On the Scalability of Time-predictable Chip-Multiprocessing

Wolfgang Puffitsch
Departement of Modeling and Information Processing
ONERA, Toulouse, France
wolfgang.puffitsch@onera.fr

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Real-time systems need a time-predictable execution platform to be able to determine the worst-case execution time statically. In order to be time-predictable, several advanced processor features, such as out-of-order execution and other forms of speculation, have to be avoided. However, just using simple processors is not an option for embedded systems with high demands on computing power. In order to provide high performance and predictability we argue to use multiprocessor systems with a time-predictable memory interface. In this paper we present the scalability of a Java chip-multiprocessor system that is designed to be time-predictable. Adding time-predictable caches is mandatory to achieve scalability with a shared memory multi-processor system. As Java bytecode retains information about the nature of memory accesses, it is possible to implement a memory hierarchy that takes the characteristics of different types of accesses into account. For tasks with low communication the measured speedup of this time-predictable system is in the range of 6 to 7 for eight processor cores, compared to execution on a single-core processor.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

Keywords

Time-predictable computer architecture

1. INTRODUCTION

Multiprocessors are considered a solution to the ever growing performance demands of embedded real-time systems. Hard real-time systems require that the underlying platform is analyzable with regard to its timing behavior. For uniprocessor systems, it is well known that some optimizations that increase the average-case performance make worst-case execution time (WCET) analysis more complex and increase the pessimism of the WCET estimate. The Java Optimized Processor (JOP) [10] is intended as a time-predictable processor to simplify WCET analysis. Although it is inevitable to sacrifice some average-case performance in favor of predictability, JOP provides good average-case performance compared to other Java processors.

When building a chip multiprocessor (CMP), it is not only necessary to use time-predictable processor cores. Also, the access to shared resources must be time-predictable. Most importantly, this concerns access to shared memory. As the bandwidth for an individual core is only a fraction of the bandwidth in a uniprocessor setting, caching becomes more important. In order to achieve good performance, while still providing a timing-analysis friendly platform, it is necessary to include caches that are suitable for WCET analysis.

Currently, there is little information as to whether a time-predictable CMP can actually provide the desired speedups. The first version of the JOP CMP system provided only moderate speedups with more than four cores [7]. In this paper, we evaluate the performance of an improved version that includes caches for heap allocated data, constants, static fields, and method dispatch tables. This *split cache* [11, 13] relaxes the memory bandwidth requirements of the CMP while still being analyzable. The results show that a time-predictable CMP can actually provide reasonable speedups.

The following section describes related work in the area of time-predictable chip-multiprocessing. Section 3 describes the rationales behind some design decisions that would be unusual in systems designed for average-case performance. Evaluation results are presented in Section 4, which are then discussed in Section 5. Section 6 finally concludes the paper.

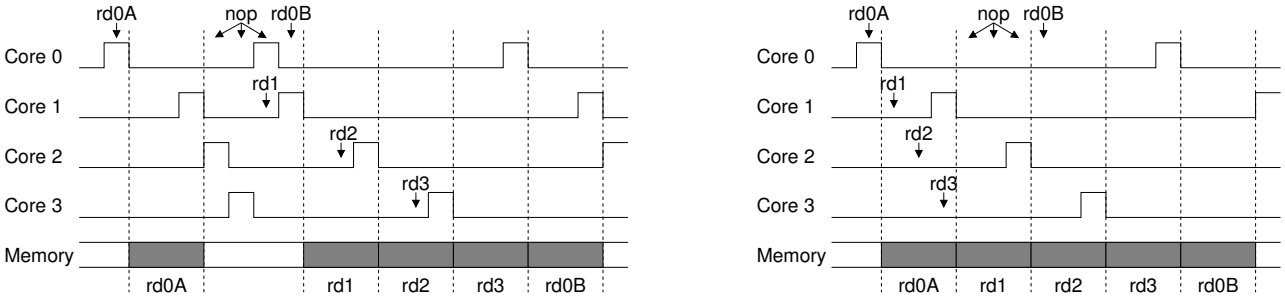
2. RELATED WORK

Not that many papers are available on the design of time-predictable CMP systems. A recent paper discusses some high-level design guidelines [1]. To simplify WCET analysis (or even make it feasible) the architecture shall be *timing compositional*. That means that the architecture has no timing anomalies or unbounded timing effects [3]. The Java processor used in the proposed time-predictable CMP fulfills those properties. For CMP systems the authors argue for bounded access delays on shared resources. This is in our opinion best fulfilled by a TDMA based memory arbitration, which combines bounded access delays with making these delays independent from accesses from other processors.

Wilhelm et al. [17] discuss the impact of cache replacement policies on static analysis in detail. For the measures considered in that paper, LRU replacement delivers the best possible results that any replacement policy can achieve. Consequently, the cache design described in Section 3.2 uses LRU replacement where applicable.

Paolieri et al. proposed a multi-core architecture that supports measurement-based WCET analysis [5]. In that paper, interferences for accesses to a shared level 2 caches are analyzed. The approach

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark



(a) Low bus traffic increases WCET

(b) Bus contention decreases WCET

Figure 1: Timing anomaly with RR arbitration

has been further extended to accesses to shared RAM [4]. In a setting with four cores, the WCET for the execution of a single thread is increased by a factor of up to 2.2, and the measured execution time by a factor of up to 1.7, compared to execution on a uniprocessor. When assuming that the task in consideration can be fully parallelized to four cores, this would be equivalent to speedups of $\frac{4}{2.2} = 1.82$ and $\frac{4}{1.7} = 2.35$, respectively.

A CMP version of the Java processor jamuth has been evaluated by Uhrig [16]. The interconnect to the shared memory is the Altera switch fabric with the Avalon bus. As jamuth is a chip-multithreaded architecture, two dimensions (chip multi-threading and multi-core) are explored. The maximum speedup, with modified versions of the JemBench benchmarks [14], in a configuration with 3 threads and 3 cores is reported as 3.5.

The SHAP Java processor project has been extended to a CMP version [19]. While it shares some of its characteristics with JOP, WCET analyzability is not mentioned among its design goals. The evaluation results in Section 4 demonstrate that an analysis friendly design does not necessarily perform worse than designs with other objectives.

3. SYSTEM DESIGN

The JOP chip-multiprocessor (CMP) design is driven by the intention to keep the system worst-case execution time analyzable. Optimizing for WCET instead of optimizing for average-case throughput leads to different solutions in the design space. This section gives the rationale for, from an average-case optimization standpoint probably uncommon, design decisions. To keep access to the shared resource main memory predictable, a TDMA based memory access arbitration is mandatory. As the pressure to the memory interface increases with multiple cores, more data needs to be held in core local memories. The split-cache design allows caching in a time-predictable way.

3.1 TDMA and Round-Robin Arbitration

In order to achieve timing predictability, the latency of memory accesses must be bounded. In a CMP setting, the arbitration of memory accesses must ensure that the memory latencies for all cores are bounded. To achieve this, we implemented two arbiters: a time-division multiple access (TDMA) arbiter and a round-robin (RR) arbiter.

The TDMA arbiter reserves a slot for each CPU in which it may access the memory exclusively. Each slot is long enough to contain a full memory access. Therefore, accesses by one core do not affect memory accesses of other cores. The RR arbiter works similarly to the TDMA arbiter, but uses flexible slot lengths. If no memory access occurs, the slot length is a single cycle. In case of memory accesses, the slot is extended to fit the current memory access. In theory, it would be possible to eliminate the cycle for cores without pending memory requests, i.e., to grant access to the next processor with a pending memory request immediately. However, this results in complex and slow hardware and would therefore in practice impair the performance of the overall system. Therefore, we accept to pay one wasted clock cycle per idle core to only perform local arbitration and have a scalable memory arbiter.

For both arbiters the latency for accesses is bounded. In a CMP with N cores, a core has to wait at most $N-1$ memory accesses before it is granted access itself. However, the arbiters differ in their average-case performance and the analyzability of the worst-case performance.

The TDMA arbiter always provides the same performance to a core, regardless of the workload on other cores. The performance does not depend on the amount of contention. This also simplifies WCET analysis, because the worst-case behavior can be analyzed locally, without considering other cores. Also, measurements to estimate the tightness of analytical WCET bounds can be kept simple. This arbiter is therefore a good choice when WCET analyzability is of utmost importance.

WCET analysis is considerably more complex for the RR arbiter. The performance depends on the level of contention. Intuitively, performance increases with low contention. However, this is not always the case. Figure 1 shows such a timing anomaly where high bus traffic leads to a lower execution time than low bus traffic with RR arbitration.

In both Figures 1a and 1b, Core 0 executes a sequence of five operations: first, it issues a read, rd0A, then it executes three operations that do not access memory (denoted by nop), and finally it issues a second read, rd0B. Cores 1 to 3 all issue a read operation, rd1 to rd3. The signals labeled Core 0 to Core 3 show when the arbiter is ready to serve accesses of the respective core; accesses from a core are passed on only if the respective signal is high.

In Figure 1a, rd0A can be served immediately. Afterwards, the three nops are executed. The second read of Core 0, rd0B, misses the slot for Core 0. Instead, rd1, which has been issued in the cycle before, is served. The reads from the other cores, rd2 and rd3, are served subsequently. After these reads have finished, rd0B is passed on. In Figure 1b, rd0A can also be served immediately. However, rd1 to rd3 are issued earlier, and are performed back to back with rd0A. Consequently, rd0B catches the next free slot for Core 0, and is served earlier than in Figure 1a.

The performance of the scenario depicted in Figure 1b is the same as for a TDMA arbiter. The scenario in Figure 1a performs actually worse than with TDMA arbitration. WCET analysis has to assume that with RR arbitration *each* memory access exhibits its worst-case behavior. In contrast, the predefined arbitration pattern of a TDMA arbiter allows the analysis to reason about sequences of

instructions (e.g., basic blocks), leading to shorter WCETs for these sequences. However, as the results in Section 4 show, this does not affect the average-case performance. Therefore, we suggest using the RR arbiter when average-case performance is more important than WCET analyzability.

3.2 Split Caches

With respect to caching, memory is usually divided into instruction memory and data memory. This cache architecture was proposed in the first RISC architectures [6] to resolve the structural hazard of a pipelined machine where an instruction has to be fetched concurrently to a memory access. This division enabled WCET analysis of instruction caches.

In former work we have argued that data caches should be split into different memory type areas to enable WCET analysis of data accesses [11, 13]. We have shown that a JVM accesses quite different data areas (e.g., the stack, the constant pool, method dispatch table, class information, and the heap), each with different properties for the WCET analysis. For some areas, the addresses are statically known; some areas have type dependent addresses (e.g., access to the method table); for heap allocated data the address is only known at runtime. Therefore, the caches are organized to simplify the analysis. Different memory areas are cached in different caches:

Method cache An instruction cache that caches whole methods. Misses can occur only upon calls and returns; all other instructions result in cache hits.

Stack cache Stack data is placed in this cache. The contents are replaced on task switches and accesses to this cache are always hits.

Constant cache A cache for constant data with known addresses. As addresses are known, a direct-mapped cache is sufficient. No cache coherence is necessary for constant data.

Static data cache Similar to the constant cache, but as data may change, cache coherence is necessary.

Object cache A fully associative cache that exploits knowledge about the object layout.

Fully associative cache A small cache for single words with unknown addresses.

The stack cache and method cache are an integral part of the JOP pipeline. The stack cache is part of the thread context and is exchanged on a thread switch. The size of the method cache, and therefore the maximum method size, can be configured. The default configuration in JOP is 4 KB for the method cache. Java methods tend to be small (99% smaller than 512 B [9]). Only class initializers for large arrays are quite large. We execute those large method at JVM boot time with a small interpreting JVM that runs on top of JOP. In that case, the method is directly executed from the main memory without using the method cache. The other data caches are optional and are located in a separate data cache unit. The inclusion or exclusion as well as the size can be configured.

For data where the address is statically known or can be inferred by a type analysis, a direct mapped caches are used [13]. For data with unpredictable addresses, two different caches are used. For objects, an *object cache* [12] is used, which uses object references as tags and caches the first *N* fields of an object. In an earlier paper [2], we have shown that such an *object cache* can be integrated into our WCET analysis tool [15]. For other data with unpredictable addresses, a fully associative cache with LRU replacement that caches single words is used. Due to its limited size, it is used only for data where accesses in the near future are likely, i.e., this cache is not used for array fields.

Memory accesses can be classified whether the accessed data requires to be held coherent or is core local. Accesses to static variables and object fields must follow the Java memory model, which requires some coherence mechanism. Accesses to constant data such as the constant pool or the method table are implicitly cache coherent. Stack allocated data is thread local in Java and needs no cache coherence protocol.

3.2.1 Cache Coherence

When designing a CMP system, one must consider cache coherence mechanisms. Depending on the memory model, different mechanisms can be provided to programmers and compilers to establish the guarantees of the memory model.

For a Java processor, it is natural to use a memory model that is compliant with the Java memory model. The Java memory model can be implemented by using write-through caches and invalidating caches when acquiring locks and reading from volatile variables [8]. While such a coarse-grain cache coherence implementation might sacrifice some performance, it is very convenient from a WCET analysis point of view. In the proposed cache coherence scheme, all actions related to cache coherence are local actions that are visible to the WCET analysis. The timing of accessing a variable is the same regardless of whether the variable is actually shared with a different core, and the cache state is independent from the behavior of other cores.

Using write-through caches also simplifies WCET analysis in a uniprocessor setting. With a write-back cache, the analysis would have to reason about whether a cache line actually needs to be written back to memory on a cache miss. Some current WCET tools assume an additional write back for each cache miss [1]. For a write-through cache, the cost of writes can be attached to the instruction that issues the write, without introducing further pessimism.

4. EVALUATION

We chose four benchmarks from the JemBench [14] benchmark suite, with different characteristics to demonstrate the scalability of our design. Two additional single-threaded benchmarks were used to evaluate the performance impact of the split cache design.

4.1 Platform

The evaluation compares CMP versions of JOP with different cache configurations. All configurations contain a cache for instructions (the method cache) and stack allocated data (the stack cache). The *cached* and *uncached* configurations in the following discussion are configurations where other types of data are cached/uncached.

The method cache is 4 KB in size, divided into 32 blocks. The stack cache is 2 KB large. Configurations that cache accesses to other memory areas contain four more caches: a direct mapped cache with 1 KB for constant data with known addresses, a direct mapped cache with 1 KB for data with known addresses that requires cache coherence, a 16-way object cache that caches the first 8 words of objects, and a 16-way fully associative cache with LRU replacement for other data with unknown addresses.

Accesses to the constant pool and the method tables go through the direct mapped cache without cache coherence, while the direct mapped cache for cache coherent data is used for accesses to static variables. The object cache is used for accesses to object fields. Most other accesses go through the fully associative cache; only accesses to array fields and some JVM-internal accesses bypass the caches.

The development board used in this evaluation is the DE2-70 board from Altera, which features a Cyclone II FPGA (EP2C70), and 2 MB of synchronous SRAM. Accesses to this RAM have a latency of 3 cycles. For round-robin arbitration, the worst-case latency for a memory access from a core is $3 * N$ cycles, as the processor in the worst case has to wait for accesses from all other cores and the completion of its own access. For TDMA arbitration, the worst-case latency is $3 * (N + 1)$ cycles, because it might happen that the core just missed the beginning of its current slot and needs to wait for its completion. JOP is clocked at 90 MHz in all configurations presented in this section.

4.2 Benchmarks

The following benchmarks have been chosen from the embedded Java benchmark suite JemBench [14]:

LiftCMP is derived from a real-world application that controls an industrial lift. Each core executes one such single-threaded application, without any cooperation or synchronization. This benchmark therefore evaluates the performance of independent threads.

Matrix benchmarks the performance of matrix multiplication. While there is some computational complexity, its performance mostly depends on the available memory bandwidth.

Raytrace is a computationally complex benchmark that depends on the performance of floating-point operations. Parallelization is limited to six threads.

Queens computes solutions to the N-Queens problem. It contains a notable amount of synchronization, which has to be handled efficiently for good performance.

Kfl is a single-threaded benchmark, derived from a real-world application that tilts up a railway contact wire. It is used as an additional benchmark for the evaluation of the cache performance.

UdpIp is is a benchmark derived from the implementation of a simple UDP/IP stack. It is also used to evaluate the cache performance.

4.3 Scalability

Figure 2 shows the benchmark results scaled to the uncached uniprocessor version. Four configurations are evaluated: TDMA arbitration without caching (TDMA), TDMA arbitration with cache (TDMA $), RR arbitration without cache (RR), and RR arbitration with cache (RR $). Naturally, there is no arbitration in the uniprocessor versions.

The LiftCMP benchmark, shown in Figure 2a, is limited in its scalability by its memory bandwidth demands for the uncached configurations. Only speedups of up to 2.9 can be achieved. In contrast, the cached configurations scale up to 6.1 for TDMA $ and 6.9 for RR $. The results suggest that caching has a considerably higher impact on the performance than the arbitration policy.

For the Matrix benchmark (Figure 2b), the configurations without caches do not scale very well, with speedups of around 2.0 at most. The configuration with caches scales considerably better, with RR $ providing a speedup of up to 3.6 with eight cores. TDMA $ achieves speedups of up to 3.16. The memory bandwidth demands of the Matrix benchmark limit the scalability, but caching enables significantly higher speedups. This benchmark also shows the effect that additional cores do not always improve performance. The inner loop of the matrix multiplication has a very regular memory access pattern. For cases where this pattern does not fit the slot pattern of the memory arbiter well, a performance degradation can be observed. The effect is visible in particular for the RR configuration with three cores and the TDMA $ configuration with six cores.

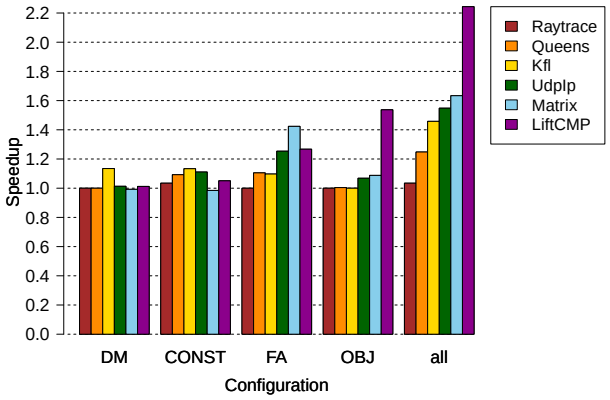


Figure 3: Performance impact of individual caches for JOP with eight cores and TDMA arbitration

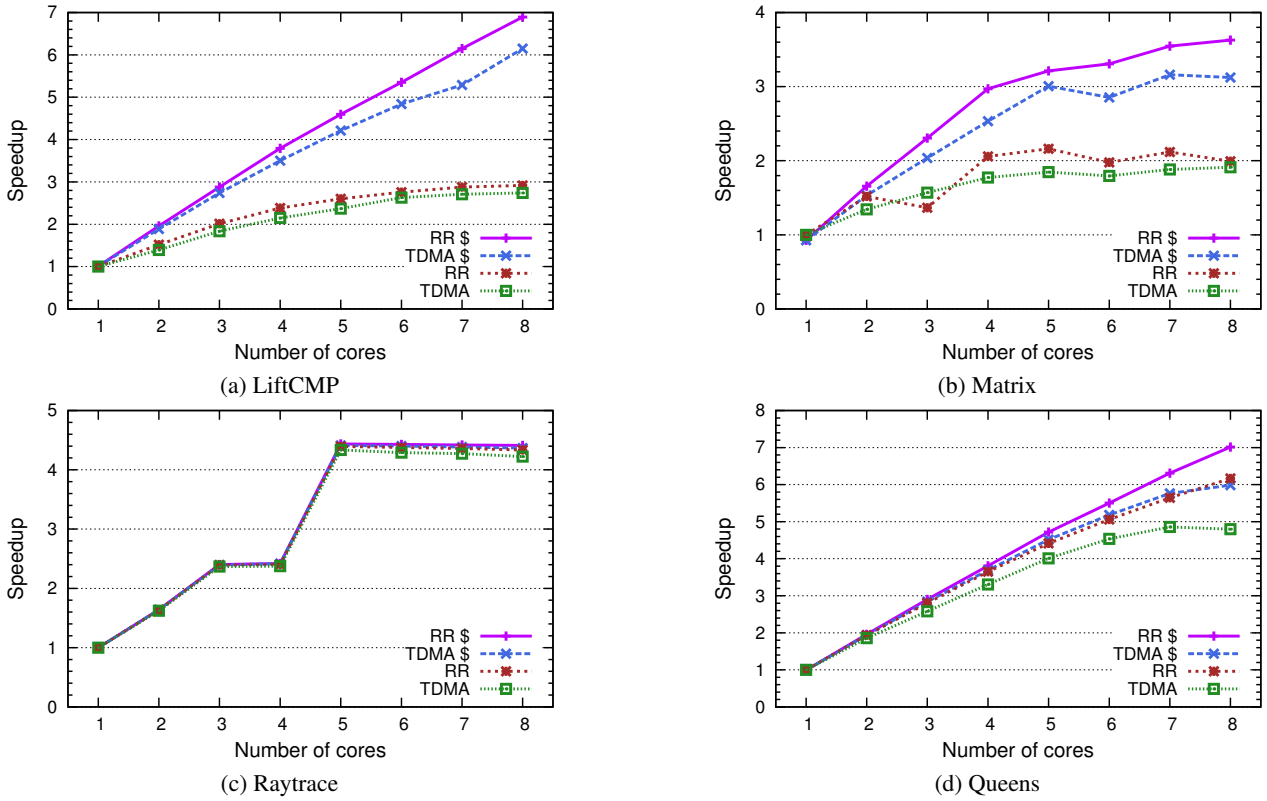
The results for the Raytrace benchmark in Figure 2c do not vary greatly between the different configurations. Most of the computation time is spent for floating-point computation, leading to low memory bandwidth demands, and speedups of around 4.2 to 4.4. Only for configurations with more than five cores, there is a notable difference between TDMA and the other three configurations. Please note that the benchmark cannot scale beyond six cores, simply because it consists of only six threads. However, it apparently already reaches its maximum performance for five cores. Supposedly due to an unfortunate assignment of threads to cores, performance does not improve between three and four cores.

The performance of the Queens benchmark reaches a speedup of around 4.8 for the plain TDMA configuration, as shown in Figure 2d. Using either TDMA $ or RR provides similar speedups of up 6.0 and 6.2, respectively. The scalability of the TDMA $ configuration is limited by synchronization—memory bandwidth is reserved for cores even if they are blocked by other cores. The RR configuration can use this available bandwidth, but suffers from its high memory bandwidth demands. The combination of RR arbitration and caching combines the benefits, leading to speedups that scale almost linearly with the number of cores for RR $, up to a factor of 7.0.

4.4 Cache Performance

Figure 3 shows the performance impact of the individual caches in the split cache in a configuration with eight cores and TDMA arbitration. DM shows the results with just the direct-mapped cache for static variables enabled, CONST with just the cache for constant data, FA with just the fully associative cache, and OBJ with just the object cache. The results with all caches enabled is labeled all.

The DM and CONST configurations provide only minor performance enhancements. As only a fraction of all instructions accesses static or constant data, this is not surprising. Even if all such accesses would be cache hits, we could not expect huge performance gains. In contrast, the FA configuration increases performance significantly. This is relatively surprising, given the small size of this cache. This can be explained by the fact that a significant proportion of instructions accesses heap allocated data. Therefore, even a small cache with moderate hit rates can provide significant speedups. As most of the benchmarks are not written in a particularly object-oriented



(a) LiftCMP (b) Matrix

(c) Raytrace (d) Queens

Figure 2: Benchmark results scaled to the uncached uniprocessor configuration

style, we do not see a major effect of the object cache for these benchmarks. The exception for this is the LiftCMP benchmark, for which the object cache improves performance by more than 50%. The LiftCMP benchmark also benefits the most from caching, with a perfomance increase by a factor or 2.2 if all caches are enabled.

4.5 Comparison with Another Java CMP

In this section, we compare the performance of the JOP CMP system with the CMP version of the SHAP Java processor [19]. While SHAP is in some regards similar to JOP, it is not explicitly designed for time-predictability.

The SHAP CMP system has been evaluated with two different FPGA boards: the Altera DE2 board and the Xilinx ML505. The DE2 board has a asynchronous 16-bit memory and 32-bit memory accesses take 2 cycles. On the Cyclone II FPGA (EP2C35) of the DE2 board, SHAP can be clocked at 60 MHz. The ML505 features a 32-bit synchronous memory with a latency of 3 cycles, but a bandwidth of 1 cycle per access is achieved through pipelining. The maximum clock frequency on the high performance Virtex-5 FPGA of that board is 80 MHz.

In order to compare the performance of the JOP CMP system with the performance of SHAP on the DE2, we implemented a variant of JOP on the DE2-70 board with 6 cycles memory access latency. This corresponds to the memory latency that earlier implementations of JOP had on the DE2 board. For the scope of this comparison, we ignore the fact that it would not be possible to implement a JOP CMP with 8 cores on the DE2 board due to the smaller FPGA.

Figure 4 compares the results for JOP and SHAP on different FPGA boards.[1] For JOP, we show the results for the cached versions with TDMA arbitration, labeled JOP DE2-70 TDMA $ for the "fast" variant, and JOP DE2-70* TDMA $ with 6 cycles memory latency. The results of the fastest configuration of JOP on the DE2-70 board, with caching and RR arbitration, are labeled JOP DE2-70 RR $. For SHAP, we show the results for the DE2 and the ML505 board (SHAP DE2 and SHAP ML505).

Even with slowed-down memory accesses, JOP performs considerably better than SHAP on the DE2 board. While the performance of SHAP on that board saturates for five and six cores, JOP scales up to eight cores.

Although the ML505 board contains a more powerful/faster FPGA and provides a higher memory bandwidth than the DE2-70 board, JOP outperforms SHAP on the respective boards for the LiftCMP benchmark. Still, both CMP systems provide reasonable scalability up to eight cores. The result in the PhD thesis of Zabel [18] indicate that the performance of SHAP for this benchmark would saturate at around ten cores.

The results in Figure 4 demonstrate that a time-predictable solution does not have to be slow and can even provide better performance on the same FPGA platform than a CMP with different design goals. Time-predictable caches are effective and can to some degree compensate slower access to main memory, as demonstrated by the similar or superior performance of JOP on the DE2-70 board compared to SHAP on the ML505 board.

5. DISCUSSION

The results in Section 4 show that a time-predictable CMP scales reasonably well and provides a performance that is comparable to other Java CMPs. Only for the Matrix benchmark the memory

[1]The numbers of for the SHAP CMP system are according to the JTRES 2010 paper [19]. We thank Martin Zabel for providing us the raw data of the benchmark runs.

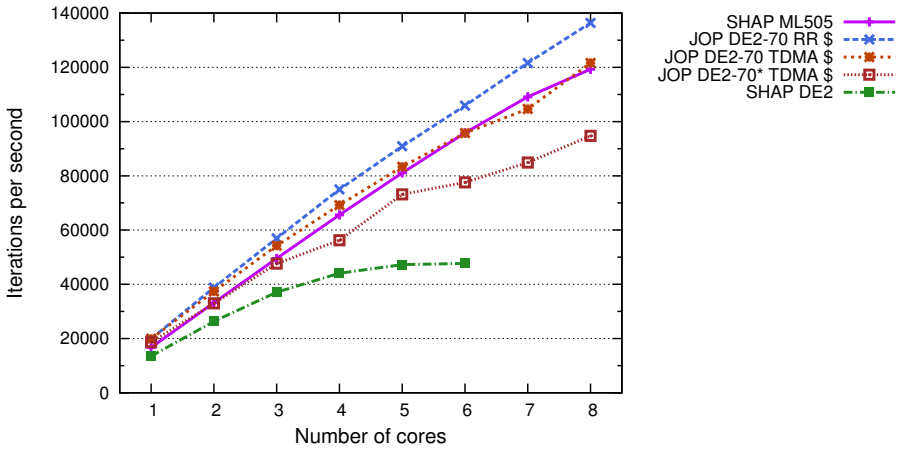


Figure 4: JOP and SHAP scaling on different FPGA platforms for LiftCMP

bandwidth requirements impair the scalability significantly. For both the Matrix and the LiftCMP benchmark, caching has a greater impact on the performance than the arbitration policy. For configurations that are severely limited in their performance by the bandwidth requirements (most notably the uncached configurations with six or eight cores), the performance gains by RR arbitration are minimal. Therefore, the better predictability of TDMA arbitration probably outweighs the performance gains for applications with similar characteristics.

RR arbitration outperforms TDMA arbitration for the Queens benchmark. In situations where a core executes within a critical section while other cores wait for the respective lock, the RR arbiter provides the unused bandwidth to the core holding the lock. While this enhances the performance in measurements, such situations are difficult to model for static analysis. It has to be proven that other cores always wait for the held lock when executing a certain program fragment. Otherwise, the analysis cannot assume that the additional bandwidth is actually available.

The evaluation of the impact of individual caches showed that both an object cache and a small fully associative cache with LRU replacement can enhance performance significantly. As the caches of JOP CMP are designed specifically to simplify static analysis, they can not only increase the measured performance, but can also decrease the analytical WCET [13].

6. CONCLUSION

In this paper we report on the scalability of a chip-multiprocessor Java processor, which is designed to enable WCET analysis. Although design optimization for WCET is different from optimizations for average-case throughput, the JOP CMP design scales reasonable for parallel workloads. As the pressure on the memory bandwidth increases with several processor cores on a shared memory architecture, we have added additional caches to the processor cores: direct mapped caches for data where the address can be statically predicted and highly associative caches for data with unpredictable addresses. The resulting speedup of a eight core system compared to a uniprocessor system is between 3 and 7, depending on the type of application.

7. REFERENCES

[1] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[2] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, 24(8):753–771, 2012.

[3] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

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

[5] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *The 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, Austin, Texas, USA, 20-24, June 2009. ACM.

[6] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[7] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[8] Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2009)*, pages 90–99, New York, NY, USA, 2009. ACM.

[9] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[10] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[11] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[12] Martin Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.

[13] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, DOI: 10.1007/s11241-012-9159-8:1–28, 2012. doi: 10.1007/s11241-012-9159-8.

[14] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[15] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[16] Sascha Uhrig. Evaluation of different multithreaded and multicore processor configurations for soPC. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 9th International Workshop, SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2009.

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

[18] Martin Zabel. *Effiziente Mehrkernarchitektur für eingebettete Java-Bytecode-Prozessoren*. PhD thesis, Technische Universität Dresden, Fakultät Informatik, 2011.

[19] Martin Zabel and Rainer G. Spallek. Application requirements and efficiency of embedded java bytecode multi-cores. In *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 46–52, New York, NY, USA, 2010. ACM.