Design Space Exploration of Object Caches with Cross-Profiling

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

Walter Binder
Faculty of Informatics
University of Lugano, Switzerland
Email: walter.binder@usi.ch

Alex Villazón
Centro de Investigaciones de Nuevas
Tecnologías Informáticas (CINTI)
Universidad Privada Boliviana, Bolivia
Email: avillazon@upb.edu

***Abstract*—To avoid data cache trashing between heap-allocated data and other data areas, a distinct object cache has been proposed for embedded real-time Java processors. This object cache uses high associativity in order to statically track different object pointers for worst-case execution-time analysis. However, before implementing such an object cache, an empirical analysis of different organization forms is needed. We use a cross-profiling technique based on aspect-oriented programming in order to evaluate different object cache organizations with standard Java benchmarks. From the evaluation we conclude that field access exhibits some temporal locality, but almost no spatial locality. Therefore, filling long cache lines on a miss just introduces a high miss penalty without increasing the hit rate enough to make up for the increased miss penalty. For an object cache, it is more efficient to fill individual words within the cache line on a miss.**

***Keywords*-cache organizations; embedded Java processors; cache architecture evaluation**

I. Introduction

Real-time systems need to be time-predictable. The worst-case execution time (WCET) needs to be known for the schedulability analysis. As execution time measurement is not a safe estimation of the WCET, program and execution platform need to be analyzed statically [20].

Data cache hits and misses are hard to predict statically. The caching of different data areas (e.g., constants, stack, heap) in the same cache is the main obstacle to a tight analysis. In [13] the authors suggest to split the cache for different data areas. For caching of constants, static data, and stack frames, a standard cache organization is analyzable.

However, the analysis of caches for heap-allocated objects is challenging. The addresses of the objects, which is the input for standard cache analysis, are only known at runtime. As a solution to this issue, an object cache with high associativity has been proposed [14]. The cache content is tracked with symbolic addresses, employing a least-recently used or first-in first-out replacement policy. Evaluating the best tradeoff between hardware resources for the object cache and low miss cycles per field access is the topic of this paper. The cache organization is evaluated with standard Java benchmarks and compared with a direct mapped cache.

We base our evaluation on the Java processor JOP [12] and the chip-multiprocessor (CMP) version of it [10]. JOP is an implementation of the Java Virtual Machine (JVM) in hardware. The JVM bytecodes represent the instruction set of JOP. JOP has been designed to be time-predictable to enable WCET analysis [16]. In the original design, only instructions and stack data are cached. Heap-allocated data is not cached at all. However, when building CMP systems, the pressure on the memory bandwidth increases. To achieve scalability of embedded CMP systems, caching of constants, static data, and heap allocated data has been added to JOP.

The evaluation of cache organizations presented in this paper is based on a new cross-profiling technique using aspect-oriented programming (AOP) [8]. Thanks to cross-profiling [1], it is possible to rapidly evaluate architectural changes with large benchmark suites.

In [6] we have evaluated different organizations of the object cache with a WCET analysis approach. As only quite simple programs are WCET analyzable, we could not verify how the object cache scales for larger workloads. With cross-profiling, as presented in this paper, we are able to evaluate the object cache with large Java benchmarks.

In this paper we evaluate cache solutions for heap allocated objects to enable independent WCET analysis of data caches. The evaluation shows that access to object fields exhibit some temporal locality, but almost no spatial locality. Therefore, only small object caches are needed to enable the analyzable split cache design. The proposed fully associative object cache, which enables WCET analysis of access to heap allocated objects, performs just slightly worse than a direct mapped cache for objects, which is practically not analyzable for heap allocated objects. We show that time-predictable computer architecture can provide significant performance improvement without defeating WCET analysis.

This paper is structured as follows. In Section II we discuss related work and Section III describes the object cache organization. AOP-based cross-profiling for the object cache is described in Section IV. The methodology to retrieve hit rates and miss penalties is given in Section V. Results for a direct mapped and a fully associative object cache are presented in Section VI. The paper is concluded in Section VII.

II. Related Work

In this section, we discuss related work in the area of object caches, cache optimization, cross-profiling, and aspect-oriented programming.

A. Object Cache

One of the first proposals of an object cache [21] appeared within the Mushroom project [22]. The Mushroom project investigated hardware support for Smalltalk-like object oriented systems. The cache is indexed by a combination of the object identifier and the field offset. Different combinations, including xoring of the two fields, are explored to optimize the hit rate. Considering only the hit rate caches with a block size of 32 and 64 bytes perform best. However, under the assumption of realistic miss penalties caches with 16 and 32 bytes lines size result in lower average access times per field access. This result is a strong argument against just comparing hit rates.

A dedicated cache for heap allocated data is proposed in [17]. Similar to our proposed object cache, the object layout is handle based. The object reference with the field index is used to address the cache. Cache configurations are evaluated with a simulation in a Java interpreter. For different cache configurations (up to 32 KB) average case field access times between 1.5 and 5 cycles are reported. For most benchmarks the optimal block size was found to be 64 bytes. The proposed object cache is also used to cache arrays, whereas our object cache is intended for *normal* objects only. Therefore, the array accesses favor a larger block size to benefit from spatial locality. Object access and array access are quite different from the WCET analysis point of view. The field index for an object access is statically known, whereas the array index usually depends on a loop iteration.

Wright et al. propose a cache that can be used as object cache and as conventional data cache [23]. To support the object cache mode the instruction set is extended with a few object oriented instructions such as load and store of object fields. The object layout is handle based and the cache line is addressed with a combination of the object reference (called object id) and part of the offset within the object. The main motivation of the object cache mode is in-cache garbage collection of the youngest generation.

All proposed object caches are optimized for average case performance. It is common to use a hash function by xoring part of the object identifier with the field offset in order to equally distribute object within the cache. However, this hash function defeats WCET analysis of the cache content. In contrast, our proposed object cache is designed to maximize the ability to track the cache state in the WCET analysis [6].

B. Cache Evaluation

In [19] compile-time analysis is used to reduce the simulation time of an instruction cache. The cache simulation is linked into the benchmark executable to avoid generating long traces. Knowledge of basic blocks is used to reduce the number of invocations of the cache simulator.

Kroupis and Soudris present a instruction cache estimation based on program analysis [9]. Their assumption is that the program structure is the same at the high-level (C) and low-level (assembler). Similar to WCET analysis they build a control flow graph and analyze loops for the cache access pattern.

Janapsatya et. al argue that for embedded systems that are designed to run a single application the cache shall be optimized, with respect for energy consumption, for this application [7]. In their cache simulation tool they simultaneous evaluate different cache configurations.

Ghosh and Givargis present an analytical approach to cache optimization for embedded systems [4]. The presented algorithm takes a trace and the desired cache misses as input and outputs cache configurations that meet the constraints.

C. Cross-Profiling and Aspect-Oriented Programming

Most related work on cross-profiling aims at estimating the execution time of a program on a given target, while executing that program on a host. For example, cross-profiling techniques have been used to simulate parallel computers [2]. As the host processor may have a different instruction set than the target processor, cross-profiling tries to match up the basic blocks on the host and on the target machines, changing the estimates on the host to reflect the simulated target.

Many researchers have identified cross-cutting concerns in hardware design that can be modularized with aspects. Kiczales' original paper on AOP [8] discusses aspect-oriented approaches in hardware design. The Ptolemy project [3] studies modeling, simulation, and design of concurrent, real-time, embedded systems, focusing on the assembly of concurrent components. Interactions between heterogeneous components in Ptolemy are handled with concepts that are similar to aspects.

III. The Object Cache

In a modern object-oriented language, data is usually allocated on the heap. The addresses for the objects are only known at runtime. It is possible to analyze local cache effects with unknown addresses for a set-associative cache. For an n-way set associative cache the history for n different addresses can be tracked symbolically [13]. As the concrete addresses are unknown, a single access influences *all* sets in the cache. The analysis reduces the effective cache size to a single set.

We propose to implement the cache architecture exactly as it results from this analysis – a small, fully associative cache. The emphasis of the object cache is on associativity instead of capacity.

The object cache is organized to cache a whole object per cache line. Each cache line can only contain a single object. Objects cannot cross cache lines. If the object is larger than the cache line, the fields at higher indexes are not cached. While this might sound like a drastic restriction, it is the only way to keep the cache contents WCET analyzable for data with statically unknown addresses. In the evaluation section, we will show that this design is still efficient.

On standard caches, on a miss a full cache line is filled. However, with the proposed caching of a complete object in a line, the line size might become longer than typical. Therefore, it is also an option to fill only part of the line (e.g., single fields) on a miss. In the evaluation section we compare both configurations. For the fill of individual fields one valid bit per field, instead one per cache line, is needed. This extension increases the hardware for the tag memory as the valid bit is included in the tag memory. For single cycle hit detection, the tag memory and the valid bits need to be implemented in dedicated registers. The data memory itself can use on-chip memory as one cycle read latency can be tolerated on the cache data itself.

Furthermore, the cache organization is optimized for the object layout of JOP. The objects are accessed via an indirection called the handle. This indirection simplifies compaction during garbage collection. The tag memory contains the pointer to the handle (the Java reference) instead of the effective address of the object in the memory. If the access is a hit, additional to the field access the cost for the indirection is zero – the address translation has already been performed. The effective address of an object can only be changed by the garbage collection. For a coherent view of the object graph between the mutator and the garbage collector, the handle cache needs to be updated or invalidated after the move. The object fields can stay in the cache.

To enable static WCET analysis the cache is organized as write-through cache. A write-back organization leads to conservative WCET estimates as on each possible read miss another write back needs to be accounted for in the analysis.

On CMPs shared data need to be hold coherent and consistent between the core local caches and main memory. Standard cache coherence and consistence protocols are expensive to implement and limit the number of cores in a multiprocessor system. The Java Memory Model (JMM) [5] allows for a simple form of cache coherence protocol [11]. With a write-through cache, the cache can be held consistent, according to the rules of the JMM, by invalidating the cache on the start of a synchronized block (bytecode monitorenter), when invoking a synchronized method, and when reading from a volatile variable.

In the evaluation we assume that the cache line is not allocated on a write. The object cache is only used for objects and not for arrays. The access behavior for array data is quite different as it explores spatial locality instead of temporal locality. Therefore, a cache organized as two prefetch buffers is more adequate for array data. The details of a time-predictable cache organization for array data is not considered in this paper.

More details of a concrete implementation of the object cache can be found in a complementary paper [14].

IV. AOP-Based Cross-Profiling

Cross-profiling is a form of dynamic program analysis, where a program is executed in a *host* environment in order to gather dynamic metrics for a *target* environment [1]. That is, cross-profiling simulates relevant activities of an embedded target while executing programs on a host. In our case, the host is any state-of-the-art JVM running on a standard machine for software development, whereas the target is an embedded Java processor, such as JOP. With cross-profiling, programs are instrumented in order to compute dynamic metrics that represent an execution of the program (with the same input data) on the target. In our case, the host and the target have the same instruction set (i.e., JVM bytecodes).

One benefit of cross-profiling is the ability to execute large workloads on the host, which could not execute on the embedded target because of resource constraints. In the case of Java, there is a large variety of standard benchmark suites, such as DaCapo.[1] On an embedded Java processor, such as JOP, none of these benchmarks could be executed, since these benchmarks have considerable memory footprints. In addition, they all require a file system, which is often not available in embedded systems. Furthermore, cross-profiling allows to investigate many different cache configurations with moderate effort. With cross-profiling, we can leverage all available Java benchmarks in order to gather more data than would be possible on the embedded target processor.

Although the large benchmarks cannot be executed on current embedded systems, we are confident that they represent typical object oriented workloads that are representative for object oriented applications on future embedded systems. As there is currently a lack of (good) standard benchmarks tailored for embedded Java systems, the ability to use standard Java benchmark suites is an important advantage. Furthermore, in previous work we confirmed high accuracy of our cross-profiling technique with workloads that could be executed on the embedded target [1].

In prior work [1], we promoted cross-profiling as an effective method for evaluating the impact of processor design alternatives on performance, focusing on embedded Java processors and on optimizations of the instruction pipeline and of instruction caches. In contrast to our prior work, which was based on low-level code instrumentation techniques, the results presented in this paper were obtained with aspect-oriented programing (AOP) [8]. An aspect for cross-profiling is easier to define, tune, and extend, compared with a functionally equivalent implementation using low-level code instrumentation tools.

[1]http://dacapobench.org/

The advantage of using AOP for dynamic program analysis stems from the convenient high-level programming model offered by *join points* (representing specific points in the execution of a program), *pointcuts* (denoting a set of join points of interest), and *advices* (the analysis code to be executed when the control flow in the base program reaches a join point of interest). Our cross-profiling aspect has pointcuts to intercept object allocation, field access (read from and write to instance fields and static fields), and lock acquisition respectively lock release. The concrete implementation of our AOP-based cross-profiler is presented in an accompanying technical report [15].

For cache simulation, the object addresses, the object size, and the offsets for the individual fields within the object need to be known. Our cross-profiling aspect calculates the object address and size at object creation and uses reflection to extract the field offset. As the use of reflection to explore the fields of a type (include the supertypes' fields) is computationally expensive, it is done only once for each type, and the information regarding position and size are kept in a hash table. All data structures used by our object cache simulators are thread-safe, since we are analyzing also multi-threaded workloads.

The results presented in this paper were obtained with MAJOR [18], a Java-based AOP framework that guarantees that all bytecodes executed in the base program (including bytecodes in the Java class library) can be analyzed by the cross-profiling aspect at runtime.

V. Evaluation Methodology

For the evaluation of the object cache we consider several different system configurations. The main memory is varied between a fast SRAM memory and a higher latency SDRAM. We consider single-core and CMP systems. The difference between a full cache coherence protocol and the simplified version with a cache flush are compared. Finally, we explore the difference between single word and full cache line loads on a cache miss.

A. Benchmark Complexity

For the evaluation of different object cache organizations we use the DaCapo benchmark suite. To keep the execution time of the benchmarks with the cache simulation reasonable, we execute DaCapo with the small workload. As we are not benchmarking the JVM or a garbage collection implementation, this workload is large enough for our purpose. Table I shows some runtime statistics of the benchmarks with the small workload: the number of different object types encountered, number of allocated objects (bytecode new), allocated memory (not including arrays), and the number of object field reads (bytecode getfield).

Several hundred different object types, 90 thousand to 2 million allocated objects, and 3 to 180 million field reads represent a workload complex enough for the evaluation of the object cache.

Table I
Object oriented runtime behavior of the DaCapo benchmarks

Benchmark	Types	Objects	Memory	Field read
antlr	226	89655	3217 KB	10384746
bloat	393	2000113	50820 KB	34478024
chart	645	2003916	57070 KB	71280095
fop	756	153463	5066 KB	2970474
hsqldb	251	246416	7268 KB	5826300
jython	688	1907084	48243 KB	180312830
luindex	230	236261	7493 KB	19665635
lusearch	234	1002858	33233 KB	49167802
xalan	440	838577	36016 KB	91516743

Table II
Access times for a memory read operations in clock cycles

			8 core CMP		
	burst	1 CPU	min.	avg.	max.
SRAM	1 word	2	2	9.5	17
SRAM	2 words	4	4	19.5	35
SRAM	4 words	8	8	39.5	71
SDRAM	1 word	12	12	59.5	107
SDRAM	2 words	14	14	69.5	125
SDRAM	4 words	18	18	89.5	162

B. System Configurations

The best cache configuration is dependent on the properties of the next level in the memory hierarchy. Longer latencies favor longer cache lines. 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 (a 16 bit words). 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.

Furthermore, a single processor configuration and a chip-multiprocessor (CMP) configuration of 8 processor cores are compared. The CMP configuration is according to an implementation of a JOP CMP system on the Altera DE2-70 board. The memory access is arbitrated in TDMA mode with a minimum slot length s to fulfill a read request according to the cache line length. For n CPUs the TDMA round is $n \times s$ cycles. The effective access time depends on the phasing between the access request and the TDMA schedule. In the best case, the access is requested at the begin of the slot for the CPU and is $t_{min} = s$ cycles. In the worst case, the request is issued just in the second cycle of the slot and the CPU has to wait a full TDMA round till the start of the next slot:

$$t_{max} = n \times s - 1 + s = (n+1) \times s - 1$$

The average case access time is

$$\begin{aligned} t_{avg} &= \frac{t_{max} + t_{min}}{2} = \frac{(n+1) \times s - 1 + s}{2} \\ &= \frac{(n+2) \times s - 1}{2} \end{aligned}$$

Table II shows the memory access times for the different configurations and different burst lengths.

VI. Object Cache Evaluation

Two organizations of the object cache, a direct mapped standard cache and a fully associative object cache, are evaluated with the DaCapo benchmarks. The direct mapped cache configuration gives a baseline to which the proposed object cache can be compared with. We show the hit rates for all benchmarks and various cache configurations in tabular form. Due to space limitation only some benchmark results are shown in the following section. We picked the antlr benchmark, as it is in the middle field on the hit rate, to show detailed numbers on variations of the cache and line sizes and the difference between field and cache line fill on a miss in the object cache. The complete measurements are available in an accompanying technical report [15].

A. The Baseline

Table III shows the hit rates of different sized direct mapped caches for the DaCapo benchmarks. The hit rate is the number of field accesses that can be served by the cache divided by the number of accesses. An access that cannot be served by the cache is called a miss or cache miss. On a miss, data needs to be fetched from the main memory and the resulting addition time is called miss penalty.

Even with a very small cache of just 32 bytes there is a reasonable hit rate around 60%, except for the benchmarks hsqldb and xalan. The hit rate increases to around 90% for a cache size of 1/4 KB and for most benchmarks increasing the cache size above 1 to 2 KB gives less than 1% improvement. On the other end of the size spectrum, even with a cache of 128 KB, the hit rate does not approach 99%. We conclude that there is high locality in the small, as the hit rate with small caches is considerable. There are limits in locality that render caches bigger than a few KB useless. The table also shows different line sizes for some cache sizes. Longer cache lines usually increase the hit rate, except for very small caches.

Hit rate is only one property of a cache. The other important property is the penalty that needs to be payed on a cache miss. Longer cache lines usually result in a better hit rate, but the time to fill the cache line, the miss penalty, is higher. For memories with a high latency and high bandwidth, spatial locality in the access pattern will favor larger cache lines. If data nearby the actual address is also fetched, future access to those data will be a hit. This spatial locality works very well for an instruction cache and sequential access to arrays. However, access to object fields is less regular and the optimal line size of the cache will be different.

In Table IV the miss penalty per field read for different cache organizations and main memories is shown for the antlr benchmark. The top half of the table (CC) shows the results for a cache with standard cache coherency protocol; the bottom half of the table (Flush) shows the miss penalties for caches with flush on monitorenter. The lowest miss penalty is marked bold in the tables. The overall miss penalty of an application is calculated by multiplying the number of misses by the average cache load time as give in Table II. The miss cycles per field read in Table IV is the overall miss penalty divided by the number of field reads. The result is the number of (additional) clock cycles per field read.

For a main memory with a low latency (SRAM) an increase in the cache line length also increases the miss penalty – there is no latency that can be amortized by transferring data in burst mode to the cache line. However, even with the SDRAM memory longer cache lines lead to higher miss penalties on caches up to 1 KB. The relative decrease in miss rate is not enough to compensate for the increase in access time. And a cache line of 16 Bytes even decreases the hit rate. For a cache of 2 KB the increase in the line length slightly reduces the miss penalty.

The same trend for SRAM and SDRAM based main memory is also reflected in the CMP configuration. The main difference is that the miss penalty is about a factor of 5 higher for a 8 core CMP system than for a uniprocessor system.

The cache coherence with cache flush on monitorenter limits the maximum useful size of the cache. For the antlr benchmark we see a clear limit of the cache size of 2 KB.

B. Variation of the Object Cache

Table V shows the hit rate of different object cache configurations with the DaCapo benchmark. A full cache coherence protocol is assumed. The hit rate is slightly less than with a direct mapped cache.

More interesting is the actual miss penalty per field access (bytecode getfield). Table VI shows those numbers for the benchmark antlr. The upper part of the tables shows the average miss penalty when the cache update is only performed for single fields. The cache line tracks valid entries with a valid flag for each word. The bottom half

Table III
DIRECT MAPPED CACHE HIT RATE

Cache		Benchmark								
Size	Line	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	xalan
32 B	4 B	66.4 %	62.9 %	63.4 %	63.5 %	30.3 %	57.2 %	77.2 %	65.6 %	41.9 %
32 B	8 B	65.5 %	63.1 %	62.6 %	68.7 %	27.3 %	47.6 %	79.5 %	61.3 %	39.1 %
32 B	16 B	67.9 %	57.6 %	63.1 %	70.7 %	27.2 %	45.1 %	78.4 %	39.4 %	30.6 %
256 B	4 B	92.6 %	77.3 %	82.9 %	79.1 %	58.2 %	95.7 %	89.0 %	89.5 %	70.5 %
256 B	8 B	93.3 %	81.7 %	83.4 %	85.6 %	54.8 %	94.1 %	90.8 %	90.0 %	72.6 %
256 B	16 B	89.1 %	84.7 %	84.7 %	89.0 %	62.0 %	93.2 %	92.1 %	88.2 %	72.3 %
1 KB	4 B	93.4 %	82.5 %	88.6 %	81.6 %	71.3 %	97.6 %	92.5 %	93.4 %	82.8 %
1 KB	8 B	94.1 %	86.5 %	89.3 %	88.0 %	71.8 %	97.4 %	94.2 %	95.1 %	85.2 %
1 KB	16 B	90.1 %	89.8 %	90.3 %	92.0 %	77.8 %	97.6 %	95.7 %	95.9 %	86.1 %
2 KB	16 B	98.5 %	91.7 %	92.6 %	92.6 %	82.7 %	98.3 %	96.7 %	97.1 %	90.1 %
4 KB	16 B	98.6 %	93.1 %	93.6 %	93.0 %	86.6 %	98.7 %	97.2 %	97.6 %	92.9 %
8 KB	16 B	98.7 %	93.9 %	94.2 %	93.4 %	89.6 %	99.0 %	97.5 %	97.8 %	94.8 %

Table IV
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE ANTLR BENCHMARK

				Miss cycles per field read			
Cache				Uniprocessor		8 core CMP	
Type	Size	Line	Hit rate	SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	66.5 %	0.67	4.02	3.18	19.95
	32 B	8 B	65.6 %	1.38	4.82	6.72	23.93
	32 B	16 B	68.0 %	2.56	5.76	12.65	28.66
	256 B	4 B	92.6 %	0.15	0.88	0.70	4.38
	256 B	8 B	93.3 %	0.27	0.94	1.30	4.65
	256 B	16 B	89.1 %	0.87	1.96	4.30	9.75
	1 KB	4 B	93.4 %	0.13	0.80	0.63	3.94
	1 KB	8 B	94.1 %	0.24	0.82	1.15	4.09
	1 KB	16 B	90.1 %	0.80	1.79	3.93	8.90
	2 KB	4 B	96.3 %	**0.07**	0.44	**0.35**	2.18
	2 KB	8 B	97.7 %	0.09	0.32	0.44	1.58
	2 KB	16 B	98.5 %	0.12	**0.27**	0.59	**1.34**
Flush	32 B	4 B	63.6 %	0.73	4.37	3.46	21.67
	32 B	8 B	64.1 %	1.44	5.03	7.01	24.98
	32 B	16 B	66.3 %	2.70	6.07	13.32	30.17
	256 B	4 B	80.8 %	0.38	2.31	1.83	11.44
	256 B	8 B	84.2 %	0.63	2.22	3.09	11.00
	256 B	16 B	82.5 %	1.40	3.14	6.90	15.63
	1 KB	4 B	80.9 %	0.38	2.29	1.81	11.35
	1 KB	8 B	84.4 %	0.63	2.19	3.05	10.88
	1 KB	16 B	82.8 %	1.38	3.10	6.80	15.42
	2 KB	4 B	82.5 %	**0.35**	2.10	**1.66**	10.42
	2 KB	8 B	86.6 %	0.54	1.88	2.62	9.33
	2 KB	16 B	90.1 %	0.79	1.79	3.92	8.88
	4 KB	16 B	90.1 %	0.79	**1.78**	3.91	**8.86**

Table V
OBJECT CACHE HIT RATE

Cache			Benchmark								
Size	Line	Assoc.	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	xalan
64 B	64 B	1 way	41.4 %	31.0 %	43.9 %	48.2 %	10.5 %	28.7 %	66.9 %	22.0 %	7.1 %
128 B	128 B	1 way	41.4 %	30.9 %	43.9 %	49.2 %	10.7 %	28.9 %	66.9 %	21.9 %	5.7 %
256 B	256 B	1 way	41.4 %	30.9 %	43.9 %	50.5 %	10.7 %	28.9 %	66.9 %	21.9 %	5.6 %
128 B	64 B	2 way	55.2 %	55.7 %	59.5 %	63.7 %	19.8 %	45.3 %	77.8 %	73.1 %	25.3 %
256 B	128 B	2 way	55.2 %	55.7 %	59.4 %	65.3 %	20.0 %	45.5 %	77.6 %	73.9 %	22.8 %
512 B	256 B	2 way	55.2 %	55.7 %	59.5 %	68.7 %	20.0 %	45.5 %	77.6 %	73.9 %	22.7 %
256 B	64 B	4 way	79.8 %	64.6 %	67.6 %	68.9 %	26.0 %	76.0 %	81.5 %	84.7 %	48.3 %
512 B	128 B	4 way	79.8 %	64.6 %	66.9 %	71.5 %	26.3 %	76.2 %	81.9 %	85.6 %	49.8 %
1 KB	256 B	4 way	79.8 %	64.6 %	67.0 %	75.2 %	26.4 %	76.2 %	81.9 %	85.6 %	49.8 %
512 B	64 B	8 way	94.7 %	73.9 %	76.2 %	71.2 %	39.4 %	95.7 %	84.6 %	88.8 %	60.2 %
1 KB	128 B	8 way	94.8 %	73.9 %	76.5 %	74.1 %	39.9 %	95.9 %	85.1 %	89.8 %	63.6 %
2 KB	256 B	8 way	94.8 %	73.9 %	76.6 %	78.1 %	39.9 %	95.9 %	85.1 %	89.8 %	63.6 %
4 KB	256 B	16 way	95.7 %	77.3 %	80.0 %	79.8 %	44.4 %	97.3 %	88.4 %	92.3 %	72.9 %
8 KB	256 B	32 way	96.1 %	79.7 %	87.0 %	81.2 %	49.1 %	97.8 %	90.8 %	93.4 %	79.6 %

Table VI
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE ANTLR BENCHMARK

					Miss cycles per field read			
Cache					Uniprocessor		8 core CMP	
Type	Size	Line	Assoc.	Hit rate	SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	41.4 %	1.17	7.03	5.57	34.86
	128 B	128 B	1 way	41.4 %	1.17	7.03	5.57	34.86
	256 B	256 B	1 way	41.4 %	1.17	7.03	5.57	34.86
	128 B	64 B	2 way	55.2 %	0.90	5.37	4.25	26.63
	256 B	128 B	2 way	55.2 %	0.90	5.37	4.25	26.64
	512 B	256 B	2 way	55.2 %	0.90	5.37	4.25	26.64
	256 B	64 B	4 way	79.8 %	0.40	2.43	1.92	12.03
	512 B	128 B	4 way	79.8 %	0.40	2.43	1.92	12.03
	1 KB	256 B	4 way	79.8 %	0.40	2.43	1.92	12.03
	2 KB	256 B	8 way	94.8 %	0.11	0.63	0.50	3.13
	4 KB	256 B	16 way	95.7 %	0.09	0.52	0.41	2.56
	8 KB	256 B	32 way	96.1 %	**0.08**	**0.47**	**0.37**	**2.33**
CC Line	64 B	64 B	1 way	68.1 %	10.20	22.96	50.89	66.85
	128 B	128 B	1 way	68.1 %	20.42	45.95	101.96	117.91
	256 B	256 B	1 way	68.1 %	40.85	91.91	204.07	220.03
	128 B	64 B	2 way	83.5 %	5.26	11.86	26.29	34.54
	256 B	128 B	2 way	83.5 %	10.56	23.75	52.70	60.95
	512 B	256 B	2 way	83.5 %	21.11	47.50	105.48	113.73
	256 B	64 B	4 way	93.7 %	2.01	4.54	10.09	13.25
	512 B	128 B	4 way	93.7 %	4.04	9.08	20.16	23.32
	1 KB	256 B	4 way	93.7 %	8.07	18.17	40.34	43.50
	2 KB	256 B	8 way	98.5 %	1.97	4.43	9.83	10.60
	4 KB	256 B	16 way	98.8 %	1.47	3.31	7.36	7.94
	8 KB	256 B	32 way	99.0 %	**1.29**	**2.90**	**6.43**	**6.93**

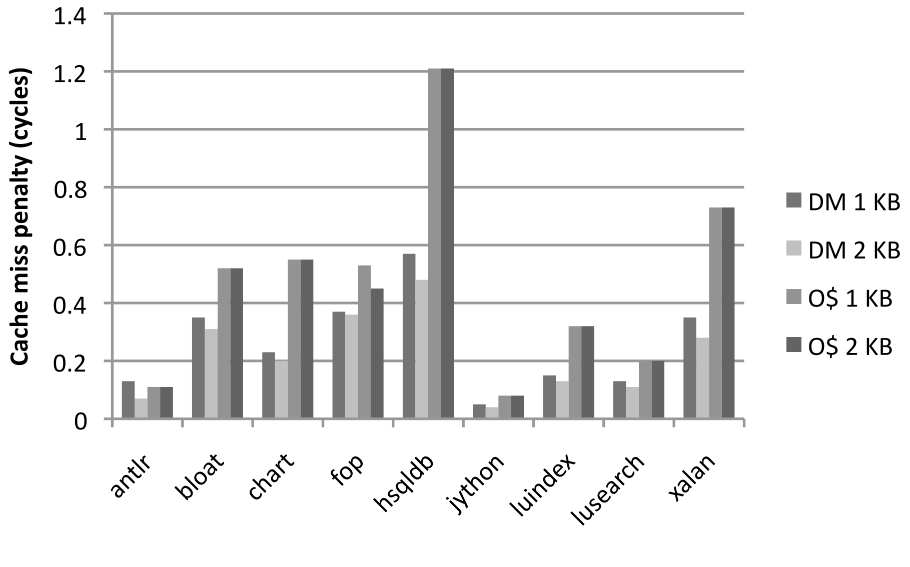

Figure 1. Miss penalty in clock cycles for different cache configurations

shows the miss penalty when the whole cache line is filled on a miss. The hit rate slightly increases due to some spatial locality. However, the cost for the cache line fill is way to high, even for the SDRAM configuration. The miss penalty renders the cache configuration with a complete line fill on a miss useless.

From the benchmarks results we conclude that with a single cache set and an associativity between 8 and 16 gives a reasonable hit rate. To compensate for the few cache lines, the individual cache lines (which can hold a single object) are considerable larger than in standard caches. With this long cache lines a fill of the complete line is impractical. The miss penalty is simply too high. The low spatial locality of object accesses even renders line fill with a high latency memory (SDRAM) on medium sized cache lines useless. Therefore, the suggested solution is to load individual words on a cache miss. Additional to the tag memory a valid bit for each cache word is needed for this configuration.

We have implemented exactly this object cache for the Java processor JOP [14]. The profiling results guided the concrete implementation. The number of cache lines and the number of fields is configurable in the hardware. Individual words are loaded on a miss.

C. Comparison

Figure 1 shows the miss penalty for all benchmarks for a few cache configurations and the SRAM based main memory. The bars labeled DM represent the direct mapped cache and O$ stands for object cache. As we have seen that the hit rate does not improve much above 1 and 2 KB for caching of objects, we have selected 1 and 2 KB configurations for the overview. The best configuration for the direct mapped cache and the object cache have been selected. The direct mapped cache line is 4 bytes (one word). The object cache is configured with single field update on a miss, as the former table showed that a full cache line fill is not beneficial. Both object cache sizes use an associativity of 8 and the resulting line sizes are 128 and 256 bytes.

The direct mapped cache performs slightly or considerable better than the object cache. However, the variation between the different benchmarks is way higher than the difference between the direct mapped cache and the object cache. The two sizes of the object cache perform in almost all cases similar. That means that the bottleneck for the object cache is on the number of cache lines (the associativity) and not on the cache size.

Furthermore, it has to be stressed that the direct mapped caching of objects is not WCET analyzable, but the object cache is. Therefore, we accept a reduction in the average case performance to gain on time predictability. The optimization for the worst case is different from the optimization for the average case.

VII. Conclusion

In this paper we have explored different organizations of an object cache for an embedded Java processor and for chip-multiprocessor versions of the Java processor. Aspect-oriented cross-profiling allows collecting cache hit/miss data for large workloads that are too big to be executed on an embedded system.

Based on a detailed quantitative evaluation of object cache organizations, we conclude that access to heap allocated objects exhibits only minor spatial locality. The major contribution to cache hits comes from temporal locality. Due to the low spatial locality, it is more beneficial to update single words in a cache line instead of filling the whole line on a miss. For the fully associative cache organization, this also holds for a main memory based on SDRAM devices with longer latency, but high burst bandwidth.

The proposed fully associative object cache performs almost as well as a standard cache organization, but is WCET analyzable. Furthermore, the dedicated cache for heap allocated objects enables tighter WCET analysis of the data cache for other data where the addresses are known statically. The object cache is an example that shows that time-predictable computer architecture can provide significant performance improvement without defeating WCET analysis.

References

[1] Walter Binder, Martin Schoeberl, Philippe Moret, and Alex Villazon. Cross-profiling for Java processors. *Software: Practice and Experience*, 39/18:1439–1465, 2009.

[2] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[3] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[4] Arijit Ghosh and Tony Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):419–440, 2004.

[5] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

[6] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object cache. 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.

[7] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. Finding optimal l1 cache configuration for embedded systems. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 796–801, Piscataway, NJ, USA, 2006. IEEE Press.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] N. Kroupis and D. Soudris. High-level estimation methodology for designing the instruction cache memory of programmable embedded platforms. *Computers Digital Techniques, IET*, 3(2):205 –221, march 2009.

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

[11] 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.

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

[13] 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.

[14] 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)*, 2011.

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

[16] 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.

[17] N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.

[18] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Comprehensive Aspect Weaving for Java. *Science of Computer Programming*, 2010.

[19] David B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. *SIGMETRICS Perform. Eval. Rev.*, 20(1):13–22, 1992.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[21] Ifor Williams and Mario Wolczko. An object-based memory architecture. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130, Martha's Vineyard, MA (USA), September 1990.

[22] Ifor W. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, 1989.

[23] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. *Sci. Comput. Program*, 62(2):145–163, 2006.