A Template for Predictability Definitions with Supporting Evidence*

Daniel Grund[1], Jan Reineke[2], and Reinhard Wilhelm[1]

1 **Saarland University, Saarbrücken, Germany.** grund@cs.uni-saarland.de
2 **University of California, Berkeley, USA.** reineke@eecs.berkeley.edu

Abstract

In real-time systems, timing behavior is as important as functional behavior. Modern architectures turn verification of timing aspects into a nightmare, due to their "unpredictability". Recently, various efforts have been undertaken to engineer more predictable architectures. Such efforts should be based on a clear understanding of predictability. We discuss key aspects of and propose a template for predictability definitions. To investigate the utility of our proposal, we examine above efforts and try to cast them as instances of our template.

Digital Object Identifier 10.4230/OASIcs.PPES.2011.22

1 Introduction

Predictability resounds throughout the embedded systems community, particularly throughout the real-time community, and has lately even made it into the Communications of the ACM [12]. The need for predictability was recognized early [25] and has since been inspected in several ways, e.g. [3, 26, 10]. Ongoing projects in point try to "reconcile efficiency and predictability" (Predator[1]), to "reintroduce timing predictability and repeatability" by extending instruction-set architectures (ISA) with control over execution time (PRET [7, 13]), or "guarantee the analyzability and predictability regarding timing" (MERASA [27]).

The common tenor of these projects and publications is that past developments in system and computer architecture design are ill-suited for the domain of real-time embedded systems. It is argued that if these trends continue, future systems will become more and more unpredictable; up to the point where sound analysis becomes infeasible — at least in its current form. Hence, research in this area can be divided into two strands: On the one hand there is the development of ever better analyses to keep up with these developments. On the other hand there is the exercise of influence on system design in order to avert the worst problems in future designs.

We do *not* want to dispute the value of these two lines of research. Far from it. However, we argue that both are often built on sand: Without a better understanding of "predictability", the first line of research might try to develop analyses for inherently unpredictable systems, and the second line of research might simplify or redesign architectural components that are in fact perfectly predictable. To the best of our knowledge there is no agreement — in the form of a formal definition — what the notion "predictability" should mean. Instead the criteria for predictability are *based on intuition* and arguments are made on a *case-by-case basis*. In the analysis of worst-case execution times (WCET) for instance, simple

* The research leading to these results has received funding from or was supported by the European Commission's Seventh Framework Programme FP7/2007-2013 under grant agreement no 216008 (Predator) and by the High-Confidence Design for Distributed Embedded Systems (HCDDES) Multidisciplinary University Research Initiative (MURI) (#FA9550-06-0312).

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

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 22–31

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in-order pipelines like the ARM7 are deemed more predictable than complex out-of-order pipelines as found in the PowerPC755. Likewise static branch prediction is said to be more predictable than dynamic branch prediction. Other examples are TDMA vs. FCFS arbitration and static vs. dynamic preemptive scheduling.

The agenda of this article is to stimulate the discussion about predictability with the long-term goal of arriving at a definition of predictability. In the next section we present key aspects of predictability and therefrom derive a template for predictability definitions. In Section 3 we consider work of the last years on improving the predictability of systems and try to cast the intuitions about predictability found in these works in terms of this template. We close this section by discussing the conclusions from this exercise with an emphasis on commonalities and differences between our intuition and that of others.

2 Key Aspects of Predictability

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

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

Consequently, a system is predictable if one can foretell facts about its future, i.e. determine interesting things about its behavior. In general, the behaviors of such a system can be described by a possibly infinite set of execution traces (sequences of states and transitions). However, a prediction will usually refer to derived properties of such traces, e.g. their length or a number of interesting events on a trace. While some properties of a system might be predictable, others might not. Hence, the first aspect of predictability is the *property to be predicted*.

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

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

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

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

▶ **Proposition 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.*

Refinements

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

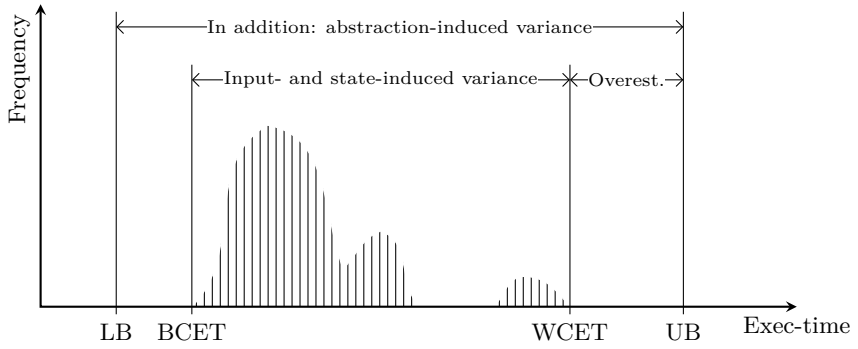

Figure 1 Distribution of execution times ranging from best-case to worst-case execution time (BCET/WCET). Sound but incomplete analyses can derive lower and upper bounds (LB, UB).

- For instance, one could refine Proposition 1 by taking into account the complexity/cost of the analysis that determines the property. However, the clause "by *any* analysis not more expensive than X" complicates matters: The key aspect of inherence requires a quantification over all analyses of a certain complexity/cost.
- Another refinement would be to consider different sources of uncertainty separately to capture only the influence of one source. We will have an example of this later.
- One could also distinguish the extent of uncertainty. E.g. is the program input completely unknown or is partial information available?
- It is desirable that the predictability of a system can be determined automatically, i.e. computed.
- It is also desirable that predictability of a system is characterized in a compositional way. This way, the predictability of a composed system could be determined by a composition of the predictabilities of its components.

2.1 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

- property to be predicted,
- sources of uncertainty, and
- quality measure.

In Section 3 we consider work of the last years on improving the predictability of systems. We then try to cast the possibly even unstated intuitions about predictability in these works in terms of this template. But first, we consider one instance of predictability in more detail to illustrate this idea.

2.2 An Illustrative Instance: Timing Predictability

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

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

- The sources of uncertainty are the *program input* and the *hardware state* in which execution begins. Figure 1 illustrates the situation and displays important notions. Typically, the initial hardware state is completely unknown, i.e. the prediction should be valid for all possible initial hardware states. Additionally, schedulability analysis cannot handle a characterization of execution times in the form of a function depending on inputs. Hence, the prediction should also hold for all admissible program inputs.
- Usually, schedulability analysis requires a characterization of execution times in the form bounds on the execution time. Hence, a reasonable quality measure is the quotient of BCET over WCET; the smaller the difference the better.
- The inherence property is satisfied as BCET and WCET are inherent to the system.

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

▶ **Definition 2.** 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 3** (Timing predictability)**.** Given uncertainty about the initial hardware state $Q \subseteq \mathcal{Q}$ and uncertainty about the program input $I \subseteq \mathcal{I}$, the timing predictability of a program p is

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

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

Refinements

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

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

▶ **Definition 4** (State-induced timing predictability)**.**

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

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

▶ **Definition 5** (Input-induced timing predictability)**.**

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

Example for state-induced timing unpredictability

A system exhibits a *domino effect* [14] if there are two hardware states q_1, q_2 such that the difference in execution time of the same program starting in q_1 respectively q_2 may be arbitrarily high, i.e. cannot be bounded by a *constant*. For instance, the iterations of a program loop never converge to the same hardware state and the difference in execution time increases in each iteration.

In [22] Schneider describes a domino effect in the pipeline of the PowerPC 755. It involves the two asymmetrical integer execution units, a greedy instruction dispatcher, and an instruction sequence with read-after-write dependencies.

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

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

3 Supporting Evidence?

In recent years, significant efforts have been undertaken to design more predictable architectural components. As we mentioned in the introduction, these efforts are usually based on sensible, yet informal intuitions of what makes a system predictable. In this section, we try to cast these intuitions as instances of the predictability template introduced in Section 2.1.

We summarize our findings about how existing efforts fit into our predictability template in Tables 1 and 2. For each approach we determine the *property* it is concerned with, e.g. execution time, the *source of uncertainty* that makes this property unpredictable, e.g. uncertainty about program inputs, and the *quality measure* that the approach tries to improve, e.g. the variation in execution time. Whenever the goals that are explicitly stated in the referenced papers do not fit into this scheme, we determine whether the approach can still be explained within the scheme. In that case, we provide appropriate characterizations in parentheses. In the following sections, we supplement the two tables with brief descriptions of the approaches.

3.1 Branch Prediction

Bodin and Puaut [5] and Burguière and Rochange [6] propose WCET-oriented static branch prediction schemes. Bodin and Puaut specifically try to minimize the number of branch mispredictions along the worst-case execution path, thereby minimizing the WCET. Using static branch prediction rather than dynamic prediction is motivated by the difficulty in modeling complex dynamic schemes and by the incurred analysis complexity during WCET estimation. The approaches are evaluated by comparing WCET estimates for the generated static predictions with WCET estimates for the dynamic scheme, based on conservative approximations of the number of mispredictions.

Table 1 Part I of constructive approaches to predictability.

Approach	Hardware unit(s)	Property	Source of uncertainty	Quality measure
WCET-oriented static branch prediction [5, 6]	Branch predictor	Number of branch mispredictions	Analysis imprecision (Uncertainty about initial predictor state)	Statically computed bound (Variability in mispredictions)
Time-predictable execution mode for superscalar pipelines [21]	Superscalar out-of-order pipeline	Execution time of basic blocks	Analysis imprecision (Uncertainty about the pipeline state at basic block boundaries)	Qualitative: analysis practically feasible (Variability in execution times of basic blocks)
Time-predictable Simultaneous Multithreading [2, 16]	SMT processor	Execution time of tasks in real-time thread	Uncertainty about execution context, i.e., other tasks executing in non-real-time threads	Variability in execution times
CoMPSoC: a template for composable and predictable multiprocessor system on chips [9]	System on chip including network on chip, VLIW cores and SRAM	Memory access and communication latency	Concurrent execution of unknown other applications	Variability in latencies
Precision-Timed Architectures [13]	Thread-interleaved pipeline and scratchpad memories	Execution time	Uncertainty about initial state and execution context	Variability in execution times
Predictable out-of-order execution using virtual traces [28]	Superscalar out-of-order pipeline and scratchpad memories	Execution time of program paths	State of features such as caches, branch predictors, etc. and input values of variable latency instructions	Variability in execution times
Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems [29]	Pipeline, memory hierarchy, and buses	Execution time, memory access latencies, latencies of bus transfers	Uncertainty about the pipeline state, the cache state, and about concurrently executing applications	Variability in execution times and memory access latencies

3.2 Pipelining and Multithreading

Rochange and Sainrat [21] propose a time-predictable execution mode for superscalar pipelines. They simplify WCET analysis by regulating instruction flow of the pipeline at the beginning of each basic block. This removes all timing dependencies within the pipeline between basic blocks. Thereby it reduces the complexity of WCET analysis, as it can be performed on each basic block in isolation. Still, caches have to be accounted for globally. 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."

Barre et al. [2] and Mische et al. [16] propose modifications to simultaneous multithreading (SMT) architectures. They 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.

3.3 Comprehensive Approaches

Hansson et al. [9] propose CoMPSoC, a template for multiprocessors with predictable and composable timing. By predictability they refer to the ability to determine lower bounds on performance. By composability they mean that the composition of applications on one platform does not have any influence on their timing behavior. Predictability is achieved by VLIW cores and no use of caches or DRAM. Composability is achieved by TDM arbitration on the network on chip and on accesses to SRAMs.

Lickly et al. [13] present a precision-timed (PRET) architecture that uses a thread-interleaved pipeline and scratchpad memories. The thread-interleaved pipeline provides high overall throughput and constant execution times of instructions in all threads, at the sacrifice of single-thread performance. PRET introduces new instructions into the ISA to provide control over timing at the program level.

Whitham and Audsley [28] refine the approach of Rochange [21]. Any aspect of the pipeline that might introduce variability in timing is either constrained or eliminated: scratchpads are used instead of caches, dynamic branch prediction is eliminated, variable duration instructions are modified to execute a constant number of cycles, exceptions are ignored. Programs are statically partitioned into so-called traces. Within a trace, branches are predicted perfectly. Whenever a trace is entered or left, the pipeline state is reset to eliminate any influence of the past.

Wilhelm et al. [29] give recommendations for future architectures in time-critical embedded systems. Based on the principle *to reduce the interference on shared resource*, they recommend to use caches with LRU replacement, separate instruction and data caches, and so-called *compositional* architectures, such as the ARM7. Such architectures do not have domino effects and exhibit little state-induced variation in execution time.

3.4 Memory Hierarchy

In the context of the Java Optimized Processor, Schoeberl [23] introduces the so-called *method cache*: instead of caching fixed-size memory blocks, the method cache caches entire Java methods. Using the method cache, cache misses may only occur at method calls and returns. Due to caching variable-sized blocks, LRU replacement is infeasible. Metzlaff et al. [15] propose a very similar structure, called *function scratchpad*, which they employ within an SMT processor.

Schoeberl et al. [24] propose dedicated caches for different types of data: methods (instructions), static data, constant, stack data, and heap data. For heap data, they propose a small, fully-associative cache. Often, the addresses of accesses to heap data are difficult, or in case of most memory allocators, impossible to predict statically. In a normal set-associative cache, an access with an unknown address may modify any cache set. In the fully-associative case, knowledge of precise memory addresses for heap data is unnecessary.

Puaut and Decotigny [18] propose to statically lock cache contents to eliminate intra-

Table 2 Part II of constructive approaches to predictability.

Approach	Hardware unit(s)	Property	Source of uncertainty	Quality measure
Method Cache [23, 15]	Memory hierarchy	Memory access time	(Uncertainty about initial cache state)	Simplicity of analysis
Split Caches [24]	Memory hierarchy	Number of data cache hits	Among others, uncertainty about addresses of data accesses	(Percentage of accesses that can be statically classified)
Static Cache Locking [18]	Memory hierarchy	Number of instruction cache hits	Uncertainty about initial cache state and interference due to preempting tasks	Statically computed bound (Variability in number of hits)
Predictable DRAM Controllers [1, 17]	DRAM controller in multi-core system	Latency of DRAM accesses	Occurrence of refreshes and interference by concurrently executing applications	Existence and size of bound on access latency
Predictable DRAM Refreshes [4]	DRAM controller	Latency of DRAM accesses	Occurrence of refreshes	Variability in latencies
Single-path paradigm [19]	Software-based	Execution time	Uncertainty about program inputs	Variability in execution times

task cache interference and inter-task cache interferences (in preemptive systems). They introduce two low-complexity algorithms to statically determine which instructions to lock in the cache. To evaluate their approach, they compare statically guaranteed cache hit rates in unlocked caches with hit rates in locked caches.

Akesson et al. [1] and later Paolieri et al. [17] propose the predictable DRAM controllers *Predator* and *AMC*, respectively. 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 predictable access schemes, which allow to bound the latencies of individual memory requests, and predictable arbitration mechanisms: CCSP in *Predator* and TDM in *AMC*, allow to bound the interference between different clients.

Bhat and Mueller [4] eliminate interferences between DRAM refreshes and memory accesses, so that WCET analysis can be performed without considering refreshes. Standard memory controllers periodically refresh consecutive rows. Their idea is to instead execute these refreshes in bursts and refresh all lines of a DRAM device in a single or few bursts. Such refresh bursts can then be scheduled in periodic tasks and taken into account during schedulability analysis.

3.5 Discussion

The predictability view of most efforts can indeed be cast as instances of the predictability template introduced in Section 2.1. Also, different efforts *do* require different instantiations: Properties found include: execution time, number of branch mispredictions, number of cache misses, DRAM access latency. Sources of uncertainty include: initial {processor|cache|branch predictor} state, but also program inputs, and concurrently executing applications. Most disagreement between the predictability template and the views taken in the analyzed efforts arises at the question of the quality measure: Many approaches use existing static analysis approaches to evaluate the predictability improvement. This *does not* establish that an approach improves predictability. However, as the inherent predictability is often hard to determine, this is still useful. Designers of real-time systems need analysis methods that will provide useful guarantees. So, from a practical point of view, system A *will* be considered more predictable than system B if some analysis for A are more precise than for B. In such cases, further research efforts should clarify whether A is indeed more predictable than B. Overapproximating static analyses provide upper bounds on a system's inherent predictability. Few methods exist so far to bound predictability from below.

4 Related Work

Here we want to discuss related work that tries to capture the essence of predictability or aims at a formal definition.

Bernardes [3] considers a discrete dynamical system (X, f), where X is a metric space and f describes the behavior of the system. Such a system is considered predictable at a point a, if a predicted behavior is sufficiently close to the actual behavior. The actual behavior at a is the sequence $(f^i(a))_{i\in\mathbb{N}}$ and the predicted behavior is a sequence of points in δ-environments, $(a_i)_{i\in\mathbb{N}}$, where $a_i \in B(f(a_{i-1}), \delta)$, and the sequence starts at $a_0 \in B(a, \delta)$.

Stankovic and Ramamritham [25] already posed the question about the meaning of predictability in 1990. The main answers given in this editorial is that "it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made." Hence, it is rather seen as the existence of successful analysis methods than an inherent system property.

Henzinger [10] describes predictability as a form of determinism. Several forms of nondeterminism are discussed. Only one of them influences observable system behavior, and thereby qualifies as a source of uncertainty in our sense. There is also a short discussion how to deal with such nondeterminism: Either avoid it by building systems bottom-up using only deterministic components or achieve top-level determinism by hiding lower-level nondeterminism by a deterministic abstraction layer. [25] discusses a similar approach.

Thiele and Wilhelm [26] describe threats to timing predictability of systems, and proposes design principles that support timing predictability. Timing predictability is measured as difference between the worst (best) case execution time and the upper (lower) bound as determined by an analysis.

In a precursor of this article, Grund [8] also attempts to formally capture predictability. It is argued, as opposed to almost all prior attempts, that predictability should be an inherent system property.

Kirner and Puschner [11] describe time-predictability as the ability to calculate the duration of actions and explicitly includes the availability of efficient calculation techniques. Furthermore, a "holistic definition of time-predictability" is given. It combines the predictability of timing, as given in [8] and in Equation 1; and the predictability of the worst-case timing, as given in [26].

[20] does not aim at a general definition of predictability. Instead the predictability of caches, in particular replacement policies, is considered. Two metrics are defined that indicate how quickly uncertainty, which prevents the classification of hits respectively misses, can be eliminated. As these metrics mark a limit on the precision that *any* cache analysis can achieve, they are inherent system properties.

5 Summary and Future Work

The most severe disagreement between our opinion on predictability and those of others concerns the inherence property. We think that the aspect of inherence is indispensable to predictability: Basing the predictability *of a system* on the result of *some* analysis of the system is like stating that sorting takes exponential time only because nobody has found a polynomial algorithm yet!

Modern computer architectures are so complex that arguing about properties of their timing behavior as a whole is extremely difficult. We are in search of compositional notions of predictability, which would allow us to derive the predictability of such an architecture from that of its pipeline, branch predictor, memory hierarchy, and other components. Future work should also investigate the relation of predictability to other properties such as robustness, composability and compositionality.

References

1 B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *CODES+ISSS '07*, pages 251–256, 2007.

2 J. Barre, C. Rochange, and P. Sainrat. A predictable simultaneous multithreading scheme for hard real-time. In *Architecture of computing systems '08*, pages 161–172, 2008.

3 N. C. Bernardes, Jr. On the predictability of discrete dynamical systems. *Proc. of the American Math. Soc.*, 130(7):1983–1992, 2001.

4 B. Bhat and F. Mueller. Making DRAM refresh predictable. In *ECRTS '10*, 2010.

5 F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. In *ECRTS '05*, pages 33–40, 2005.

6 C. Burguiere, C. Rochange, and P. Sainrat. A case for static branch prediction in real-time systems. In *RTCSA '05*, pages 33–38, 2005.

7 S. Edwards and E. Lee. The case for the precision timed (PRET) machine. In *DAC '07*, pages 264–265, 2007.

8 D. Grund. Towards a formal definition of timing predictability. Presentation at RePP 2009 workshop. `http://rw4.cs.uni-saarland.de/~grund/talks/repp09-preddef.pdf`.

9 A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

10 T. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philos. Trans. Royal Soc.: Math., Phys. and Engin. Sciences*, 366(1881):3727–3736, 2008.

11 R. Kirner and P. Puschner. Time-predictable computing. In *SEUS '11*, volume 6399 of *LNCS*, pages 23–34, 2011.

12 Edward Lee. Computing needs time. *Comm. of the ACM*, 52(5):70–79, 2009.

13 B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *CASES '08*, pages 137–146, 2008.

14 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '09*, pages 12–21, 1999.

15 S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *MEDEA '08*, pages 38–45, 2008.

16 J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *ICCD '08*, pages 371–376, 2008.

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

18 I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02*, page 114, 2002.

19 P. Puschner and A. Burns. Writing temporally predictable code. In *WORDS '02*, page 85, 2002.

20 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

21 C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Computing Frontiers '05*, pages 307–314, 2005.

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

23 M. Schoeberl. A time predictable instruction cache for a Java processor. In *JTRES '04*, pages 371–382, 2004.

24 M. Schoeberl, W. Puffitsch, and B. Huber. Towards time-predictable data caches for chip-multiprocessors. In *SEUS '09*, pages 180–191, 2009.

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

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

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

28 J. Whitham and N. Audsley. Predictable out-of-order execution using virtual traces. In *RTSS '08*, pages 445–455, 2008.

29 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. *Trans. on CAD of Integrated Circuits and Syst.*, 28(7):966–978, 2009.