Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 758480, 17 pages
doi:10.1155/2009/758480

Research Article

Time-Predictable Computer Architecture

Martin Schoeberl

Institute of Computer Engineering, Vienna University of Technology, 1040 Vienna, Austria

Correspondence should be addressed to Martin Schoeberl, mschoebe@mail.tuwien.ac.at

Received 8 August 2008; Accepted 6 December 2008

Recommended by Bernhard Rinner

Today's general-purpose processors are optimized for maximum throughput. Real-time systems need a processor with both a reasonable and a known worst-case execution time (WCET). Features such as pipelines with instruction dependencies, caches, branch prediction, and out-of-order execution complicate WCET analysis and lead to very conservative estimates. In this paper, we evaluate the issues of current architectures with respect to WCET analysis. Then, we propose solutions for a time-predictable computer architecture. The proposed architecture is evaluated with implementation of some features in a Java processor. The resulting processor is a good target for WCET analysis and still performs well in the average case.

1. Introduction

Standard computer architecture is driven by the following paradigm: *Make the common case fast and the uncommon case correct* [1]. However, this design approach leads to architectures where the worst-case execution time (WCET) is high and hard to predict by static analysis. For real-time systems, we have to design architectures with the following paradigm: *Make the worst case fast and the whole system easy to analyze*.

Classic enhancements in computer architectures are pipelining, instruction and data caching, dynamic branch prediction, out-of-order execution, speculative execution, and fine-grained chip multithreading. These features are increasingly harder to model for the low-level WCET analysis. Execution history is the key to performance enhancements, and also the main issue for WCET analysis. Thus, we need techniques to manage the execution history.

Pipelines should be simple, with minimum dependencies between instructions. It is agreed that caches are mandatory to bridge the gap between processor speed and memory access time. Caches in general, and particularly data caches, are usually hard to analyze statically. Therefore, we are introducing caches that are organized to speed up execution time and provide tight WCET bounds. We propose three different caches: (1) an instruction cache for full methods, (2) a stack cache, and (3) a small, fully associative buffer for heap access. Furthermore, the integration of a program—or compiler—managed scratchpad memory can help to tighten bounds for hard-to-analyze memory access patterns.

Out-of-order execution and speculation result in processor models that are too complex for WCET analysis. We discuss that the transistors are better used onchip multiprocessors (CMPs) with simple in-order pipelines. Real-time systems are naturally multithreaded and thus map well to the explicit parallelism of chip multiprocessors.

We propose a multiprocessor model with one processor per thread. Thread switching and schedulability analysis for each individual core disappears, but the access to the shared resource main memory still needs to be scheduled.

We have implemented most of the proposed concepts for evaluation in a Java processor. The Java processor JOP [2] is intended for real-time and safety critical applications written in a modern object-oriented language. It has to be noted that all concepts can also be applied to a standard RISC processor. The following list points out the key arguments for a time-predictable computer architecture.

(i) There is a mismatch between performance-oriented computer architectures and worst-case analyzability.

(ii) Complex features result in increasingly complex models.

(iii) Caches, a very important feature for high performance, need new organization.

(iv) Thread level parallelism is natural in embedded systems. Exploration of this parallelism with simple chip multiprocessors is a valuable option.

(v) One thread per processor obviates the classic schedulability analysis and introduces scheduling of memory access.

Catching up with WCET analysis of features that enhance the average-case performance is not an option for future real-time systems. We need a sea change and should take the constructive approach by designing computer architectures where predictable timing is a first-order design factor.

The contributions of the paper are twofold: (1) an extensive overview is given of processor features that make WCET estimation difficult, (2) solutions for a time-predictable architecture that can be implemented in RISC, CISC, or VLIW style processors are provided. The implementations of some of the proposed concepts in the context of a Java processor, as described in Section 5, have been previously published in [3, 4].

The paper is organized as follows. Section 2 presents related work on real-time architectures. In Section 3, we describe the main issues that hamper tight WCET estimates of actual processors. We propose solutions for these issues in Section 4. In Section 5, we evaluate the proposed time-predictable computer architecture with an implementation of a Java processor in an FPGA. Section 6 concludes the paper.

2. Related Work

Bate et al. [5] discuss the usage of modern processors in safety critical applications. They compare commercial off-the-shelf (COTS) processors with a customized processor developed specifically for the safety critical domain. While COTS processors benefit from a large user base and the resulting maturity of the design process, customized processors provide the following advantages:

(i) design in conjunction with the safety argument,

(ii) design for good worst-case performance,

(iii) using only features that can be easily analyzed,

(iv) the processor can be treated as a *white box* during verification and testing.

Despite these advantages, few research projects exist in the field of WCET-optimized hardware. Thiele and Wilhelm [6] discuss that a new research discipline is needed for time-predictable embedded systems to "match implementation concepts with techniques to improve analyzability."

Similarly, Edwards and Lee discussed as "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [7]. A first simulation of their PRET architecture is presented in [8]. PRET implements the SPARC V8 instruction set architecture (ISA) in a six-stage pipeline and performs chip level multithreading for six threads to eliminate data forwarding and branch prediction. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a TDMA scheme, called memory wheel, similar to the TDMA-based arbiter used in the JOP CMP system [9]. The SPARC ISA is extended with a *deadline* instruction that stalls the current thread until the deadline is reached. This instruction is used to perform time-based, instead of lock-based, synchronization for access to shared data.

Berg et al. identify the following design principles for a time-predictable processor: "recoverability from information loss in the analysis, minimal variation of the instruction timing, noninterference between processor components, deterministic processor behavior, and comprehensive documentation" [10]. The authors propose a processor architecture that meets these design principles. The processor is a classic five-stage RISC pipeline with minimal changes in the instruction set: it handles function calls with an explicit instruction for simpler reconstruction of the control flow graph and construction of 32-bit immediate values with two instructions to avoid immediate values in the code segment. The memory system has to be organized in Harvard-style with dedicated busses to the FLASH memory for the code and the SRAM memory for the data. The replacement strategy of caches has to be least-recently used (LRU).

Heckmann et al. provide examples of problematic processor features in [11]. The most problematic features found are the replacement strategies for set-associative caches. A pseudo-round-robin replacement strategy of the 4-way set-associative cache in the ColdFire MCF 5307 effectively renders the associativity useless for WCET analysis. The use of a single 2-bit counter for the whole cache destroys age information within the cache sets. The analysis of that cache results in effectively modeling only a quarter of the cache as a direct-mapped cache. Similarly, a pseudo-LRU replacement strategy for an 8-way set-associative cache of the PowerPC 750/755 uses an age counter for each set. Here, only half of the cache is modeled by the analysis. Slightly more complex pipelines, with branch prediction and out-of-order execution, need an integrated pipeline and cache analysis to provide useful WCET bounds. Such integrated analysis is complex and also demanding with respect to the computational effort. In conclusion, Heckmann et al. suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches, (2) locally deterministic update strategies for caches, (3) static branch prediction, and (4) limited out-of-order execution. The authors discuss for restriction of processor features of actual processors (of the time) for embedded systems, but do not provide suggestions for *additional* or *alternative* features for a time-predictable processor.

The VISA approach [12] adapts a complex simultaneous multithreading processor that can be reconfigured to a simple single-issue pipeline. The complexity of the processor can be dynamically disabled at runtime. WCET analysis is performed for the simple pipeline. A task is divided into subtasks and each subtask is assigned a checkpoint. The task is executed on the complex pipeline and only if the checkpoint is missed, the processor is switched to the simple mode. The checkpoint is inserted early enough to complete the subtask on the simple pipeline before the deadline. The

available slack time, when the task is executed on the fast, complex pipeline, is utilized for energy saving.

Puschner and Burns discuss for a single-path programming style [13] that results in a constant execution time. In that case, WCET can easily be measured. However, this programming paradigm is quite uncommon and restrictive. Single-path programming can be inefficient when the control flow is data-dependent. A processor, called SPEAR [14], was especially designed to evaluate the single-path programming paradigm. A single predicate bit can be set with a compare instruction whereby several instructions (e.g., move, arithmetic operations) can be predicated. The SPEAR implements a three-stage in-order pipeline and uses onchip memories for instruction and data instead of caches.

Complex hardware and software architectures hinder hierarchical timing analysis [15]. A radical simplification of the whole system to avoid unwanted timing interactions is proposed—single path programming, execution of a single task/thread per core, simple in-order pipelines, and statically scheduled access to shared memory in CMPs.

Whitham discusses that the execution time of a basic block has to be independent of the execution history [16]. As a consequence, his MCGREP architecture reduces pipelining to two stages (fetch and execute) and avoids caches all together. To reduce the WCET, Whitham proposes to implement the time-critical functions in microcode on a reconfigurable function unit (RFU). The main processor implements an RISC ISA as a microprogrammed, sequential processor. The interesting approach in MCGREP is that the RFUs implement the same architecture and microcode as the main CPU. Therefore, mapping a sequence of RISC instructions to microcode for one or several RFUs is straightforward. With several RFUs, it is possible to explicitly extract instruction level parallelism (ILP) from the original RISC code in a similar way to VLIW architectures.

Whitham and Audsley extend the MCGREP architecture with a trace scratchpad [17]. The trace scratchpad caches microcode and is placed after the decode stage. It is similar to a trace cache found in newer Pentium CPUs to cache the translated micro-operations. The differences from a cache are that the execution from the trace scratchpad has to be explicitly started and the scratchpad has to be loaded under program control. The authors extract ILP at the microcode level and schedule the instructions statically—similar to a VLIW architecture.

3. WCET Analysis Issues

The WCET of tasks is the necessary input for schedulability analysis. Measuring the WCET is not a safe option. Only static WCET analysis can provide safe upper bounds of execution times.

WCET analysis can be separated into high-level and low-level analysis. The high-level analysis is a mature research topic [18–20]. An overview of WCET-related research can be found in [21, 22]. The main issues that need to be solved are in the low-level analysis. The processors that can be analyzed are usually several generations behind actual architectures [11, 23, 24] (e.g., Thesing models, in his Ph.D. thesis [25], the MPC755 variant of the PowerPC 750). The PowerPC 750 was introduced in 1997 and the MPC755 was *not recommended for new designs* in 2006.

The main issues in low-level analysis are features that increase average performance. All these features, such as multilevel caches, branch target buffer, out-of-order execution, and speculation, include a state that heavily depends on a large execution history. This caching of the execution history is actually fundamental for performance enhancements. However, it is the history which is hard to model for WCET analysis. A long history leads to a state explosion for the final WCET calculation. Low-level WCET analysis thus usually performs simplifications and uses conservative estimates. One example of this conservative estimate is to classify a cache access as a miss, if the outcome of the cache access is unknown.

Lundqvist and Stenström have shown that this intuitive assumption can be wrong on dynamically scheduled microprocessors [26]. They provide an example of such a timing anomaly in which a cache hit can cause a longer execution time than a cache miss. The principles behind these timing anomalies are further elaborated in [27].

3.1. Pipeline Dependencies. Simple pipelines, similar to the original Berkeley/Stanford RISC design [28], are easy to model for WCET analysis. In a nonstalled pipeline, the execution time latency corresponds to the length of the pipeline. The effective execution time itself is only a single cycle. What makes pipeline analysis necessary are stalls introduced by dependencies within the pipeline. Those stalls are introduced by

(1) data dependencies between instructions,

(2) control dependencies between instructions.

In one of the first RISC designs, the MIPS [29], these dependency hazards are explicitly exposed to the compiler. They have to be resolved by the compiler with instruction scheduling for delayed branches and for the single cycle delay between a memory load and the data use. Therefore, these effects are also recognized by the WCET tool. More advanced pipelines avoid exposing stalls from the ISA in order to avoid too many (compiler) target variations and retain binary compatibility between processor versions. Nevertheless, this information is needed for WCET analysis.

Dependencies within a basic block can be easily modeled. The challenge is to merge the effects from different basic blocks and across function boundaries. In [30], the *timing schema* [31] is extended to include the pipeline information. Timing schema is a tree-based WCET analysis. After the determination of basic block execution times, the control flow graph is processed in a bottom-up manner until a final WCET bound is available. Branches are merged with the higher WCET bound as result. For the extension, the pipeline is represented by reservation stations, and the state at the head and tail of a basic block is considered when basic blocks are merged.

Pipelines with timing dependencies can result in an unbounded effect, called *long timing effect* (LTE) [32]. This means that an instruction far back in the history (longer than the pipeline length) influences the execution time of the current instruction. These LTEs can be negative or positive. A positive LTE means longer execution time. An instruction with a possible positive LTE needs a safe approximation of that effect for the pipeline analysis.

More complex pipelines can be analyzed with abstract interpretation, but the analysis time can become impractical. Berg et al. [10] report that up to 1000 states per instruction are needed for the model of the PowerPC 755. This processor was introduced in 1998 and we expect a considerable growth of the states that need to be tracked by abstract interpretation for newer processors.

3.2. Instruction Fetch. The instruction fetching is often decoupled from the main memory or the instruction cache by a prefetch unit. This unit fills the prefetch queue with instructions independently of the main pipeline. This form of prefetching is especially important for a variable length instruction set as the x86 ISA or the bytecode instructions of the Java virtual machine (JVM). The fill status of the prefetch queue depends on the history of the instruction stream. The possible length of this history is unbounded. To model this queue for a WCET tool, we need to cut the maximum history and assume an empty queue at such a cut point.

In [33], the authors model the 4-byte-long prefetch queue of an Intel 80188. Even for this simple prefetch queue, the authors have to perform some simplifications in their approach to handle the resulting complexity due to the interaction between the instruction execution and the instruction prefetch (the consuming and the producing ends of the queue).

3.3. Caches. Between the middle of the 1980s and 2002, CPU performance increased by around 52% per year, but memory latency decreased only by 9% [1]. To bridge this growing gap between CPU and main memory performance, a memory hierarchy is used. Several layers with different tradeoffs between size, speed, and cost form that memory hierarchy. A typical hierarchy consists of

(1) register file,

(2) per-processor level 1 instruction and data cache,

(3) onchip, shared unified level 2 cache,

(4) offchip level 3 cache,

(5) main memory,

(6) hard disc for virtual memory.

The only layer under the control of the compiler is the register file. The rest of the memory hierarchy is usually not visible—it is not part of the ISA abstraction. Placement of data in the different layers is performed automatically by the hardware for caches and by the OS for virtual memory management. The access time for a word located in a memory block paged out by the OS is several orders of magnitude higher than a level 1 cache hit. Even the access times to the level 1 cache and to the main memory differ by two orders of magnitudes.

Cache memories for the instructions and data are classic examples of the *make the common case fast* paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Much effort has been expended on research to integrate the instruction cache into the timing analysis of tasks [34, 35], on the cache's influence on task preemption [36, 37], and on integration of the cache analysis with the pipeline analysis [38]. The influence of different cache architectures on WCET analysis is described in [11].

A unified cache for data and instructions can easily destroy all the information on abstract cache states. Access to n unknown addresses in an n-way set-associative cache results in the state *unknown* for all cache lines. Modern processors usually have separate instruction and data caches for the level 1 cache. However, the level 2 cache is usually shared. Most CMP systems also share the level 2 cache between the different cores. The possible interactions between concurrent threads running on different cores are practically impossible to model.

Data caches are considerably harder to analyze than instruction caches. For some data accesses, especially for data allocated on the heap, the addresses cannot be predicted. However, access to the stack can be predicted statically. A data cache that caches heap and stack content suffers from the same problem as a unified instruction and data cache: an unknown address for a heap access will evict one block from all sets in the abstract cache state and will increase the age of all cache blocks.

In a recent paper, Reineke et al. analyzed the predictability of different cache replacement policies [39]. It is shown that LRU performs best with respect to predictability. Pseudo-LRU and FIFO perform similarly, as both perform considerably worse than LRU. In an 8-way set-associative setting, pseudo-LRU and FIFO take more than twice as long as LRU to recover from lost information.

3.4. Branch Prediction. Accurate branch prediction is of utmost importance to keep long pipelines filled. The penalty of a wrongly predicted conditional branch is typically almost as long as the pipeline length. Modern branch predictors guess the outcome primarily from results of earlier branches. They heavily rely on the execution history, an effect we want to avoid for a tight worst-case prediction. Global branch predictors and caches have a similar issue: as soon as a single index into the branch history is unknown, the whole information of branch prediction is lost for the analysis at that point.

Two-level branch predictors are not suitable for time-predictable architectures [40]; for example, on the Pentium III, Pentium 4, and UltraSparc III, a decrease in the number of loop iterations can actually result in an increase of the execution time. This is another form of timing anomaly [26].

Branch prediction also interferes with cache contents. When the analysis cannot anticipate the outcome of the

prediction, both branch directions need to be considered for cache analysis.

3.5. Instruction Level Parallelism. Some microprocessors try to extract ILP from the instruction stream, that is, execute more than one instruction per clock cycle. ILP extractions can be done either statically by the compiler or dynamically by the hardware.

Processors with static-scheduled ILP are known as very long instruction word (VLIW) processors. The main issue of VLIW processors is that the pipeline details are exposed at the ISA. The compiler has to group parallel instructions and needs to consider pipeline constraints. Some processors rely on the compiler to resolve data dependencies and do not stall the pipeline. Therefore, each new generation of VLIW processors needs a new compiler back end. However, this issue is actually an advantage for low-level WCET analysis, as these details are needed for the pipeline analysis.

Dynamically scheduled, superscalar microprocessors combine several parallel execution units with an out-of-order execution to extract the ILP from the instruction stream. In current processors, about hundred instructions (e.g., 128 in the Pentium 4 [1]) can be in flight at each cycle. Analysis of a realistically sized application with an accurate processor model is thus (almost) impossible. Even modeling the pipeline states for basic blocks leads to a state space explosion; and modeling only basic blocks would result in very long penalties for the branches—on a later version of the Pentium 4, a simple instruction takes at least 31 clock cycles from fetch to retire [1].

Despite this complexity, in [41], a hypothetical out-of-order executing microprocessor is modeled for WCET analysis. Verification of the proposed approach on a real processor is missing. We think modeling out-of-order processors is practically not feasible.

3.6. Chip Multithreading. Dynamic extraction of ILP is limited to about two instructions per cycle on current processors, such as Pentium 4 and AMD Opteron [1]. Another path to speed up multithreaded workloads is the extraction of thread-level parallelism (TLP). The concept of TLP in a single processor is quite old—it was used in the CDC 6600, a supercomputer from the 1960s—but is now being reconsidered in all desktop and server processors. Fine-grained multithreading can hide the latency of load/use hazards or a cache miss for one thread by the execution of other threads.

The main issue with multithreading in real-time systems arises when the execution time of one thread depends on the state of a different thread. The main source of timing interactions in a CMP comes from shared caches and shared main memory. In the worst case, all latency hiding has to be ignored by the analysis and the sum of the execution times of several threads is the same as the serial execution on a single-threaded CPU. In addition, multithreaded processors usually share the level 1 caches. Therefore, each thread invalidates the abstract cache state of the other threads.

Dynamic ILP and TLP can be combined for simultaneous multithreading (SMT). With this technique, independent threads can be active in the same pipeline stage. This results in a higher utilization of processor resources that are already available for the ILP extraction. Modeling the fine-grained interaction of different SMT threads for WCET analysis seems, at least to the author, an intractable problem.

3.7. Chip Multiprocessors. Due to the power wall [1], CMP systems are now state-of-the-art in desktop and server processors. There are three different CMP systems: (1) multicore versions of superscalar architectures (Intel/AMD), (2) multicore chips with simple RISC processors (Sun Niagara), and (3) the cell architecture.

Mainstream desktop processors from Intel and AMD include two or four out-of-order executing processors. These processors are replications of the original, complex cores that share a level 2 cache and the memory bus. Cache coherence protocols on the chip keep the level 1 caches coherent and consistent. Furthermore, these cores also support SMT, sometimes also called hyperthreading.

Sun took a completely different approach with their Niagara T1 [42] by abandoning their superscalar architecture. The T1 contains 8 simple RISC cores, each supporting 4 threads, scheduled round-robin. 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 a single floating point unit which is shared by all 8 processors. Each core implements a six-stage, single-issue pipeline similar to the original five-stage RISC pipeline. Such a simple pipeline brings WCET analysis back into consideration.

The cell multiprocessor [43–45] takes an approach similar to a distributed memory multiprocessor. The cell contains, beside a PowerPC microprocessor, 8 synergistic processors (SPs). The SPs contain 256 KB onchip memory that is incoherent with the main memory. The PowerPC, the 8 SPs, and the memory interface are connected via a network consisting of four independent rings. Communication between the cores in the network has to be set up explicitly. All memory management, for example, transfer between SPs or between onchip memory and main memory, is under program control, resulting in a new programming model. The time-predictable memory access to the onchip memory and the in-order pipeline of the SPs should be a reasonable target for WCET analysis. The challenge is to include the explicit memory transfers between the cores and the main memory into the analysis.

Intel recently announced a CMP system named Larrabee [46]. Larrabee is intended as a replacement for graphic processing units from other vendors. It is notable that Intel uses several dual issue, in-order x86 cores. They discuss that for some workloads, in-order pipelines are more power efficient than out-of-order cores. The design is based on the first Pentium processor, enhanced with multithreading support and vector instructions.

The main source of timing interactions in a CMP comes from the shared level 2 (and probably level 3) cache and the shared main memory. The shared memory provides an

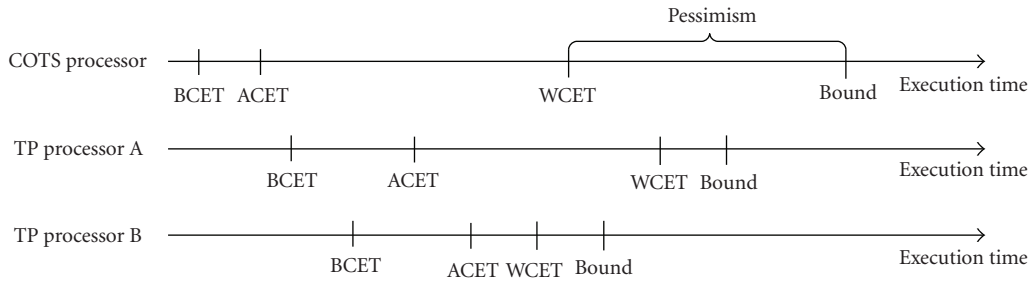

Figure 1: Distribution of the best-case, average-case, and worst-case execution times and the WCET bound of a task on different architectures. A time-predictable processor has less pessimism and the average-case execution time is not important.

easy-to-use programming model at the cost of unpredictable access time to the data. A shared level 2 cache is practically not analyzable due to the interthread interference. This is the same issue as with multithreading with a shared level 1 cache.

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.

Yan and Zhang analyze a shared instruction cache on a dual core system that executes two threads [47]. To restrict the set of conflicting cache blocks, they introduce the category *always-except one hit* for level 2 cache blocks. Assuming threads A and B, a cache block c is classified as *always-except one hit* for thread A when c is part of a loop in thread A, c conflicts with a block used by thread B, and the conflicting block in thread B is not part of a loop in thread B. However, the approach has two drawbacks: (1) for n threads/cores, several categories (up to $n - 1$) need to be introduced; (2) *not in a loop* is not a proper model for real-time threads as these are usually periodic.

The memory arbitration algorithm determines the worst-case access time to the main memory. Any fairness-based arbitration is, at least, difficult to integrate into WCET analysis. Priority-based arbitration can only guarantee access time for the core with the highest priority because lower priority cores can be blocked indefinitely.

3.8. Documentation. To model the processor for the low-level analysis, an accurate documentation of the processor internals is needed. However, this information is often not available or sometimes simply wrong [32]. For actual processors, the documentation of the internals is usually not disclosed. Over time, due to reverse engineering and less competition with other processors, more information becomes available. This is probably another reason why WCET analysis is about 10 years behind the processor technology.

3.9. Summary. While conventional techniques in designing processor architectures increase average throughput, they are not feasible for real-time systems. The influence of these architectural enhancements is at best hardly WCET analyzable. From a survey of the literature, we found that modeling a new version of a microprocessor and finding all undocumented details is usually worth a full Ph.D. thesis.

We discuss that trying to catch up on the analysis side with the growing complexity of modern computer architectures is not feasible. A paradigm shift is necessary. Computer architecture has to be redefined or adapted for real-time systems. Predictable and *analyzable* execution time is of primary importance.

4. Time-Predictable Architecture

We propose a computer architecture designed especially for real-time applications. We do not want to restrict features only, but we also want to actively add features that enhance performance and are time-predictable.

Figure 1 illustrates the aim of a time-predictable architecture, showing the distribution of the different execution times for a task: they are best-case execution time (BCET), average-case execution time (ACET), worst-case execution time (WCET), and the bound of the WCET that an analysis tool can provide. The difference between the actual WCET and the bound is caused by the pessimism of the analysis resulting from two factors: (a) certain information, for example, infeasible execution paths, not being known statically and (b) the simplifications to make the analysis computationally practical. For example, infeasible execution paths may significantly impact the WCET bound because the static analysis cannot prove that these paths may never be executed. Similarly, dynamic features such as speculative execution and pipelining often need to be modeled conservatively to prevent an explosion of the analysis complexity.

The first time line shows the distribution of the execution times for a commercial off-the-shelf (COTS) processor. The other two time lines show the distribution for two different time-predictable processors.

Variant A depicts a time-predictable processor with a higher BCET, ACET, and WCET than a standard processor. Although the WCET is higher than the WCET of the standard processor, the pessimism of the analysis is lower and the resulting WCET bound is lower as well. Even this type of processor is a better fit for hard real-time systems than today's standard processors.

Processor B shows an architecture where the BCET and ACET are further increased, but the WCET and the WCET

Table 1: Architectural issues for WCET analysis of standard processors and proposed architectural solutions.

	Standard processor WCET issues	Time-predictable processor
Pipeline	Dependencies and shared state	Simple pipeline
Instruction fetch	Unbounded timing effects	Avoid prefetch queue, use double buffer
Caches	Replacement policy, abstract cache state destruction by unknown addresses	Method cache, stack cache, and a highly associative, small heap cache
Branch prediction	Long history in dynamic predictors	Static branch prediction
Superscalar architectures	Timing anomalies	Avoid, instead use CMP and/or VLIW
Chip-multithreading	Interthread interference (cache, pipeline)	Avoid, instead use CMP
Chip-multiprocessors	Intercore interference via shared memory, cache	TDMA scheduled memory access

bound are decreased. Our goal is to design an architecture with a low WCET bound. For hard real-time systems, the likely increase in the ACET and BCET is acceptable because the complete system needs to be designed to reduce the WCET. It should be noted that a processor designed for low WCET will never be as fast in the average case as a processor optimized for ACET. Those are two different design optimizations. We define a time-predictable processor as "under the assumption that only feasible execution paths are analyzed, a time-predictable processor's WCET bound is close to the real WCET."

In the following, we propose time-predictable solutions or replacements, if possible, for the issues we identified in the Section 6. Table 1 summarizes the issues of standard processors for WCET analysis and the proposed architectural solutions.

4.1. Pipeline Dependencies. Pipelining is a major architectural feature to speed up program execution. Different stages of an instruction are overlapped and, therefore, executed in parallel. The theoretical throughput of a scalar pipeline is one instruction per clock cycle.

In contrast to Whitham [16], we think that a time-predictable architecture should be pipelined. The pipeline should be simple and dependencies between instructions avoided, or at least minimized, to avoid unbounded timing effects.

4.2. Instruction Fetch. To avoid a prefetch queue, with probably unbounded execution-time dependencies over a stream of instructions, a fixed-length instruction set is recommended. Variable length instructions can complicate instruction cache analysis because an instruction can cross a block boundary. The method cache, as proposed in the following section, avoids this issue. Either all instructions of a function, independent of their length, are in the cache, or none of them.

Fetching variable-sized instructions from the method cache can be performed in a single cycle. The method cache is split into two interleaved memories banks. Each of the two cache memories needs a read port wide enough for a maximum-sized instruction. Accessing both memories concurrently with a clever address calculation overcomes the boundary issue for variable-sized instruction access.

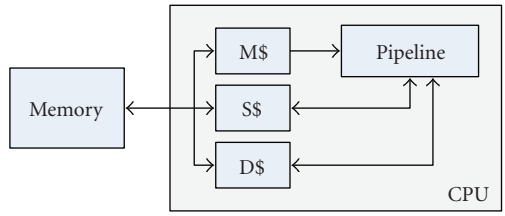

Figure 2: Cache configuration for the time-predictable architecture. The method cache M$ caches instructions of a full method/function. The data cache D$ is augmented by a stack cache S$ to avoid cache trashing of stack allocated data with heap allocated data.

4.3. Caches. To reduce the growing gap between the clock frequency of the processor and memory access times, multilevel cache architectures are commonly used. Since even a single-level cache is problematic for WCET analysis, more levels in the memory architecture are practically not analyzable. The additional levels also increase the latency of the memory access on a cache miss.

For the cache analysis, the addresses of the memory accesses need to be predicted. The addresses for the instruction fetch are easy to determine, and access to stack allocated data, for example, function arguments and local variables, is also quite regular. The addresses can be predicted when the call tree is known.

The addresses for heap-allocated data are very hard to predict statically—the addresses are only known during runtime (we found no publication that describes analysis of the data cache for heap-allocated data). Without knowing the address, a single access influences all sets in the cache.

To avoid corruption of the abstract cache state in the analysis by data accesses, separate instruction and data caches are mandatory [11]. Furthermore, we propose to split the data cache into a cache for stack-allocated data and a cache for global- or heap-allocated data. As stack allocated data is guaranteed thread local, the stack cache can be further simplified for CMP systems.

For the instruction cache, we propose a new form of organization where whole functions are loaded on a miss on call or return. Figure 2 shows the proposed organization of the three caches.

4.3.1. The Instruction Cache. We propose a new form of organization for the instruction cache: the *method cache* [3], which has a novel replacement policy. A whole function or method is loaded into the cache on a call or return. This cache fill strategy pools all the cache misses of a function. All instructions except call and return are guaranteed cache hits. Only the call tree needs to be analyzed during the cache analysis. With the proposed cache organization, the cache analysis can be performed independently of the pipeline analysis.

Filling the cache on call and return only removes another source of interference: there is no competition for the main memory access between instruction cache and data cache. In traditional architectures, there is a subtle dependency between the instruction cache and memory access for a load or store instruction. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instruction. With a data cache, this situation can be even worse. The worst-case scenario for the memory stall time for an instruction fetch or a data load is two miss-penalties when both cache reads are a miss.

The main restriction of the method cache is that a whole method needs to fit into the cache. For larger methods, software- and hardware-based options are possible to resolve this issue. The compiler can split large methods into several shorter methods. At the hardware level, there are two options for methods that are too large: the cache can be disabled or the method cache can be switched into a direct-mapped mode.

If we avoid absolute jumps within a method, we can use a relative program counter within the method and place a method at each position within the cache. This property is fulfilled with Java bytecode, but can also be enforced by the compiler for C code.

For a full method load into the cache, we need to know the length of the method. This information is available in the Java class file. For compiled C code, this information can be provided in the executable. A simple convention, implemented in the linker, is to store the method length one word before the actual method starts. In order to use the method cache in an RISC processor, the ISA is extended with a prefetch instruction to force the cache load. The prefetch instruction can be placed immediately before the call or return instruction. It can also be scheduled earlier to hide the cache load latency.

4.3.2. The Stack Cache. Access patterns to stack allocated data are different from heap- or static-allocated data. Addresses into the stack are easy to predict statically because the allocation addresses of stack frames can be predicted by the analysis of the call tree. Furthermore, a new stack frame for a function call does not need to be cache-consistent with the main memory. The involved cache blocks need no cache fill from the main memory.

To benefit from these properties for WCET analysis, we propose to split the data cache into a stack cache and a cache for static- and heap-allocated data (it is possible to further split the data cache into a cache for static data and heap data). The organization of the cache for static- and heap-allocated data, further referred to as data cache, will be proposed in the following section.

The regular access pattern to the stack cache will not benefit from set associativity. Therefore, the stack cache is a simple direct-mapped cache. The stack contains local variables and the write frequency is higher than for other memory areas. The high frequency mandates a write back organization.

A stack cache is similar to a windowed register file as implemented in the Berkeley RISC processor [28]. A stack cache can be organized to exchange data with the main memory on a stack frame basis. When the cache overflows, which happens only during a call, the oldest frame or frames have to be moved to the memory. A frame needs to be loaded from the memory only when a function returns. Exchange with the main memory can be implemented in hardware, microcode, or with compiler visible machine instructions.

If the maximum call depth results in a stack that is smaller than the stack cache, all accesses will be a cache hit. A write back occurs first when the program reaches a call depth resulting in a wrap around within the cache. A cache miss can occur only when the program goes up in the call tree and needs access to a cache block that was evicted by a call down in the call tree.

Figure 3 shows the call and return behavior of a program over time and the changing stack cache window. The stack grows downwards in the figure. The dashed box shows a possibility to enforce a write back at some program point. The following stack changes fit into the enforced stack window and no memory transactions are necessary.

On a return, the previously used cache blocks can be marked empty because function local data is not accessible after the return (it could be accessed in C by returning a pointer to the stack data; however, this is undefined and considered poor programming practic). As a result, cache lines will never need to be written back on a cache wrap around after return. The stack cache activity can be summarized in the following way.

(1) A cache miss can only occur after a return. The first miss is at least *one cache size* away from a leaf in the call tree.

(2) Cache write back can only occur after a function call. The first write back is *one cache size* away from the root of the call tree.

We can make the misses and write backs more predictable by forcing them to occur at explicit points in the call tree. At these points, the cached stack frames are written back to the main memory and the whole stack cache is marked empty. If we place the flush points at function calls in the call tree that are within *one cache size* from the leaf functions, all cache accesses into that area are guaranteed hits. This algorithm can actually improve WCET because most of the execution time of a program is spent in inner loops further down the call tree.

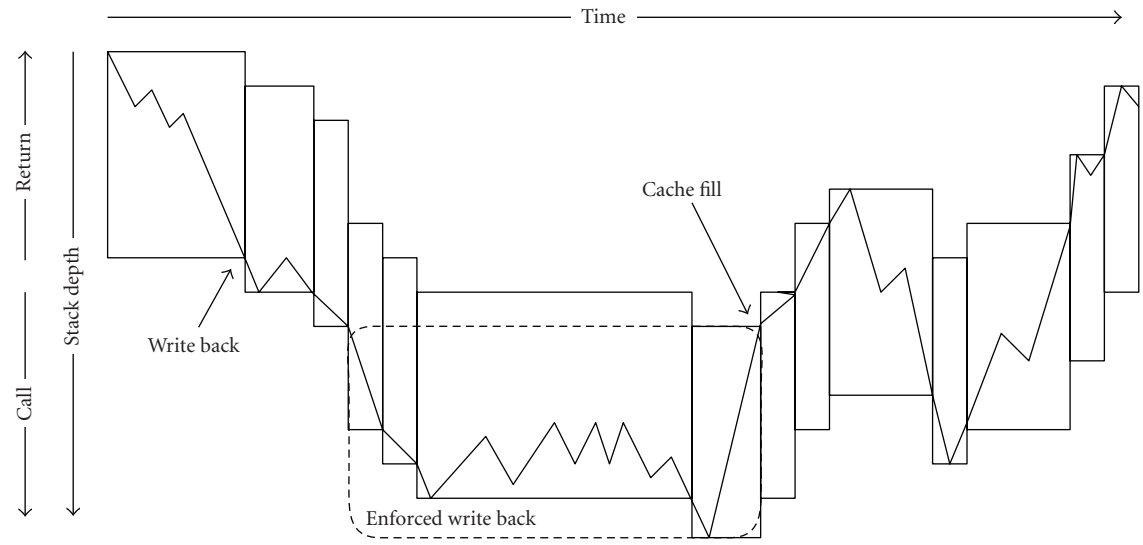

Figure 3: Stack usage for call and return and the resulting stack cache window. When the window overflows on a call, a write back of old frames is necessary. The stack cache fill is caused by an underflow after a return. Enforcing a write back of the whole stack cache can guarantee hits for subsequent, more deeply nested functions.

Stack data is usually not shared between threads and no cache coherence and consistence protocol—the major bottleneck for CMP scaling—needs to be implemented for a CMP system.

4.3.3. The Data Cache. For conservatively written programs with statically allocated data, the address of the data is known after program linking. Value analysis results in a good prediction of read and write addresses. The addresses are the input for the cache analysis. In [48], control tasks, from a real-time benchmark provided by Airbus, were analyzed. For this benchmark, 90% of the memory accesses were predicted precisely.

In a modern object-oriented language, data is usually allocated on the heap. The address for these objects is only known at runtime. Even when using such a language in a conservative style, where all data is allocated during an initialization phase, it is not easy to predict the resulting addresses. The order of the allocations determines the addresses of the objects. When the order becomes unknown at one point in the initialization phase, the addresses for all following allocations cannot be determined precisely.

It is possible to analyze local cache effects with unknown addresses for an LRU set-associative cache. For an n-way associative cache, the history for n different addresses can be tracked. Because the addresses are unknown, a single access influences *all* sets in the cache. The analysis reduces the effective cache size to a single set.

The local analysis for the LRU-based cache is illustrated by a small example with a four-word cache. The example cache allocates a cache block on a write. Table 2 shows a code fragment with access to heap-allocated data (objects a, b, c, and d). The cache state after the load or store instruction is shown in the right section of the table. The leftmost column of the cache state represents the youngest element, the rightmost column the oldest (the LRU element). We assume a 4-way set-associative cache for the example. Therefore, we can locally track four different and *unknown* addresses. After the first two constant assignments, we know that a.v and b.v are in the cache. The following load of b.v is trivially a hit and the store into c.v changes the cache content and the age of a.v and b.v. All following loads are hits and only change the age ordering of the cache elements. In this small example we dealt with four different and unknown addresses, but could classify all read accesses as hits for a four-word cache.

Table 2: An example of analyzable accesses to three heap-allocated objects with a four-word LRU cache. The cache content after the execution of a statement is depicted in the right section of the table.

		Cache state			
Instruction	Memory	Youngest			Oldest
a.v = 123;	store a.v	a.v	—	—	—
b.v = 456;	store b.v	b.v	a.v	—	—
c.v = b.v;	load b.v	b.v	a.v	—	—
	store c.v	c.v	b.v	a.v	—
d.v = b.v;	load b.v	b.v	c.v	a.v	—
	store d.v	d.v	b.v	c.v	a.v
b.v = a.v;	load a.v	a.v	d.v	b.v	c.v
	store b.v	b.v	a.v	d.v	c.v

We propose to implement the cache architecture exactly as it results from this analysis—a small, fully associative cache with an LRU replacement policy. This cache organization is similar to the victim cache [49], which adds associativity to a direct-mapped cache. A small, fully associative buffer holds discarded cache blocks. The replacement policy is LRU.

LRU is difficult to calculate in hardware and only possible for very small sets. Replacement of the oldest block gives an

approximation of LRU. The resulting FIFO strategy can be used for larger caches. To offset the less predictable behavior of the FIFO replacement [39], the cache has to be much larger than an LRU-based cache.

4.3.4. The Scratchpad Memory. A common method for avoiding data caches is an onchip memory called scratchpad memory, which is under program control. This program managed memory entails a more complicated programming model although it can be automatically partitioned [50, 51]. A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [52] and the allocation of data in the scratchpad memory [53, 54] can be optimized for the WCET. A comparison between locked cache blocks and a scratchpad memory with respect to the WCET can be found in [55].

Exposing the scratchpad memory at the language level can further help to optimize the time-critical path of the application.

4.4. Branch Prediction. As the pipelines of current general-purpose processors become longer to support higher clock rates, the penalty of branches also increases. This is compensated by branch prediction logic with branch target buffers. However, the upper bound of the branch execution time is the same as without this feature.

Simple static branch prediction (e.g., backward branches are assumed taken, forward branches not taken) or compiler-generated branch predictions are WCET-analyzable options. One-level dynamic branch predictors can be analyzed [56]. The branch history table has to be separate from the instruction cache to allow independent modeling for the analysis.

4.5. Instruction Level Parallelism. Statically scheduled VLIW processors are an option for a time-predictable architecture. The balance between the VLIW width and the number of cores in a CMP system depends on the application domain. For control-oriented applications, we assume that a dual-issue VLIW is a practical architecture. DSP-related applications can probably fill more instruction slots with useful instructions.

Dynamically scheduled superscalar architectures are not considered as an option for a time-predictable architecture. The amount of hardware that is needed to extract ILP from a single thread is better spent on a (VLIW-based) CMP system.

4.6. Chip Multithreading. Fine-grained multithreading within the pipeline is in principle not an issue for WCET analysis. The scheduling algorithm of the threads needs to be known and must not depend on the state of the threads. Round-robin scheduling is a time-predictable option. The execution time for simple instructions simply increases by a factor equal to the number of threads. The benefit of hiding pipeline stalls due to data dependencies or branches results in a lower factor for these instructions. Execution of n tasks on an n-way multithreading pipeline takes less (predictable) time than executing these tasks serially on a single threaded processor. However, cache misses, even if a single cache miss could be hidden, result in interference between the different threads because the memory interface is a shared resource.

Fine-grained multithreading resolves the data dependencies for a thread within the pipeline: the thread is only active in a single pipeline stage. Therefore, the forwarding network can be completely removed from the processor. This is an important simplification of the pipeline because the forwarding multiplexer is often part of the critical path that restricts the maximum clock frequency.

To avoid cache thrashing, each thread needs—in addition to its own register file—its own instruction and data cache, which reduces the effectively shared transistors to the pipeline itself. We think that the cost is too high for the small performance enhancement. Therefore, also duplicating the pipeline—resulting in a CMP solution—will result in a better performance/cost factor.

SMT is not an option as the interaction between the threads is too complex to model.

4.7. Chip Multiprocessors. Embedded applications need to control and interact with the *real*-world, a task that is inherently parallel. Therefore, these systems are good candidates for CMPs. We discuss that the transistors required to implement superscalar architectures are better used on complete replication of simple cores.

CMP systems share the access bandwidth to the main memory. To build a time-predictable CMP system, we need to schedule the access to the main memory in a predictable way. A predictable scheduling can only be time-based, where each core receives a fixed time slice. This scheduling scheme is called time division multiple access (TDMA). The time slices do not need to be of equal size. The execution time of uncached loads and stores and the cache miss-penalty depend on this schedule and, therefore, for accurate WCET analysis, the complete schedule needs to be known.

Assuming that enough cores are available, we propose a CMP model with a single thread per processor. In that case, thread switching and schedulability analysis for each individual core disappears. Since each processor executes only a single thread, the WCET of that thread can be as long as its deadline. When the period of a thread is equal to its deadline, 100% utilization of that core is feasible. For threads that have enough slack time left, we can increase the WCET by decreasing their share of the bandwidth on the memory bus. Other threads with tighter deadlines can, in turn, use the freed bandwidth and run faster. The usage of the shared resource main memory is adjusted by the TDMA schedule. The TDMA schedule itself is the input for WCET analysis for all threads. Finding a TDMA schedule, where all tasks meet their deadlines, is thus an iterative optimization problem.

Figure 4 shows the analysis tool flow for the proposed time-predictable CMP with three tasks. First, an initial arbiter schedule is generated, for example, one with equal time slices. That schedule and the tasks are the input of WCET analysis performed for each task individually. If all tasks meet their deadline with the resulting WCETs, the system is schedulable. If some tasks do not meet their

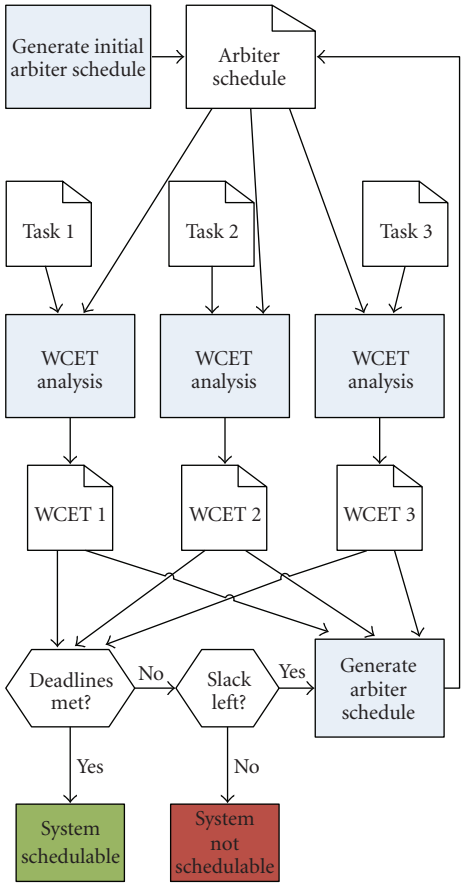

Figure 4: Tool flow for a CMP-based real-time system with one task per core and a static arbiter schedule. If the deadlines are not met, the arbiter schedule is adapted according to the WCETs and deadlines of the tasks. After the update of the arbiter schedule, the WCET of all tasks needs to be recalculated.

deadline and other tasks have some slack time available, the arbiter scheduler is adapted accordingly. WCET analysis is repeated, with the new arbiter schedule until all tasks meet their deadlines or no slack time for an adaption of the arbiter schedule is available. In the latter case, no schedule for the system is found.

4.8. Documentation. The hardware description language VHDL was originally developed to document the behavior of digital circuits. Today, digital hardware can be synthesized from a VHDL description. Therefore, the VHDL code for the processor is the ideal form of documentation. VHDL code can also be simulated and all interactions between different components are observable.

An open-source design enables the WCET tool provider to check the real processor when the documentation is missing; documentation errors are also easier to find. Sun provides the Verilog source of their Niagra T1 [42] as open-source under the GNU GPL (http://www.opensparc.net/opensparc-t1/downloads.html).

5. Evaluation

In this section, we evaluate some of the proposed time-predictable architectural features with JOP [2], an implementation of a Java processor. We have chosen to natively support Java as it is the language which will be used for future safety critical systems [57, 58]. Java's intermediate representation, the Java class file, is analysis friendly and the type information can be reconstructed from the class file. Executing bytecodes—the instruction set of the Java virtual machine (JVM)—directly in the hardware allows WCET analysis at the bytecode level. The translation step from bytecode to machine code, which introduces timing inaccuracies, can be avoided.

5.1. The Java Processor JOP. The major design goal of JOP is the time-predictable execution of Java bytecodes [59]. All functional units, and especially the interactions between them, are carefully designed to avoid any timing dependency between bytecodes.

JOP dynamically translates the Java bytecodes to a stack-based microcode that can be executed in a three-stage pipeline. The translation takes exactly one cycle per bytecode. Compared to other forms of dynamic code translation, the scheme used in JOP does not add any variable latency to the execution time and is, therefore, time-predictable.

JOP contains a simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed. The short pipeline (four stages) results in short conditional branch delays; a difficult to analyze branch prediction logic or a branch target buffer can be avoided.

All microcode instructions have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. Loads and stores of object fields are handled explicitly. The absence of timing dependencies between bytecodes results in a simple processor model for the low-level WCET analysis.

The proposed architecture is open-source and all design files are available (http://www.jopdesign.com/). The instruction timing of the bytecodes is documented.

5.1.1. Method Cache. JOP contains the proposed method cache. The default configuration is 4 KB, divided into 16 blocks of 256 Bytes. The replacement strategy is FIFO.

WCET analysis of the method cache and of standard instruction caches is currently under development. Therefore, we perform only average-case measurements for a comparison between a time-predictable cache organization and a standard cache organization. With a simulation of JOP, we measure the cache misses and miss-penalties for different configurations of the method cache and a direct-mapped cache. The miss-penalty and the resulting effect on the execution time depend on the main memory system. Therefore, we simulate three different memory technologies: static RAM (SRAM), synchronous DRAM (SDRAM), and double data rate (DDR) SDRAM. For the SRAM, a latency of 1 clock cycle and an access time of 2 clock cycles per 32 bit

Table 3: Direct-mapped cache, average memory access time.

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	8 B	**0.18**	0.25	0.19
1 KB	16 B	0.22	**0.22**	0.16
1 KB	32 B	0.31	0.24	**0.15**
2 KB	8 B	**0.11**	0.15	0.12
2 KB	16 B	0.14	**0.14**	**0.10**
2 KB	32 B	0.22	0.17	0.11

Table 4: Method cache, average memory access time.

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	16 B	0.36	0.21	0.12
1 KB	32 B	0.36	0.21	0.12
1 KB	64 B	0.36	0.22	0.12
1 KB	128 B	0.41	0.24	0.14
2 KB	32 B	0.06	0.04	0.02
2 KB	64 B	0.12	0.08	0.04
2 KB	128 B	0.19	0.11	0.06
2 KB	256 B	0.37	0.22	0.13

word are assumed. For the SDRAM, a latency of 5 cycles (3 cycles for the row address and 2 cycles for the CAS latency) is assumed. The SDRAM delivers one word (4 bytes) per cycle. The DDR SDRAM has a shorter latency of 4.5 cycles and transfers data on both the rising and falling edges of the clock signal.

The resulting miss-cycles are scaled to the bandwidth consumed by the instruction fetch unit. The result is the number of cache fill cycles per fetched instruction byte: in other words, the average main memory access time in cycles per instruction byte. A value of 0.1 means that for every 10 fetched instruction bytes, one clock cycle is spent to fill the cache.

Table 3 shows the result for different configurations of a direct-mapped cache. For the evaluation, we used an adapted version of the real-time application Kfl (the Kfl benchmark is also used in Section 5.4), which is a node in a distributed control application. As the embedded application is quite small (1366 LOC), we simulated small instruction caches. The best performing configuration depends on the relationship between memory bandwidth and memory latency. The data in bold emphasize the best block size for the different memory technologies. As expected, memories with a higher latency and bandwidth perform better with larger block sizes. For small block sizes, the latency clearly dominates the access time. Although the SRAM has half the bandwidth of the SDRAM and a quarter of the DDR SDRAM, it is faster than the SDRAM memories with a block size of 8 byte. In most cases, a block size of 16 bytes is fastest.

Table 4 shows the average memory access time per instruction byte for the method cache. Because we load full methods, we have chosen larger block sizes than for a standard cache. All configurations benefit from a memory system with a higher bandwidth. The method cache is less latency sensitive than the direct-mapped instruction cache. For the small 1 KB cache, the access time is almost independent of the block size. The capacity misses dominate. From the 2 KB configuration, we see that smaller block sizes result in less cache misses. However, smaller block sizes result in more hardware for the hit detection since the method cache is in effect fully associative. Therefore, we need a balance between the number of blocks and the performance.

The cache conflict is high for the small configuration with 1 KB cache. The direct-mapped cache, backed up with a low-latency main memory, performs better than the method cache. When high-latency memories are used, the method cache performs better than the direct-mapped cache. This is expected as the long latency for a transfer is amortized when more data (the whole method) is filled in one request.

A small block size of 32 bytes is needed in the 2 KB method cache to outperform the direct-mapped cache with the low-latency main memory as represented by the SRAM. For higher latency memories (SDRAM and DDR), a method cache with a block size of 128 bytes outperforms the direct-mapped instruction cache.

The comparison does not show if the method cache is more easily predictable than other cache solutions. It shows that caching full methods performs similarly to standard caching techniques.

5.1.2. Stack Cache. In JOP, a simplified version of the proposed stack cache is implemented. The JVM uses the stack not only for the activation frame and for local variables but also for operands. Therefore, the two top elements of the stack are implemented as registers [4]. With this configuration, we can avoid the write-back pipeline stage.

The fill and spill between the stack cache and the main memory is simplified. The cache content is exchanged only on a thread switch. Therefore, the maximum call depth is restricted by the onchip cache size. In a future version of JOP, we intend to relax this limitation. The cache fill will be performed on a return and the write back on invoke when necessary. A stack analysis tool will add a marker to the methods where a full cache write back will be performed and the stack access in methods deeper in the call tree will be guaranteed hits. Heap-allocated data and static fields are not cached in the current implementation of JOP.

5.1.3. Branch Prediction. In JOP, branch prediction is avoided. This results in pressure on the pipeline length. The microprogrammed core processor has a pipeline length of as little as three stages resulting in a branch execution time of three cycles in microcode. The two slots in the branch delay can be filled with instructions or *nop*. With the additional bytecode fetch and translation stage, the overall pipeline is four stages and results in a four-cycle execution time for a bytecode branch.

5.2. WCET Analysis. Bytecode instructions that do not access memory have a constant execution time. Most simple bytecodes are executed in a single cycle. Table 5 shows

example instructions and their timing. The access time to object, array, and class fields depends on the timing of the main memory. With a memory with r_{ws} wait states for a read access, the execution time for, for example, getfield, is

$$t_{\text{getfield}} = 11 + 2r_{ws}. \quad (1)$$

To demonstrate that JOP is amenable to WCET analysis, we have built an IPET-based WCET analyzer [60]. While loop bounds are annotated at the source level, the analysis is performed at the bytecode level. Without dependencies between bytecodes, the pipeline analysis can be omitted. The execution time of basic blocks is calculated simply by adding the execution time of individual bytecodes. For the method cache, we have implemented a simplified analysis where only leaf nodes in the call tree are considered. A return from such a leaf node is a guaranteed hit. (The maximum method size is restricted to half of the cache size.) Invocation of a leaf node in a tight loop (without invocations of other methods) is classified as a miss for the first iteration and a hit for the following iterations. For small benchmarks, the overestimation of the WCET is around 5%. For two real applications (Lift and Kfl) the analysis resulted in an overestimation of 56% and 116%. It should be noted that the overestimation is calculated by comparison with measurement-based WCET estimation, which is not a safe approach.

Another indication that JOP is a WCET *friendly* design is that other real-time analysis projects use JOP as the primary target platform. Harmon has developed a tree-based WCET analyzer for interactive back-annotation of WCET estimates into the program source [61]. Bøgholm et al. have developed an integrated WCET and scheduling analysis tool based on model checking [62].

5.3. Comparison with picoJava. We compare the time-predictable JOP design with picoJava [63, 64], a Java processor designed for average-case performance. Simple bytecodes are directly supported by the processor. Most of them execute in a single cycle. More complex bytecodes trap to a software routine. However, the invocation time of the trap depends on the cache state and is between 6 cycles in the best case and 426 cycles in the worst case—a factor in the order of two magnitudes. Some of the trapped instructions (e.g., invokevirtual) can be replaced at runtime by a *quick* version (e.g., invokevirtual_quick). This replacement results in different execution times for the first execution of some code and following executions.

To speed up sequences of stack operations, picoJava can fold several instructions into an RISC style three register operation, for example, the sequence: load, load, add, store. This feature compensates for the inefficiency of a stack machine. However, the folding unit depends on a 16-byte instruction buffer with all the resulting unbounded timing effects of a prefetch queue.

However, picoJava implements a 64-word stack buffer as discrete registers. Spill and fill of that stack buffer is performed in background by the hardware. Therefore, the stack buffer closely interacts with the data cache. The interference between the folding unit, the instruction buffer, the instruction cache, the stack buffer, the data cache, and the memory interface causes complications in modeling picoJava for WCET analysis.

Also, picoJava is about 8 times larger than JOP and can be clocked at less than half of the frequency of JOP in the same technology [65]. Therefore, the small size of a time-predictable architecture naturally leads to a CMP system.

Table 5: Execution time of simple bytecodes in cycles.

Instruction	Cycles	Funtion
iconst_0	1	load constant 0 on TOS
bipush	2	load a byte constant on TOS
iload_0	1	load local variable 0 on TOS
iload	2	load a local variable on TOS
dup	1	duplicate TOS
iadd	1	integer addition
isub	1	integer subtraction
ifeq	4	conditional branch

Table 6: Application benchmark performance on different Java systems. The table shows the benchmark results in iterations per second—a higher value means higher performance.

	Kfl	UdpIp	Lift
Cjip	176	91	
Jamuth	3400	1500	
EJC	9893	2882	
SHAP	11570	5764	12226
aJ100	14148	6415	
JOP	18275	8467	18649
picoJava	23813	11950	25444
CACAO/YARI	39742	17702	38437

5.4. Performance. One important question remains: is a time-predictable processor slow? We evaluate the average-case performance of JOP by comparing it with other embedded Java systems: Java processors from industry and academia as well as two just-in-time (JIT) compiler-based systems. For the comparison, we use JavaBenchEmbedded, (available at http://www.jopwiki.com/JavaBenchEmbedded) a set of open-source Java benchmarks for embedded systems. However, Kfl and Lift are two real-world applications adapted with a simulation of the environment to run as stand-alone benchmarks. Udplp is a simple client/server test program that uses a TCP/IP stack written in Java.

Table 6 shows the raw data of the performance measurements of different embedded Java systems for the three benchmarks. The numbers are iterations per second whereby a higher value represents better performance. Figure 5 shows the results scaled to the performance of JOP.

The numbers for JOP are taken from an implementation in the Altera Cyclone FPGA [66], running at 100 MHz. JOP is configured with a 4 KB method cache and a 1 KB stack cache.

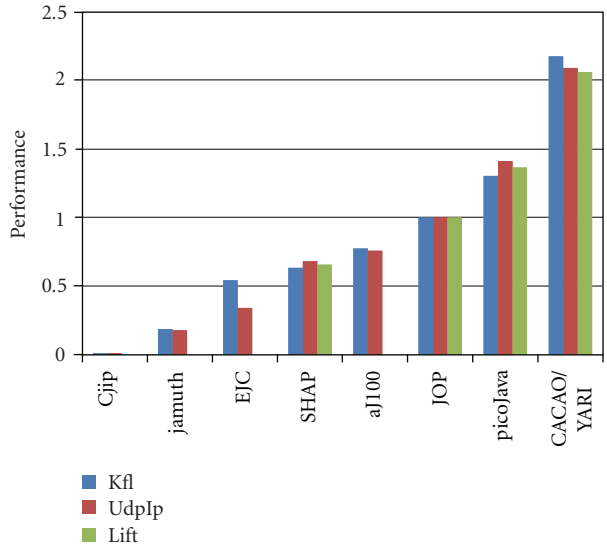

Figure 5: Performance comparison of different Java systems with embedded application benchmarks. The results are scaled to the performance of JOP.

Table 7: Resource consumption and maximum operating frequency of JOP, YARI, and picoJava.

Soft-core	Logic cells	Memory	Frequency
JOP	3,300	7.6 KB	100 MHz
YARI	6,668	18.9 KB	75 MHz
picoJava	27,560	47.6 KB	40 MHz

Cjip [67] and aJ100 [68] are commercial Java processors, which are implemented in an ASIC and clocked at 80 and 100 MHz, respectively. Both cores do not cache instructions. However, aj100 contains a 32 KB onchip stack memory. jamuth [69] and SHAP [70] are Java processors that are implemented in an FPGA. Also, jamuth is the commercial version of the Java processor Komodo [71], a research project for real-time chip multithreading; and it is configured with a 4 KB direct-mapped instruction cache for the measurements. The architecture of SHAP is based on JOP and enhanced with a hardware object manager. Also, SHAP implements the method cache [72]. The benchmark results for SHAP are taken from the SHAP website (http://shap.inf.tu-dresden.de/, accessed July, 2008); and it is configured with a 2 KB method cache and 2 KB stack cache.

Also, picoJava [73] is a Java processor developed by Sun. picoJava is no longer produced and the second version (picoJava-II) was available as open-source Verilog code. Puffitsch implemented picoJava-II in an FPGA (Altera Cyclone-II) and the performance numbers are obtained from that implementation [65]. picoJava is configured with a direct-mapped instruction cache and a 2-way set-associative data cache. Both caches are 16 KB.

EJC [74] is an example of a JIT system on an RISC processor (32-bit ARM720T at 74 MHz). The ARM720T contains an 8 KB unified cache. To compare JOP with a JIT-based system in exactly the same hardware, we use the research JVM CACAO [75] on top of the MIPS compatible soft-core YARI [76]. YARI is configured with a 4-way set-associative instruction cache and a 4-way set-associative write-through data cache. Both caches are 8 KB.

The measurements do not provide a clear answer to the question of whether a time-predictable architecture is slow. JOP is about 33% faster than the commercial Java processor aJ100. However, picoJava is 36% faster than JOP and the JIT/RISC combination is about 111% faster than JOP. (The numbers of CACAO/YARI are from [76]. In the mean time, YARI has been enhanced and outperforms JOP by a factor of 2.8.) We conclude that a time-predictable solution will never be as fast in the average case as a solution optimized for the average case.

5.5. Hardware Area and Clock Frequency. Table 7 compares the resource consumption and maximum clock frequency of a time-predictable processor (JOP), a standard MIPS architecture (YARI), and a complex Java processor (picoJava), when implemented in the same FPGA. The streamlined architecture of JOP results in a small design: JOP is half the size of the MIPS core YARI, and compared to picoJava consumes about 12% of the resources. JOP's size allows implementing a CMP version of JOP even in a low-cost FPGA. The simple pipeline of JOP achieves the highest clock frequency of the three designs. From the frequency comparison, we can estimate that the maximum clock frequency of JOP in an ASIC will also be higher than a standard RISC pipeline in an ASIC.

5.6. JOP CMP System. We have implemented a CMP version of JOP with a fairness-based arbiter [77]. All cores are allotted an equal share of the memory bandwidth. Each core has its own method cache and stack cache. Heap-allocated data is not cached in this design.

When comparing a JOP CMP system against the complex Java processor picoJava, a dual core version of JOP is about 5% slower than a single picoJava core, but consumes only 22% of the chip resources. With four cores, JOP outperforms picoJava by 30% with size of 43% of picoJava.

A configurable TDMA arbiter for a time-predictable CMP system and the integration of the arbitration schedule into the WCET tool [60] is presented in [9].

5.7. Summary. A model of a processor with accurate timing information is essential for tight WCET analysis. The architecture of JOP and the microcode are designed with this in mind. Execution time of bytecodes is known cycle accurately [59]. It is possible to analyze the WCET at the bytecode level [78] without the uncertainties of an interpreting JVM [79] or generated native code from ahead-of-time compilers for Java.

6. Conclusion

In this paper, we discuss a time-predictable computer architecture for embedded real-time systems that supports

WCET analysis. We have identified the problematic microarchitecture features of standard processors and provided alternative solutions when possible.

Dynamic features, which model a large execution history, are problematic for WCET analysis. Especially interferences between different features result in a state space explosion for the analysis. The proposed architecture is an in-order pipeline with minimized instruction dependencies. The cache memory consists of a method cache containing whole methods and a data cache that is split for stack-allocated data and heap-allocated data. The pipeline can be extended to a dual-issue pipeline when the instructions are compiler scheduled. For further performance enhancements, we propose a CMP system with time-sliced arbitration of the main memory access. Running each task on its own core in a CMP system eliminates scheduling, and the related cache thrashing, from the analysis. The schedule of the memory access becomes an input for WCET analysis. With nonuniform time slices, the arbiter schedule can be adapted to balance the utilization of the individual cores.

The concept of the proposed architecture is evaluated by a real-time Java processor, called JOP. We have presented a brief overview of the architecture. A simple four-stage pipeline and microcoded implementation of JVM bytecodes result in a time-predictable architecture. The proposed method and stack caches are implemented in JOP. The resulting design makes JOP an easy target for the low-level WCET analysis of Java applications.

We compared JOP against several embedded Java systems. The result shows that a time-predictable computer architecture does not need to be slow. A streamlined, time-predictable processor design is quite small. Therefore, we can regain performance by the exploration of thread level parallelism in embedded applications with a replication of the processor in a CMP architecture.

The proposed processor has been used with success to implement several commercial real-time applications [80]. JOP is open-source under the GNU GPL, and all design files and the documentation are available at http://www.jopdesign.com/.

We plan to implement some of the suggested architectural enhancements in an RISC-based system in the future. We will implement the proposed stack cache and the method cache in YARI [76], an open-source, MIPS ISA-compatible RISC implementation in an FPGA.

A scratchpad memory for JOP is implemented and the integration into the programming model is under investigation. We will add a small fully associative data cache to JOP. This cache will also serve as a buffer for a real-time transactional memory for the JOP CMP system. We will investigate whether a standard cache for static data is a practical solution for Java.

Acknowledgment

The author thanks Wolfgang Puffitsch and Florian Brandner for the productive discussions on the topic and suggestions for improving the paper.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 4th edition, 2006.

[2] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 265–286, 2008.

[3] M. Schoeberl, "A time predictable instruction cache for a Java processor," in *Proceedings of the On the Move to Meaningful Internet Systems: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '04)*, vol. 3292 of *Lecture Notes in Computer Science*, pp. 371–382, Springer, Agia Napa, Cyprus, October 2004.

[4] M. Schoeberl, "Design and implementation of an efficient stack machine," in *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW '05)*, Denver, Colo, USA, April 2005.

[5] I. Bate, P. Conmy, T. Kelly, and J. McDermid, "Use of modern processors in safety-critical applications," *The Computer Journal*, vol. 44, no. 6, pp. 531–543, 2001.

[6] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[7] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proceedings of the 44th ACM/IEEE Annual Conference on Design Automation (DAC '07)*, pp. 264–265, ACM Press, San Diego, Calif, USA, June 2007.

[8] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08)*, E. R. Altman, Ed., pp. 137–146, ACM Press, Atlanta, Ga, USA, October 2008.

[9] C. Pitter, "Time-predictable memory arbitration for a Java chip-multiprocessor," in *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '08)*, pp. 115–122, Santa Clara, Calif, USA, September 2008.

[10] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, L. Thiele and R. Wilhelm, Eds., vol. 03471 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungsund Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[11] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[12] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pp. 350–361, ACM Press, San Diego, Calif, USA, June 2003.

[13] P. Puschner and A. Burns, "Writing temporally predictable code," in *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '02)*, pp. 85–94, IEEE Computer Society, San Diego, Calif, USA, January 2002.

[14] M. Delvai, W. Huber, P. Puschner, and A. Steininger, "Processor support for temporal predictability—the SPEAR design example," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS '03)*, pp. 169–176, Porto, Portugal, July 2003.

[15] P. Puschner and M. Schoeberl, "On composable system timing, task timing, and WCET analysis," in *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Prague, Czech Republic, July 2008.

[16] J. Whitham, *Real-time processor architectures for worst case execution time reduction*, Ph.D. thesis, University of York, York, UK, 2008.

[17] J. Whitham and N. Audsley, "Using trace scratchpads to reduce execution times in predictable real-time architectures," in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pp. 305–316, St. Louis, Mo, USA, April 2008.

[18] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCTES '95)*, pp. 88–98, ACM Press, La Jolla, Calif, USA, November 1995.

[19] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times—a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[20] J. Gustafsson, *Analyzing execution-time of object-oriented programs using abstract interpretation*, Ph.D. thesis, Uppsala University, Uppsala, Sweden, 2000.

[21] P. Puschner and A. Burns, "Guest editorial: a review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, 2000.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, et al., "The worst-case execution-time problem-overview of methods and survey of tools," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[23] J. Engblom, A. Ermedahl, M. Södin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 437–455, 2003.

[24] K. D. Nilsen and B. Rygg, "Worst-case execution time analysis on modern processors," *ACM SIGPLAN Notices*, vol. 30, no. 11, pp. 20–30, 1995.

[25] S. Thesing, *Safe and precise worst-case executiontime prediction by abstract interpretation of pipeline models*, Ph.D. thesis, University of Saarland, Saarland, Germany, 2004.

[26] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pp. 12–21, IEEE Computer Society, Phoenix, Ariz, USA, December 1999.

[27] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder, "Principles of timing anomalies in superscalar processors," in *Proceedings of the 5th International Conference on Quality Software (QSIC '05)*, pp. 295–303, IEEE Computer Society Press, Melbourne, Australia, September 2005.

[28] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985.

[29] J. L. Hennessy, "VLSI processor architecture," *IEEE Transactions on Computers*, vol. 33, no. 12, pp. 1221–1246, 1984.

[30] S.-S. Lim, Y. H. Bae, G. T. Jang, et al., "An accurate worst case timing analysis for RISC processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 7, pp. 593–604, 1995.

[31] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 875–889, 1989.

[32] J. Engblom, *Processor pipelines and static worst-case execution time analysis*, Ph.D. thesis, Uppsala University, Uppsala, Sweden, 2002.

[33] N. Zhang, A. Burns, and M. Nicholson, "Pipelined processors and worst case execution times," *Real-Time Systems*, vol. 5, no. 4, pp. 319–343, 1993.

[34] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," in *Proceedings of the Real-Time Systems Symposium (RTSS '94)*, pp. 172–181, San Juan, Puerto Rico, USA, December 1994.

[35] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the timing analysis of pipelining and instruction caching," in *Proceedings of the 16th Real-Time Systems Symposium*, pp. 288–297, Pisa, Italy, December 1995.

[36] C.-G. Lee, J. Hahn, Y.-M. Seo, et al., "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

[37] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pp. 204–212, IEEE Computer Society Press, Brookline, Mass, USA, June 1996.

[38] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 1, pp. 53–70, 1999.

[39] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[40] J. Engblom, "Analysis of the execution time unpredictability caused by dynamic branch prediction.," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, pp. 152–159, IEEE Computer Society, Toronto, Canada, May 2003.

[41] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for WCET analysis," *Real-Time Systems*, vol. 34, no. 3, pp. 195–227, 2006.

[42] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[43] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, pp. 258–262, San Francisco, Calif, USA, February 2005.

[44] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 589–604, 2005.

[45] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[46] L. Seiler, D. Carmean, E. Sprangle, et al., "Larrabee: a many-core ×86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–15, 2008.

[47] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pp. 80–89, St. Louis, Mo, USA, April 2008.

[48] C. Ferdinand, R. Heckmann, M. Langenbach, et al., "Reliable and precise WCET determination for a real-life processor," in *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT '01)*, T. A. Henzinger and C. M. Kirsch,

Eds., vol. 2211 of *Lecture Notes in Computer Science*, pp. 469–485, Springer, Tahoe City, Calif, USA, October 2001.

[49] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pp. 364–373, Seattle, Wash, USA, May 1990.

[50] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 318–326, ACM Press, San Jose, Calif, USA, October-November 2003.

[51] M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 802–815, 2006.

[52] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, vol. 2006, pp. 217–226, Dresden, Germany, July 2006.

[53] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 600–605, Munich, Germany, March 2005.

[54] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pp. 223–232, IEEE Computer Society, Miami, Fla, USA, December 2005.

[55] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*, pp. 1484–1489, EDA Consortium, Nice, France, April 2007.

[56] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Systems*, vol. 18, no. 2-3, pp. 249–274, 2000.

[57] A. Wellings, "Is Java augmented with the RTSJ a better realtime systems implementation technology than Ada 95," *Ada Letters*, vol. 23, no. 4, pp. 16–21, 2003.

[58] Java Expert Group, "Java specification request JSR 302: Safety critical Java technology," http://jcp.org/en/jsr/detail?id=302.

[59] M. Schoeberl, "A time predictable Java processor," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 800–805, Munich, Germany, March 2006.

[60] M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," in *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '06)*, vol. 177, pp. 202–211, ACM Press, Paris, France, October 2006.

[61] T. Harmon and R. Klefstad, "Interactive back-annotation of worst-case execution time analysis for Java microprocessors," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pp. 209–216, Daegu, Korea, August 2007.

[62] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen, "Model-based schedulability analysis of safety critical hard real-time Java programs," in *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '08)*, pp. 106–114, ACM Press, Santa Clara, Calif, USA, September 2008.

[63] Sun, "picoJava-II Microarchitecture Guide," Sun Microsystems, March 1999.

[64] Sun, "picoJava-II Programmer's Reference Manual," Sun Microsystems, March 1999.

[65] W. Puffitsch, *picoJava-II in an FPGA*, M.S. thesis, University of Technology, Vienna, Austria, 2007.

[66] Altera, "Cyclone FPGA Family Data Sheet," ver. 1.2, April 2003.

[67] Imsys, Im1101c (the Cjip) technical reference manual, v0.25, 2004.

[68] aJile, "aj-100 real-time low power Java processor," preliminary data sheet, 2000.

[69] S. Uhrig and J. Wiese, "Jamuth: an IP processor core for embedded Java real-time systems," in *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pp. 230–237, ACM Press, Vienna, Austria, September 2007.

[70] M. Zabel, T. B. Preußer, P. Reichel, and R. G. Spallek, "Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture," in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD '07)*, pp. 59–62, Lübeck, Germany, August 2007.

[71] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and Th. Ungerer, "Real-time event-handling and scheduling on a multithreaded Java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19–31, 2003.

[72] T. B. Preußer, M. Zabel, and R. G. Spallek, "Bump-pointer method caching for embedded Java processors," in *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pp. 206–210, ACM Press, Vienna, Austria, September 2007.

[73] J. M. O'Connor and M. Tremblay, "picoJava-I: the Java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, 1997.

[74] EJC, "The ejc (embedded Java controller) platform," http://www.embedded-web.com/index.html.

[75] A. Krall and R. Grafl, "CACAO—a 64-bit JavaVM just-in-time compiler," in *Proceedings of the Workshop on Java for Science and Engineering Computation (PPoPP '97)*, G. C. Fox and W. Li, Eds., ACM Press, Las Vegas, Nev, USA, June 1997.

[76] F. Brandner, T. Thorn, and M. Schoeberl, "Embedded JIT compilation with CACAO on YARI," Tech. Rep. RR 35/2008, Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria, June 2008.

[77] C. Pitter and M. Schoeberl, "Performance evaluation of a Java chip-multiprocessor," in *Proceedings of the 3rd International Symposium on Industrial Embedded Systems (SIES '08)*, pp. 34–42, La Grande Motte, France, June 2008.

[78] G. Bernat, A. Burns, and A. Wellings, "Portable worst-case execution time analysis using Java byte code," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS '00)*, pp. 81–88, Stockholm, Sweden, June 2000.

[79] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable Java byte code WCET analysis framework," in *Proceedings of the 7th International Conference on Real-Time Computing and Applications (RTCSA '00)*, pp. 39–48, Cheju Island, Korea, December 2000.

[80] M. Schoeberl, "Application experiences with a real-time Java processor," in *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008.