Java Environment for Parallel Real-Time Development

Project Number 216682

D2.6– JOP Scalability Report

Version 1.0
4 October 2010
Final

Public Distribution

Wolfgang Puffitsch, Martin Schoeberl, Benedikt Huber
Technical University of Vienna, Technical University of Denmark

Project Partners: **aicas**, **EADS Deutschland**, **FZI**, **RadioLabs**, **SkySoft Portugal**, **SYSGO**, **Technical University Cluj-Napoca**, **The Open Group**, **Technical University of Vienna**, **University of York**, **Technical University of Denmark**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neu-Strasse 18 76131 Karlsruhe, Germany Tel:+49 721 663 968 23 Fax:+49 721 663 968 93 E-mail:siebert@aicas.com	**EADS Deutschland** Frederic Lamy Woerthstrasse 85 89077 Ulm, Germany Tel: +49 731 392 7427 Fax: +49 731 392 2074 27 E-mail: frederic.lamy@eads.com
FZI Gábor Szeder Haid-und-Neu-Strasse 10-14 76131 Karlsruhe, Germany Tel: +49 721 965 4266 Fax: +49 721 965 4259 E-mail:szeder@fzi.de	**RadioLabs** Filippo Corsini via Cavaglieri 26 00173 Rome, Italy Tel: +39 069 727 8250 Fax: +39 069 727 8268 E-mail:filippo.corsini@radiolabs.it
SkySoft Portugal José Neves Av. D. João II, Torre Fernão Magalhães, 7º 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 Fax: +351 21 386 6493 E-mail:jose.neves@skysoft.pt	**SYSGO** Jacques Brygier 5, Rue Hans List, Batiment F 78290 Croissy-sur-Seine, France Tel: +33 1 300 912 63 Fax: +33 1 301 504 48 E-mail:jacques.brygier@sysgo.fr
Technical University Cluj-Napoca Gheorghe Sebestyen-Pal G. Baritiu 26-28 40027 Cluj-Napoca, Romania Tel:+40 264 401 476 Fax:+40 264 594 491 E-mail:gheorghe.sebestyen@cs.utcluj.ro	**Technical University of Vienna** Wolfgang Puffitsch Treitlstrasse 3 1040 Vienna, Austria Tel: +43 15 880 118 208 Fax: +43 15 869 149 E-mail:wpuffits@mail.tuwien.ac.at
The Open Group Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail:s.hansen@opengroup.org	**University of York** Andrew Wellings Heslington Hall York YO10 5DD, United Kingdom Tel: +44 1904 432 742 Fax: +44 1904 432 767 E-mail:andy@cs.york.ac.uk
Technical University of Denmark Martin Schoeberl Richard Petersens Plads Building 322, room 232 2800 Lyngby, Denmark Tel: +45 45253743 E-mail:masca@imm.dtu.dk	

Contents

Document Control

Version	Status	Date
0.1	Document start	17 June 2010
0.2	Performance figures	10 September 2010
0.3	First draft for partner review	29 September 2010
1.0	Final version	4 October 2010

Executive Summary

This documents contains the deliverable *D2.6 JOP Scalability Report* of work-package 2 of the JEOPARD project due 33 months after project start as stated in the Description of Work. This document presents the evaluation of the chip-multiprocessor version of JOP with respect to scalability. In addition, this document provides rationales for some design decisions, as requested at the first review meeting.

1 Introduction

The report evaluates the scalability of a chip-multiprocessor version of the Java processor JOP [5]. Although JOP is intended as a time-predictable processor to enable worst-cases execution time (WCET) analysis, the average case performance shall not suffer.

The first version of the JOP CMP system, as described in D 2.1 and in [5], provides only moderate speedup with more than 4 cores. We have improved the cache for heap allocated data, constants, static fields, and method dispatch table to relax the memory bandwidth requirements of a CMP system. We call this the split-cache design [8, 13] . The resulting CMP system provides a reasonable speedup in the average case.

The data caches are organized as split cache to enable WCET analysis of the data cache. Caching of constant, static fields, and the method dispatch table is easy to analyze. For heap allocated objects we have co-developed the object cache and the WCET analysis for this cache type.

2 Rational for Design Decisions

The JOP chip-multiprocessor (CMP) design is driven by the intention to keep the system worst-cases execution time (WCET) analyzable. Optimizing for WCET instead of optimizing for average case throughput leads to different solutions in the design space. This section gives the rational for, probably uncommon, design decisions.

2.1 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 [4] 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 [8, 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:

- An instruction cache for complete methods (method cache)
- A stack cache
- A cache for static data
- An object cache for heap allocated objects
- A cache for constants

The stack cache and method cache are an integral part of the JOP pipeline. The other data caches are optional and are located in a separate data cache unit. The inclusion or exclusion can be configured.

The size of all caches is configurable. Details on the stack cache and method cache can be found in Deliverable D 2.4.

For data, where the address is statically known or can be inferred by a type analysis a direct mapped cache is used (see D 2.4). For heap allocated objects we propose to use an object cache. Different variations of the objects cache are described in the following subsection. In [3] we have shown that the proposed object cache can be integrated into our WCET analysis tool [14].

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.

2.1.1 The Object Cache

When accessing statically unknown addresses, it is impossible to predict which cache line from one way is affected by the access. From the analysis' point of view, a n-way set-associative cache is reduced to n cache lines when all addresses are unknown. Simpler cache organizations than a fully associative cache are therefore pointless for the analysis. However, such caches are expensive and must therefore be kept small. In contrast, accesses to datums with statically known addresses can be classified as hits or misses even for simple direct-mapped caches. For such a cache, accessing an unknown address would void information about all other accesses. Therefore, splitting the cache simplifies the static analysis and allows for a more precise hit/miss classification. For most data areas standard cache organizations are a reasonable fit. Only for heap allocated data we need a special organization – the object cache for objects and a solution that benefits mainly from spatial locality for arrays. The WCET analysis driven exploration of the object cache organization is the topic of [3].

The object cache is intended for embedded Java processors such as JOP [7], jamuth [15], or SHAP [16]. Within a hardware implementation of the Java virtual machine (JVM), it is quite easy to distinguish between different memory access types. Access to object fields is performed via bytecodes `getfield` and `putfield`; array accesses have their own bytecode and also accesses to the other memory areas of a JVM. The instruction set of a *standard* processor contains only untyped load and store instructions. In that case a Java compiler can use the virtual memory mapping to distinguish between different access types.

The object cache architecture is optimized for WCET analysis instead of average case performance. To track individual cache lines symbolically, the cache is fully associative. Without knowing the address of an object, all cache lines in one way map to a single line in the analysis. Therefore, the object cache contains just a single line per way. Instead of mapping blocks of the main memory to those lines, whole objects are mapped to cache lines. The index into the cache line is the field index. To compensate for the resulting small cache size with one cache line per way, we also explore quite large cache lines. To reduce the resulting large miss penalty we also consider to fill only the missed word into the cache line. To track which words of a line contain a valid entry, one valid bit per word is added to the tag memory.

The object cache is a data cache with cache lines indexed by unique object identifiers (the Java reference). The object cache is thus similar to a virtually-addressed cache, except that the cache index is unique and therefore there are no aliasing issues (different object identifiers map to different objects). In our system, the tag memory contains the pointer to the handle (the Java reference) instead

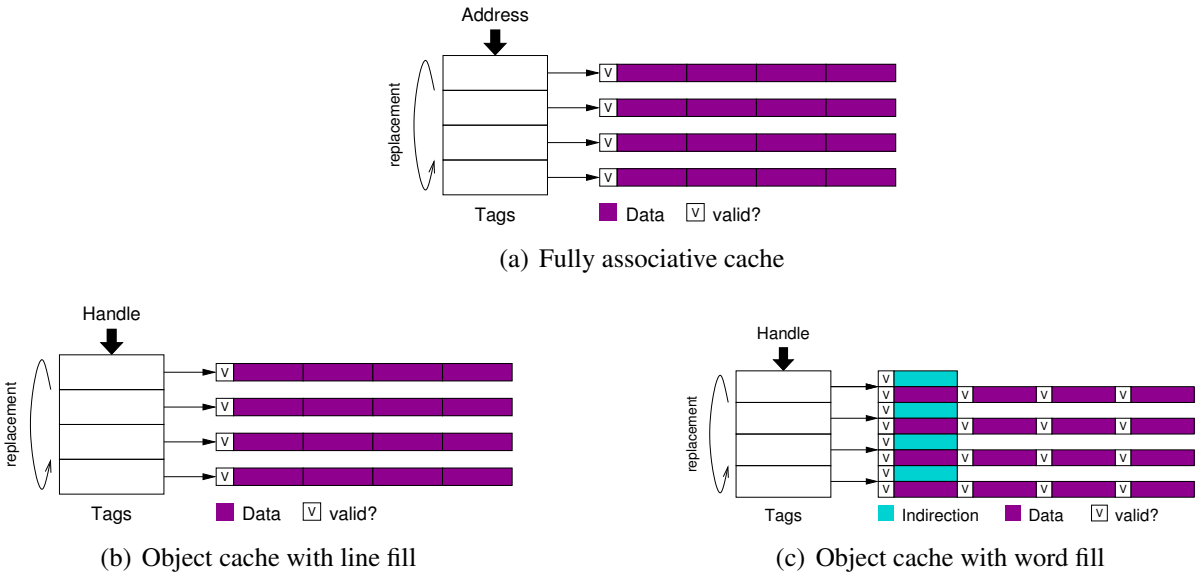


Figure 1: Comparison of a fully associative cache and object cache variants

of the effective address of the object in memory. This simplifies changing the address of an object during the compacting garbage collection. For a coherent view of the object graph between the mutator and the garbage collector, the cached address of an object needs to be updated or invalidated after the move. The cached fields, however, are not affected by changing the object's address, and can stay in the cache. Furthermore, the object cache reduces the overhead of using handles. If an access is a hit, the cost for the indirection is zero – the address translation has been already performed.

The object cache is organized to cache one object per cache line. If an object needs more space than available in one cache line, fields with higher indices are not cached. The decision not to cache fields with higher field indices than words in the cache line simplifies the tag memory and the hit detection. If we would allow that different fields of one object can map to the same word in the cache line, we would need an additional tag entry per field. To compensate for possible misses with large objects, we explore quite long cache lines. The cost for the cache line is less then the cost of the tag memory, as all tags have to be compared in parallel, but the cache line needs only be read out. For an implementation in an FPGA this means that the tag memory has to implemented with discrete registers, but the cache lines can be implemented in standard on-chip memory blocks.

As objects thus cannot cross cache lines, the number of words per cache line is one important design decision to be taken. To avoid that less frequently accessed fields are cached, a compile time optimization may rearrange the order of object fields. Both benchmarking results and the results of the static analysis may be used to classify the access frequency of fields.

Figure 1 outlines the differences between a fully associative data cache, and object caches with word and line fill. While the fully associative cache in Figure 1(a) uses the actual address as tag, the object caches use the handle. When filling the whole cache line, it is not necessary to keep the indirection pointer in the cache (Figure 1(b)). Once an object is cached, the indirection is resolved implicitly. For an object cache with word fill policy (Figure 1(c)), each word requires a valid flag, and the indirection must also be cached to allow for efficient loading of words that are not yet cached.

To simplify static cache analysis we chose to organize the cache as write through cache. Write back is harder to analyze statically, as on each possible miss another write back needs to be accounted for.

Furthermore, a write-through cache simplifies the cache coherence protocol for a chip multiprocessor (CMP) system [6]. The caches are invalidated upon `monitorenter` and reads from volatile variables. This scheme is compliant with the Java memory model. Although the coherence implementation is probably less efficient than other coherence protocols, cache invalidation is always a core-local action. The cache state does not depend on the behavior of other cores. WCET analysis is therefore possible for the cache coherence protocol.

The object cache is only used for objects and not for arrays. This is because arrays tend to be larger than objects, and their access behavior rather exposes spatial locality, in contrast to the temporal locality of accesses observed for objects. Therefore, we believe that a cache organized as a small set of prefetch buffers is more adequate for array data. As on the one hand arrays use a different set of bytecodes and are thus distinguishable from ordinary objects, but on the other hand have a structure similar to objects, this decision does not restrict our choices for further exploration.

2.2 TDMA and Round-Robin Memory 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 is 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.

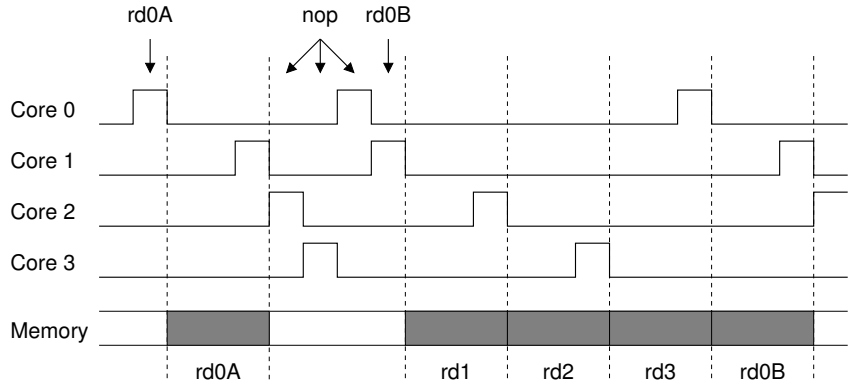
For both arbiters, the latency for access is bounded—in a CMP with N cores, a core has to wait at most $N - 1$ memory accesses before it is granted access itself. However, the arbiters differ in their average-case performance and the analyzability of the average case performance.

The TDMA arbiter provides the same performance to a core, without interference from 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. This arbiter is therefore a good choice when WCET analyzability is of utmost importance.

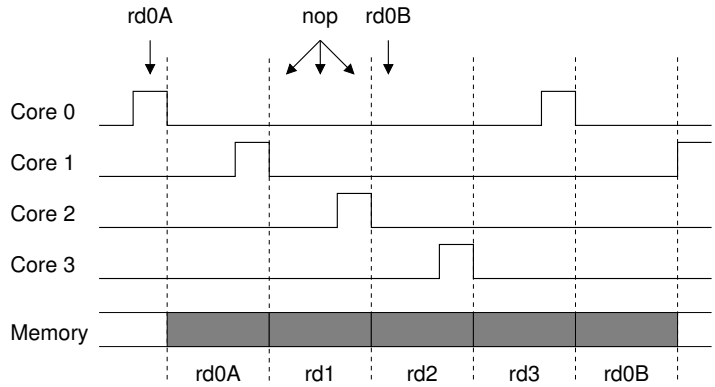
WCET analysis is considerably more complex for the RR arbiter. The performance depends on the level of contention. Intuitively, performance increases with low contention. However, this is not always the case. Figure 2 shows such an timing anomaly where high bus traffic leads to a lower execution time than low bus traffic with RR arbitration.

In both Figures 2(a) and 2(b), 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.

In Figure 2(a), rd0A can be served immediately. Afterwards, the three nops are executed. As there were no other accesses, rd0B misses the next slot for Core 0, and is delayed until rd1 to rd3 have been served.



(a) Low bus traffic increases WCET



(b) Bus contention decreases WCET

Figure 2: Timing anomaly with RR arbitration

In Figure 2(b), rd0A can also be served immediately. However, rd1 to rd3 have been 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 2(a).

The performance of the scenario depicted in Figure 2(b) is the same as for a TDMA arbiter. The scenario in Figure 2(a) performs actually worse than with TDMA arbitration. However, as the results in Section 3 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.

2.3 Transactional Memory

We have implemented hardware transactional memory [2] in the context of the JOP CMP [10]. Transactional memory (TM) is considered as a better scaling alternative than explicit locks. Furthermore, the usage of TM is simpler than using locks, as the programmer uses atomic sections instead of individual lock. The atomic sections are represented by a method annotation (`@atomic`) and a bytecode manipulation tool translates those methods to the transactions with retry on a fail.

In [9] we have shown that the maximum retry of conflicting transactions is bounded when transactions of individual threads are displaced by a minimum time interval. The later is implicitly enforced when using periodic threads, which are common in real-time systems. Without further analysis the worst case retry time is the same as the maximum blocking time of a single, global lock. However,

detecting non-conflicting transactions is possible with program analysis [9]. Therefore, the model of atomic sections simplifies the programming model and shifts the burden to find tight blocking times to program analysis.

3 Evaluation

We chose four benchmarks, from the benchmark suite JemBench [12], with different characteristics to demonstrate the scalability of our design. Two of the benchmarks have been used in the initial JOP CMP design document D 2.1. Therefore, the improvements on the scalability can be seen by comparing the current results with the results reported in D 2.1.

3.1 Platform

The evaluation compares CMP versions of JOP with different heap cache configurations. All configurations contain a cache for the instructions (the method cache) and stack allocated data (the stack cache). The *uncached* configurations in the following discussion are configurations where heap allocated data is not cached.

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 three 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, and a 16-way fully associative cache with LRU replacement for data with unknown addresses.

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. The board would have allowed a latency of only 3 cycles for memory accesses, but we chose to assume a latency of 6 cycles for two reasons: First, this configuration is similar to the configurations used in earlier experiments, which were conducted on a DE2 board. Second, scalability is easy to achieve with very fast main memory. We believe that a setting with memory with a higher latency (and therefore makes scalability more challenging) is more realistic.

JOP is clocked at 100 MHz in all configurations presented in this section.

3.2 Benchmarks

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

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

Raytrace is a computationally complex benchmark that depends on the performance of floating-point operations. Parallelization is limited to 6 threads.

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

To highlight different aspects for the evaluation, we present the data in three different formats.

- Figure 3 displays the speedups for the benchmark relative to the uniprocessor version. While uncached CMP variants are scaled to a uniprocessor without heap cache, cache CMP configurations are scaled to a cached uniprocessor. This representation highlights the scalability of the CMP versions.
- Figure 4 shows the benchmark results scaled to the uncached configuration of a multiprocessor with a TDMA arbiter. This emphasizes the contribution of the caches and the arbiter to the overall performance.
- In Figure 5, the benchmark results are scaled to the uncached uniprocessor version, which is the smallest configuration. The figure displays the possible performance gains. In contrast to Figure 3, speedups of that are greater than the number of cores are possible, because the cached uniprocessor version is already faster than the uniprocessor without cache.

There are four main configurations in Figures 3, 4, and 5: TDMA arbitration without heap 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.

3.3 Scalability

Figure 3 shows the benchmarks results scaled to uniprocessor configurations, such that configurations with caches are compared to a cached uniprocessor, and configurations without caches to a uniprocessor without cache. This allows us to investigate how well the different configurations scale without considering absolute performance.

For the Matrix benchmark, the configurations without caches scale poorly, with speedups of around 1.5 at most. The configuration with caches scale considerably better, with RR $ providing a speedup of around 3 with four cores. Six cores provide a speedup of 2.4 for TDMA $. The memory bandwidth demands of the Matrix benchmark limit the scalability, but caching enables considerably higher speedups. When the memory bandwidth is saturated, additional cores provide no additional performance scaling, but impose more synchronization overhead. Therefore, the performance decreases with 6 or 8 cores.

The performance of the Queens benchmark saturates at a speedup of around 3 for the plain TDMA configuration. Using either RR or TDMA $ provides similar speedups of up to 5. 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 $.

The results for the Raytrace benchmark 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 to 4.5. Only for configurations with six and eight cores, there is a notable difference beteen TDMA and the other three configurations. Please note that the benchmark cannot scale beyond six cores, simply because it consists of only six threads. This also explains the super-linear speedup from four to six cores, because the scheduling overhead disappears for six cores.

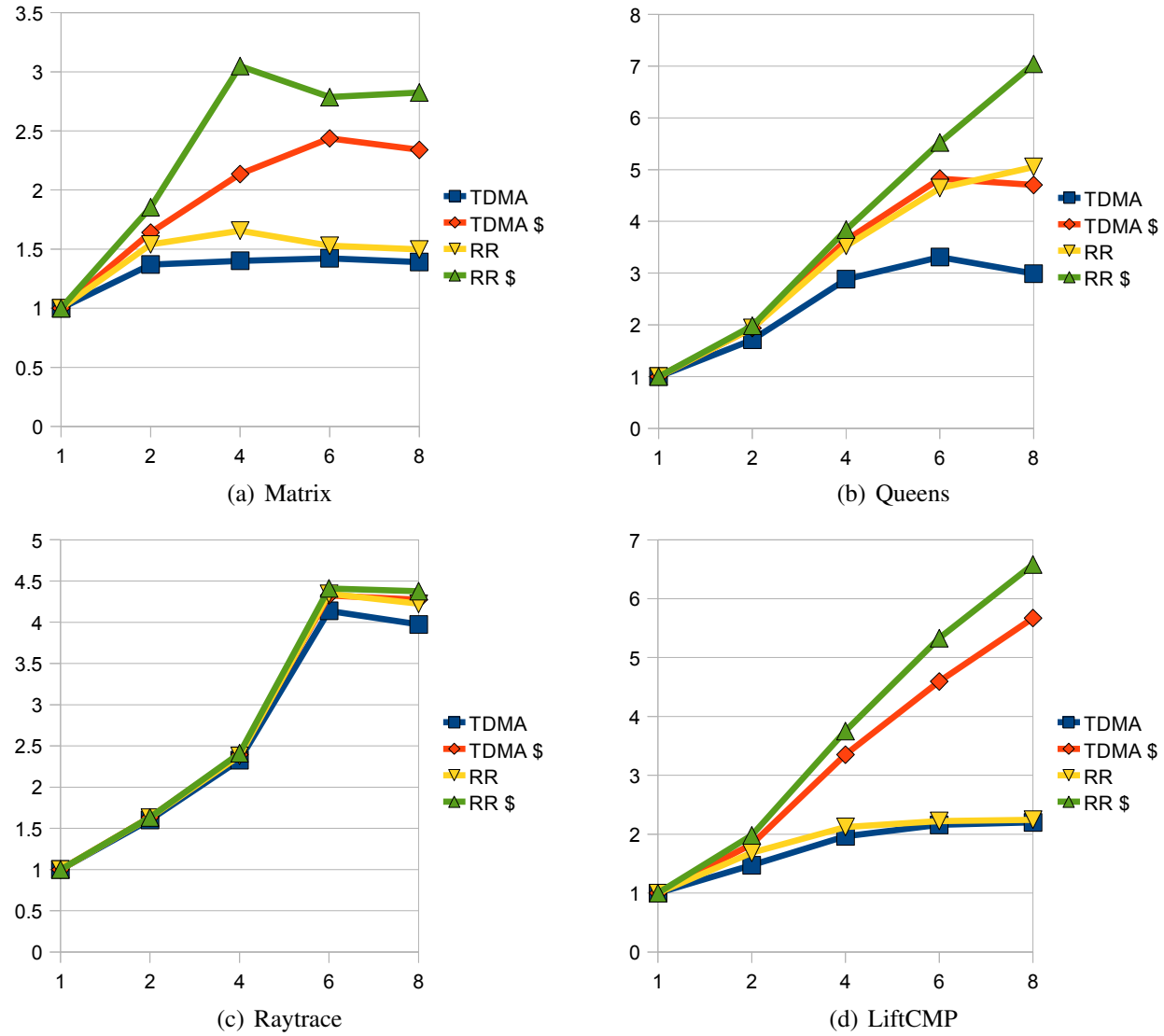


Figure 3: Benchmark results scaled to uniprocessor

The LiftCMP benchmark is limited in its scalability by its memory bandwidth demands for the uncached configurations. Only speedups of around 2.2 can be achieved. In contrast, the cached configurations scale up to 5.7 for TDMA $ and 6.6 for RR $.

3.4 Feature Impact

The graphs in Figure 4 emphasize the observations from the previous section. For the Matrix and LiftCMP benchmark, caching can be singled out as most important feature for performance. They have considerably memory bandwidth demands and little synchronization. For the Raytrace benchmark, both the RR arbitration and the caches contribute to performance enhancements. Their impact is limited though. The Queens benchmark demonstrates that the scalability of TDMA arbitration is limited if there is a considerable amount of synchronization. Combining RR arbitration and caching enables considerably higher performance for eight cores.

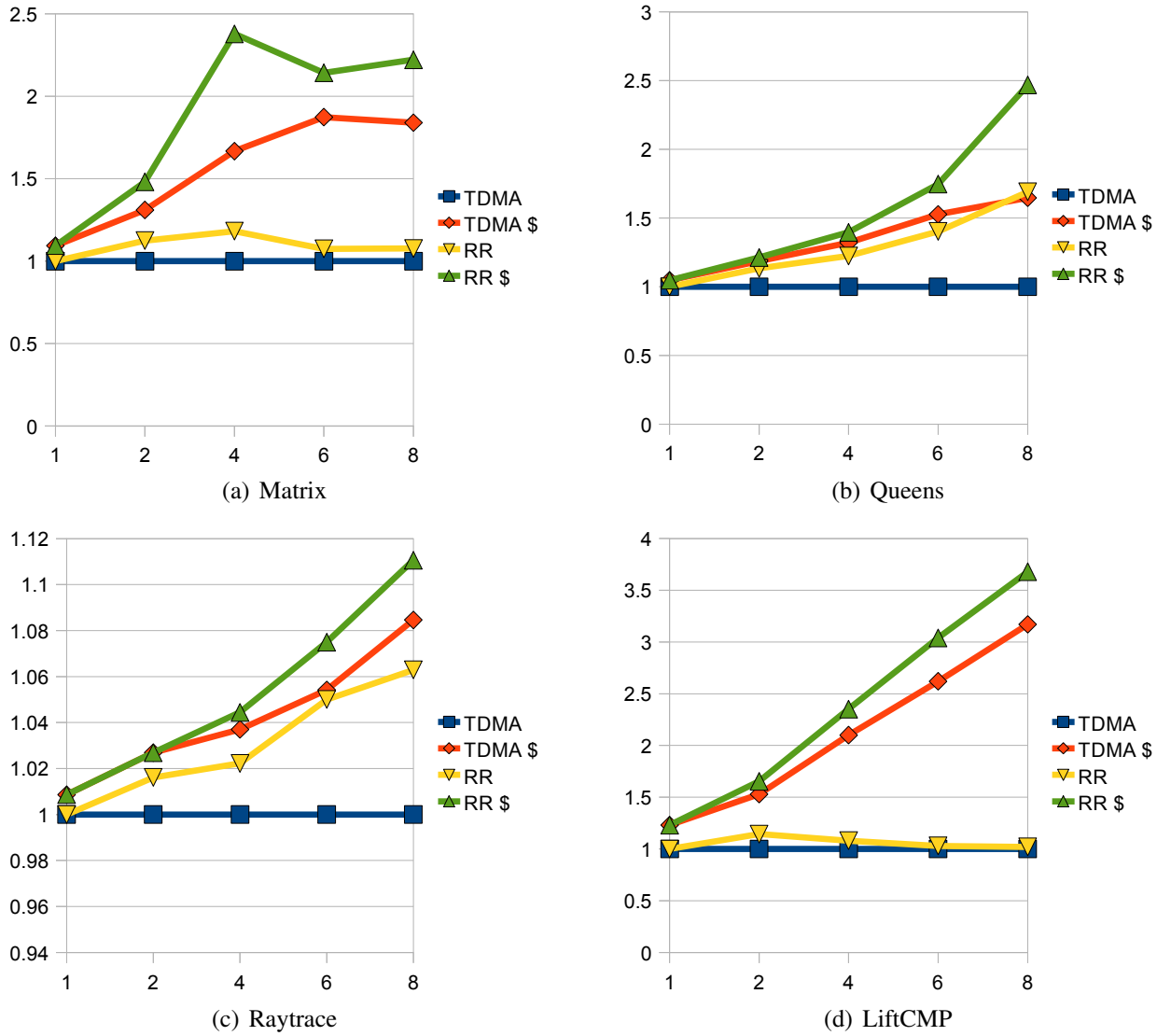


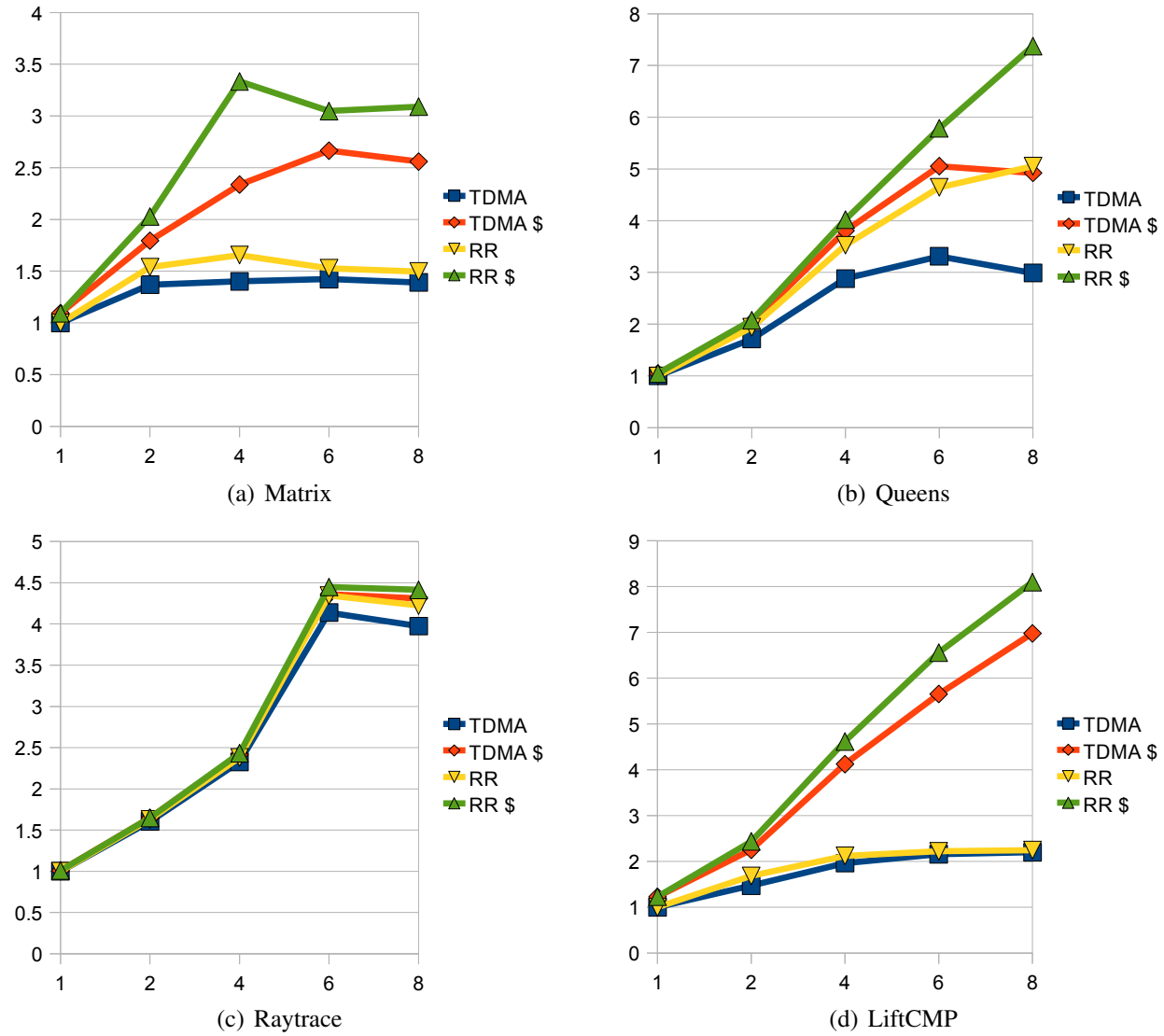
Figure 4: Benchmark results scaled to uncached TDMA multiprocessor

The usefulness of caching increases with the number of cores. For the uniprocessor version, the minor performance improvements hardly justify spending the additional hardware resources. When adding more cores, the effectiveness of the caches increases, and it becomes more likely that the improved performance warrants the additional costs.

3.5 Performance

Figure 5 shows the achievable speedups, compared to the plain, uncached uniprocessor configuration. The results are similar to the results displayed in Figure 3, but the speedups for the cached versions are slightly higher. Especially notable is the LiftCMP benchmark, where a speed up of 8.1 can be achieved with eight cores. This speedup is the result of combining heap caching and multiprocessing.

For all benchmarks the best performance could be achieved with the RR arbiter. However, as explained in Section 2.2, it may be difficult to guarantee the performance through static analysis. For



(a) Matrix (b) Queens

(c) Raytrace (d) LiftCMP

Figure 5: Benchmark results scaled to uncached uniprocessor

most benchmarks TDMA arbitration, which simplifies static analysis, leads to fairly similar speedups. An exception is the Queens benchmark, where synchronization limits the scalability.

For eight cores, RR $ achieves an average speedup of 5.7; TDMA $ achieves a speedup of 4.7. We believe that these speedups are reasonable, especially when considering that the processor design was driven by predictability rather than average case performance.

3.6 WCET Driven Object Cache Evaluation

The possible configurations of the object cache have been evaluated with our WCET analysis tool [14] to find the best configuration for embedded Java systems. For the caching of heap allocated objects we have evaluated different design options, such as cache line length and line or word fill. The detailed results can be found in [3] and some average case evaluations with cross-profiling in [11].

The details of the object cache analysis are not the focus of this deliverable, but some insights into its workings are necessary to interpret the evaluation results. As a consequence of the fact that the dataflow analysis abstracts the actual program, we do not always know the exact object a variable points to, but only have a set of objects the variable may point to at hand. Therefore, we perform a persistence analysis instead of a hit/miss classification, and try to identify scopes where accesses to an object are persistent. A particularly simple criteria for persistency in an object cache with associativity N is that at most N *distinct* objects are accessed within one scope. The main challenge for the analysis is thus to identify the number of distinct objects possibly accessed in a scope.

With this form of persistence analysis it is irrelevant if the replacement policy is LRU or FIFO. With *classic* cache analysis, FIFO replacement results in less hit classifications than LRU replacement [1].

We investigate two different behaviors on a cache miss: The first is to fill the whole cache line on a miss, loading all fields into the cache at once. This might be attractive if the memory has a longer latency for the first word accessed. The second option is to only load the missed field, which requires an additional tag bit for each word. It would also be possible to consider tradeoffs between these two extremes, e.g., loading four words on a cache miss, though this is not explored here.

For the evaluation of the object cache we consider several different system configurations: (1) the main memory is varied between a fast SRAM memory and a SDRAM memory with a higher latency; (2) uniprocessor and a 8 core chip-multiprocessor are considered. Finally we explore the difference between single word and full cache line loads on a cache miss. To compare the WCET analysis results with average case measurements the same configurations as in [11] are used.

The best cache configuration is dependent on the properties of the next level in the memory hierarchy. Longer latencies favor longer cache lines to spread the latency over possible hits due to spatial locality. Therefore, we evaluate two different memory configuration that are common in embedded systems: static memory (SRAM) and synchronous DRAM (SDRAM). For the SRAM configuration we assume a latency of two cycles for a 32 bit word read access. As an example of the SDRAM we select the IS42S16160B, the memory chip that is used on the Altera DE2-70 FPGA board. The latency for a read, including the latency in the memory controller, is assumed to be 10 cycles. The maximum burst length is 8 locations. As the memory interface is 16 bit, four 32 bit words can be read in 8 clock cycles. The resulting miss penalty for a single word read is 12 clock cycles, for a burst of 4 words 18 clock cycles. For longer cache lines the SDRAM can be used in page burst mode. With page burst mode, up to a whole page can be transferred in one burst. For shorter bursts the transfer has to be explicitly stopped by the memory controller. We assume the same latency of 10 clock cycles in the page burst mode.

For the evaluation of different object cache configurations we show the results of two single core benchmarks, *Lift* and *UDP/IP*, from the benchmark suit JemBench [12]. Further results can be found in [3].

Table 1 displays the detailed analysis results for the UDP/IP benchmark and Table 2 for the Lift benchmark. More results are available in an accompanying technical report [11]. The results are shown for a cache configuration with single word fill on a miss, complete line fill on a miss, and as a reference a cache configuration for single words, as we have presented it in [13].

From the results we can see that the maximum *analyzable* hit rate without line fill is between 46 % and 99 % [11]. This hit rate can be achieved with a moderate associativity between 2 and 16 way, depending on the program. Furthermore, even the simple configuration of a fully associative cache, as shown by the single field rows, results in a reasonable hit rate with a moderate associativity.

When the cache is configured with full line fill the hit rate naturally is increased as some fields that are later used will be loaded on the line fill. Longer lines result in higher hit rates. However, the miss penalty also increases and so the miss cycles per field access. The best configuration depends on the relation between latency and bandwidth of the main memory. For a main memory with a short latency, as represented by the SRAM configuration, individual field loads on a miss give a better miss cycles per access rate than filling the whole cache line.

For the SDRAM memory the optimal line size and whether individual fields should be filled is not so clear. There is at least one configuration for every benchmark, where a line fill configuration with 8 to 32 bytes per line (depending on the benchmark) performs better than the individual field fill. However, there is no single line size, which gives better results on line fill than on word fill for all benchmarks. Moreover, the performance gained using the optimal line fill configuration is relatively small. Choosing the line size optimal for one benchmark, results in a higher miss penalty for other benchmarks than using a word fill configuration. This result is a little bit different from average-case measurements with DaCapo application benchmarks [11]. With the DaCapo benchmarks single field fill was always more efficient than loading whole cache lines, which indicates only small spatial locality in heap allocated objects. For these reasons, we lean slightly towards filling only individual fields even with a SDRAM main memory.

4 Conclusion

In this deliverable 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 cache for heap allocated, shared data. The resulting speedup of a 8 core system compared to a uniprocessor system is between 3 and 7, depending on the type of application.

Table 1: Object cache hit rate and miss penalty per field access for the `UDP/IP` benchmark.

	Cache				Field access cost per access			
					Uniprocessor		8 core CMP	
Type	Size	Line	Assoc.	Hit rate	SRAM	SDRAM	SRAM	SDRAM
word fill	0 B	128 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	8 B	1 way	25.00 %	1.50	9.00	12.75	120.75
	16 B	16 B	1 way	58.33 %	0.83	5.00	7.08	67.08
	32 B	32 B	1 way	54.17 %	0.92	5.50	7.79	73.79
	64 B	64 B	1 way	54.17 %	0.92	5.50	7.79	73.79
	8 B	4 B	2 way	2.08 %	1.96	11.75	16.65	157.65
	16 B	8 B	2 way	27.08 %	1.46	8.75	12.40	117.40
	32 B	16 B	2 way	60.42 %	0.79	4.75	6.73	63.73
	64 B	32 B	2 way	62.50 %	0.75	4.50	6.38	60.38
	128 B	64 B	2 way	62.50 %	0.75	4.50	6.38	60.38
	16 B	4 B	4 way	2.08 %	1.96	11.75	16.65	157.65
	32 B	8 B	4 way	27.08 %	1.46	8.75	12.40	117.40
	64 B	16 B	4 way	60.42 %	0.79	4.75	6.73	63.73
	128 B	32 B	4 way	66.67 %	0.67	4.00	5.67	53.67
	256 B	64 B	4 way	66.67 %	0.67	4.00	5.67	53.67
line fill	0 B	128 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	8 B	1 way	29.17 %	1.75	8.83	14.71	114.04
	16 B	16 B	1 way	70.83 %	1.58	4.50	12.96	46.96
	32 B	32 B	1 way	66.67 %	5.33	8.67	43.00	101.67
	64 B	64 B	1 way	66.67 %	10.67	14.00	85.67	197.67
	8 B	4 B	2 way	2.08 %	1.96	11.75	16.65	157.65
	16 B	8 B	2 way	31.25 %	1.54	8.42	13.02	110.69
	32 B	16 B	2 way	72.92 %	1.04	3.75	8.60	43.60
	64 B	32 B	2 way	79.17 %	3.33	5.42	26.88	63.54
	256 B	128 B	2 way	79.17 %	13.33	15.42	106.88	243.54
	16 B	4 B	4 way	2.08 %	1.96	11.75	16.65	157.65
	32 B	8 B	4 way	31.25 %	1.54	8.42	13.02	110.69
	64 B	16 B	4 way	72.92 %	1.04	3.75	8.60	43.60
	256 B	64 B	4 way	83.33 %	5.33	7.00	42.83	98.83
	512 B	128 B	4 way	83.33 %	10.67	12.33	85.50	194.83
single field	0 B	4 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	4 B	2 way	4.17 %	1.92	11.50	16.29	154.29
	16 B	4 B	4 way	62.50 %	0.75	4.50	6.38	60.38
	32 B	4 B	8 way	66.67 %	0.67	4.00	5.67	53.67
	64 B	4 B	16 way	66.67 %	0.67	4.00	5.67	53.67

Table 2: Object cache hit rate and miss penalty per field access for the Lift benchmark.

Cache					Cache hit rate			
					Uniprocessor		8 core CMP	
Type	Size	Line	Assoc.	Hit rate	SRAM	SDRAM	SRAM	SDRAM
word fill	0 B	0 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	41.86 %	1.84	11.05	15.66	148.29
	8 B	8 B	1 way	3.92 %	1.98	11.89	16.85	159.59
	16 B	16 B	1 way	31.31 %	1.73	10.37	14.69	139.11
	8 B	4 B	2 way	90.70 %	1.66	9.95	14.09	133.46
	16 B	8 B	2 way	86.27 %	1.61	9.68	13.72	129.93
	32 B	16 B	2 way	88.89 %	1.23	7.37	10.44	98.86
	64 B	32 B	2 way	89.55 %	0.95	5.68	8.05	76.26
	128 B	64 B	2 way	90.54 %	0.23	1.37	1.94	18.36
	256 B	128 B	2 way	89.47 %	0.21	1.26	1.79	16.95
	16 B	4 B	4 way	93.02 %	1.65	9.89	14.02	132.75
	32 B	8 B	4 way	88.24 %	1.61	9.63	13.64	129.22
	64 B	16 B	4 way	89.90 %	1.22	7.32	10.36	98.15
	128 B	32 B	4 way	90.30 %	0.94	5.63	7.98	75.56
	256 B	64 B	4 way	90.99 %	0.22	1.32	1.86	17.65
	512 B	128 B	4 way	89.91 %	0.20	1.21	1.71	16.24
line fill	0 B	0 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	41.86 %	1.84	11.05	15.66	148.29
	8 B	8 B	1 way	5.88 %	2.55	12.42	21.41	158.88
	16 B	16 B	1 way	33.33 %	3.45	12.00	28.43	137.70
	8 B	4 B	2 way	90.70 %	1.66	9.95	14.09	133.46
	16 B	8 B	2 way	90.20 %	1.64	9.62	13.92	128.52
	32 B	16 B	2 way	94.95 %	1.31	7.18	11.04	94.62
	64 B	32 B	2 way	95.52 %	1.25	5.63	10.40	74.40
	128 B	64 B	2 way	97.30 %	0.89	1.42	7.21	19.84
	256 B	128 B	2 way	97.37 %	1.68	1.95	13.50	30.76
	16 B	4 B	4 way	93.02 %	1.65	9.89	14.02	132.75
	32 B	8 B	4 way	94.12 %	1.61	9.50	13.63	127.11
	64 B	16 B	4 way	96.97 %	1.24	7.03	10.47	93.21
	128 B	32 B	4 way	97.76 %	1.04	5.29	8.71	70.39
	256 B	64 B	4 way	98.65 %	0.47	0.87	3.83	12.04
	512 B	128 B	4 way	98.68 %	0.84	0.97	6.75	15.38
single field	0 B	4 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	4 B	2 way	0.00 %	2.00	12.00	17.00	161.00
	16 B	4 B	4 way	14.47 %	1.71	10.26	14.54	137.70
	32 B	4 B	8 way	84.65 %	0.31	1.84	2.61	24.71
	64 B	4 B	16 way	89.47 %	0.21	1.26	1.79	16.95
	128 B	4 B	32 way	89.91 %	0.20	1.21	1.71	16.24

References

[1] Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, July 2010.

[2] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.

[3] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object caches. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, pages 26–35, New York, NY, USA, 2010. ACM.

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

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

[6] 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.

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

[8] 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.

[9] Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 326–333, Sierre, Switzerland, March 2010. ACM Press.

[10] Martin Schoeberl and Peter Hilber. Design and implementation of real-time transactional memory. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 279–284, Milano, Italy, August 2010. IEEE Computer Society.

[11] Martin Schoeberl, Benedikt Huber, Walter Binder, Wolfgang Puffitsch, and Alex Villazon. Object cache evaluation. Technical report, Technical University of Denmark, 2010.

[12] 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.

[13] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number LNCS 5860, pages 180–191. Springer, November 2009.

[14] 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.

[15] Sascha Uhrig and Jörg 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 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[16] Martin Zabel, Thomas B. Preusser, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.