Timing Validation of Automotive Software

Daniel Kästner[1], Reinhard Wilhelm[2], Reinhold Heckmann[1], Marc Schlickling[12], Markus Pister[12], Marek Jersak[3], Kai Richter[3], Christian Ferdinand[1]

[1] AbsInt GmbH, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany
[3] Symtavision GmbH, Braunschweig, Germany

Abstract Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. During the last years sophisticated analysis tools for timing analysis at the code-level, controller-level and networked system-level have been developed. This trend is exemplified by two tools: AbsInt's timing analyzer aiT, and and Symtavision's SymTA/S. aiT determines safe upper bounds for the execution times (WCETs) of non-interrupted tasks. SymTA/S computes the worst-case response times (WCRTs) of an entire system from the task WCETs and from information about possible interrupts and their priorities. A seamless integration between both tools provides for a holistic approach to timing validation: starting from a system model, a designer can perform timing budgeting, performance optimization and timing verification, thus covering both the code and the system aspects. However, the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform. Especially on multi-core architectures this aspect becomes of critical importance. This paper describes an industry-strength tool flow for timing validation, and discusses prerequisites at the hardware level for ascertaining high analysis precision.

1 Introduction

Developers of safety-critical real-time systems have to ensure that their systems react within given time bounds. Tests and measurements help to detect violations of time bounds, but cannot prove their absence, unless all possible scenarios are covered. Face to the complexity of today's embedded software and the complexity of contemporary hardware architectures this is virtually impossible. Moreover, tests and measurements are only available late in the development cycle. Tools for static program analysis can obtain results valid for all possible system runs and inputs, even before the first real prototypes are available. Examples for such tools are AbsInt's timing analyzer aiT, and Symtavision's SymTA/S tool. aiT is a code-level analysis tool determining safe upper bounds for the execution times (WCETs) of non-interrupted tasks. SymTA/S works at the system level;

it computes the worst-case response times (WCRTs) of an entire system from the task WCETs and from information about possible interrupts and their priorities.

A tool coupling between aiT and SymTA/S paves the path towards a holistic timing validation tool chain. The system developer creates a system model in SymTA/S, consisting of a task graph, information on task scheduling (priorities, time slots, etc.), and information on task activation (time tables, interrupts, etc.). To determine the WCRTs of the tasks with possible interrupts, the WCETs of the non-interrupted tasks are required. To obtain these WCETs, SymTA/S sends requests to aiT. Then aiT asks the developer for necessary information on hardware configuration and executables, determines the requested WCETs, and sends them back to SymTA/S. The coupling of SymTA/S with aiT allows SymTA/S to determine end-to-end timings in an early development phase, with automatic identification of problematic system configurations and automatic system optimization as a next step. In the EU projects INTEREST and INTERESTED the tool coupling is extended towards model-based code generation tools such as ASCET[8] and SCADE[7] to support the entire development process from the model till the implementation with formally verified timing behavior.

Static-analysis-based methods can give timing guarantees even for complex processor architectures exhibiting a huge execution-time variability and a strong dependency of the execution time on the initial execution state. Nevertheless, the efficiency and the precision of the result of timing analysis highly depends on the hardware architecture. As an example, a cache with a random replacement strategy does not allow for a cache analysis with good precision. The design of the internal and external buses, if done wrongly, leads to hardly analyzable behavior and great loss in precision. Multi-core architectures with shared caches, finally, will create a space of interleavings of interactions on these caches that will make sound and precise timing analysis practically infeasible at all.

The trend in automotive embedded systems is towards unifying frameworks like AUTOSAR. Such frameworks aim at managing the increasing functional complexity and allowing for better reconfigurability and maintainability. Standardized interfaces allow to compose components, independently developed by different suppliers, on ECUs. A runtime environment provides basic services, like intra- or inter-ECU communication between components. At a lower level AUTOSAR abstracts from the underlying hardware, the actually deployed ECUs. From a functional point of view this framework is appealing because of the gained compositionality.

The AUTOSAR timing model currently being developed concerns mainly the integration of scheduling requirements. The success of scheduling analysis depends on the predictability of the execution times of the AUTOSAR-"runnables", the basic building blocks of a software component. When multiple components are mapped to a hardware architecture where a high degree of interference between the components cannot be avoided (e.g., due to shared caches or buses) execution times of runnables may vary considerably and the possibilities to predict safe and precise execution time bounds can be rather limited. This limits the success of the scheduling analysis and this counteracts the idea of

composing software components. Thus, the applicability of the AUTOSAR idea depends on availability of architectures on which software composition doesn't lead to unpredictable timing behavior.

In the following, we present a tool flow for validating timing behavior based on aiT and SymTA/S. In Sec. 2-Sec. 3 we discuss the influence of the hardware architecture on the timing validation process, based on experience with static timing-analysis in the embedded-systems industry [30,13] and theoretical insights [23,22,11]. Sec. 5 first address hardware issues that have to be respected both in single core and in multi-core architectures. Face to the increased complexity of multi-core designs, there these aspects become even more critical than they are in the single core domain. Sec. 5 concludes with discussing specific issues for multi-core architectures.

2 The System Level: Schedulability Analysis

Scheduling analysis is a systematic approach that automatically finds and evaluates critical timing situations resulting from function and system integration. Such *corner case identification* is the opposite of traditional test-based methods: instead of massive testing to try to find all corner cases, scheduling analysis systematically constructs scenarios leading to worst-case timing.

SymTA/S [14] is Symtavision's tool for timing and scheduling analysis and optimization for controllers, networks and entire systems. SymTA/S computes the worst-case response times (WCRTs) of tasks and the worst-case end-to-end communication delays. It takes into account the worst-case execution times of the tasks as well as information about RTOS scheduling, bus arbitration, possible interrupts and their priorities. The graphical user interface of SymTA/S offers ways to specify a system architecture, select scheduling on controllers and arbitration on buses, map functions to controllers and communication to busses, and to describe dataflow, activation conditions, deadlines and other timing constraints.

The analysis results are displayed in a variety of ways. The most powerful and easy to understand are *Gantt-Charts* that visualize to the designer why and under which conditions deadlines can be violated. There are two main use cases for SymTA/S:

1. *timing design / budgeting* during early design stages and
2. *timing verification* during later design stages.

The added value for verification is exemplified in Figure 1: the upper part of the diagram displays a typical timing trace, showing a response time of 6.9 ms for a task executed every 10 ms – well below the 10 ms deadline. In the lower part, SymTA/S scheduling analysis has constructed a worst-case schedule leading to a WCRT of 9 ms for the 10 ms task – still below the deadline, but much closer. The key message is that no other schedule will produce a longer WCRT. SymTA/S thus safeguards against deadline violations resulting from worst-case schedules. Furthermore, the Gantt-display enables the designer to check the reasoning of

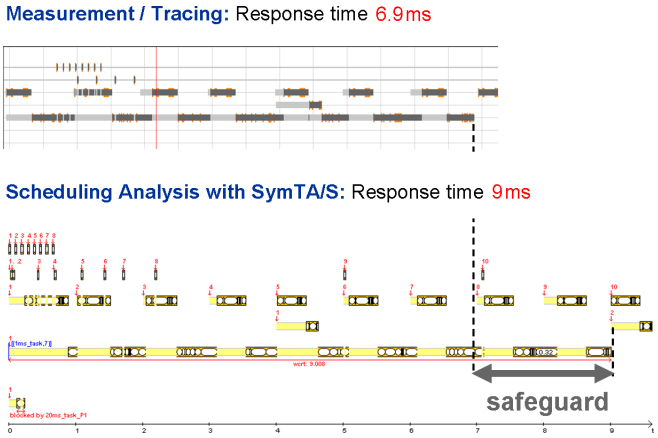

Figure 1. Safeguarding timing using scheduling analysis (as compared to measurement)

SymTA/S and thus to see if some important information has been omitted in the model.

Furthermore SymTA/S also supports the system design stage. With a *what-if* scheduling analysis it is easily possible to estimate how additional functions and their scheduling will influence overall system timing, and whether deadlines can be safely met for a specific design alternative. As a result, timing budgets for individual functions of sub-systems can be derived early on, and given to a designer as part of a requirements specification. Additionally, SymTA/S offers a plugin for *design-space exploration* allowing designers to automatically evaluate the strengths and weaknesses of alternative designs with respect to timing and performance. A plugin for *sensitivity analysis* allows users to automatically determine the amount of extra load (e.g., caused by additional functions) permissible without violating deadlines.

3 The Code Level: Static Timing Analysis

aiT computes safe upper bounds on the worst-case execution times (WCETs) of sequential tasks. For a precise computation of the WCET, aiT operates on the executable. If available, aiT can also read the source files for further information. The WCET is computed in several phases [9] (see Figure 2).

In the first step a *decoder* reads the executable and reconstructs its control flow [27]. Then, *value analysis* determines lower and upper bounds for the values in the processor registers for every program point and execution context, which lead to bounds for the addresses of memory accesses (important for cache analysis and if memory areas with different access times exist). Value analysis can also determine that certain conditions always evaluate to true or always evaluate

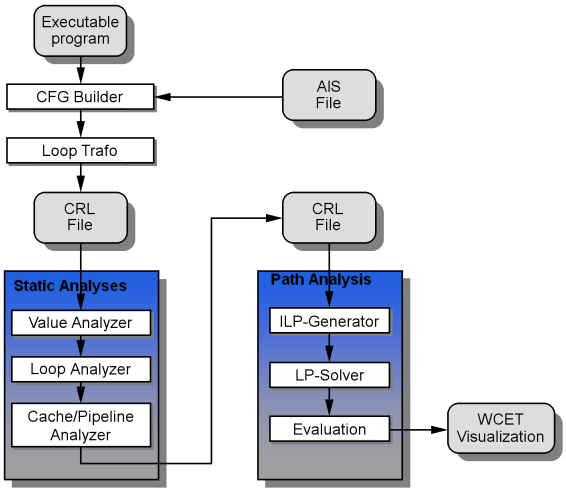

Figure 2. Phases of WCET Computation

to false. As consequence, certain paths controlled by such conditions are never executed. Thus value analysis can detect and mark some unreachable code.

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. aiT tries to determine the number of loop iterations by *loop bound analysis* [10], but succeeds in doing so for simple loops only. Bounds for the remaining loops must be provided as specifications in a separate parameter file (.ais file) or annotations in the C source. The *micro-architectural analysis* [6,29,11] determines bounds on the execution time of individual basic blocks by performing an abstract interpretation of the program. It thereby takes into account the processor's pipeline, caches, and speculation concepts: static cache analyses determine safe approximations to the contents of caches at each program point. All accesses into main memory are classified into hits, misses, or accesses of unknown nature. A pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues and functional units etc., and for the classification of memory references by the cache analysis etc [26]. Ignoring these average-case-enhancing features would result in imprecise bounds. Using this information, *path analysis* determines a safe estimate of the WCET. The program's control flow is modeled by an integer linear program [16,28] so that the solution to the objective function is the predicted worst-case execution time for the input program.

After a successful analysis, aiT reports its results in several ways: aiT can produce a graphical output showing the call graph and control flow graph of the analyzed part of the application. Alternatively, aiT can write a text report meant to be human readable, and a more formal XML report. These reports contain detailed results for all analyzed routines in all calling contexts, including specific results for the first few iterations of loops vs. a result for the remaining iterations.

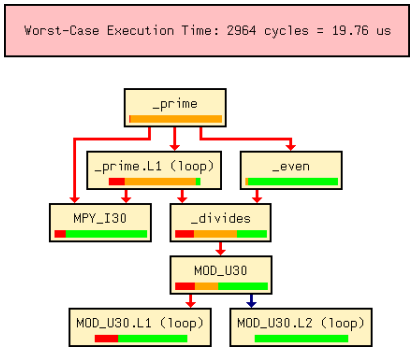

Figure 3. Call graph with WCET result

4 The Interaction between SymTA/S and aiT

Timing analysis is a novel domain, and the requirements for coupling code-level and system-level tools were not suitably covered by any existing exchange format. Therefore, the concept of "Timing Cookies" has been developed in the INTEREST project to avoid the duplication of the sophisticated user-interfaces of the tools. The information exchanged is stored in such a Timing Cookie in the first round of communication between the tools. During the next round of communication this information is retrieved from that cookie so that invariant parts do not have to be re-entered manually.

The Timing Cookie Exchange Format XTC is defined as an XML schema. It is organized hierarchically since some information can be reused for different analyses. For instance, a CPU configuration can be reused for different runnables sharing the same CPU. The information in the common section is structured in four blocks: general, CPU, runnable (i.e., an atomic piece of software[4]), and mode (a specific control-flow path through the runnable).

The interaction between SymTA/S and aiT follows the following pattern: From a system model, SymTA/S launches a request for WCET information for specific pieces of code (see Figure 4). This request is tagged with a unique ID and sent to aiT in an XTC. If necessary, aiT queries the user for all missing information required to service the request. For the first request issued for a system model, this typically includes the type of processor, the location of the executable code, the starting point of the analysis etc. When aiT answers the request by sending an XTC with a response back to SymTA/S, it stores this information in the private aiT-part of the cookie. This aiT-specific information will be included in subsequent requests so that aiT can use the information already gathered without the need to ask the user again.

[4] AUTOSAR [2] terminology has been adopted

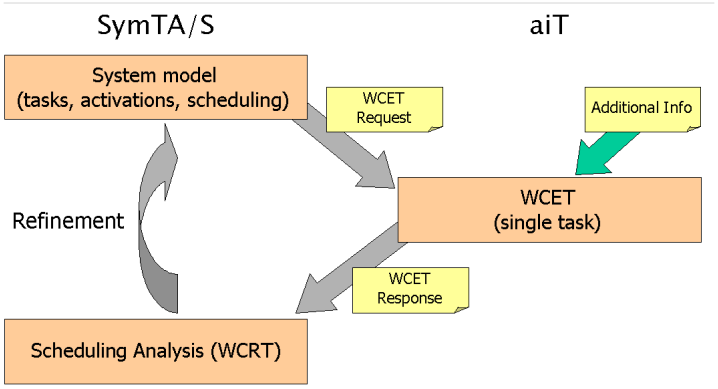

Figure 4. Flow of requests and responses

5 Hardware and Predictability

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. Multi-core designs combine two or more independent cores into a single die. Cores in a multi-core device may share a single coherent cache at the highest on-device cache level or may have separate caches. The processors also share the same interconnect to the rest of the system. Each core independently implements the typical hardware features described above for the single core domain.

The consequence is that the execution time of individual instructions, and thus the contribution of one execution of an instruction to the program's execution time can vary widely. This variation 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.

For WCET computation, 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 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. How much is lost depends both on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software.

Most of the problems posed to timing analysis are caused by the *interference on shared resources*. Resources are shared for cost, energy and performance

reasons. Different users of a shared resource may often access the resource in a statically unknown way. Different access sequences may result in different states of the resource. The different sequences may already exhibit different execution times, and the resulting resource states may again cause differences in the future timing behavior. An out-of-order processor will execute an instruction stream in one sequential order. Exhaustive exploration could, in principle, identify this one sequence for each input and initial state, while in practice, this is infeasible. Thus, this order is assumed to be not statically known. This forces the analysis to consider all possible sequences. Different sequences may have different effects on the cache contents. Examples of shared resources with interferences are buses and memory: Buses are used by several masters, which may access the buses in unpredictable ways. On a multi-core system the interference on shared resources is significantly increased wrt a single core system. Memory and caches are shared between several processors or cores. One thread executed on one core does not know when accesses by another thread on another core will happen.

In the following we will first investigate the predictability of common elementary hardware features: pipelines, caches, and buses. This discussion applies both for single core and multi-core architectures, since they determine the "internal´´ behavior of each core in a multi-core design. In Sec. 5.4 we additionally discuss specific features of multi-core designs.

5.1 Pipelines

For non-pipelined architectures one can simply add up the execution times of individual instructions to obtain a bound on the execution time of a basic block. Pipelines increase performance by overlapping the executions of different instructions. Hence, a timing analysis cannot consider individual instructions in isolation. Instead, they have to be considered collectively – together with their mutual interactions – to obtain tight timing bounds. Superscalar- and out-of-order execution increase the number of possible interleavings. The larger the buffers (e.g., fetch buffers, retirement queues, etc.) are the longer lasts the influence of past events. Dynamic branch prediction, cache-like structures, and branch history tables increase history dependence even more.

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. To compute a precise bound on the execution time of a basic block, the analysis needs to exclude as many *timing accidents*, i.e., incidents that cause an increase of an instruction's execution time, 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. Then the analysis must take all alternative scenarios into account. One could be tempted to design the analysis such that

only the locally most expensive pipeline transition is chosen. However, in the presence of *timing anomalies* [17,23] this approach is unsound.

The notion of *timing anomalies* was introduced by Lundqvist and Stenström in [17]. In the context of WCET analysis, [23] 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. A scenario where this can occur is when the cache miss penalty prevents the branch unit from misspeculating and prefetching along the wrong path. An especially severe timing anomaly is the so-called *domino effect* [17] that causes the difference in execution time of the same program starting in two different hardware states to become arbitrarily high. 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 [25], Schneider describes a domino effect in the pipeline of the PowerPC 755. Another example is given by Berg [3] who considers the PLRU replacement policy of caches. Thus, in general, the analysis has to follow all possible successor states.

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.
- **Compositional architectures with constant-bounded effects:** These exhibit timing anomalies but no domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant number of cycles to the local worst-case path. The Infineon TriCore is assumed, but not formally proven, to belong to this class.
- **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.

5.2 Caches

To obtain tight bounds on the execution time of a task, timing analyses *must* take into account the cache architecture. The cache analysis 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. The precision of a cache analysis is strongly dependent on the predictability of the cache architecture, especially on its replacement policy. The three most common replacement strategies are the following:

– The LRU(Least Recently Used) replacement strategy uses age bits to discard the least recently used cache line in case of a cache miss. When a cache hit occurs, the age information of all cache lines is updated. It is used in the Freescale PPC603e core and the MIPS 24K/34K.
– With the FIFOstrategy the cache is organized like a queue: new elements are inserted at the front evicting elements at the end of the queue. In contrast to LRU, hits do not change the queue. FIFO is used in the Intel XScale and some ARM9 and ARM11 based processor cells.
– PLRU(Pseudo-LRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k-1$ "tree bits" pointing to the line to be replaced next. For an in detail explanation of PLRU consider [22,1]. It is used in the PowerPC 75x and the Intel Pentium II-IV.

In [21] the influence of these three replacement strategies on the precision of static cache analyses is analyzed. The results show that LRU-replacement has the best predictability properties of all replacement policies. Employing other policies, like PLRU or FIFO, yields less precise WCET bounds, because fewer memory accesses can be classified as hits of misses. This also has the consequence that timing analysis has to explore more possibilities so that the efficiency is lower than with LRU.

Static cache analyses usually cannot make any assumptions about the initial cache contents. Cache contents on entrance depend on previously executed tasks. Even assuming a completely empty cache may not be conservative as shown in [3,21], the notable exception being LRU. FIFO and PLRU are much more sensitive to their state than LRU. Depending on its state, FIFO(k) may have up to k times as many misses and arbitrarily more hits, on the same access sequence. PLRU and LRU coincide at associativity 2. For greater associativities, the same access sequence under a PLRU strategy may incur arbitrarily many more misses for one starting state than for another. For PLRU(8), the number of hits of the same access sequence with different starting states may differ by a factor of 11. In [21] also the aggregated effect of the initial cache setting on WCET has been investigated for a realistic hardware setting. For a 4-way set-associative FIFO cache with a cache miss penalty of 50 cycles, the worst-case execution time may be a factor of 3 higher than the measured time of the same access sequence, only due to the influence of the initial cache state. If PLRU were used as a replacement policy the difference could be even greater.

This is especially detrimental for measurement-based approaches [18,4,32]. Measurement would trivially be sound if all initial states and inputs would be covered. Due to their huge number this is usually not feasible. Some measurement-based approaches consider distributions of execution times (execution time profiles) of program snippets, which are then composed according to the control flow. Each program fragment is measured with a subset of the possible initial states and inputs so that the maximum of the measured execution times is in general an underestimation of the worst-case execution time. Using corrective

bounds for the so-called execution time profiles can introduce large pessimism since they don't exploit context and flow information [19].

Relatively simple architectures without any performance-enhancing features like pipelines, caches, etc., exhibit the same timing independently of the initial state. For such architectures, measurement-based timing analysis is sound [32]. [5] and [32] propose to lock the cache contents [20,31] and to flush the pipeline at program points where measurement starts. This is not possible on all architectures and it also has a detrimental effect on both the average- and the worst-case execution times of tasks.

5.3 Buses

A bus is a subsystem for transferring data between different components inside a computer, between a computer and its peripheral devices, or between different computers. Examples are system buses like the *60x-bus* [12], internal computer buses like *PCI* and external computer buses like *CAN* or *FlexRay*.

In general, busses are clocked with a lower frequency than the CPU. The number of possible displacements of phase between CPU- and bus-clock signal is bounded, i.e. at the start of a CPU cycle the bus cycle can only be in a finite number of states. For example, if the CPU operates at $f_{CPU} = 100$ MHz and the bus at $f_{BUS} = 25$ MHz, there are 4 different states. In general, the number of states is determined by $\frac{f_{CPU}}{\gcd(f_{CPU}, f_{BUS})}$.

Analyzing timing behavior of memory accesses is special because these accesses cross the CPU/bus clock boundary. Since the time unit for timing analyzes is one CPU cycle, the analysis needs to know when the next bus cycle begins. Otherwise it would have to account for the worst case: the bus cycle has just begun and the CPU needs to wait nearly a full bus cycle to perform a bus action. This pessimism would lead to less precise WCET bounds. Therefore, the displacement of phase has to modeled within a micro-architectural analysis so that the search space for the analysis is augmented by the number of different bus-clock-states.

Parallel buses (e.g., SCSI) introduce further complication. The execution of consecutive memory accesses can be overlapped, i.e. for two accesses, the address phase of the second access can be overlapped with the data phase of the first access (*bus pipelining*). Pipelined buses need to arbitrate the incoming bus requests, e.g. if there is an instruction fetch and a memory access at the same time, the arbitration logic needs to decide which bus request is issued first.

Asynchronous mechanisms such as *DMA* or *DRAM* refresh cannot be analyzed with the methods described so far. A DMA transmission and a DRAM refresh and their associated costs cannot be contributed to the execution of an instruction. The costs of a DRAM refresh must be amortized over time. A similar approach can be used if the frequency of DMA is statically known.

5.4 Multi-core Architectures

There is a tendency towards the use of multi-core architectures for their good energy/performance ratio. Shared memories (Flash, RAM) and peripherals are connected to the cores by shared buses or cross-bars. Conflicts when accessing shared resources are usually resolved by assigning fixed priorities. Depending on the architecture, conflicts on shared resources can be expected to happen frequently. For example, if the cores have no private RAM, a potential conflict might occur for each access (typically 20-30% of all executed instructions). Examples for current automotive multi-core architectures are the Infineon TriCore TC1797, the Freescale MC9S12X and the Freescale MPC5516. They consist of a powerful main processor and a less powerful co-processor. For future automotive multi-core architectures we see a design trend towards the use of identical cores mostly with shared memories.

Under the aspect of predictability, some existing and upcoming multi-core architectures are unacceptable because of the interference of the different cores on shared resources such as caches and buses. The execution time of a task running on one core typically depends on the activities on the other cores. Static worst-case execution time analysis usually assumes the absence of interferences. The additional time (or penalty) caused by interferences must be bounded for a scheduling analysis. For architectures with domino effects and timing anomalies inside the cores the additional inter-core interferences represent a huge obstacle to determining such a bound. Especially the unconstrained use of shared caches can make a sound and precise analysis of the cache performance impossible. The set of potential interleavings of the threads running on the different cores result in a huge state space to be explored resulting in poor precision.

There exist first approaches to the analysis of the cache performance of shared caches in multi-core systems. All approaches implicitly assume *fully timing compositional architectures* (see Sec. 5.1). They compute the cache footprint of preempted and preempting tasks, determine the intersection, and assume the rest as being eliminated (cf. [15]). This approach is neither context-sensitive nor flow-sensitive and therefore overly pessimistic. For a fully timing-compositional architecture, Schlieker, Ivers, and Ernst [24] determine upper bounds of the penalties by computing the number of potential conflicts when accessing shared memory by counting the number of memory accesses possibly generated on different cores. Thus it becomes apparent that in order to achieve good predictability results on a multi-core system choosing a fully timing compositional intra-core architecture combined with separate caches is of utmost importance.

6 Conclusion

Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. In order to provide software and system designers with an efficient way to verify timing properties of ECU software, the code-level timing analysis tool aiT and the system-level timing analysis tool SymTA/S have been coupled. Starting from a system model, a designer can perform timing budgeting,

performance optimization and timing verification, thus covering both the code and the system aspects. XML Timing Cookies (XTC) provide for a user-friendly, open, and efficient tool integration. SymTA/S communicates with aiT via XTC, sending analysis requests and receiving responses. While providing a holistic tool flow for timing validation to system designers, the precision of the results and the efficiency of the analysis methods depend on the predictability of the execution platform.

This is particularly important face to the trend towards unifying frameworks like AUTOSAR in automotive embedded systems. The goal is to establish a standard in which components, possibly independently developed by different suppliers, can be integrated on ECUs by standardized interfaces. To this end AUTOSAR abstracts from the underlying hardware, the actually deployed ECUs. The AUTOSAR timing model currently being developed concerns mainly the integration of scheduling requirements. However, the success of scheduling analysis depends on the predictability of the execution times of the AUTOSAR-"runnables". Thus, the applicability of the AUTOSAR idea depends on availability of architectures on which software composition doesn't lead to unpredictable timing behavior.

The experience with the use of static timing analysis methods and the tools based on it in the automotive and the aeronautics industries is positive. Static-analysis-based methods can give timing guarantees even for complex processor architectures exhibiting a huge execution-time variability and a strong dependency of the execution time on the initial execution state. However, when multiple components are mapped to a hardware architecture where a high degree of interference between the components cannot be avoided (e.g., due to shared caches or buses) execution times of runnables may vary considerably and the possibilities to predict safe and precise execution time bounds can be rather limited. This limits the success of the scheduling analysis and this counteracts the idea of composing software components. In contrast, choosing a fully timing compositional intra-core architecture combined with separate caches will lead to good predictability results. In consequence, the underlying hardware architecture has to be chosen with timing predictability in mind.

This paper discusses the most important hardware components affecting timing predictability and summarizes their effect on the applicability of measurement-based approaches and on the efficiency and precision of static analysis methods. An industry-strength tool flow for timing validation is presented, and the prerequisites at the hardware level for ascertaining high analysis precision are detailed.

References

1. H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 267–272, New York, NY, USA, 2004. ACM Press.
2. T. AUTOSAR Development Partnership. Automotive Open System Architecture (AUTOSAR). URL: http://www.autosar.org, 2003.

3. C. Berg. PLRU cache domino effects. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.
4. G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, Washington, DC, USA, 2002. IEEE Computer Society.
5. J.-F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In R. Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
6. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, 2002.
7. Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com/products/scade-suite.
8. ETAS Group. ASCET Software Products. http://www.etas.com/en/products/ascet_software_products.php.
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 *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.
10. C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in WCET analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *Lecture Notes in Computer Science*, pages 12–52. Springer-Verlag, 2007.
11. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.
12. Freescale Semiconductor, Inc. *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*, 2004. Rev. 0.1.
13. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.
14. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. *IEEE Proceedings on Computers and Digital Techniques*, 152(2), Mar. 2005.
15. C.-G. Lee, J. Hahn, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 264, Washington, DC, USA, 1996. IEEE Computer Society.
16. Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.
17. T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 12–21, December 1999.
18. S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, Munich, Germany, Sept. 2002.
19. S. M. Petters, P. Zadarnowski, and G. Heiser. Measurements or static analysis or both? In C. Rochange, editor, *WCET*, 2007.

20. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.
21. J. Reineke and D. 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.
22. J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.
23. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.
24. S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in MPSoCs. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 288–293. ACM Press New York, NY, USA, 2006.
25. J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.
26. J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.
27. H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.
28. H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, Dec. 1998.
29. S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
30. S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.
31. X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282, 2003.
32. I. Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.