern for reactive real-time systems.

4.2.1 The Berkeley-Columbia PRET language

The original version of PCO (Fig. 3(a)) was introduced to illustrate the programming of the Berkeley-Columbia PRET architecture [63]. The programming language is a multi-threaded version of C, extended with a special *deadline instructions*, 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.[4] Hence, a `DEAD` instruction can only enforce a *lower bound* on the execution time of code segment. By

[4] 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 [63].



Producer

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

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned 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;
}
```

Observer

```
int main() {
  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, by Lickly et al. 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-Extended. Scheduling requirements are specified with explicit thread priorities (1–4). Physical timing is specified separately, here with `sleep()`.

Figure 3: Variants of the PCO example which extend the original PCO [63] with preemptions.

assigning well-chosen values to the DEAD instructions, it is therefore possible to design predictable multi-threaded systems, where problems such as race conditions will be avoided thanks to the interleaving resulting from the 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 [63] guarantees that.

In Fig. 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 lock-step, 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 OS. However, this comes at the price of a brittle, low-level, non-portable scheduling style.

As it turns out, this lock-step 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.

4.2.2 Synchronous C and PRET-C

Synchronous C (originally introduced as SyncCharts in C [107]) and PRET-C [90, 3] are both light-weight, 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 with 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.

Fig. 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 Fig. 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. This allows, for example, a direct synthesis from SyncCharts [104].

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 [83]. The family of *reactive processors* follows an intermediate approach where a synchronous program is compiled into machine code that is then run on a processor with an instruction set architecture (ISA) that directly implements synchronous reactive control flow constructs [108]. W.r.t. predictability, the main advantage of reactive processors is that they offer direct ISA support for crucial features of the languages (e.g., preemption, synchronization, inter-thread communication), therefore allowing a very fine control over the number of machine 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 [37, 63].

The first reactive processor, called REFLIX, was presented by [92], and this group has since then developed a number of follow-up designs [118]. 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 [38]. Benchmarking results show that this architecture provides a 26% decrease in the WCRT compared to a stand-alone MB.

Similarly, the KEP platform (Kiel Esterel Processor) includes a *Tick Manager* that minimizes reaction time jitter and can detect timing overruns [61]. 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 aforementioned deterministic semantics and guiding principles, such as the 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 the KEP, to compute safe estimates for the Tick Manager [20]). This flow-graph based approach was further improved by Mendler et al. with a modular, algebraic approach that also takes signal valuations into account to exclude infeasible paths [71]. Besides, Logothetis et al. used timed Kripke structures to compute tight bounds on synchronous programs [66].

Similarly, a WCRT analyzer was developed for PRET-C programs running on ARPRET [90]. First, the PRET-C program is compiled and each node of its control-flow graph (CFG) is decorated with the number machine cycles required to execute it on ARPRET. Then, this decorated CFG is translated into a timed automaton which is analyzed with UPPAAL to compute the WCRT [90]. To further improve the performances of this WCRT analyzer, infeasible execution paths can be discarded, by combining the abstracted state-space of the program with expressive data-flow information [4].

4.5 Conclusions and future work

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 re-implemented with PRET-C or SC, showing that these languages are easy to use [3, 4].

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, e.g., 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* 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 Related Work

A very first approach to integrate WCET techniques into a compiler was presented by [21]. Flow facts used for timing analysis were annotated manually via source-level pragmas but are not updated during optimization. This turns the entire approach tedious and error-prone. Additionally, the compiler targets the Intel 8051, i.e. an inherently simple and predictable machine without pipeline and caches etc.

While mapping high-level code to object code, compilers apply various optimizations so that the

correlation between high-level flow facts and the optimized object code becomes very low. To keep track of the influence of compiler optimizations on high-level flow facts, co-transformation of flow facts is proposed by [40]. However, the co-transformer has never reached a fully working state, and several standard compiler optimizations can not be modeled at all due to insufficient data structures.

Techniques to transform program path information which keep high-level flow facts consistent during GCC's standard optimizations have been presented by [55]. Their approach was thoroughly tested and led to precise WCET estimates. However, compilation and timing analysis are done in a decoupled way. The assembly file generated by the compiler is passed to the timing analyzer together with the transformed flow facts. Additionally, the proposed compiler is only able to process a subset of ANSI-C, and the modeled target processor lacks pipelines and caches.

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

5.2 Structure of the WCET-aware C Compiler WCC

The most advanced compiler for timing predictable systems is the WCET-aware C Compiler [110] developed within the ArtistDesign NoE. This section presents WCC in more detail as a case study on how compilers for timing predictable systems could look like. WCC is an ANSI-C compiler for Infineon TriCore processors that are heavily used in the automotive industry. The following subsections describe the key components turning WCC into a unique compiler for real-time systems. A complete description of the compiler's infrastructure is given in [43].

Specification of Memory Hierarchies

The performance of many systems is dominated by the memory subsystem. Obviously, timing estimates also heavily depend on the memories. In the WCC environment, it is up to the compiler to provide the WCET analyzer with detailed information about the underlying memory hierarchy. Thus, the compiler uses an infrastructure to specify memory hierarchies. Furthermore, it exploits this memory hierarchy infrastructure to apply memory-aware optimization by assigning parts of a program to fast memories.

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

Integration of Static WCET Analysis into the Compiler

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

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

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

Flow Fact Specification and Transformation

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

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

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

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

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

5.3 Examples of WCET-aware Optimizations

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

Scratchpad Memory Allocation and Cache Locking

As already motivated in Section 3.2, scratchpad memories (SPMs) or locked caches are ideal for WCET-centric optimizations since their timing is fully predictable. Optimizations allocating parts

of a program's code and data onto these memories have been studied intensely in the past [111, 28, 100].

WCC exploits scratchpads by placing parts of a program into an SPM [41] using *integer linear programming (ILP)*. Inequations model the structure of a program's *control flow graph (CFG)*. Constants model the worst-case timing per basic block when being allocated to slow main memory or to the fast SPM. This way, the ILP is always aware of that path in the CFG leading to the longest execution time and can thus optimally minimize the WCET. Besides scratchpads, the compiler also supports cache locking using a similar optimization approach [81].

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

Code Positioning

Code positioning is a well-known compiler optimization improving the I-cache behavior. A contiguous mapping of code fragments in memory avoids overlapping of cache sets and thus decreases the number of cache conflict misses. Code positioning as such was studied in many different contexts in the past, like e. g. to avoid jump-related pipeline delays [121] or at granularity of entire functions [67] or tasks [45].

WCC's code positioning [42] aims to systematically reduce I-cache conflict misses and thus to reduce the WCET of a program. It uses a *cache conflict graph (CG)* as the underlying model of a cache's behavior. Its nodes represent either functions or basic blocks of a program. An edge is inserted whenever two nodes interfere in the cache, i. e. potentially evict themselves from the cache. Using WCC's integrated timing analysis capabilities, edge weights are computed which approximate the number of possible cache misses that are caused during the execution of a CG node.

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

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

Cache Partitioning for Multi-Task Systems

The cache-related optimizations presented so far cannot handle multi-task systems with preemptive scheduling, since it is difficult to predict the cache behavior during context switches. Cache partitioning is a technique for multi-task systems to turn I-caches more predictable. Each task of a system is exclusively assigned a unique cache partition. The tasks in such a system can only evict cache lines residing in the partition they are assigned to. As a consequence, multiple tasks do not interfere with each other any longer w.r.t. the cache during context switches. This allows to apply static timing analysis to each individual task in isolation. The overall WCET of a multi-task system using partitioned caches is then composed of the worst-case timing of the single tasks given

a certain partition size, plus the overhead for scheduling and context switches.

WCET-unaware cache partitioning has already been examined in the past. Cache hardware extensions and associativity- and set-based cache partitioning have been proposed in [32] and [73], resp. [74] presents ideas for compiler support for software-based cache partitioning which serves as basis for WCC's cache partitioning. Software-based cache partitioning scatters the code of each task over the address space such that tasks are solely mapped to only those cache lines belonging to the task's partition. WCC's cache partitioning [82] again relies on ILP to optimally determine the individual tasks' partition sizes.

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

5.4 Conclusions and Future Work

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

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

6 Building Real-Time Applications on Multicores

6.1 Background

Multicore processors bring a great opportunity for high-performance and low-power embedded applications. Unfortunately, the current design of multicore architectures is mainly driven by performance, not by considering timing predictability. Typical multicore architectures [2] integrate a growing number of cores on a single processor chip, each equipped with one or two levels of private caches. The cores and peripherals usually share a memory hierarchy including L2 or L3 caches and DRAM or Flash memory. An interconnection network offers a communication mechanism between the cores, the I/O peripherals and the shared memory. A shared bus can hold a limited number of components as in the ARM Cortex A9 MPCORE. Larger-scale architectures implement more complex Networks on Chip (NoC), like meshes (e.g. the Tile64 by Tilera) or crossbars (e.g. the P4080 by Freescale), to offer a wider communication bandwidth. In all cases, conflicts among accesses from various cores or DMA peripherals to the shared memory must be arbitrated either in the network or in the memory controller. In the following, we distinguish between *storage resources* (e.g. caches) that keep information for a while, generally for several cycles and *bandwidth resources* (e.g. bus or interconnect) that are typically reallocated at each cycle.

6.2 Timing Interferences and Isolation

The timing behavior of a task running on a multicore architecture depends heavily on the arbitration mechanism of the shared resources and other tasks' usage of the 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 [51] at run time. The analysis can be improved by taking task lifetimes into account: tasks that cannot be executed concurrently due to the scheduling algorithm and inter-task dependencies should not be considered as possibly conflicting. Along this line of work, Li *et al.* [62] propose an iterative approach to estimate the WCET bounds of tasks sharing L2 caches. To further improve the analysis precision, the timing behaviour of cache access may be modeled and analyzed using abstract interpretation and model checking techniques [70]. Other approaches aim at determining the extra execution time of a task due to contention on the memory bus [6, 94]. Decoupling the estimation of memory latencies from the analysis of the pipeline behaviour is a way to enhance analysability. However, it is safe for fully timing-compositional systems only.

Spatial and Temporal Isolation

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

The PRedictable Execution Model [80] 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 [91, 7]. To predict latencies, the alignment of basic block time-stamps to the allocated bus slots can be analyzed [31]. However, TDMA-based arbitration is not so common in multicore processors on the market due to performance reasons.

To make the latencies to shared bandwidth resources predictable (boundable), hardware solutions rely on bandwidth partitioning techniques, e.g. round-robin arbitration [77]. Software-controlled cache partitioning schemes allocate private partitions to tasks. For example, *Page-coloring* [46]

allocates the cache content of each task to certain areas in the shared cache by mapping the virtual memory addresses of that task to proper physical memory regions. Then the avoidance of cache interference does not come for free, as the explicit management of cache space adds another dimension to the scheduling and complicates the analysis.

6.3 System-Level Scheduling and Analysis

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

Global Scheduling

One may allow all tasks to compete for execution on all cores. Global scheduling is a realistic option for multcore systems, on which the task migration overhead is much less significant compared with traditional loosly-coupled multiprocessor systems thanks to the hardware mechanisms like on-chip shared cache. So a rapidly increasing interest rises in the study of global scheduling since the late 1990s, around the same time as the major silicon vendors such as IBM and AMD started the development of multicore processors. Global multiprocessor scheduling is a much more difficult problem than uniprocessor scheduling, as first pointed out by Liu in 1969 [64]: *The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.*

One may simply adopt a global task queue and map the released tasks onto the parallel processing cores using single-processor scheduling algorithms such as RM and EDF. Unfortunately these algorithms suffer from the so-called *Dhall effect* [36], namely some system with utilization arbitrarily close to 1 can be infeasible no matter how many processors are added to the system. This result leads to the negative view that global scheduling is widely considered unsuitable for real-time systems. One way to overcome the Dhall effect is fairness scheduling [11], which splits up the task's execution into small pieces and interleaves them with other tasks, to keep the execution of a task to progress in the speed proportional to its workload. Fairness scheduling and its variants [5] can achieve optimality, but is usually considered impracticable to implement due to the run-time overheads.

The major obstacle in precisely analyzing global scheduling and thereby fully exploring its potential is that global scheduling suffers from *timing anomalies*, i.e., a schedulable system can become unschedulable by a parameter change that appears to be harmless. In uniprocessor fixed-priority scheduling the critical instant is the situation where all the interfering tasks release their first instance simultaneously and all the following instances are released as soon as possible. Unfortunately, the critical instant in global scheduling is in general unknown. The critical instant in uniprocessor scheduling, with a strong intuition of resulting in the maximal system workload, does not necessarily lead to the worst-case situation in global fixed-priority scheduling [59]. Therefore, the analysis of global scheduling requires to explore all the possible system behavior.

A large body of works has been done on the efficient analysis of global scheduling by over-approximation. The common approach is to derive an upper bound on the total workload of a task system. Much work has been done on tightening the workload estimation by excluding impossible system behavior from the calculation (e.g. [18, 47]). The work in [47] established the

concept of *abstract critical instant* for global fixed-priority scheduling, namely the worst-case response time of a task occurs under the situation that all higher-priority tasks, except at most $M-1$ of them (M is the number of processors), are released in the same way as the critical instant in uniprocessor fixed-priority scheduling. Although the *abstract critical instant* does not provide an accurate worst-case release pattern, it restricts the analysis to a significantly smaller subset of the overall state space.

The uniprocessor scheduling algorithms like RM and EDF lose their optimality on multicores, which gives rise to the question of what are actually the *good* global scheduling strategies? The fundamental work on global scheduling [35] showed that global EDF, although it can not guarantee deadlines under full workload (100% utilization) any longer, still maintains a weaker concept of optimality in the sense of guaranteeing bounded tardiness (response time) under full workload. In contrast, global fixed-priority scheduling is proved to be able to guarantee bounded tardiness (response time) under a more restricted condition [47].

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 [29]. The design and analysis of partitioned scheduling is relatively simple: as soon as the system has been partitioned into subsystems that will be executed on individual processors each, the traditional uniprocessor real-time scheduling and analysis techniques can be applied to each individual subsystem/processor. The system partitioning is similar to the bin-packing problem [33], for which efficient heuristics are known although it is in general intractable. Similar to the bin-packing problem, partitioned scheduling suffers from resource waste due to fragmentation. Such a waste will be more significant, as the multi core evolves in the direction to integrate a larger number of less powerful cores and the workload of each task becomes relatively heavier comparing with the processing capacity of each individual core. Theoretically, the worst-case utilization bound of partitioned scheduling can not exceed 50% regardless of the local scheduling algorithm on each processor [29].

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 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. [54, 57, 48]). Recently, a notable result is obtained in [48], which generalizes the famous Liu and Layland's utilization bound $N \times (2^{\frac{1}{N}} - 1)$ [64] for uniprocessor fixed priority scheduling to multicores by a semi-partitioned scheduling algorithm using RM [64] on each core. This result is further extended to generalize various parametric utilization bounds (for example the 100% utilization bound for *harmonic* task systems) to multi cores [49]. Another hybrid approach combining global and partitioned scheduling is *clustered scheduling* [13], which partitions the processor cores into subsets (called a cluster each), and uses global scheduling to schedule the subset of tasks assigned to each cluster. Clustered scheduling suffers less resource waste than partitioned scheduling, and may reduce the context switch penalty than global scheduling on hardware architectures where cores are actually grouped into clusters with closer resource sharing.

Implementation and Evaluation

To evaluate the performance and applicability of different scheduling paradigms in RTOS supporting multicore architectures, LITMUS^{RT} [27], 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. [13]). The run-time overheads include mainly the scheduler latency (typically several tens μs in Linux [119]) and cache-related costs, which depends on the application work space characterization, and can vary between several μs and tens of ms [12, 119]. Their studies indicate that partitioned scheduling and global scheduling have both pros and cons, but partitioned scheduling performs better for hard real-time applications [13]. Clustered scheduling exhibits competitive performance on cluster-based multi-core architectures as it mitigates both the high run-time overhead in global scheduling and the resource waste of fragmentation in partitioned scheduling. Recently, evaluations have also been done with semi-partitioned scheduling algorithms [14], together with the work in [119, 19], indicating that semi-partitioned scheduling is indeed a promising scheduling paradigm for multicore real-time system. The work of [119] shows that on multicore processors equipped with shared caches and high-speed inter-connections, task migration overhead is typically with the same order of magnitude as intra-core context-switch overhead; for example, on an Intel Core-i7 4-cores machine running LINUX, the typical costs for task migration are in the scale of one to two hundred μs for a task with one MB working size.

6.4 Conclusion and Challenges

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

7 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. On the IC level this is still common practice while at the level of distributed systems handling of errors, e.g. 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 systems 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 semiconduc-

tor 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 [22]. 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 finite 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 systems 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 SIL (safety integrity level) classification of the IEC61508 [52]. Using redundancy in space, these reliability requirements can be directly 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 systems design.

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 worst case execution time by the time to correct an error. Given the unbounded statistic error models, however, the time for repetitions cannot be bounded with a guaranteed worst case execution time. This dilemma can, however, be solved in the same way as in the case of redundancy in space, i.e. by introducing a probabilistic threshold. This way, predictability can be re-established in a form that is appropriate to design and verify safety and time critical systems, even in the presence of hardware errors.

7.1 An Example - 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) [23]. CAN connects distributed systems, consisting of an arbitrary number of ECUs 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 [44].

The CAN protocol applies state of the art CRC checks to detect the occurrence of transmission errors. Subsequently a fully automated error signaling mechanisms 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 [102], [103]. 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. Non-affected 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 statistic 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 $R(t) = P(nofailurein[0;t])$. 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 [53].

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 80's, where theory of linear block codes has been applied to derive the residual error probability for CRC's of different length [116], [117]. In [30], 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 in [102]. 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 [26] 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 [75]. Weakly-hard real time constraints for the CAN, i.e., constraints that are allowed to be missed from time to time, have been considered under the aspect of errors in [25]. 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 inter-arrival 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 in [96]. 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 verified according to its own safety requirements. In [96] 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 [97], 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, i.e. 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 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 mechanism 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) [56]. 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 that the voter is a single point of failure [109], 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) [106]. 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 [89]. 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 [93]. Communication systems which might suffer from burst errors must use more powerful ECCs such as Reed-Solomon-Codes [112], 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, i.e., only probabilistic thresholds for the timing behaviour can be given anymore. One example is the previously mentioned retransmission mechanism of CAN. CAN uses cyclic redundancy codes (CRC) to detect errors. Whenever an error is detected, 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. 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 [78].

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 have been proposed. The N-version programming approach [8] 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 [79], which in turn raise new issues concerning their applicability for systems with high reliability requirements [58]. A simplified variant of N-version programming is to execute the same implementation of a function multiple times [84]. This can be realized in a time-multiplexed mode on the same CPU (re-execution) 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, re-execution 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 to subject of 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 is presented in [98]. 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, e.g., 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 [9] replicates only selected tasks and performs regular errors 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 straight forward 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.

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.

8 Conclusions

In this paper, we have surveyed some recent advances regarding techniques for building timing predictable embedded systems. A previous survey [101] examined the then state-of-the-art regarding techniques for building predictable systems, and outlined some directionss ahead. We can now see that interesting developments have occurred along several of them.

In [101], one suggested path was to integrate timing analysis across several design layers. The development of the WCC compiler, and of timing-predictable synchronous languages, are offering a solution to this problem, at least on task-level. Another suggestion was to develop better coordination of shared resources: this is becoming critical with the advent of multicores. Although good solutions for predictable systems on multicore are not yet available, the understanding of the necessary elements towards this goal has increased significantly. But perhaps the main obstacle for building truly predictable systems is that although it is in many respects understood how to build predictable systems, the building blocks for actually realizing them are not available in today's processor platforms.

References

[1] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *Proc. 5th IEEE/ACM Int'l Conf. on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 251–256. ACM, 2007.

[2] David H. Albonesi and Israel Koren. Tradeoffs in the design of single chip multiprocessors. In *Proc. IFIP WG10.3 Working Conf. Parallel Architectures and Compilation Techniques*, PACT '94, pages 25–34. ACM, 1994.

[3] S. Andalam, P.S. Roop, and A. Girault. Predictable multithreading of embedded applications using PRET-C. In *Proc. Int'l Conf. on Formal Methods and Models for Codesign*, MEMOCODE'10, pages 159–168, Grenoble, France, July 2010. IEEE Computer Society.

[4] S. Andalam, P.S. Roop, and A. Girault. Pruning infeasible paths for tight WCRT analysis of synchronous programs. In *Proc. Design Automation and Test in Europe*, DATE'11, pages 204–209, Grenoble, France, March 2011. IEEE.

[5] James H. Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68:157–204, February 2004.

[6] Björn Andersson, Arvind Easwaran, and Jinkyu Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7:4:1–4:4, January 2010.

[7] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. 21st Int'l Conf. on VLSI Design*, VLSID '08, pages 103–110, 2008.

[8] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Trans. Software Engineering*, 11(12):1491–1501, 1985.

[9] Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability Analysis for MPSoCs with Mixed-Critical, Hard Real-Time Constraints. In *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Taipeh, Taiwan, October 2011. ACM.

[10] Jonathan Barre, Christine Rochange, and Pascal Sainrat. A predictable simultaneous multithreading scheme for hard real-time. In *Proc. 21st Int'l Conf. on Architecture of computing systems*, ARCS '08, pages 161–172, 2008.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proc. twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, 1993.

[12] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proc. 7th annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '10, 2010.

[13] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proc. 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 14–24, 2010.

[14] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Is semi-partitioned scheduling practical. In *Proc. 23rd Euromicro Conf. Real-Time Systems*, ECRTS '11, 2011.

[15] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003. Special issue on embedded systems.

[16] C. Berg. PLRU cache domino effects. In *Proc. 6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, WCET '06, 2006.

[17] G. Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[18] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. 28th IEEE Int'l Real-Time Systems Symposium*, RTSS '07, pages 149–160, 2007.

[19] Konstantinos Bletsas and Bjorn Andersson. Implementing slot-based task-splitting multiprocessor scheduling. In *Proc. 6th IEEE Int'l Symp. on Industrial Embedded Systems*, SIES '11, 2011.

[20] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems*, 2008:1–21, 2008.

[21] Hans Börjesson. Incorporating worst case execution time in a commercial c-compiler. Master's thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden, 1996.

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

[23] R.ÃGmbH Bosch. CAN Specification 2.0, 1991.

[24] D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, UK, December 1994. ERA Technology.

[25] I. Broster, G. Bernat, and A. Burns. Weakly hard real-time constraints on controller area network. In *Proceedings of 14th Euromicro Conf. on Real-Time Systems*, pages 134–141, Vienna, Austria, 2002. IEEE Computer Society.

[26] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proc. 23rd Real-Time Systems Symp.*, pages 269–278, Austin, USA, 2002. Probabilistic analysis of CAN with faults, IEEE Computer Society.

[27] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. 27th IEEE Int'l Real-Time Systems Symposium*, RTSS' 06, pages 111–126, 2006.

[28] Antonio Marti Campoy, Isabelle Puaut, Angel Perles Ivars, et al. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *Proc. 17th Euromicro Conf. Real-Time Systems*, ECRTS '05, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[29] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods and Models*. Chapman Hall/CRC, Boca, 2004.

[30] J. Charzinski. Performance of the error detection mechanisms in CAN. In *Proc. 1st Int'l CAN Conf.*, pages 1–20, Mainz, Germany, 1994.

[31] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proc. 13th Int'l Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, 2010.

[32] Derek Chiou, Larry Rudolph, Strinivas Devadas, and Boon S. Ang. Dynamic cache partitioning via columnization. Technical Report 430, Massachusetts Institute of Technology, Cambridge, United States, 1999.

[33] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Co., 1997.

[34] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, pages 36–42, May 2010.

[35] UmaMaheswari C. Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proc. 26th IEEE Int'l Real-Time Systems Symposium*, RTSS'05, pages 330–341, Miami, FL, USA, December 2005.

[36] S K Dhall and C L Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[37] S.A. Edwards and E.A. Lee. The case for the precision timed (PRET) machine. In *Proc. 44th Design Automation Conf.*, DAC'07, pages 264–265, San Diego, USA, June 2007. IEEE.

[38] S.A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP J. on Embedded Systems*, 2007, 2007. Article ID 52651.

[39] Ali El-Haj-Mahmoud, Ahmed S. AL-Zawawi, Aravindh Anantaraman, and Eric Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proc. 2005 Int'l Conf. on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 213–224, 2005.

[40] Jakob Engblom. Worst-case execution time analysis for optimized code. Master's thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden, 1997.

[41] Heiko Falk and Jan C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *Proc. 46th Design Automation Conf.*, DAC '09, pages 732–737, New York, NY, USA, 2009. ACM.

[42] Heiko Falk and Helena Kotthaus. WCET-driven cache-aware code positioning. In *Proc. Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '11, pages 145–154, New York, NY, USA, October 2011. ACM.

[43] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *The Int'l Journal of Time-Critical Computing Systems (Real-Time Systems)*, 46(2):251–300, October 2010.

[44] J. Ferreira, A. Oliveira, P. Fonseca, and JA Fonseca. An experiment to assess bit error rate in CAN. In *Proceedings of 3rd Int'l Workshop of Real-Time Networks (RTN)*, pages 15–18, Catania, Italy, 2004. IEEE Computer Society.

[45] Gernot Gebhard and Sebastian Altmeyer. Optimal task placement to improve cache performance. In *Proc. Int'l Conf. on Embedded Software*, EMSOFT '07, pages 259–268, New York, NY, USA, September 2007. ACM.

[46] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proc. 7th ACM Int'l Conf. on Embedded software*, EMSOFT '09, pages 245–254, 2009.

[47] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Proc. 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 387–397, 2009.

[48] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pages 165–174, Stockholm, Sweden, April 2010.

[49] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Parametric utilization bounds for fixed-priority multiprocessor scheduling. In *Proc. 26th IEEE Int'l Parallel & Distributed Processing Symposium*, IPDPS '12, 2012.

[50] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[51] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proc. 30th IEEE Real-Time Systems Symposium*, RTSS'09, pages 68–77, Washington, DC, USA, December 2009.

[52] IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems (IEC 61508), 2010.

[53] ISO. Road vehicles - Functional safety (ISO 26262), 2011.

[54] Shinpei Kato and Nobuyuki Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proc. 8th ACM Int'l Conf. on Embedded software*, EMSOFT '08, pages 139–148, 2008.

[55] Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *Proc. 13th Euromicro Conf. Real-Time Systems*, ECRTS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[56] R.E. Kuehn. Computer redundancy: design, performance, and future. *IEEE Trans. Reliability*, 18(1):3–11, 1969.

[57] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proc. 21st Euromicro Conf. Real-Time Systems*, ECRTS '09, pages 239–248, 2009.

[58] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proc. IEEE*, 82(1):25–40, 1994.

[59] Sylvain Lauzac, Rami G. Melhem, and Daniel Mossé. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proc. 2008 Euromicro Conf. Real-Time Systems*, ECRTS'98, pages 188–195, Berlin, Germany, June 1998.

[60] G. LeGoff. Using synchronous languages for interlocking. In *Int'l Conf. on Computer Application in Transportation Systems*, 1996.

[61] Xin Li and Reinhard von Hanxleden. Multithreaded reactive programming – the kiel esterel processor. *IEEE Trans. Computers*, 61:337–349, 2012.

[62] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proc. 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 57–67, 2009.

[63] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proc. Compilers, Architectures, and Synthesis of Embedded Systems*, CASES'08, pages 137–146, Atlanta, USA, October 2008. ACM.

[64] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[65] Isaac Liu, Jan Reineke, and Edward A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *Proc. 44th Asilomar Conf. Signals, Systems, and Computers*, pages 2111–2115, November 2010.

[66] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *Proc. 24th Int'l Real-Time Systems Symposium*, RTSS'03, pages 256–264, Cancun, Mexico, December 2003. IEEE Computer Society.

[67] Paul Lokuciejewski, Heiko Falk, and Peter Marwedel. WCET-driven cache-based procedure positioning optimizations. In *Proc. 20th Euromicro Conf. Real-Time Systems*, ECRTS '08, pages 321–330, Washington, DC, USA, July 2008. IEEE Computer Society.

[68] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–21, 1999.

[69] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 12–21, December 1999.

[70] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proc. 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, 2010.

[71] Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Proc. Conf. Design, Automation and Test in Europe*, DATE'09, pages 93–98, Nice, France, 2009. IEEE.

[72] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *Proc. 26th Int'l Conf. on Computer Design*, ICCD '08, pages 371–376, 2008.

[73] A. M. Molnos, M. J. M. Heijligers, S. D. Cotofana, and J. T. J. van Eijndhoven. Cache partitioning options for compositional multimedia applications. In *Proc. 15th Annual Workshop on Circuits, Systems and Signal Processing*, ProRISC '04, pages 86–90, November 2004.

[74] Frank Mueller. Compiler support for software-based cache partitioning. In *Proc. Workshop on Languages, Compilers and Tools for Real-Time Systems*, LCTES '95, pages 125–133, New York, NY, USA, June 1995. ACM.

[75] N. Navet, Y.Q. Song, and F. Simonot. Worst-case Deadline Failure Probability in Real-Time Applications Distributed over CAN (Controller Area Network). *Journal of System Architectures*, 46:606–617, 2000.

[76] M. Paolieri, E. Quinones, F.J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90, 2009.

[77] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proc. 36th annual international symposium on Computer architecture*, ISCA '09, pages 57–68, 2009.

[78] D. Paret and R. Riesco. *Multiplexed networks for embedded systems: CAN, LIN, Flexray, Safe-by-Wire...* John Wiley & Sons Inc, 2007.

[79] B. Parhami. Voting algorithms. *IEEE Trans. Reliability*, 43(4):617–629, 1994.

[80] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proc. Conf. Design, Automation and Test in Europe*, DATE '10, pages 741–746, 2010.

[81] Sascha Plazar, Heiko Falk, Jan C. Kleinsorge, and Peter Marwedel. WCET-aware static locking of instruction caches. In *Proc. Int'l Symp. on Code Generation and Optimization*, CGO '12, New York, NY, USA, April 2012. ACM.

[82] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. Wcet-aware software based cache partitioning for multi-task real-time systems. In *Proc. 9th Int'l Workshop on Worst-Case Execution Time Analysis*, WCET '09, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany.

[83] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[84] L.L. Pullum. *Software fault tolerance techniques and implementation*. Artech House Publishers, 2001.

[85] Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. Reports of SFB/TR 14 AVACS 36, SFB/TR 14 AVACS, March 2008. ISSN: 1860-9821, http://www.avacs.org.

[86] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proc. seventh IEEE/ACM/IFIP Int'l Conf. on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 99–108, 2011.

[87] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proc. 6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, WCET '06, 2006.

[88] Christine Rochange and Pascal Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. 2nd conference on Computing frontiers*, CF '05, pages 307–314, 2005.

[89] K. Rokas, Y. Makris, and D. Gizopoulos. Low cost convolutional code based concurrent error detection in FSMs. In *Proc. 18th IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems, 2003*, pages 344–351. IEEE, 2003.

[90] P.S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis of synchronous C programs. In *Proc. Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES'09, Grenoble, France, October 2009. ACM.

[91] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. 28th IEEE Int'l Real-Time Systems Symposium*, RTSS '07, pages 49–60, 2007.

[92] Zoran A. Salcic, Partha S. Roop, Morteza Biglari-Abhari, and Abbas Bigdeli. REFLIX: A processor core for reactive embedded applications. In *Proc. 12th Int'l Conf. on Filed Programmable Logic and Applications*, volume 2438 of *FPL'02*, pages 945–945, Montpellier, France, September 2002. Springer.

[93] A.M. Saleh, J.J. Serrano, and J.H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. Reliability*, 39(1):114–122, 1990.

[94] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proc. Conf. Design, Automation and Test in Europe*, DATE '10, pages 759–764, 2010.

[95] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[96] M. Sebastian and R. Ernst. Reliability Analysis of Single Bus Communication with Real-Time Requirements. In *Proc. 2009 15th IEEE Pacific Rim Int'l Symp. on Dependable Computing*, pages 3–10, Shanghai, China, 2009. IEEE Computer Society.

[97] Maurice Sebastian, Philip Axer, and Rolf Ernst. Utilizing Hidden Markov Models for Formal Reliability Analysis of Real-Time Communication Systems with Errors. In *17th IEEE Pacific Rim Int'l Symp. on Dependable Computing*, Pasadena, USA, 2011. IEEE Computer Society.

[98] J.C. Smolens, B.T. Gold, J. Kim, B. Falsafi, J.C. Hoe, and A.G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Int'l Conf. Architectural Support for Progr. Lang. and Operating Systems*, pages 224–234, Boston, USA, 2004. ACM, ACM.

[99] J. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1990.

[100] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, et al. WCET centric data allocation to scratchpad memory. In *Proc. 26th IEEE Real-time Systems Symposium*, RTSS '05, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[101] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

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

[103] K. Tindell, A. Burns, and A.J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[104] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proc. Conf. Design, Automation and Test in Europe*, DATE'11, pages 563–566, Grenoble, France, March 2011. IEEE.

[105] T. Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99, 2010.

[106] J.H. Van Lint. *Introduction to coding theory*, volume 86. Springer Verlag, 1999.

[107] Reinhard von Hanxleden. SyncCharts in C—a proposal for light-weight, deterministic concurrency. In *Proc. Int'l Conf. on Embedded Software*, EMSOFT'09, pages 225–234, Grenoble, France, March 2009. IEEE.

[108] Reinhard von Hanxleden, Xin Li, Partha Roop, Zoran Salcic, and Li Hsien Yoong. Reactive processing for reactive systems. *ERCIM News*, 67:28–29, October 2006.

[109] J.F. Wakerly. Microcomputer reliability improvement using triple-modular redundancy. *Proceedings of the IEEE*, 64(6):889–895, 1976.

[110] WCET-aware compilation. http://ls12-www.cs.tu-dortmund.de/research/activities/wcc, January 2012.

[111] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. Design Automation and Test in Europe*, DATE '05, pages 600–605, Washington, DC, USA, March 2005. IEEE Computer Society.

[112] S.B. Wicker and V.K. Bhargava. *Reed-Solomon codes and their applications*. Wiley-IEEE Press, 1999.

[113] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[114] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – verview of methods and survey of tools. *ACM Transaction in Embedded Computing Systems*, 7:36:1–36:53, May 2008.

[115] Reinhard Wilhelm, Christian Ferdinand, Christoph Cullmann, Daniel Grund, Jan Reineke, and Benoît Triquet. Designing predictable multicore architectures for avionics and automotive systems. In *Workshop on Reconciling Performance with Predictability (RePP)*, October 2009.

[116] J. K. Wolf, A. Michelson, and A. Levesque. On the probability of undetected error for linear block codes. *IEEE Trans. Communications*, 30(2):317 – 325, feb 1982.

[117] J.K. Wolf and II Blakeney, R.D. An exact evaluation of the probability of undetected error for certain shortened binary CRC codes. In *Military Communications Conference (MILCOM 88) '21st Century Military Communications - What's Possible?'*, pages 287 –292, San Diego, USA, October 1988. IEEE.

[118] Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. Starpro — a new multithreaded direct execution platform for esterel. *Electron. Notes Theor. Comput. Sci.*, 238:37–55, June 2009.

[119] Yi Zhang, Nan Guan, Yanbin Xiao, and Wang Yi. Implementation and empirical comparison of partitioning-based multi-core scheduling. In *Proc. 6th IEEE Int'l Symp. on Industrial Embedded Systems*, SIES'11, pages 248–255, Västerås, Sweden, June 2011.

[120] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by optimizing worst-case paths. In *Proc. 11th IEEE Real Time on Embedded Technology and Applications Symposium*, RTAS '05, pages 138–147, Washington, DC, USA, 2005. IEEE Computer Society.

[121] Wankang Zhao, David Whalley, Christopher Healy, et al. Improving WCET by applying a WC code-positioning optimization. *ACM Trans. on Architecture and Code Optimization*, 2(4):335–365, December 2005.