ON COMPOSABLE SYSTEM TIMING, TASK TIMING, AND WCET ANALYSIS[1]

Peter Puschner[2] and Martin Schoeberl[3]

Abstract
The complexity of hardware and software architectures used in today's embedded systems make a hierarchical, composable timing analysis impossible. This paper describes the source of this complexity in terms of mechanisms and side effects that determine variations in the timing of single tasks and entire applications. Based on these observations, the paper proposes strategies to reduce the complexity. It shows the positive effects of these strategies on the timing of tasks and on WCET analysis.

1. Introduction

Until the early 90s of the 20th century the computer architectures used in embedded hard real-time systems were relatively simple. In the light of these simple architectures, hierarchical (composable) timing models – models that separated the low-level task timing issues from the real-time scheduling problem at the high level – had been conceived. Worst-case execution-time analysis (WCET analysis) had started to become an independent field of investigation within real-time systems research.

Since then, researchers working on WCET analysis have developed methods to identify (in)feasible paths through pieces of code and strategies to compute WCET estimates for code running on different hardware architectures. Over the years, the results of WCET analysis allowed its researchers to compute WCET estimates for code running on more and more complex computer architectures.

While new computer hardware had entered the stage and advances in WCET analysis were made to deal with these changes (e.g., to model the effects of instruction pipelines and caches on task timing), the overall timing models still remained unchanged. I.e., although the temporal (de)composability of task execution times had been lost, real-time schedulers and schedulability analysis still use the strategies that had been conceived at a time when task execution times were independent. The only available measure to deal with the lack of composability in task timing is the addition of an extra analysis step. This analysis uses information including the periods, priorities, and the physical-memory maps of tasks to make a pessimistic assessment of the worst-case effects of the timing interactions between tasks, see, e.g., [14].

Tools that assess the worst-case side effects of task executions on overall system timing are valuable at the moment. In the long run, however, we will have to get rid of the side effects of tasks on system timing instead of analyzing these effects. Only the elimination of the side effects provides the basis for a development and analysis process that is hierarchical and thus much less complex than what is currently state of the art. It is therefore the purpose of this paper to identify the properties of current

[1]The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 214373 (ARTISTDesign).
[2]Vienna University of Technology, A1040 Vienna, Austria; email: peter@vmars.tuwien.ac.at
[3]Vienna University of Technology, A1040 Vienna, Austria; email: mschoebe@mail.tuwien.ac.at

task-execution models that adversely affect the decomposability of tasks and make timing analysis difficult. Based on these findings we introduce a restrictive hardware and software architecture that allows us to return to a hierarchical timing analysis of manageable complexity. The discussion will highlight the consequences of this proposal on task design, on task timing characteristics, and – as a consequence – on WCET analysis.

2. Hierarchic Timing Analysis

Traditionally the timing analysis of hard real-time systems, comprising the CPU-time scheduling or schedulability analysis and the worst-case execution-time analysis, is a hierarchical process. At the lower level, tasks are analyzed for their worst-case execution time (WCET). The results of this low-level analysis are used either for constructing a CPU schedule that meets the timing constraints of the application or for the purpose of schedulability analysis – the latter ensures that a task set characterized by its task parameters (including execution times) will indeed be scheduled correctly.

A hierarchic decomposition as sketched above reduces the complexity of the timing analysis and the real-time systems engineering process. According to [13], such a decomposition requires that the subsystems are nearly decomposable, i.e., *the interactions among the subsystems are weak but not negligible* [13].

2.1. A Model of Simple Tasks

In this work, our main interest is to investigate phenomena related to the time consumption of tasks. For this purpose we assume that all tasks are *S-tasks* [6], simple tasks that do not have any synchronization points inside. As a precondition to the execution of each task instance we assume that all inputs for the task are available. During task execution there is no I/O or other blocking. A simple task produces outputs by writing to defined locations within its local memory – the availability of the outputs is part of the postcondition of each task execution. Outputs are read and further processed (copied or transmitted in a message) by the operating system after the completion of the task.

For the sake of simplicity we assume that tasks are *stateless*, i.e., they do not preserve any data values between invocations. Note that tasks that need a state can be converted to stateless tasks by making their state an input/output variable. Each instance of such a converted task reads this input/output variable after its start and writes its contents back as an output before it terminates. So the next instance can read in the "state" again, and so on.

The interface between a task and its environment (the other tasks and the physical world) is characterized by its control properties, temporal constraints, the functional intent and its data properties [6]. Tasks communicate with the environment with the help of the operating system, that reads resp. writes the interface (input/output) variables of the tasks. One can observe that the interactions between the tasks of a computer system influence the execution times of the tasks. Some interactions (e.g., the exchange of data) are due to phenomena that are observable parts of the task interface. Other interactions (e.g., the modification of the contents of a shared cache which, in turn, influences the execution time of other tasks) cannot be traced back to the task interface. The latter are called side effects.

During the execution of a task instance, the values of the interface variables of the task are not defined. Therefore it is not safe to access these variables from outside the task while the task is in progress. The termination of the task commits the outputs. The outputs can then be propagated and processed

by the task environment.

2.2. Simple and Complex Computer Architectures

In *simple computer architectures* [6], the timing of each instruction can be determined locally, from the knowledge of the instruction, the operands of the instruction, and a small execution context of the instruction (often consisting only of the instruction itself). In these architectures, the execution of an instruction can influence the execution time of another instruction in another task only via the explicit change of values in a data memory that it shares with one or more other tasks. There are, however, no implicit effects (side effects) of task behaviour on the execution times of other tasks. This way we have clearly defined interactions between subsystems (tasks) that can be taken into account by the scheduler, resp. schedulability analysis at the system level. Timing interactions between tasks are limited to the effects of the data exchanged at the interfaces of the tasks.

In *complex computer architectures* the interactions between the different subsystems (tasks) are in general not weak. This is because interactions are no longer restricted to a limited amount of explicit timing dependencies that are caused by data sharing (including access to physically shared memory as well as message communication) via the interfaces of the tasks. These architectures are characterized by a number of additional timing effects due to the competition in the reservation of and access to scarce, shared system resources. Depending on the particular computer system architecture shared resources include pipelines for processing instructions and loading data, instruction and data cache memories, and fast on-processor caches or registers for speculative branch and trace prediction. Besides these mechanisms, the sharing of buses and memories for instructions and data among the processors of chip-multiprocessing systems adds another source for temporal side effects between tasks.

Because complex computer architectures are more and more used in embedded real-time systems, we have to be aware of these implicit side effects that influence execution times. We have to identify and analyze these interactions, and we have to investigate into appropriate ways of eliminating the hidden side effects, thus avoiding that the reasoning about the temporal properties of real-time systems becomes unmanageably complex.

3. Interactions of Task Timing

In the following we investigate the timing interactions between tasks in more detail. We describe both the reasons for the interactions and the resulting phenomena that can be observed.

3.1. Task Interactions in Simple Architectures

In general the execution times of simple tasks vary. Such variations are due to effects of differences in the inputs that are passed to the task via its interface.

Variable, data-dependent execution times of CPU instructions: A number of processors implement (part of) the CPU instructions in micro code where more complex instructions execute repetitive steps over a number of CPU clock cycles. In some cases the number of steps depends on the actual operands which leads to execution-time variations (e.g., shift, multiply, and division).

Consequences for task timing analysis: If the operands of instructions with data-dependent timing

are not exactly known at analysis time, static analysis has to use pessimistic abstractions instead. In measurement-based analyses, variable instruction timing in general increases the number of different execution-time combinations of instructions that have to be compared in search for the worst case. Roughly, every instruction with k different execution times increases the number of different scenarios by a factor k. To deal with this increase in complexity, one either needs to use knowledge about the instruction behaviour when generating input data for measurements or, alternatively, increase the number of test cases significantly to obtain results of sufficient quality.

Different execution paths: In numerous algorithms the conditions of conditional branches that determine the actual instructions executed during an execution directly or indirectly depend on the task inputs. Upon execution, differences in inputs result in different test results in conditionals which, in turn, produce different execution traces or paths, and thus possibly different execution times.

Consequences for task timing analysis: As every conditional branch essentially doubles the number of possible execution paths – an effect that multiply occurs in loops – the number of possible execution paths of a task is usually too high to analyze the timing of all possible paths one by one. As a consequence, static timing analysis uses pessimistic abstractions which, in turn, can lead to overestimations in the result of WCET analysis.

3.2. Task Interactions in Complex Architectures

In addition to the above-mentioned explicit timing interactions between tasks, the following mechanisms of complex computer architectures give rise to temporal side effects.

Intra-task effects on hardware state: Let us at this point consider a single periodic task that is perfectly shielded from external side effects. Still, the different instances of this task, operating on different inputs, may execute on different paths. This leaves the hardware of the computer system (e.g., the instruction cache, branch prediction buffers, etc.) in different states when the task completes. As the hardware state at termination equals the start state of the next instance of the tasks, each execution of a task influences the timing of subsequent instances of the task. Depending on whether the hardware state evolves monotonously and converges to a fixed point after a certain number of executions or not (as in the case of conditional cache conflicts within single task instances), the state effects on execution times may stabilize or not.

Effects on task timing: One observes variations in execution time due to different hardware states at the task start. In particular the first instance of a task cannot benefit from state changes (e.g., the loading of cache lines) of previous executions, thus usually consuming much more time than follow-up instances. Depending on whether there are conflicting state effects within task instances, timing effects between instances may disappear after the state has reached its fixed point or not. Even if a such a fixed point exists, the number of executions needed to reach the fixed point is in general unknown.

Consequences for task timing analysis: Which starting state should be assumed? What is the "worst-case starting state", i.e., the starting state from which an execution of maximum duration starts? Do we really want to consider the worst-case starting state in the execution time analysis given that the cost for building up the state in the first instance of a task is usually much higher than the state-dependent cost of successive executions, or shall we discern between the first/first N – what N? – and all other executions that follow?

In static WCET analysis, the pessimistic approximation of possible start states leads to pessimism in the computed WCET bounds. In measurement-based analyses, intra-task state effects increase the space of parameters that are relevant for execution times, which has a negative effect on the percentage of possible situations that can be assessed with a given set of resources.

External hardware state modifications **between invocations** ***(no preemption):*** So far we assumed that the hardware state of a task is not modified between successive executions. In real-world scenarios, however, other tasks and operating system activities alter the state left by the task, and thus the starting state of further instances.

Effects on task timing: The effects on timing are not limited to the intra-task effects mentioned above. Although state changes from prior executions of the task may still be effective, other tasks and operating system activities interfere with the task state, i.e., there are task-external influences on the task execution times as well.

Consequences for timing analysis: As above, there is the question about which start states are relevant for WCET analysis, resp. which abstractions should be used for the analysis. In contrast to the previous discussion, timing effects are not local to the task, however. Therefore, besides the WCET analysis one needs some extra, global analysis to account for task interferences between task executions and their effects on the overall timing of the real-time computer system. A question in this context is: What are useful abstractions for each of these analysis steps in order to achieve a clean separation between WCET analysis and the timing analysis that accounts for the interferences?

External modification of state **during execution** ***(preemption):*** In systems with preemptive scheduling, the situation gets even more complex as preemptions may have almost arbitrary effects on a the state of a task during its execution.

Effects on task timing: The effects of preemptions on the state of a task depend on a number of factors, e.g., the number of preemptions, the state of the task at preemption time, the state modifications performed by the preempting code.

Consequences for timing analysis: As above, in addition possible interferences during task preemptions have to be considered, which again adds complexity/pessimism to the analysis. A simple hierarchical timing analysis that decomposes into a low level WCET analysis and a high-level scheduling or schedulability analysis is beyond reach because of the strong interactions between the two levels.

Dynamic state-sensitive resource allocation and scheduling: Actual out-of-order processors perform speculative execution even over predicted branches where the branch outcome is not yet known. As a consequence around 100 instructions[2] can be in the pipeline on the fly between instruction fetch and instruction retirement.

Effects on task timing: The execution time of a single instruction depends on a very large execution history. Assuming a flushed pipeline on a basic block start is not an option anymore.

Consequences for timing analysis: Modeling the state of about 100 instructions per clock cycle and the speculative execution will result in a state space explosion. The situation can get worse in the

[2]On a Pentium 4 the minimum latency of an instruction between fetch and retire is 31 clock cycles and up to 3 instructions can be fetched each cycle [1]. Register renaming restricts the number of micro operation in execution to 128.

presence of so-called timing anomalies, i.e., when the dynamic scheduling of instructions can lead to non-monotonic timing relationships between instruction sequences and the constituents (parts) of these sequences. The latter poses a major obstacle to a safe compositional timing analysis.

3.3. Task Interactions in Chip-Multiprocessors

Due to the ever increasing transistor budget [8] for a chip several functions (or IP cores) can now be integrated into a single chip. This integration is often referred to as System-on-a-Chip (SoC). SoC is categorized into two types:

Heterogenous multiprocessors contain a number of different IP cores (e.g., general purpose processor, DSP co-processors, small memory units) on a single chip. The cores are connected either by point-to-point links or by a network on chip (NoC). Those systems are called multi-processor SoC (MPSoC).

Homogenous multiprocessors contain several identical processors with some on-chip memory. Those systems are also referred as chip multiprocessors (CMP).

Both architectures are common in embedded systems. Due to the power wall [1] CMP systems are now also state-of-the-art in desktop and server processors. In this paper we consider CMP systems and the impact of the shared memory on the timing analysis. Three, quite different CMP architectures are state-of-the-art: (1) multicore versions of super-scalar architectures (Intel/AMD), (2) multicore chips with simple RISC processors (Sun Niagara), and (3) the CELL architecture.

Most cores for CMP allow fine-grain multithreading within a single core. Multithreading in a core can hide latencies due to cache misses. With simultaneous multithreading (SMT) more than one thread can execute in a single pipeline stage when enough functional units are available. Multithreading increases throughput for server type workloads due to higher processing resource utilization; the individual task execution time increases.

Complex processor CMP: Mainstream desktop processors from Intel and AMD include two or four out-of-order executing processors. Those processors are just replications of the original, complex cores that share a 2nd level cache and the memory bus. Cache coherence protocols on the chip keep the level 1 caches coherent and consistent. Furthermore, those cores also support SMT, sometimes also called hyper-threading.

RISC based CMP: Sun took a completely different approach with their Niagara T1 [5] by abandoning the super-scalar architecture that tries to extract instruction level parallelism (ILP) from sequential code. Eight simple cores implement fine-grain multithreading to support thread level parallelism often found in server workloads. Each core consists of a simple six-stage, single-issue pipeline similar to the original five-stage RISC pipeline. The additional pipeline stage adds fine-grained multithreading. Four threads are supported on each core that are scheduled in round-robin fashion. With 8 cores the Niagara can execute 32 independent threads of execution. When a thread stalls due to a cache miss or a load-use dependency it is skipped in the schedule. The first version of the chip contains just a single FPU that is shared by all 8 processors.

Local memory based CMP: The Cell multiprocessor [2, 3, 4] takes an approach similar to a distributed memory multiprocessor. The Cell contains, beside a PowerPC microprocessor, 8 synergistic proces-

sors (SP). The SPs contain 256 KB on-chip memory instead of a cache. The PowerPC, the 8 SP, and the memory interface are connected via a 4 ring network. Communication between the cores in the network has to be setup explicitly. All memory management, e.g. transfer between SPs or between on-chip memory and main memory, is under program control, resulting in a new programming model.

Simultaneous multithreading: The tight coupling of the CMP cores introduces several timing interactions that are hard to predict. The simplest form of multiprocessing within a single pipeline is introduced by fine-grain or simultaneous multithreading. The hardware managed threads of execution interact in a very fine grain manner: each stall in one thread influences the execution time of the other threads. The best WCET estimates we can provide for hardware multithreading is the same time as executing those threads serially on the same pipeline.

Keeping caches coherent and consistent: Cache coherence protocols (bus snooping or directory based) enforce a coherent and consistent view of the main memory. These protocols exchange the cache information between all cores on each memory access and introduce a high variability of the cache access time even when the access is a cache hit.

Shared caches and memory: Probably the main source of timing interaction comes from the shared 2nd (and probably 3rd) level of cache and the shared main memory. The shared memory provides an easy-to-use programming model at the cost of unpredictable access time to the data. With global multiprocessor scheduling a task can migrate from one core to another – even within a single period of execution. A migrated tasks completely looses its L1 cache state.

4. Avoiding Unwanted Interactions

We have seen that a number of factors contribute to variations in task execution times. Some of the effects are malign as they are not local to a single task execution but invalidate the hardware state that other tasks or other instances of the same task have built up. These interactions cause side effects that obstruct a hierarchical timing analysis.

In this section we propose some ways to eliminate these interactions. The central idea is to protect the time-relevant state of a task from dynamic changes that make it unpredictable. To this end, we aim at the spatial separation of tasks and we replace dynamic run-time decisions by unalterable, pre-planned control mechanisms where all decisions have been taken offline, at implementation time. In detail, our solution builds on the following mechanisms:

- The use of single-path code in all tasks,
- The execution of a single task/thread per core,
- The use of simple in-order pipelines, and
- Statically scheduled access to shared memory in CMPs.

4.1. Use of Single-Path Code

When considering simple architectures, we think that data-dependent instruction execution times can be eliminated easily. In fact there are a number of processors with constant instruction execution

times around. Timing variations due to the execution of different execution paths and the intra-task effects on the hardware state that occur in complex architectures can be eliminated by transforming the code into so-called single-path code [11]. In the single-path transformation [9], all control dependencies in the code are removed. Instead, the input-dependent conditionals are replaced by predicated instructions that have invariable execution times.

Effects on task timing: As single-path code always executes the same trace, there are no execution-time variations due to multiple paths in simple architectures. For single-path task implementations, execution-time variations due to intra-task timing effects are restricted to the warm-up phase of the task. As all executions of a task run the same trace the hardware state stabilizes after a limited number of executions. After this fixed point has been reached, the timing of the task remains constant. To avoid that the single-path conversion yields code with very poor performance, we suggest the use of *WCET-oriented* programming strategies and algorithms [10].

Consequences for timing analysis: The timing analysis of single-path tasks is trivial. On simple architectures it is sufficient to measure the execution of a single task instance, with any input data, to obtain the (single) execution time of the task. For the complex architectures, the execution time of isolated tasks can be measured after a limited number of executions, once the hardware state has stabilized at its fixed point.

4.2. Execution of a Single Task per Core

Both types of external modifications (inter-task effects) of the hardware state of a task – those occurring between invocations and those due to preemptions – can only be eliminated by protecting the hardware state against influences from other tasks. One way to achieve this is saving and restoring the state whenever a task completes or a task gets preempted. As the administrative overhead for this state management seems to be pretty high, we propose a more rigorous shielding of tasks that benefits from the current trends in hardware development – assigning each task/thread to a dedicated core of a chip multiprocessor. As the used simple tasks do not access shared data during their execution, each processor builds up its own private state and a spatial separation of the timing relevant state of the tasks is achieved. The timing impact of accesses to shared data for the purpose of communication and I/O (as performed by the operating system) does, of course, need special consideration. The latter will be discussed below (see Section 4.4).

Effects on task timing and timing analysis: Assigning each task to a dedicated processor core eliminates all of the mentioned inter-task timing effects. This way, task timing analysis only has to consider task-internal effects on the state. This, in turn, can reduce the state space of the execution-time anaysis significantly, thus yielding tighter (static analysis) or safer (measurement-based analysis) results of WCET analysis. In addition to simplifying the task timing analysis, the elimination of task interactions reduces the side effects on task timing, thus allowing for a better composibility in the overall timing-analysis process.

4.3. Simple in-order CMP Pipelines

Extracting ILP from sequential code in one task with speculating out-of-order pipelines consume a lot of resources and is hard to analyze. For time predictable systems the transistor budget for future CMPs is better spent by replication of simple RISC pipelines. The additional available cores can be utilized to shield individual tasks as proposed in the former section. We assume local data and

instruction memory per core either program managed or organized as a cache.

Current trends in computer architecture actually simplify WCET analysis. Besides the Intel/AMD approach, the CMP pipelines are simpler than the former mainstream complex processors. The architectures target at thread-level parallelism instead of ILP – an abstraction that can be better handled in real-time systems.

The pipeline of Sun's Niagara CMP is a simple in-order pipeline where timing anomalies [7] are not an issue anymore. The CELL SP elements are dual-issue SIMD in-order pipelines. The SP contains no cache and no virtual memory. In the CELL processor each data exchange between the cores has to be setup under program control. Therefore, we can apply the time triggered approach of data exchange on the CELL architecture.

Effects on task timing and timing analysis: Single issue, in-order execution pipelines are well understood for WCET analysis. The speedup due to pipelining can be modeled for basic blocks and also for larger constructs, such as loops.

4.4. Statically Scheduled Access to Shared Memory

In a CMP system the competition for shared resources shifts from the CPU to the memory bandwidth. Imagine an extreme CMP system with more CPUs than tasks to execute. In that case there is no competition for the CPU – we even can avoid scheduling at all. However, all CPUs access the single global memory. A shared resource where the access has to be scheduled, e.g., through an arbiter. Even when this example is not practical at the moment, it shows the trend towards integrating the competition for the memory bus into the timing analysis.

We consider a static, preplanned scheduling of the memory access [12] for all cores. The arbitration of the memory access is time sliced. Integrating the knowledge of the access time slices into the WCET analysis provides safe estimates for load/store instructions and instruction cache fills. The time slicing does not have to be regular. We can introduce a relative boost (longer slices) for some cores at the cost of other cores that run tasks with enough slack time.

To avoid hard-to-predict task-migration (from one core to another) costs we pin each task to a dedicated core. If a CPU supports hardware multithreading, only one of the virtual CPUs can be used. The other virtual CPUs need to be disabled.

Consequences for timing and schedulability analysis: The scheduling of the memory access has to be integrated into the WCET analysis. With a static schedule of the memory arbitration the access time property is well known and independent from the activity of the other cores. With few tasks – or even a single task – executing on a core, the traditional scheduling for the CPU resource disappears. Scheduling is performed at the memory access level. The low overhead of a task switch at the memory arbitration allows fine grain access control: either time sliced down to a single memory access or a percentage based bandwidth scheduling are feasible.

5. Summary and Conclusion

In this paper we investigated into the problems of nowadays timing analysis. We showed that, due to the properties of the used hardware and software architectures, tasks cannot be considered indepen-

dent in their execution. As a consequence, task timing is not an isolated property, which makes the analysis of both task timing and system/application timing highly complex.

We analyzed the reasons for the complexity of timing analysis and identified ways to make a hierarchical compositional timing analysis possible. Our solutions utilizes the architectural features of chip multiprocessors that bring along performance and the parallelism we need to reduce the resource competition between tasks.

- For each *single task* we make task timing easier to predict and stable, the latter meaning that each execution of a task has the same execution time. Regarding software, we use the single-path conversion to reduce the number of execution paths (or traces) to one, and WCET-oriented programming to get reasonable performance. On the hardware side we use processors with constant instruction execution times. Further, we rely on in-order pipelines to eliminate the effects of dynamic instruction scheduling, a central source of timing anomalies.
- When considering the *whole task set* of an application, our main goal was to eliminate the inter-task timing effects. By allocating each task to a dedicated CPU core we avoid those timing interferences that are due to the competition for scarce CPU and memory resources. The pre-runtime, offline planning of all accesses to shared memory removes all other interferences, which are due to the – necessary and inevitable – data exchange between tasks.

To summarize, the introduced mechanisms simplify the structure of single tasks and shield different concurrent tasks from one another, both in the spatial and in the temporal domain. The mechanisms both simplify the overall timing analysis – in that they make a hierarchical timing analysis possible – and in parallel simplify the execution characteristics of tasks, thus paving the way back to a simple WCET analysis.

References

[1] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[2] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262, 2005.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *j-IBM-JRD*, 49(4/5):589–604, 2005.

[4] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006.

[5] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multi-threaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[6] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[7] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[9] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[10] Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[11] Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[12] 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 *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*, pages 49–60, Dec. 2007.

[13] Herbert Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 3rd edition, 1996.

[14] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. 17th Euromicro Conference on Real-Time Systems*, pages 41–48, Jul. 2005.