Predictability Considerations in the Design of Multi-Core Embedded Systems*

Christoph Cullmann[1], Christian Ferdinand[1], Gernot Gebhard[1], Daniel Grund[2], Claire Maiza (Burguière)[2], Jan Reineke[3], Benoît Triquet[4], Reinhard Wilhelm[2]

1: AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany

2: Department of Computer Science, Saarland University, Germany

3: Department of EECS, University of California, Berkeley, USA

4: Airbus Operations S.A.S., Toulouse, France

Abstract: Embedded systems with hard real-time constraints need sound timing-analysis methods for proving that these constraints are satisfied. Computer architects have made this task harder by improving average-case performance through the introduction of components such as caches, pipelines, out-of-order execution, and different kinds of speculation. This article argues that some architectural features make timing analysis very hard, if not infeasible, but also shows how smart configuration of existing complex architectures can alleviate this problem.

Keywords: Real-time systems, worst-case execution time, timing analysis, design for predictability

1. Introduction

This paper is concerned with multi-core architectures for embedded control systems with high predictability requirements that are to be used in the automotive and aeronautics industries. Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. Experience with the use of static timing analysis methods and the tools based on them in the automotive and the aeronautics industries is positive. However, both, the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform. In fact, the architecture determines whether a static timing analysis is practically feasible at all and whether the most precise obtainable results are precise enough. The architecture is usually designed to improve average-case performance. The predictability of the performance-enhancing features is not a criterion for such average-case designs. Yet, the dependence on the architectural development is of growing concern to the developers of timing analysis tools and their customers, the developers in industry. The problem reaches a new level of severity with the advent of multi-core architectures in the embedded domain. Based on experience with static timing-analysis of single core architectures in the embedded-systems industry [30, 32], theoretical insights [21, 19, 10], and characteristics of avionics and automotive applications, we give advice concerning future multi-core computer architectures for time-critical systems. Furthermore, as such a predictable multi-core architecture has not yet been implemented, we show how to configure available multi-cores in a way suitable for static timing analysis.

*The research reported herein has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No 216008 (PREDATOR). The findings of this paper represent no commitment from the side of Airbus as to their future architectures.

2. Foundations and Context

2.1 AUTOSAR and IMA

Growing software complexity in the embedded domain has led to the development of standardized frameworks which focus on integrating components, possibly developed by different suppliers, on Electronic Control Units (ECUs). Examples are AUTOSAR in the automotive domain and the IMA architecture in the aeronautics industry. Both IMA and AUTOSAR are claimed to support compositionality and composability; the behavior of a system is determined by the behavior of the system's components and the type of composition (*compositionality*), and the behavior of individual components should not change by the composition (*composability*). For time-critical systems, composability of the timing behavior means that the modification of one component only influences its own timing behavior and not that of other components. This depends on the availability of architectures on which software composition does not lead to unpredictable timing behavior.

Applications in our context are vehicle functions that are mapped to computational units. An application may consist of several *tasks*. Engine control, electronic stability program, flight control and guidance would all be considered applications in this sense.

2.2 Static Timing Analysis

Exact worst-case execution times are impossible or very hard to determine, even for the restricted class of real-time programs with their usual coding rules. Therefore, these guarantees consist of safe and precise upper bounds on the execution times of tasks. The combined requirements for timing analysis methods are:

• *soundness*, to ensure the reliability of the guarantees,

• *efficiency*, to make them useful in industrial practice, and

• *precision of the results*, to increase the chance to prove the satisfaction of the timing requirements.

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are in part responsible for the gap between WCETs and upper bounds and between BCETs and lower bounds. How much is lost depends on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software. The methods used to determine upper bounds and lower bounds are very similar.

In modern microprocessor architectures, caches, pipelines, and all kinds of speculation are key features for improving (average-case) performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions, and thus the contribution to the program's execution time, can vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases:

• The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e., all operands are ready, no resource conflicts with other currently executing instructions exist.

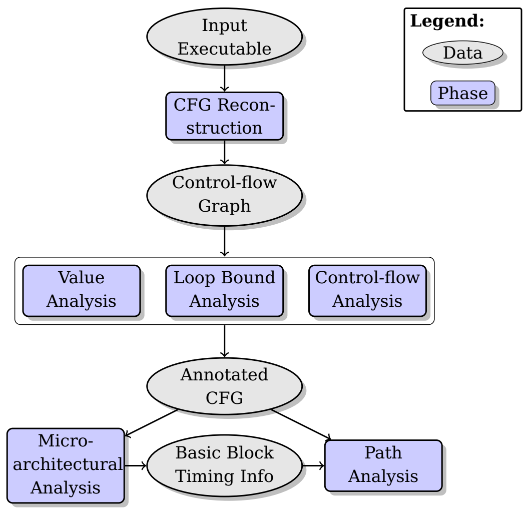

Figure 1: Components of a timing-analysis framework and their interaction.

• "Everything goes wrong", i.e., instruction and/or operand fetches miss the cache, resources needed by the instruction are occupied, etc.

We will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases the *timing penalty* of this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters a timing accident depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history.

3. Single-Core Timing Analysis

3.1 Timing Analysis Framework

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged [12, 28, 8]. Figure 1 gives a general view on this architecture. One can distinguish three major building blocks:

• Control-flow reconstruction and static analyses for control and data flow [27, 25, 6, 11, 26].

• Micro-architectural analysis, which computes lower and upper bounds on execution times of basic blocks [7, 29, 10].

• Path analysis, which computes the shortest and longest execution paths through the whole program.

The commercially available tool aiT by AbsInt GmbH implements this architecture, cf. http://www.absint.de/ait. The tool is employed in the aeronautics and automotive industries. aiT has been successfully used to determine precise bounds on execution times of real-time software [10, 9, 30, 13]. As in this paper we only investigate the influence of the architecture, the following will only be concerned with the micro-architectural analysis phase:

Micro-architectural analysis [7, 29, 10] determines lower and upper bounds of the execution times of basic blocks performing an abstract interpretation of the program execution on the particular architecture, taking into account its pipeline, caches, memory buses, and attached peripheral devices. By means of an abstract model of the hardware architecture, the pipeline analysis simulates the execution of each instruction. The cache analysis provides safe approximations of the contents of the caches at each program point. Complex architectural features are the main challenges for this analysis phase.

3.2 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.

Since most parts of the pipeline state influence timing, current abstract models 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, speculative execution, 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 of the pipeline may depend on such missing 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 most expensive pipeline transition is chosen. However, in the presence of *timing anomalies* [14, 21] this approach is unsound. Thus, in general, the analysis has to follow all possible successor states.

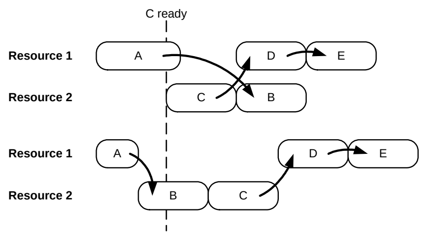

Figure 2: Scheduling anomaly.

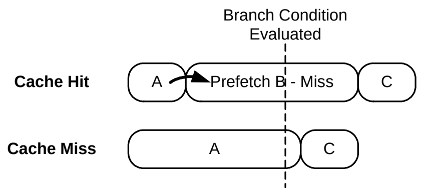

Figure 3: Speculation anomaly. A and B are prefetches. If A hits, B can also be prefetched and might miss the cache.

3.3 Timing Anomalies and Domino Effects

The notion of *timing anomalies* was introduced by Lundqvist and Stenström in [14]. In the context of WCET analysis, [21] presents a formal definition. 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 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, as shown in Figure 3. A mispredicted branch results in unnecessary instruction fetches, which might miss the cache. In case of cache hits the processor may fetch more instructions.

A system exhibits a *domino effect* [14] if there are two hardware states s, t such that the difference in execution time (of the same program starting in s, t

respectively) may be arbitrarily high, i.e. cannot be bounded by a *constant*. E.g., given a program loop, the executions never converge to the same hardware state and the difference in execution time increases in each iteration. The existence of domino effects is undesirable for timing analysis. Otherwise, one could safely discard states during the analysis and make up for it by adding a predetermined constant.

Unfortunately, domino effects show up in real hardware. In [24], Schneider describes a domino effect in the pipeline of the PowerPC 755. Berg [3] provided another example considering the PLRU replacement policy of caches. Section 3.6 discusses a cache domino effect in more detail.

3.4 Classification of Architectures

Architectures can be classified into three categories depending on whether they exhibit timing anomalies or domino effects.

- *Fully timing compositional architectures:* The (abstract model of an) architecture does not exhibit timing anomalies. Hence, the analysis can safely follow local worst-case paths only. One example for this class is the ARM7. Actually, the ARM7 allows for an even simpler timing analysis. On a timing accident all components of the pipeline are stalled until the accident is resolved. Hence, one could perform analyses for different aspects (e.g. cache, bus occupancy) separately and simply add all timing penalties to the best case execution time.
- *Compositional architectures with constant-bounded effects:* These exhibit timing anomalies but no domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant number of cycles to the local worst-case path [20]. 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.

3.5 Caches

Caches are employed to hide the latency gap between memory and CPU by exploiting locality in memory accesses. To reduce traffic and management overhead, main memory is logically partitioned into a set of memory blocks. Memory blocks are cached as a whole in cache lines of equal size. On today's architectures a cache miss may take several hundred CPU cycles. Therefore, the cache performance has a strong influence on a system's overall performance.

An important part of a static timing analysis is its cache analysis, which tries to classify memory accesses as hits or misses. Memory accesses that cannot be safely classified as a hit or a miss have to be conservatively accounted for by considering both possibilities, if the analyzed hardware architecture is not fully timing compositional.

Both precision and efficiency of a cache analysis strongly depend on the predictability of the employed replacement policy [19]. The Least-Recently-Used (LRU) replacement policy has the best predictability properties. Employing other policies, like Pseudo-LRU (PLRU) or First-In-First-Out (FIFO), yield less precise WCET bounds, because fewer memory accesses can be precisely classified. Furthermore, the efficiency degrades, because the analysis has to explore more possibilities.

3.6 Cache Replacement Policies and Domino Effects

Pseudo-LRU replacement policy: PLRU is cheaper to implement than true LRU in terms of storage requirements and update logic. However, in some cases, its replacement decisions differ substantially from those of LRU, i.e. it does not replace the least-recently-used element. In the following we demonstrate this on a sequence of accesses and explain its impact on the predictability.

Pseudo-LRU (PLRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with “tree bits” pointing to the line to be replaced next; a 0 indicating the left subtree, a 1 indicating the right. After every access, all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched.

For instance, consider Figure 4. In the initial state, the tree bits point to the line containing memory block c. A miss to e evicts the memory block c which was pointed to by the tree bits. To protect e from eviction, all tree bits on the path to the root of the tree are made to point away from it. Similarly, upon the following hit to a, the bits on the path from a to the root of the tree are made to point away from a. Note that they are not necessarily flipped. Another access to a would not change the tree bits at all as they already point away from a. Finally, a miss to f eliminates d from the cache set.

In the second state a miss would evict b. However, accessing its neighbor a flips the tree bit at the root of the tree. The access to a protects b as well. The following miss evicts d instead of b. In this way, elements may survive indefinitely without ever being

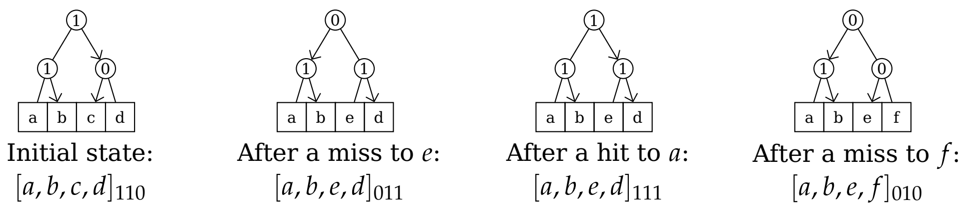

Figure 4: Updates of a PLRU cache set for the access sequence $\langle e,a,f\rangle$.

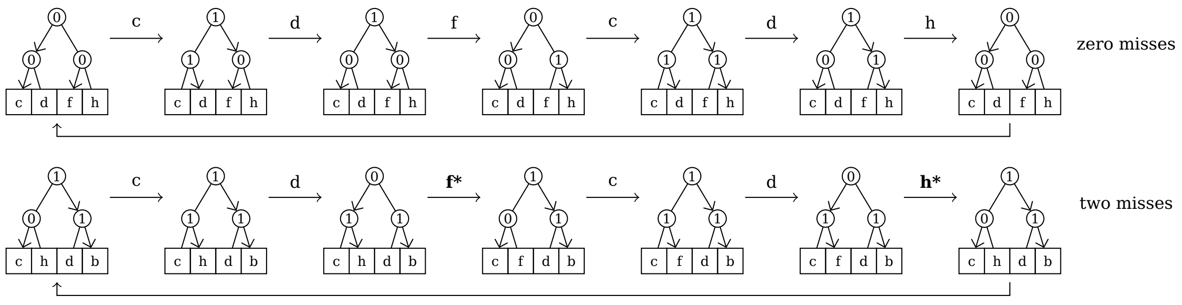

Figure 5: An example domino effect for 4-way associative caches with PLRU replacement for the access sequence (c,d,f,c,d,h): The first row features an initial cache state, where no misses occur for the given access sequence. The second row demonstrates a different initial cache state causing two misses for exactly the same access sequence. At the end of the access sequence the initial cache state is reached again. Each miss is marked by *.

accessed. This property is detrimental to the cache analysis and thus the predictability of PLRU.

Figure 5 demonstrates the domino effect caused by the PLRU replacement policy. Depending on the initial cache state, repeating the access sequence (c,d,f,c,d,h) n times either results in no misses at all or leads to $2n$ misses.

FIFO replacement policy: Similarly to PLRU, caches using the FIFO replacement policy are cheaply manufactured, as they only require a single counter pointing on the cache line to be evicted upon a miss. In [3], Berg shows that FIFO caches feature domino effects as well. In [18], the presence and extent of domino effects in different policies, including PLRU and FIFO is precisely quantified.

Round-Robin replacement policy: For caches, the round-robin replacement algorithm is equivalent to FIFO. Consequently, those caches feature the same domino effects.

Random replacement policy: Using a random replacement policy could lead to domino effects too. By chance, a cache using random replacement could behave like any strategy featuring domino effects.

3.7 Cache Write Policies

The tightness of the computed WCET bounds is also influenced by the employed cache write policy. In case that *write-through* is employed, data is written directly to main memory (and possibly to the cached block). In case of *write-back*, data is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Whenever the cache analysis cannot exclude the replacement of a dirty cache line, a write-back event might also happen. The less write-backs can be excluded, the higher the overestimation and thus the WCET bound. Furthermore, this increases the number of states the analysis has to take into account, which consequently decreases the analysis efficiency. Current analyses do not try to exclude any write-backs.

In case of write-through, the overestimation is lower since there is only one case to consider (always write to memory). However, this takes comparably longer than writing to the cache only. Hence, the absolute WCET bound must not necessarily be lower than for write-back. Nevertheless, for real systems, write-through seems to always result in lower worst-case estimates.

3.8 Preemptive Systems

There are task sets that are only schedulable in a *preemptive* scheduling regime. For instance, task sets that include high priority tasks with short deadlines are often not schedulable in non-preemptive regimes. Furthermore, each interrupt can be seen as a preempting task. However, in modern hardware architectures, preemption is not side-effect free. In *cached* systems, the major part of the preemption costs is caused by cache interferences. This part of the preemption costs is commonly referred to as the *cache-related preemption delay* (CRPD).

Memory accesses of the preempting task change the cache contents. For instance, a memory access that hits the cache during an uninterrupted execution of the task, could then miss the cache, because the execution of the preempting task leads to the eviction of the corresponding cache line. As a consequence, the approximations of the cache contents are no longer valid for a static WCET analysis of the preempted task.

The problem caused by cache interferences within a preemptive system can be solved in different ways:

Avoid cache interferences by cache partitioning [15, 33]. Each task is assigned a part of the cache, which essentially fixes the CRPD to be zero by design. The timing analysis techniques for non-preemptive execution can be applied accounting for the reduced cache size of each task.

Incorporate cache interferences during WCET analysis. Schneider's approach [23] assumes preemption at each program point to compute a safe upper bound on the execution time under preemption. If the amount of code or data is larger than the cache sizes and there is some reuse of data or instructions in the cache, this analysis largely overestimates a task's worst-case execution time.

Analyze cache interference costs separately. A bound on the CRPD is derived by means of a static analysis using data-flow analysis [2]. It computes an upper bound of preemption-induced misses assuming a constant penalty for each such miss. Being limited to timing compositional architectures, the analysis is not directly applicable to architectures featuring caches suffering from domino effects [5], as e.g. the PLRU replacement policy. During execution, a preempting task could modify the cache state such that a domino effect occurs, as Figure 5 shows.

4. Design of Predictable Multi-Cores

4.1 Interferences on Shared Resources

Most of the challenges of static timing analysis for multi-core architectures are caused by the *interference on shared resources*. Resources are shared for cost, energy, and communication reasons. Even if the sharing of a resource only slightly increases the concrete execution times of a task, it might be difficult for a static analysis to prove this: If a resource is shared among several (resource-)users, their accesses to this resource may be interleaved in a huge amount of ways, in particular if the users are not tightly synchronized. Different access sequences may result in different states of the shared resource. While the different interleaved access sequences may already exhibit different execution times, the resulting resource states may cause even more differences in the future timing behavior.

We observe two kinds of interferences: *Inherent interferences* and *virtual interferences*.

Inherent interferences on a shared resource is behavior that can actually be observed in a run of the system. Such interferences might increase the actual execution times of tasks and therefore, inherently, the WCET bounds of those tasks, too.

Virtual interferences are introduced by abstraction of the system, i.e. loss of information about the system. Although an interference might never happen in a concrete run of the system, the analysis cannot prove this, as it can only rely on its incomplete, static information. For instance, if the timing analysis for task T completely abstracts from concurrently running tasks, it has to assume an interference by another task T' every time T makes an access to a shared resource. This information loss is caused by a (total) abstraction from the set of tasks and the systems task scheduling policies, which restrict what can actually happen concurrently.

It is an open problem how to limit the information loss about concurrently running tasks by suitable abstractions. Hence, limiting inherent interferences must be a high-priority design goal: If there can be no interferences *at all* in the concrete system, it is easy for an analysis to exclude interferences even when abstracting completely from other tasks. One first principle for predictable architecture design is to strive for a good compromise between cost, performance, and predictability, *concerning the sharing or duplication of resources.*

4.2 Design Principles

The PROMPT (PRedictability Of Multi-Processor Timing) architecture design principles, see [31], aim at embedded hard real-time systems in the automotive and the aeronautics industry requiring *efficiently predictable good worst-case performance*.

The small amount of sharing existing in the set of applications allows to design a target architecture with little interference on shared resources and thus little variance of execution times and high predictability. Our principle is *Architecture follows Application*. The goals of this design discipline is to *improve the worst-case performance* and to *make the derivation of reliable and precise timing guarantees efficiently feasible*. This design discipline will support the IMA and AUTOSAR movements in the aeronautics and the automotive industries. We conjecture that without this or a similar design discipline the required modular development process will not be realizable without an unacceptable loss of guaranteed performance.

We expect that the improved precision of the execution-time bounds will

• increase the chance to show the satisfaction of timing requirements, and thereby

• avoid the need of over-commissioning and save resources.

The architecture is designed in a multi-phase process. It starts with the design or the selection of the cores that exhibit good predictability as discussed in Section 3. Then the set of applications is considered:

• *Hierarchical privatization* will decompose the set of applications according to their sharing characteristics on the shared global state. The resulting partitioning of the set of applications could be used to define an isomorphically structured target architecture with no more shared resources than required by the set of applications.

• *Sharing of lonely resources* would introduce sharing of costly and infrequently accessed resources. Input/output devices will most likely have to be shared, for cost and space reasons.

• *Controlled socialization* would try to satisfy cost constraints with an acceptable loss of predictability. It would introduce sharing while controlling the loss in predictability.

The main problem is to determine safe and sufficiently small delays for the access to shared resources. There seem to exist three alternatives: *Deterministic access protocols* [22], "cumulative" analyses using bound functions [17] and resource access arbitration with bounded delays [1, 16]. A deterministic protocol can be computed for the accesses to shared resources from the access patterns to these shared resources. This deterministic protocol will allow to control the worst-case length of an access delay and to derive safe and precise bounds on the overall execution times.

Cumulative approaches use upper bounds on the resource consumption by interfering tasks to determine safe bounds on the access delays. Instances of the third approach use arbitration schemes that guarantee bounds on the delays for the accesses to shared resources. The derivation of such bounds depends on the arbitration protocol of the bus used to access shared resources. In [1] such an arbitration scheme is complemented with a resource front-end to completely isolate the temporal behavior of tasks accessing a predictable shared resource. As shown in [4], a hybrid arbitration scheme featuring properties of round robin and TDMA exhibits both predictability and performance.

4.3 Design Guidelines

The recommendations made in the previous sections lead to the following design guidelines for predictable multi-core architectures for hard real-time systems. The first three guidelines aim at the predictability of a single core, whereas the remaining guidelines discuss the predictability of the overall system.

1. *A fully timing compositional architecture*: Exhaustive enumeration of architectural states is practically infeasible. Therefore, an abstract hardware model of the analyzed architecture is used. Timing anomalies in combination with interferences on shared resources both introduce a high computational complexity of a static timing analysis and lead to imprecise WCET bounds. Furthermore, only within timing compositional architectures additional delays can be bounded by a constant (e.g. due to an access to a shared resource or due to a preemption or interrupt).

2. *Disjoint instruction and data caches*: In case of uncertainty about data access or if the order between a data and an instruction cache access can not be precisely determined, the interferences between data and instructions accesses impairs the precision and additionally leads to an inefficient analysis.

3. *Caches with LRU replacement policy*: Employing replacement strategies like FIFO or PLRU yields less precise WCET bounds and less efficient timing analysis. Employing such strategies even introduces domino effects.

4. *A shared bus protocol with bounded access delay*: An unbounded access delay leads to an unbounded execution time of tasks that access the shared resource. Guarantee of the timing constraints can be given only in case of a bounded access delay.

5. *Private caches*: The uncertainty about cache contents of shared caches impairs the precision and leads to a more complex analysis.

6. *Private memories*, or, *only share lonely resources*: The delay to access a shared resources depends on the utilization of the resource. Hence, introducing additional sharing may lead to a system that is not schedulable.

5. Smart Configuration of Existing Multi-Cores

Section 4 discussed how one could design predictable multi-core architectures from scratch. However, currently available multi-cores were not developed with WCET analysis in mind. Consequently, they exhibit timing anomalies, poorly analyzable cache replacement policies, or fully shared memory. Leaving all those average-case performance-enhancing features enabled renders static timing analysis almost inapplicable.

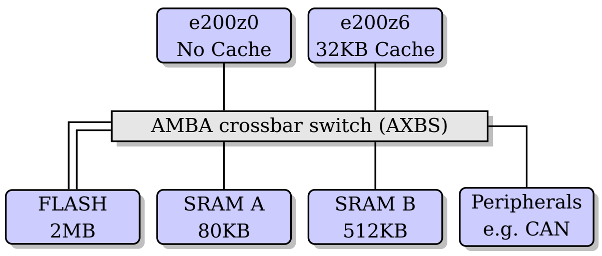

Figure 6: MPC5668G block diagram.

This section complements the former one by showing how to configure multi-core processors in a way such that static timing analysis becomes easier. In the following, we make a configuration proposal for two recent multi-core architectures—one from the automotive domain, the other one from the avionics domain.

5.1 MPC5668G – An Automotive Processor

Performance requirements of embedded automotive processors are constantly growing as more complex features are integrated in the overall system. Often a new application is implemented on a separate device. Adding more and more features to an automotive system is thus costly in terms of power consumption, heat dissipation, and space (e.g. cabling).

Consequently, there is a need to reduce the number of ECUs inside a vehicle. To do so, one could employ an embedded multi-core architectures that provides enough computational power, interfaces to peripheral devices (e.g. devices connected to CAN buses, or FlexRay) and integrate several automotive features on such single chip. This would lower the power consumption, reduce the overall heat dissipation, and solve the space problem. A typical example of such an architecture is the FreeScale MPC5668G dual-core processor that supports several currently available automotive communication protocols.

The MPC5668G processor comprises an e200z6 core—a rather complex processor—and an e200z0 core, which is a stripped-down version of the e200z6. The e200z6 core utilizes a seven-stage pipeline for single-issue in-order execution and retirement of instructions. The z6 core uses an eight entry branch target buffer (BTB) for branch prediction. The BTB entries are updated using the FIFO replacement algorithm. The cache is unified, 32KB large, and 4-way (or 8-way, depending on the configuration) set-associative. Figure 6 depicts the block diagram of the MPC5668.

To ease static timing analysis, we recommend the following smart configuration:

1. *Unified versus disjoint cache*. Unified caches are more challenging to analyze. The cache should be configured such that disjoint ways are available for code fetches and data accesses (i.e. disjoint cache).

2. *Replacement policy*. The cache uses a pseudo round-robin replacement algorithm to determine which cache line to evict upon a miss. There is a single replacement counter for all cache sets. This design is prone to high performance variations and can have domino effects. To avoid this, we recommend to lock the cache down to one way for code and one way for data. The locked ways should be filled with frequently accessed data or code. This improves the analysis results.

3. *Dynamic branch prediction*. The e200z6 core uses a BTB for dynamic branch prediction, which is updated using the FIFO replacement policy. As FIFO has domino effects, the branch target buffer should be entirely disabled to make the core more predictable.

4. *Shared memories*. In general the whole memory is shared among the two cores. However, the hardware allows for some partitioning such that conflicts on the internal SRAM memory modules can be avoided. The MPC5668G features two disjoint SRAM memory modules: an 80KB module, and a 512KB module. To avoid any interferences on the internal SRAM, the application software could be designed such that one SRAM module only is used by z0 core, whereas the z6 core solely uses the other SRAM module.

5. *Shared FLASH prefetch buffers*. To reduce access delays on the internal FLASH memory, the MPC5668G core implements four prefetch buffers that allow for zero-cycle access delays in case a buffer already contains the requested data. The prefetch buffers are shared between the two processors, and are used for both instruction and data accesses. For predictability reasons, the prefetch buffers should only be enabled for one of the cores, to avoid any interferences. Furthermore, the prefetch buffers should be split up, such that disjoint buffers are used to satisfy instruction fetches and data accesses (exactly for the same reason as the cache should be split between code and data accesses). This configuration does not redeem the FLASH module of all interferences. An access of any of the cores might still be delayed by an access of the other one (address pipelining). To get rid of those inferences as well, the code executed by one core should be put into the privately used SRAM module—where applicable.

The above configuration allows for an efficient static WCET analysis that yields results that are quite precise.

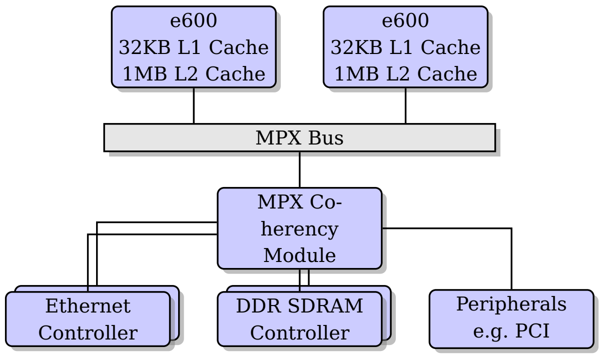

Figure 7: MPC8641D block diagram.

5.2 *MPC8641D – An Avionics Processor*

The MPC8641D is a dual-core derivate of the MPC7448, which is a complex single-core architecture employed in the avionics industry. A single-core MPC7448 consists of a e600 core with a complex, eight-level pipeline that allows out-of-order and speculative execution and features first- and second-level caches with PLRU and random replacement. Already as a single-core, this architecture is non-compositional, exhibiting both domino-effects in the pipeline and the caches. The MPC8641D tightly couples two such cores with a single shared bus. Figure 7 shows the block diagram of the MPC8641D. Each access, either for the instruction fetches or any data access must pass this one shared resource. Given the non-compositionality of the two cores, any clash on the shared bus during execution could trigger a domino effect. This makes the timing behavior of the entire system very unpredictable, unless interference on the shared bus can somehow be avoided.

The individual cores can be made more predictable by configuration: e.g. locking down the first level caches to have a LRU replacement policy and using the write-through policy, completely locking the random replacement second level cache for use as scratchpad memory and using static branch prediction in favour of the dynamic one. Still the domino-effects which are possible in their complex pipelines are not avoidable. Therefore, to get a predictable multi-core system, clashes on the shared bus need to be avoided.

For avoiding interferences upon bus accesses, two features of the *IMA architecture* and the employed cores are very helpful:

• The *IMA architecture* features time slices for the individual tasks in the ten milliseconds range, which is quite long compared to the one millisecond typically seen in the automotive domain. Inside each of this time slices, the input/output activities, which only make up a fraction of the slice, can be moved to the beginning and the end of the individual time slice to create local copies of the working set. Then the largest remaining fraction of the time slice can the be used for the lengthy computation on the local copies of the data.

• The two e600 cores are supported by private 1MB second level caches. One of the cores can use this cache as local private memory for instructions and data by locking it. This allows the avoidance of bus accesses by one core during the long computation phases of its tasks.

Given the above design of the system, in which one core works on its private memory most of the time and only short time slices are needed for bus accesses, a clever scheduling can completely avoid clashes of accesses. Therefore, the normal static timing analysis, which assumes continuous execution without interferences, can be used to deal with each of the cores separately.

6. Summary

The article presents the timing-analysis problem, its roots, and architecture-dependent complications. It sketches the currently available timing-analysis technology for single-core architectures. Extrapolating from the practical experience and theoretical insights in the single-core case shows that wrong multi-processor designs will make timing analysis infeasible. Design principles for predictable multi-processor architectures are given and finally it is shown how smart configuration of existing architectures, designed without predictability considerations, can improve predictability considerably.

7. References

[1] Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. *Euromicro Symposium on Digital Systems Design*, 0:547–555, 2009.

[2] Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[3] Christoph Berg. PLRU cache domino effects. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2006.

[4] Paolo Burgio, Martino Ruggiero, and Luca Benini. Simulating future automotive systems. Technical report, Micrel Lab, University of Bologna, 2010.

[5] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches: Pitfalls and solutions. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2009.

[6] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In *Workshop on*

Worst-Case Execution-Time Analysis (WCET), July 2007.

[7] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution-Time Analysis*. PhD thesis, Uppsala University, 2002.

[8] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution-Time Analysis*. PhD thesis, Uppsala University, 2003.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Conference on Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, 2001.

[10] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[11] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.

[12] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Real-Time Systems Symposium (RTSS)*, 1995.

[13] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Real-Time Systems*, 91(7):1038–1054, 2003.

[14] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.

[15] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[16] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *International Symposium Computer Architecture (ISCA)*, 2009.

[17] Rodolfo Pellizzoni and Marco Caccamo. Toward the predictable integration of real-time COTS based systems. In *Real-Time Systems Symposium (RTSS)*, 2007.

[18] 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.

[19] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[20] Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in presence of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2009.

[21] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2006.

[22] 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 *Real-Time Systems Symposium (RTSS)*, 2007.

[23] Jörn Schneider. Cache and pipeline-sensitive fixed-priority scheduling for preemptive real-time systems. In *Real-Time Systems Symposium (RTSS)*, 2000.

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

[25] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2007.

[26] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2007.

[27] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Universität des Saarlandes, 2002.

[28] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[29] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[30] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract-interpretation-based timing validation of hard real-time avionics software systems. In *Dependable Systems and Networks (DSN)*, June 2003.

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

[32] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009.

[33] Andrew Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.