

Object Cache Evaluation

Technical Report, Technical University of Denmark 2010

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

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

Alex Villazón
Centro de Investigaciones de Nuevas
Tecnologías Informáticas (CINTI)
Universidad Privada Boliviana, Bolivia
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-cases execution time analysis. However, before implementing such an object cache, an empirical analysis of different organization forms is needed. We use a novel cross-profiling technique based on aspect-oriented programming in order to evaluate different object cache organizations for different processor configurations with standard Java benchmarks. From the evaluation of different cache organizations we conclude that field access exhibits some temporal locality, but almost no spatial locality. Therefore, long cache lines just introduce a high miss penalty without increasing the hit rate enough to make up for the miss penalty.

Index Terms—Processor architecture evaluation, embedded systems, cross-profiling, aspect-oriented programming, Java virtual machine

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 [36].

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. It has been proposed to split the cache into different data areas [26]. For caching constants, static data, and stack frames, a simple direct mapped cache 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 [29]. The cache content is tracked with symbolic addresses, employing a least-recently used (LRU) or first-in first-out (FIFO) replacement policy. In this paper this cache organization is evaluated with standard Java benchmarks and compared with a direct mapped cache.

We base our evaluation on the Java processor JOP [25] and the chip-multiprocessor (CMP) version of it [22]. JOP is an implementation of the Java virtual machine in hardware.

The bytecodes are the instruction set of JOP. JOP has been designed to be time-predictable, enabling WCET analysis [30]. 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 heap allocated data has been added to JOP. Evaluating the best tradeoff between hardware resources and the achievable hit rate is the topic of this paper.

One possible way to evaluate different cache organizations is to run simulations on VHDL models of the design. This approach requires that the different caches are implemented as VHDL models, and it is limited to benchmarks that can be executed on the target hardware. In addition, the turnaround time for experiments is in the range of hours. For these reasons, VHDL simulation is not well suited for design space exploration of cache organizations.

The evaluation of cache organizations presented in this paper is based on a novel cross-profiling technique using aspect-oriented programming (AOP). In prior work [5], [6], [27], 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. Thanks to cross-profiling, it is possible to evaluate architectural changes with large benchmark suites within minutes.

While our prior work on cross-profiling relied on low-level bytecode instrumentation techniques, resulting in complex tools that are difficult to extend, in this paper we advocate high-level AOP techniques for the rapid development of compact and extensible cross-profilers.

The original scientific contributions of this paper are three-fold. First, we introduce different designs of object caches that help improve WCET analysis for embedded Java processors. Second, we thoroughly evaluate these cache designs with standard Java benchmarks. Third, we introduce aspect-based cross-profiling as an effective and efficient approach to design space exploration for Java processors.

This paper builds on our prior work on cross-profiling [5]. In [4] we introduced cross-profiling for embedded Java

processors. The approach presented in [4] is based on constant cycle estimates for all bytecodes. As a major limitation, it does not take the presence of hardware caches into account, which requires a runtime simulation of the cache, resulting in different cycle estimates depending on cache hit or miss.

Cross-profiling has been used for general processor architecture evaluation in the context of Java processors [27]. In this paper we concentrate on one architectural detail, the object cache, and evaluate it in greater depth.

This paper is a complementary technical report to [28].

II. THE OBJECT CACHE

The object cache is organized to cache whole objects in a cache line. Each cache line can only contain a single object. Objects cannot cross cache lines. If the object is bigger than the cache line, the fields at higher indexes are not cached. Furthermore, the implementation in JOP 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 been already 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 cache analysis the cache is organized as write through cache. Write back is hard 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 CMP system. 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 more the spatial instead of the temporal locality. Therefore, a cache organized as two prefetch buffers is more adequate for array data.

Chip-multiprocessor (CMP) systems share the memory bandwidth between the on-chip processors and the pressure to avoid memory accesses is increased. Therefore, these systems call for large, processor local caches. Furthermore, some data needs to be held consistent between the processor local caches. Cache coherence and consistence protocols are expensive to implement and limit the number of cores in a multiprocessor system. The Java memory model (JMM) allows for a simple form of cache coherence protocol [24]. With a write through cache, the caches can be held consistent according to the rules of the JMM by invalidating the cache on start of a synchronized block of method (bytecode monitorEnter). The intuitive explanation is that at that point in time one core gets an actual view of the shared memory content. The

main memory contains the latest data as all data caches are organized as write through.

III. 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 [5], [6]. 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 (e.g., Sun's HotSpot VM or IBM's J9 VM on a desktop machine or on a laptop), whereas the target is an embedded Java processor, such as JOP.

Cross-profiling is a form of simulation, where 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, JVM bytecodes [18], which simplifies cross-profiling, as it can be implemented with the aid of bytecode instrumentation techniques. For the scope of this paper, we are interested in measuring dynamic metrics related to object access. Hence, the relevant bytecodes to be intercepted by cross-profiling include object allocation, read and write access to fields, as well as lock acquisition and release (since in Java, every object has an associated intrinsic lock [12], [13]).

Using cross-profiling, the target need not physically exist to gather interesting dynamic metrics; it is sufficient that the design of the target is known such that the relevant behavior can be simulated on the host. Consequently, cross-profiling is well suited for design space exploration for embedded processors, where the effects of different design choices can be modeled and simulated on a host, in order to explore which design alternative yields best performance. Only the most promising alternative is implemented in hardware afterwards. In this way, cross-profiling helps reduce time and cost in the development of embedded processors.

Another 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 [7], SPECjbb2005 [31], or SPECjvm2008 [32], to mention some of them. The DaCapo suite is continuously updated in order to comprehensively represent a wide range of Java applications. On an embedded Java processor, such as JOP, none of these benchmarks could be executed, since they all require a file system, which is not available on JOP. In addition, many of these benchmarks have considerable memory footprints. With cross-profiling, we can leverage all available Java benchmarks in order to gather more data than would be possible on the embedding target processor (assuming it already existed). As there is currently a lack of standard benchmarks tailored for embedded Java systems, the ability to use standard Java benchmark suites is an important advantage.

While cross-profiling can be many orders of magnitude faster than full simulation of a target processor using its hardware specification [5], [6], which makes cross-profiling practical for large workloads, it also has its limitations. Because of the following three reasons, results obtained with cross-profiling may be biased towards the host.

- 1) Some virtual machine activities on the host may be performed differently on the embedded target; examples include just-in-time compilation and garbage collection. If these activities are visible to the cross-profiling, they may affect the collected dynamic metrics. However, as we are using standard JVMs on the host where the bigger part of such runtime activities is implemented in native code and not amenable to bytecode instrumentation, the gathered dynamic metrics simply exclude these activities.
- 2) Host and target may use different versions and implementations of the Java class libraries. If the applications under cross-profiling make heavy use of the Java class library, the different library implementations have an impact on the collected dynamic metrics.
- 3) Thread scheduling is different on the host and on the embedded target, which can have an impact on cross-profiling for multi-threaded applications where concurrent threads can be executed at the same time. For such applications, in general, the cross-profiling results are not exactly reproducible, because of possible different thread scheduling in each run. Furthermore, the extra bytecodes executed for cross-profiling and the hardware configuration of the host, such as the number of available CPU cores, may affect the cross-profiling results.

IV. CROSS-PROFILING OBJECT ACCESS WITH ASPECT-ORIENTED PROGRAMMING

The results presented in this paper were obtained with a new cross-profiling technique using aspect-oriented programming (AOP) [16]. In the following text, we give a short overview of AOP and present the aspect-based cross-profiler used in our evaluation.

A. Dynamic Program Analysis with AOP

AOP [16] enables the specification of cross-cutting concerns in applications, avoiding related code that is scattered throughout methods, classes, or components. Traditionally, AOP has been used for disposing of “design smells”, such as needless repetition, and for improving maintainability of applications. Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, field accesses, etc. *Advices* are executed *before*, *after*, or *around* the intercepted join points. Advices have access to contextual information of the join points.

Dynamic cross-cutting with the pointcut and advice mechanism is particularly well-suited for defining different kinds of dynamic program analyses, such as profiling [3], [21], data race detection [1], [8], or memory leak detection [17], [34]. The advantage of using aspects for dynamic program

analysis stems from the convenient high-level model offered by join points (representing specific points in the execution of a program) and pointcuts (denoting a set of join points of interest). An aspect for dynamic program analysis is easier to define, tune, and extend, compared with a functionally equivalent implementation based on low-level code instrumentation tools. In this paper, we show that cross-profiling is yet another kind of dynamic program analysis that can be conveniently expressed with AOP.

As we are interesting in aspect-based cross-profiling for Java workloads, we need to choose an appropriate AOP framework for Java. AspectJ [15] is the de facto standard for AOP in Java. It offers a compiler to translate aspects into Java classes (e.g., advice are translated into Java methods), as well as a weaver that “applies” the compiled aspects to Java application classes. The AspectJ weaver operates at the bytecode level, locates join points in the application bytecode that match pointcuts in the aspects, and inserts invocations to the compiled advice methods [14].

A practical impediment for aspect-based dynamic program analysis is that most AOP systems—including AspectJ—do not support weaving in the standard Java class library (because of bootstrapping issues and to avoid infinite recursions when advice invoke methods in the Java class library). In prior work, we developed the AOP framework MAJOR [34], [35] that addresses exactly this limitation. MAJOR supports most features in the AspectJ language and is based on the AspectJ weaver. MAJOR ensures aspect weaving with full bytecode coverage, that is, any method that has a bytecode representation can be woven, including methods in the Java class library. For aspect-based cross-profiling, full bytecode coverage is essential in order to ensure that the gathered dynamic metrics represent overall program execution on the host. Consequently, we are using MAJOR for weaving the cross-profiling aspect presented in this section.

B. Practical Considerations

With MAJOR, all JDK classes are woven, but the DIB allows each thread to state whether it wants to execute the original or the woven version of methods in the JDK.

When running a dynamic analysis aspect, in general, one should ensure that only non-woven code gets executed by the aspect, otherwise one risks perturbations and infinite recursions. For this reason, the DIB is activated during the execution (dynamic extend) of all advice methods.

For the JDK classes, we need code duplication within method bodies and the DIB mechanism, because each JDK class can be loaded only once, and we need two code versions, one woven (executed by program code - base level) and one unmodified (executed by advice methods - meta level).

For program classes, we need only a single code version, the woven one, if we assume that the dynamic analysis does not call back into program code (which is a reasonable assumption – and why one should not invoke equals() or hashCode() on objects used by the base-level program).

```

public interface OCache {
    void allocation(Object o);
    void getfield(Object o, String fieldname);
    void putfield(Object o, String fieldname);
    void getstatic(Class c, String fieldname);
    void putstatic(Class c, String fieldname);
    void moniorenter(Object o);
    void monitorexit(Object o);
}

```

Fig. 1. Interface for object cache simulation

For the classes of the dynamic analysis, only a single code version is needed, the original one. That is, one has to make sure that the aspect is not applied to these classes. This is why you need to specify the Exclusion pointcuts in the aspect are specified to avoid weaving of code invoked from the aspects. This is a general issue of AOP languages, called the conflation of base-level and meta-level. So programmers using AOP often encounter problems with mix of woven and unwoven code.

In short, with MAJOR one can safely invoke JDK methods on objects that are created in the aspect as they are protected by DIB. The aspect methods (and resulting invoked JDK methods) shall not call methods of objects that belong to the application under test (the equals() example). Usage of IdentityHashMap is safe as it does not use objects equal and hash code and it is protected by DIB.

C. Cross-profiling Aspect

Figure 1 presents the OCache interface that an object cache simulator must implement. Figure 2 shows the (simplified) cross-profiling aspect OCacheAspect that invokes an object cache simulator through the OCache interface. The object cache simulator is a singleton, specified through a system property, and instantiated by the aspect (details not shown in Figure 2).

Thanks to MAJOR [34], [35], OCacheAspect is woven into all classes linked by the JVM, except for the classes that represent the cross-profiler, which include the interface OCache, the aspect OCacheAspect, and the implementations of the object cache simulators; we assume all these classes to reside in packages that start with org.jop.cache. The pointcut exclusion() in Figure 2 excludes all classes in these packages from weaving.

The first advice in OCacheAspect (with the pointcut call(*.new(..))) intercepts all object allocations and invokes the allocation(Object) method of the object cache simulator. The second and third advice (with the pointcuts get(!static * *) resp. set(!static * *) intercept read respectively write access to all instance fields, invoking the methods getfield(Object, String) respectively setfield(Object, String) in the simulator. The first argument of type Object represents the instance where the field is accessed, whereas the second argument of type String provides a unique identifier (i.e., a signature) for the field name in the object. The AspectJ pseudo-variable thisJoinPointStaticPart provides such static information for matching join points. The third and the fourth advice (with the pointcuts get(static * *) resp. set(static * *)) intercept read and write access to static fields. The Class instance holding the accessed static field and

```

public aspect OCacheAspect {

    // create singleton OCache instance
    private static final OCache cache = ...;

    // pointcut to prevent weaving in the classes of the cross-profiler
    pointcut exclusion() : !within(org.jop.cache.**);

    // object allocation
    after() returning(Object o) : call(*.new(..)) &&
        exclusion() {
        cache.allocation(o);
    }

    // read access to instance field
    before(Object o) : get(!static * *) && target(o) &&
        exclusion() {
        Signature s =
            thisJoinPointStaticPart.getSignature();
        cache.getfield(o, s.toShortString());
    }

    // write access to instance field
    before(Object o) : set(!static * *) && target(o) &&
        exclusion() {
        Signature s =
            thisJoinPointStaticPart.getSignature();
        cache.putfield(o, s.toShortString());
    }

    // read access to static field
    before() : get(static * *) && exclusion() {
        Signature s =
            thisJoinPointStaticPart.getSignature();
        Class c = s.getDeclaringType();
        cache.getstatic(c, s.toShortString());
    }

    // write access to static field
    before() : set(static * *) && exclusion() {
        Signature s =
            thisJoinPointStaticPart.getSignature();
        Class c = s.getDeclaringType();
        cache.putstatic(c, s.toShortString());
    }

    // acquisition of intrinsic lock (including entry of synchronized method)
    before(Object o): lock() && args(o) && exclusion() {
        cache.moniorenter(o);
    }

    // release of intrinsic lock (including completion of synchronized method)
    after(Object o): unlock() && args(o) && exclusion() {
        cache.monitorexit(o);
    }

    ...
}

```

Fig. 2. Simplified aspect for cross-profiling object accesses

a String identifying the field name are passed to the methods getstatic(Class, String) respectively putstatic(Class, String) of the object cache simulator. Finally, the last two advice (with the pointcuts lock() resp. unlock()) intercept acquisition and release of intrinsic locks.¹ The lock() and unlock() pointcut designators match both synchronized methods and synchronized blocks.

¹In Java, each object has an associated intrinsic lock [12], [13]. The intrinsic lock of an object is implicitly acquired upon invocation of a synchronized method. For synchronized static methods, the intrinsic lock of the corresponding Class instance is used. The intrinsic lock is released upon normal or abnormal completion of a synchronized method. Intrinsic locks can also be explicitly acquired using synchronized{} blocks.

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
python	688	1907084	48243 KB	180312830
luindex	230	236261	7493 KB	19665635
lusearch	234	1002858	33233 KB	49167802
xalan	440	838577	36016 KB	91516743

D. Object Cache Simulation

Our implementations of the object cache simulator use reflection to map each accessed field to position and size of the field using the object memory layout of the embedded target, taking also alignment into account. 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 analysing also multi-threaded workloads.

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 is 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 suit [7].

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 6 to 180 million field reads represent a workload complex enough for the evaluation of the object cache.

B. System Configurations

A cache design is not independent 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:

TABLE II
ACCESS TIMES FOR A MEMORY READ OPERATIONS IN CLOCK CYCLES

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

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.

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

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

Table II shows the memory access times for the different configurations.

VI. OBJECT CACHE EVALUATION

Two organizations of the object cache, a direct mapped cache and a fully associative 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.

A. The Baseline

Table III shows the hit rate of a different sized direct mapped caches for the DaCapo benchmarks. Even with a very small

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 %
16 KB	16 B	98.7 %	94.3 %	94.5 %	93.5 %	92.1 %	99.1 %	97.7 %	97.8 %	96.1 %
32 KB	16 B	98.7 %	94.8 %	95.0 %	93.6 %	93.9 %	99.1 %	98.0 %	97.8 %	97.0 %
64 KB	16 B	98.7 %	95.0 %	95.4 %	93.8 %	94.9 %	99.2 %	98.1 %	97.8 %	97.6 %
128 KB	16 B	98.7 %	95.2 %	95.9 %	94.1 %	95.0 %	99.2 %	98.2 %	97.8 %	97.9 %

cache of just 32 bytes there is a noticeable 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. If the increase of the hit rate really pays off for the higher miss penalty depends on the properties of the next level in the memory hierarchy. The details on the effects of the lines size is shown later.

The proposal to avoid hardware cache coherence protocols by flushing the cache on monitorenter and access to volatile fields for CMP systems further limits the useful cache size. Table IV shows the hit rates for a cache organization with cache flush on monitorenter. The hit rate for very small cache configurations (32 to 256 Bytes) is a little bit less than the hit rates without cache flushing. More dominant is the practical limit of the cache size to 2 to 4 KB with an achievable hit rate between 53% and 90%. If this hit rate reduction pays off by the higher scalability of the cache organization needs a more complete comparison.

Hit rate is only one property of a cache. The other important property is the penalty that needs to be payed on a miss. Longer cache lines give a better hit rate, but the time to fill the cache line, the miss penalty, is higher. For memories with a high latency, 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 V the miss penalty for different cache organizations and main memories is shown for the antlr benchmark. The miss penalty is calculated by multiplying the number of misses by the average cache load time as give in Table II. In Table V the miss penalty is scaled to all object 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 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. For the bloat benchmark, shown in Table VI, longer cache lines lead to less miss cycles for cache sizes of 1 KB or larger.

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 bottom half of the table shows the miss penalties for caches with flush on monitorenter. For the antlr benchmark we see a clear limit in the useful maximum cache size of 2 KB. With the bloat benchmark the miss cycles can be reduced by 1 to 2% with larger caches.

B. Variation of the Object Cache

Table XIV shows the hit rate of different object cache configurations with the DaCapo benchmark. A full cache coherence protocol is assumed. Table XV shows the hit rate for the simplified cache coherence with full cache flush on monitorenter. 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 XVI and Table XX show those

TABLE IV
DIRECT MAPPED CACHE HIT RATE WITH CACHE FLUSH ON SYNCHRONIZED BLOCKS

Cache		Benchmark								
Size	Line	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	xalan
32 B	4 B	63.5 %	50.5 %	52.7 %	54.4 %	27.9 %	33.1 %	74.1 %	59.4 %	34.8 %
32 B	8 B	64.0 %	52.7 %	55.6 %	62.6 %	25.5 %	35.7 %	76.8 %	56.7 %	33.2 %
32 B	16 B	66.2 %	50.4 %	58.1 %	65.9 %	25.6 %	35.9 %	76.2 %	36.9 %	26.1 %
256 B	4 B	80.6 %	60.6 %	65.3 %	66.6 %	41.2 %	38.5 %	84.1 %	77.8 %	51.1 %
256 B	8 B	84.0 %	66.6 %	69.3 %	76.0 %	44.5 %	41.8 %	86.8 %	80.5 %	57.2 %
256 B	16 B	82.4 %	70.3 %	73.5 %	80.9 %	55.7 %	51.8 %	88.8 %	80.2 %	60.3 %
1 KB	4 B	80.8 %	62.9 %	70.0 %	67.6 %	42.6 %	38.7 %	86.9 %	80.2 %	53.8 %
1 KB	8 B	84.2 %	69.2 %	74.2 %	77.1 %	47.2 %	42.1 %	89.5 %	84.0 %	61.2 %
1 KB	16 B	82.6 %	73.4 %	78.4 %	82.4 %	60.6 %	52.7 %	91.6 %	86.1 %	66.5 %
2 KB	16 B	89.9 %	74.3 %	80.5 %	82.6 %	61.7 %	52.8 %	92.4 %	86.9 %	67.5 %
4 KB	16 B	90.0 %	74.8 %	81.2 %	82.6 %	62.1 %	52.9 %	92.8 %	87.1 %	68.0 %
8 KB	16 B	90.0 %	75.1 %	81.7 %	82.7 %	62.2 %	52.9 %	93.0 %	87.2 %	68.3 %
16 KB	16 B	90.0 %	75.3 %	81.9 %	82.7 %	62.3 %	52.9 %	93.2 %	87.2 %	68.4 %
32 KB	16 B	90.0 %	75.4 %	82.0 %	82.7 %	62.3 %	52.9 %	93.4 %	87.2 %	68.5 %
64 KB	16 B	90.0 %	75.4 %	82.0 %	82.7 %	62.3 %	52.9 %	93.4 %	87.2 %	68.5 %
128 KB	16 B	90.0 %	75.5 %	82.2 %	82.7 %	62.3 %	52.9 %	93.5 %	87.2 %	68.5 %

TABLE V
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE ANTLR BENCHMARK

Cache				Miss cycles per field read			
				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
	4 KB	16 B	98.6 %	0.11	0.25	0.55	1.26
	8 KB	16 B	98.7 %	0.11	0.24	0.53	1.20
	16 KB	16 B	98.7 %	0.10	0.24	0.52	1.17
	32 KB	16 B	98.7 %	0.10	0.23	0.51	1.16
64 KB	16 B	98.7 %	0.10	0.23	0.51	1.15	
128 KB	16 B	98.7 %	0.10	0.23	0.50	1.14	
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
	8 KB	16 B	90.1 %	0.79	1.78	3.91	8.86
	16 KB	16 B	90.1 %	0.79	1.78	3.91	8.85
	32 KB	16 B	90.1 %	0.79	1.78	3.91	8.85
64 KB	16 B	90.1 %	0.79	1.78	3.91	8.85	
128 KB	16 B	90.1 %	0.79	1.78	3.91	8.85	

TABLE VI
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE BLOAT BENCHMARK

Cache			Hit rate	Miss cycles per field read			
Type	Size	Line		Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	64.4 %	0.71	4.27	3.38	21.16
	32 B	8 B	64.7 %	1.41	4.94	6.88	24.53
	32 B	16 B	59.4 %	3.24	7.30	16.02	36.29
	256 B	4 B	77.7 %	0.45	2.68	2.12	13.28
	256 B	8 B	82.2 %	0.71	2.50	3.48	12.40
	256 B	16 B	85.4 %	1.16	2.62	5.75	13.03
	1 KB	4 B	82.7 %	0.35	2.08	1.65	10.32
	1 KB	8 B	86.8 %	0.53	1.85	2.58	9.18
	1 KB	16 B	90.2 %	0.79	1.77	3.88	8.79
	2 KB	4 B	84.4 %	0.31	1.88	1.49	9.31
	2 KB	8 B	88.6 %	0.46	1.60	2.23	7.95
	2 KB	16 B	91.9 %	0.65	1.46	3.20	7.25
	4 KB	16 B	93.0 %	0.56	1.25	2.75	6.23
	8 KB	16 B	93.8 %	0.50	1.12	2.46	5.58
	16 KB	16 B	94.2 %	0.46	1.04	2.28	5.16
	32 KB	16 B	94.6 %	0.43	0.96	2.12	4.80
64 KB	16 B	94.9 %	0.41	0.92	2.02	4.58	
128 KB	16 B	95.1 %	0.39	0.89	1.95	4.42	
Flush	32 B	4 B	49.7 %	1.01	6.03	4.78	29.92
	32 B	8 B	52.0 %	1.92	6.72	9.35	33.34
	32 B	16 B	49.4 %	4.05	9.10	19.98	45.27
	256 B	4 B	59.8 %	0.80	4.83	3.82	23.94
	256 B	8 B	65.9 %	1.36	4.77	6.64	23.67
	256 B	16 B	69.7 %	2.42	5.45	11.97	27.12
	1 KB	4 B	62.2 %	0.76	4.54	3.59	22.49
	1 KB	8 B	68.7 %	1.25	4.39	6.11	21.78
	1 KB	16 B	73.0 %	2.16	4.86	10.67	24.17
	2 KB	4 B	62.9 %	0.74	4.45	3.53	22.08
	2 KB	8 B	69.4 %	1.22	4.28	5.96	21.24
	2 KB	16 B	73.9 %	2.09	4.70	10.30	23.35
	4 KB	16 B	74.4 %	2.04	4.60	10.10	22.88
	8 KB	16 B	74.8 %	2.02	4.54	9.97	22.58
	16 KB	16 B	75.0 %	2.00	4.51	9.89	22.40
	32 KB	16 B	75.1 %	1.99	4.49	9.84	22.30
64 KB	16 B	75.1 %	1.99	4.48	9.82	22.26	
128 KB	16 B	75.2 %	1.99	4.47	9.81	22.23	

numbers for the benchmarks `antlr` and `hsql`. 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 shows the miss penalty when the whole cache line is filled on a miss. This hit rate slightly increases due to some spatial locality. However, the cost for the cache line fill is way too high, even for the SDRAM configuration. The miss penalty, especially seen with the `hsql` benchmark, renders the cache configuration with a complete line fill on a miss useless.

We conclude that a fully associative cache configuration with a medium line size to cache full objects is practical for a time-predictable system. To benefit from predictable hits only individual words are updated on a cache miss.

VII. RELATED WORK

In this section, we discuss related work in the area of cross-profiling, aspect-oriented programming, embedded Java processors, and object caching.

A. Cross-profiling

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 [11]. 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.

In [5], [6] we introduced CProf, a cross-profiling framework for embedded Java processors. In the case of CProf, the host and the target have the same instruction set—Java bytecodes. CProf assumes that there are constant cycle estimates for most bytecodes; recent Java processors, such as the Java Optimized Processor JOP [25], meet this requirement. In the beginning of each basic block of code, CProf inserts code to increment a counter by an estimate for the CPU cycles required for executing the basic block on the target. CProf is able to simulate instruction caches that cache whole method bodies, which are incorporated in several recent embedded Java

TABLE VII
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE CHART BENCHMARK

Cache			Miss cycles per field read				
Type	Size	Line	Hit rate	Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	63.4 %	0.73	4.39	3.48	21.77
	32 B	8 B	62.3 %	1.51	5.27	7.34	26.17
	32 B	16 B	62.9 %	2.97	6.68	14.65	33.20
	256 B	4 B	82.9 %	0.34	2.06	1.63	10.19
	256 B	8 B	83.4 %	0.67	2.33	3.24	11.56
	256 B	16 B	84.6 %	1.23	2.77	6.09	13.79
	1 KB	4 B	88.6 %	0.23	1.37	1.09	6.81
	1 KB	8 B	89.3 %	0.43	1.50	2.10	7.47
	1 KB	16 B	90.3 %	0.77	1.74	3.82	8.66
	2 KB	4 B	90.2 %	0.20	1.18	0.94	5.86
	2 KB	8 B	91.3 %	0.35	1.21	1.69	6.02
	2 KB	16 B	92.8 %	0.58	1.30	2.86	6.47
	4 KB	16 B	93.6 %	0.51	1.14	2.51	5.69
	8 KB	16 B	94.3 %	0.46	1.03	2.27	5.14
	16 KB	16 B	94.5 %	0.44	0.98	2.16	4.89
	32 KB	16 B	95.0 %	0.40	0.90	1.97	4.46
64 KB	16 B	95.4 %	0.37	0.82	1.80	4.09	
128 KB	16 B	95.9 %	0.32	0.73	1.60	3.63	
Flush	32 B	4 B	52.8 %	0.94	5.67	4.49	28.11
	32 B	8 B	55.8 %	1.77	6.19	8.62	30.73
	32 B	16 B	58.4 %	3.32	7.48	16.41	37.19
	256 B	4 B	65.4 %	0.69	4.16	3.29	20.61
	256 B	8 B	69.3 %	1.23	4.30	5.99	21.34
	256 B	16 B	73.7 %	2.10	4.74	10.39	23.54
	1 KB	4 B	70.1 %	0.60	3.59	2.84	17.80
	1 KB	8 B	74.2 %	1.03	3.62	5.04	17.95
	1 KB	16 B	78.3 %	1.73	3.90	8.56	19.40
	2 KB	4 B	71.4 %	0.57	3.44	2.72	17.04
	2 KB	8 B	75.9 %	0.96	3.38	4.70	16.76
	2 KB	16 B	80.4 %	1.57	3.53	7.76	17.57
	4 KB	16 B	81.2 %	1.50	3.38	7.42	16.82
	8 KB	16 B	81.8 %	1.46	3.28	7.19	16.29
	16 KB	16 B	81.9 %	1.45	3.25	7.14	16.17
	32 KB	16 B	82.0 %	1.44	3.24	7.10	16.10
64 KB	16 B	82.1 %	1.43	3.23	7.08	16.05	
128 KB	16 B	82.2 %	1.42	3.20	7.01	15.89	

processors, including JOP [25], SHAP [23], and CarCore [19]. CProf is implemented with low-level bytecode instrumentation techniques and therefore difficult to extend. For this reason, we adopted a new AOP-based cross-profiling approach for exploring the design of object caches. Our cross-profiling aspect is compact and easy to modify and extend, allowing us to rapidly investigate many different design alternatives.

B. Aspect-Oriented Programming

AOP has been successfully used for implementing various dynamic analysis tools, including profilers [21], memory leak detectors [17], [34], and data race detectors [8]. The use of AspectJ for profiling is explored in [21], yielding mixed results. On the one hand, it is possible to express a variety of profiling tasks as concise aspects. On the other hand, the AspectJ weaver does not support comprehensive aspect weaving (it prevents weaving the Java class library), thus resulting in incomplete profiles. Our approach to aspect-based cross-profiling avoids this restriction thanks to the use of MAJOR [34], [35], which makes comprehensive aspect weaving possible.

While AspectJ [15] supports all join points that are relevant for simulating object caches, it lacks some join points that would be needed for other cross-profiling use cases, such as estimating the number of executed CPU cycles on the target. For such use cases, one would need additional join points at the level of basic blocks of code [5]. There are several frameworks that ease the extension of AOP languages and frameworks with new pointcuts, such as Josh [10] or the AspectBench Compiler (abc) [2]. Josh [10] is an AOP language similar to AspectJ, which allows defining new pointcut designators. Josh is based on the Javassist bytecode instrumentation library [9]. abc [2] is an extensible AspectJ compiler. Its frontend is based on Polyglot [20], an extensible compiler framework for Java, whereas the backend relies on the Soot [33] framework.

VIII. CONCLUSION

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

TABLE VIII
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE FOP BENCHMARK

Cache Type	Cache		Hit rate	Miss cycles per field read			
	Size	Line		Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	63.5 %	0.73	4.38	3.47	21.70
	32 B	8 B	68.7 %	1.25	4.38	6.09	21.72
	32 B	16 B	70.7 %	2.34	5.27	11.57	26.21
	256 B	4 B	79.1 %	0.42	2.51	1.99	12.44
	256 B	8 B	85.6 %	0.58	2.02	2.81	10.03
	256 B	16 B	89.0 %	0.88	1.97	4.33	9.80
	1 KB	4 B	81.6 %	0.37	2.21	1.75	10.97
	1 KB	8 B	88.0 %	0.48	1.68	2.33	8.32
	1 KB	16 B	92.0 %	0.64	1.44	3.16	7.16
	2 KB	4 B	82.2 %	0.36	2.14	1.69	10.59
	2 KB	8 B	88.6 %	0.45	1.59	2.22	7.90
	2 KB	16 B	92.6 %	0.59	1.33	2.91	6.60
	4 KB	16 B	93.0 %	0.56	1.25	2.75	6.22
	8 KB	16 B	93.4 %	0.53	1.19	2.61	5.92
	16 KB	16 B	93.5 %	0.52	1.17	2.56	5.80
	32 KB	16 B	93.6 %	0.51	1.15	2.52	5.71
	64 KB	16 B	93.8 %	0.50	1.12	2.46	5.58
128 KB	16 B	94.1 %	0.47	1.06	2.34	5.29	
Flush	32 B	4 B	54.5 %	0.91	5.46	4.32	27.05
	32 B	8 B	62.6 %	1.49	5.23	7.28	25.96
	32 B	16 B	66.0 %	2.72	6.12	13.43	30.44
	256 B	4 B	66.7 %	0.67	3.99	3.16	19.78
	256 B	8 B	76.1 %	0.95	3.34	4.65	16.58
	256 B	16 B	81.0 %	1.52	3.42	7.50	16.98
	1 KB	4 B	67.7 %	0.65	3.87	3.06	19.19
	1 KB	8 B	77.2 %	0.91	3.19	4.44	15.83
	1 KB	16 B	82.5 %	1.40	3.16	6.93	15.70
	2 KB	4 B	67.9 %	0.64	3.85	3.05	19.09
	2 KB	8 B	77.4 %	0.90	3.16	4.40	15.70
	2 KB	16 B	82.6 %	1.39	3.12	6.85	15.53
	4 KB	16 B	82.7 %	1.38	3.11	6.82	15.46
	8 KB	16 B	82.8 %	1.38	3.10	6.79	15.39
	16 KB	16 B	82.8 %	1.38	3.09	6.79	15.39
	32 KB	16 B	82.8 %	1.37	3.09	6.79	15.38
	64 KB	16 B	82.8 %	1.37	3.09	6.79	15.38
128 KB	16 B	82.8 %	1.37	3.09	6.79	15.38	

With the quantitative evaluation of the object cache we conclude that access to heap allocated object exhibits only minor spatial locality. The major contribution to cache hits comes from temporal locality. Therefore, the achievable hit rates for small object caches in embedded systems is in the range of 70% to 90%. 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 organization this is even true for a main memory based on SDRAM devices.

Acknowledgements: This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOP-ARD).

REFERENCES

- [1] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 1–12, New York, NY, USA, Mar. 2010. ACM.
- [2] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM Press.
- [3] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Parallelizing Calling Context Profiling in Virtual Machines on Multicores. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 111–120, New York, NY, USA, 2009. ACM.
- [4] W. Binder, M. Schoeberl, P. Moret, and A. Villazon. Cross-profiling for embedded Java processors. In *Proceedings of the 5th International Conference on the Quantitative Evaluation of Systems (QEST 2008)*, pages 287–296, St Malo, France, September 2008. IEEE Computer Society.
- [5] W. Binder, M. Schoeberl, P. Moret, and A. Villazon. Cross-profiling for Java processors. *Software: Practice and Experience*, 39(18):1439–1465, 2009.
- [6] W. Binder, A. Villazon, M. Schoeberl, and P. Moret. Cache-aware cross-profiling for Java processors. In *Proceedings of the 2008 international conference on Compilers, architecture, and synthesis for embedded systems (CASES 2008)*, pages 127–136, Atlanta, Georgia, October 2008. ACM.
- [7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann.

TABLE IX
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE HSQLDB BENCHMARK

Cache			Miss cycles per field read				
Type	Size	Line	Hit rate	Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	30.3 %	1.39	8.36	6.62	41.45
	32 B	8 B	27.3 %	2.91	10.18	14.17	50.51
	32 B	16 B	27.2 %	5.82	13.10	28.74	65.12
	256 B	4 B	58.3 %	0.83	5.01	3.96	24.82
	256 B	8 B	54.9 %	1.80	6.32	8.80	31.36
	256 B	16 B	62.1 %	3.03	6.82	14.97	33.92
	1 KB	4 B	71.3 %	0.57	3.44	2.72	17.05
	1 KB	8 B	71.8 %	1.13	3.95	5.51	19.63
	1 KB	16 B	77.8 %	1.78	4.00	8.79	19.91
	2 KB	4 B	75.9 %	0.48	2.89	2.29	14.33
	2 KB	8 B	77.3 %	0.91	3.18	4.43	15.78
	2 KB	16 B	82.7 %	1.38	3.11	6.82	15.45
	4 KB	16 B	86.6 %	1.08	2.42	5.31	12.03
	8 KB	16 B	89.6 %	0.84	1.88	4.12	9.34
	16 KB	16 B	92.1 %	0.63	1.42	3.11	7.05
	32 KB	16 B	93.9 %	0.48	1.09	2.39	5.42
64 KB	16 B	94.9 %	0.41	0.92	2.03	4.59	
128 KB	16 B	95.0 %	0.40	0.90	1.98	4.48	
Flush	32 B	4 B	27.9 %	1.44	8.65	6.85	42.90
	32 B	8 B	25.5 %	2.98	10.43	14.53	51.80
	32 B	16 B	25.5 %	5.96	13.41	29.42	66.66
	256 B	4 B	41.1 %	1.18	7.07	5.60	35.05
	256 B	8 B	44.4 %	2.22	7.79	10.85	38.66
	256 B	16 B	55.6 %	3.55	7.99	17.53	39.71
	1 KB	4 B	42.5 %	1.15	6.91	5.47	34.24
	1 KB	8 B	47.0 %	2.12	7.41	10.33	36.81
	1 KB	16 B	60.6 %	3.15	7.09	15.57	35.28
	2 KB	4 B	42.7 %	1.15	6.88	5.45	34.10
	2 KB	8 B	47.6 %	2.10	7.34	10.22	36.43
	2 KB	16 B	61.6 %	3.07	6.91	15.16	34.36
	4 KB	16 B	62.0 %	3.04	6.84	15.01	34.00
	8 KB	16 B	62.2 %	3.03	6.81	14.95	33.87
	16 KB	16 B	62.2 %	3.02	6.80	14.92	33.80
	32 KB	16 B	62.2 %	3.02	6.80	14.91	33.79
64 KB	16 B	62.3 %	3.02	6.79	14.91	33.78	
128 KB	16 B	62.3 %	3.02	6.79	14.91	33.78	

The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

- [8] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, July 20–24 2008, pages 155–165, New York, NY, USA, 07 2008. ACM.
- [9] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.
- [10] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, New York, NY, USA, 2004. ACM Press.
- [11] 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.
- [12] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- [13] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.
- [14] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM.
- [15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
- [16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [17] C. Kung and C. Ju-Bing. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007*, pages 23–28, Beijing, China, 2007. IEEE Computer Society.
- [18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
- [19] S. Metzloff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)*, pages 38–45, New York, NY, USA, 2008. ACM.
- [20] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction: 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, New York, NY, Apr. 2003. Springer-Verlag.
- [21] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

TABLE X
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE JYTHON BENCHMARK

				Miss cycles per field read			
Cache			Hit rate	Uniprocessor		8 core CMP	
Type	Size	Line		SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	57.2 %	0.86	5.14	4.07	25.48
	32 B	8 B	47.6 %	2.10	7.34	10.22	36.44
	32 B	16 B	45.1 %	4.39	9.88	21.67	49.11
	256 B	4 B	95.7 %	0.09	0.51	0.41	2.54
	256 B	8 B	94.1 %	0.24	0.83	1.15	4.10
	256 B	16 B	93.2 %	0.55	1.23	2.70	6.13
	1 KB	4 B	97.7 %	0.05	0.28	0.22	1.40
	1 KB	8 B	97.4 %	0.10	0.36	0.51	1.80
	1 KB	16 B	97.6 %	0.19	0.44	0.96	2.17
	2 KB	4 B	97.9 %	0.04	0.25	0.20	1.22
	2 KB	8 B	97.9 %	0.08	0.29	0.40	1.44
	2 KB	16 B	98.3 %	0.13	0.30	0.65	1.48
	4 KB	16 B	98.7 %	0.10	0.23	0.49	1.12
	8 KB	16 B	99.0 %	0.08	0.18	0.40	0.92
	16 KB	16 B	99.1 %	0.07	0.16	0.36	0.82
	32 KB	16 B	99.1 %	0.07	0.16	0.34	0.77
64 KB	16 B	99.2 %	0.07	0.15	0.33	0.74	
128 KB	16 B	99.2 %	0.06	0.15	0.32	0.73	
Flush	32 B	4 B	33.1 %	1.34	8.03	6.35	39.80
	32 B	8 B	35.8 %	2.57	8.99	12.53	44.65
	32 B	16 B	36.0 %	5.12	11.53	25.30	57.32
	256 B	4 B	38.5 %	1.23	7.38	5.84	36.60
	256 B	8 B	41.8 %	2.33	8.15	11.35	40.46
	256 B	16 B	51.8 %	3.85	8.67	19.03	43.12
	1 KB	4 B	38.7 %	1.23	7.35	5.82	36.47
	1 KB	8 B	42.1 %	2.32	8.10	11.29	40.23
	1 KB	16 B	52.7 %	3.78	8.51	18.68	42.33
	2 KB	4 B	38.7 %	1.23	7.35	5.82	36.46
	2 KB	8 B	42.1 %	2.31	8.10	11.28	40.21
	2 KB	16 B	52.8 %	3.77	8.49	18.63	42.21
	4 KB	16 B	52.9 %	3.77	8.48	18.60	42.15
	8 KB	16 B	52.9 %	3.77	8.47	18.59	42.12
	16 KB	16 B	53.0 %	3.76	8.47	18.58	42.11
	32 KB	16 B	53.0 %	3.76	8.47	18.58	42.10
64 KB	16 B	53.0 %	3.76	8.47	18.58	42.09	
128 KB	16 B	53.0 %	3.76	8.47	18.58	42.09	

- [22] C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted for publication, 2010.
- [23] T. B. Preusser, M. Zabel, and R. G. Spallek. Bump-pointer method caching for embedded java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.
- [24] W. 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.
- [25] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
- [26] M. 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.
- [27] M. Schoeberl, W. Binder, P. Moret, and A. Villazon. Design space exploration for Java processors with cross-profiling. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems (QEST 2009)*, pages 109–118, Budapest, Hungary, September 2009. IEEE Computer Society.
- [28] M. Schoeberl, W. Binder, and A. Villazon. Design space exploration of object caches with cross-profiling. In *submitted*, 2010.
- [29] M. Schoeberl, W. Puffitsch, and B. 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.
- [30] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.
- [31] The Standard Performance Evaluation Corporation. SPEC JBB2005 (Java Business Benchmark). Web pages at <http://www.spec.org/osg/jbb2005/>, 2005.
- [32] The Standard Performance Evaluation Corporation. SPECjvm2008 Benchmarks. Web pages at <http://www.spec.org/jvm2008/>, 2008.
- [33] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
- [34] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.
- [35] A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.
- [36] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

TABLE XI
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE LUINDEX BENCHMARK

Type	Cache		Hit rate	Miss cycles per field read			
	Size	Line		Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	77.2 %	0.46	2.74	2.17	13.57
	32 B	8 B	79.5 %	0.82	2.87	4.00	14.24
	32 B	16 B	78.4 %	1.73	3.89	8.53	19.33
	256 B	4 B	89.0 %	0.22	1.32	1.04	6.54
	256 B	8 B	90.8 %	0.37	1.29	1.80	6.40
	256 B	16 B	92.1 %	0.63	1.42	3.12	7.06
	1 KB	4 B	92.5 %	0.15	0.90	0.71	4.45
	1 KB	8 B	94.2 %	0.23	0.81	1.13	4.01
	1 KB	16 B	95.7 %	0.34	0.77	1.69	3.82
	2 KB	4 B	93.7 %	0.13	0.76	0.60	3.77
	2 KB	8 B	95.4 %	0.19	0.65	0.91	3.23
	2 KB	16 B	96.7 %	0.26	0.59	1.29	2.93
	4 KB	16 B	97.2 %	0.22	0.50	1.09	2.48
	8 KB	16 B	97.5 %	0.20	0.46	1.00	2.27
	16 KB	16 B	97.7 %	0.19	0.42	0.92	2.07
	32 KB	16 B	98.0 %	0.16	0.37	0.80	1.82
	64 KB	16 B	98.1 %	0.15	0.34	0.75	1.70
128 KB	16 B	98.2 %	0.14	0.32	0.70	1.59	
Flush	32 B	4 B	74.1 %	0.52	3.10	2.46	15.39
	32 B	8 B	76.9 %	0.93	3.24	4.51	16.09
	32 B	16 B	76.2 %	1.90	4.29	9.40	21.31
	256 B	4 B	84.1 %	0.32	1.90	1.51	9.44
	256 B	8 B	86.8 %	0.53	1.84	2.57	9.15
	256 B	16 B	88.8 %	0.90	2.02	4.43	10.03
	1 KB	4 B	86.9 %	0.26	1.57	1.24	7.77
	1 KB	8 B	89.6 %	0.42	1.46	2.04	7.25
	1 KB	16 B	91.6 %	0.67	1.50	3.30	7.48
	2 KB	4 B	87.9 %	0.24	1.45	1.15	7.21
	2 KB	8 B	90.5 %	0.38	1.33	1.85	6.60
	2 KB	16 B	92.5 %	0.60	1.35	2.97	6.73
	4 KB	16 B	92.9 %	0.57	1.28	2.81	6.38
	8 KB	16 B	93.0 %	0.56	1.25	2.75	6.23
	16 KB	16 B	93.2 %	0.54	1.22	2.69	6.09
	32 KB	16 B	93.4 %	0.53	1.19	2.60	5.90
	64 KB	16 B	93.5 %	0.52	1.18	2.58	5.85
128 KB	16 B	93.5 %	0.52	1.17	2.57	5.82	

TABLE XII
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE LUSEARCH BENCHMARK

Type	Cache		Hit rate	Miss cycles per field read			
	Size	Line		Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	64.7 %	0.71	4.24	3.36	21.02
	32 B	8 B	60.0 %	1.60	5.60	7.80	27.80
	32 B	16 B	37.7 %	4.98	11.21	24.59	55.72
	256 B	4 B	89.3 %	0.21	1.29	1.02	6.38
	256 B	8 B	89.8 %	0.41	1.43	2.00	7.12
	256 B	16 B	87.8 %	0.98	2.20	4.83	10.94
	1 KB	4 B	93.4 %	0.13	0.79	0.62	3.91
	1 KB	8 B	95.0 %	0.20	0.69	0.97	3.45
	1 KB	16 B	95.8 %	0.34	0.76	1.66	3.76
	2 KB	4 B	94.3 %	0.11	0.68	0.54	3.38
	2 KB	8 B	96.0 %	0.16	0.56	0.78	2.78
	2 KB	16 B	97.1 %	0.23	0.52	1.13	2.56
	4 KB	16 B	97.6 %	0.19	0.43	0.94	2.13
	8 KB	16 B	97.8 %	0.17	0.39	0.86	1.96
	16 KB	16 B	97.8 %	0.17	0.39	0.85	1.93
	32 KB	16 B	97.8 %	0.17	0.39	0.85	1.93
	64 KB	16 B	97.8 %	0.17	0.39	0.85	1.93
128 KB	16 B	97.8 %	0.17	0.39	0.85	1.93	
Flush	32 B	4 B	58.3 %	0.83	5.00	3.96	24.80
	32 B	8 B	55.2 %	1.79	6.27	8.73	31.11
	32 B	16 B	35.2 %	5.19	11.67	25.61	58.03
	256 B	4 B	77.5 %	0.45	2.70	2.13	13.37
	256 B	8 B	80.1 %	0.80	2.79	3.88	13.83
	256 B	16 B	79.6 %	1.63	3.67	8.06	18.27
	1 KB	4 B	80.0 %	0.40	2.40	1.90	11.89
	1 KB	8 B	83.7 %	0.65	2.28	3.17	11.31
	1 KB	16 B	85.8 %	1.14	2.56	5.61	12.71
	2 KB	4 B	80.4 %	0.39	2.35	1.86	11.67
	2 KB	8 B	84.2 %	0.63	2.21	3.08	10.97
	2 KB	16 B	86.7 %	1.07	2.40	5.26	11.92
	4 KB	16 B	86.9 %	1.05	2.35	5.16	11.70
	8 KB	16 B	87.0 %	1.04	2.34	5.14	11.65
	16 KB	16 B	87.0 %	1.04	2.34	5.14	11.64
	32 KB	16 B	87.0 %	1.04	2.34	5.13	11.63
	64 KB	16 B	87.0 %	1.04	2.34	5.13	11.63
128 KB	16 B	87.0 %	1.04	2.34	5.13	11.63	

TABLE XIII
DIRECT MAPPED CACHE HIT RATE AND MISS PENALTY FOR THE XALAN BENCHMARK

Cache Type	Size	Line	Hit rate	Miss cycles per field read			
				Uniprocessor		8 core CMP	
				SRAM	SDRAM	SRAM	SDRAM
CC	32 B	4 B	40.3 %	1.19	7.16	5.67	35.52
	32 B	8 B	37.6 %	2.50	8.74	12.17	43.37
	32 B	16 B	28.0 %	5.76	12.95	28.42	64.40
	256 B	4 B	69.9 %	0.60	3.61	2.86	17.90
	256 B	8 B	71.9 %	1.12	3.94	5.48	19.55
	256 B	16 B	70.5 %	2.36	5.31	11.65	26.39
	1 KB	4 B	82.3 %	0.35	2.13	1.69	10.56
	1 KB	8 B	84.7 %	0.61	2.15	2.99	10.65
	1 KB	16 B	85.6 %	1.15	2.59	5.68	12.86
	2 KB	4 B	86.2 %	0.28	1.66	1.31	8.22
	2 KB	8 B	88.6 %	0.45	1.59	2.22	7.90
	2 KB	16 B	89.8 %	0.81	1.83	4.02	9.11
	4 KB	16 B	92.7 %	0.59	1.32	2.90	6.56
	8 KB	16 B	94.7 %	0.43	0.96	2.11	4.77
	16 KB	16 B	96.1 %	0.31	0.71	1.55	3.52
	32 KB	16 B	97.0 %	0.24	0.54	1.19	2.69
	64 KB	16 B	97.6 %	0.19	0.44	0.96	2.17
128 KB	16 B	97.9 %	0.17	0.38	0.83	1.89	
Flush	32 B	4 B	34.8 %	1.30	7.82	6.19	38.80
	32 B	8 B	33.0 %	2.68	9.38	13.06	46.54
	32 B	16 B	26.1 %	5.91	13.30	29.19	66.14
	256 B	4 B	51.2 %	0.98	5.86	4.64	29.06
	256 B	8 B	57.3 %	1.71	5.98	8.33	29.70
	256 B	16 B	59.9 %	3.21	7.22	15.84	35.88
	1 KB	4 B	53.8 %	0.92	5.54	4.39	27.49
	1 KB	8 B	61.2 %	1.55	5.43	7.56	26.94
	1 KB	16 B	66.3 %	2.69	6.06	13.30	30.14
	2 KB	4 B	54.2 %	0.92	5.50	4.35	27.27
	2 KB	8 B	61.8 %	1.53	5.34	7.44	26.53
	2 KB	16 B	67.3 %	2.62	5.89	12.93	29.30
	4 KB	16 B	67.8 %	2.57	5.79	12.71	28.80
	8 KB	16 B	68.0 %	2.56	5.75	12.62	28.60
	16 KB	16 B	68.2 %	2.54	5.72	12.55	28.44
	32 KB	16 B	68.3 %	2.54	5.71	12.53	28.39
	64 KB	16 B	68.3 %	2.54	5.70	12.52	28.36
128 KB	16 B	68.3 %	2.53	5.70	12.51	28.35	

TABLE XIV
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 %
16 KB	256 B	64 way	96.3 %	82.3 %	89.7 %	81.9 %	85.4 %	98.0 %	93.2 %	94.0 %	85.6 %
32 KB	256 B	128 way	96.4 %	84.3 %	90.4 %	82.3 %	87.7 %	98.1 %	94.0 %	94.5 %	89.2 %
64 KB	256 B	256 way	96.4 %	85.4 %	90.7 %	82.7 %	88.4 %	98.2 %	94.4 %	94.8 %	91.1 %
128 KB	256 B	512 way	96.5 %	86.1 %	90.9 %	83.0 %	88.8 %	98.2 %	94.7 %	94.9 %	92.8 %

TABLE XV
OBJECT CACHE CACHE HIT RATE WITH FLUSH ON SYNCHRONIZED BLOCKS

Cache			Benchmark								
Size	Line	Assoc.	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	xalan
64 B	64 B	1 way	40.7 %	25.3 %	41.9 %	42.3 %	10.0 %	28.1 %	65.7 %	18.5 %	5.6 %
128 B	128 B	1 way	40.7 %	25.3 %	41.9 %	43.3 %	10.1 %	28.2 %	65.7 %	18.3 %	4.4 %
256 B	256 B	1 way	40.7 %	25.3 %	41.9 %	44.6 %	10.2 %	28.2 %	65.7 %	18.3 %	4.3 %
128 B	64 B	2 way	54.6 %	42.1 %	52.8 %	57.4 %	18.7 %	33.5 %	76.0 %	68.6 %	21.9 %
256 B	128 B	2 way	54.6 %	42.1 %	52.7 %	58.9 %	18.9 %	33.7 %	75.8 %	69.4 %	19.3 %
512 B	256 B	2 way	54.6 %	42.1 %	52.8 %	62.3 %	18.9 %	33.7 %	75.8 %	69.4 %	19.2 %
256 B	64 B	4 way	78.4 %	50.2 %	56.0 %	59.4 %	24.3 %	38.7 %	79.6 %	78.9 %	43.2 %
512 B	128 B	4 way	78.4 %	50.2 %	55.4 %	61.9 %	24.6 %	38.9 %	79.9 %	79.7 %	44.6 %
1 KB	256 B	4 way	78.4 %	50.2 %	55.4 %	65.5 %	24.6 %	38.9 %	79.9 %	79.7 %	44.6 %
512 B	64 B	8 way	83.0 %	59.0 %	62.6 %	60.2 %	37.0 %	38.9 %	82.4 %	81.1 %	49.7 %
1 KB	128 B	8 way	83.0 %	59.0 %	62.8 %	63.0 %	37.5 %	39.1 %	82.9 %	82.0 %	52.5 %
2 KB	256 B	8 way	83.0 %	59.0 %	62.9 %	67.1 %	37.5 %	39.1 %	82.9 %	82.0 %	52.6 %
4 KB	256 B	16 way	83.0 %	61.8 %	65.4 %	67.8 %	41.4 %	39.2 %	85.7 %	82.6 %	54.4 %
8 KB	256 B	32 way	83.0 %	63.5 %	71.4 %	68.1 %	43.0 %	39.2 %	87.1 %	82.8 %	54.8 %
16 KB	256 B	64 way	83.0 %	64.5 %	73.6 %	68.2 %	44.3 %	39.2 %	89.1 %	82.8 %	54.8 %
32 KB	256 B	128 way	83.0 %	65.3 %	74.1 %	68.2 %	44.3 %	39.2 %	89.7 %	82.8 %	54.8 %
64 KB	256 B	256 way	83.0 %	65.7 %	74.2 %	68.2 %	44.3 %	39.2 %	90.0 %	82.8 %	54.8 %
128 KB	256 B	512 way	83.0 %	66.0 %	74.3 %	68.2 %	44.3 %	39.2 %	90.2 %	82.8 %	54.8 %

TABLE XVI
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE ANTLR BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					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
	512 B	64 B	8 way	94.7 %	0.11	0.63	0.50	3.13
	1 KB	128 B	8 way	94.8 %	0.11	0.63	0.50	3.13
	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
	16 KB	256 B	64 way	96.3 %	0.07	0.45	0.36	2.22
	32 KB	256 B	128 way	96.4 %	0.07	0.44	0.35	2.17
64 KB	256 B	256 way	96.4 %	0.07	0.43	0.34	2.12	
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
	512 B	64 B	8 way	98.4 %	0.49	1.12	2.50	3.28
	1 KB	128 B	8 way	98.5 %	0.98	2.22	4.92	5.69
	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
	16 KB	256 B	64 way	99.1 %	1.20	2.70	5.99	6.46
	32 KB	256 B	128 way	99.1 %	1.15	2.59	5.74	6.19
64 KB	256 B	256 way	99.1 %	1.10	2.48	5.50	5.93	
Flush Single	64 B	64 B	1 way	40.7 %	1.19	7.12	5.64	35.32
	128 B	128 B	1 way	40.7 %	1.19	7.12	5.64	35.32
	256 B	256 B	1 way	40.7 %	1.19	7.12	5.64	35.32
	128 B	64 B	2 way	54.6 %	0.91	5.44	4.31	27.00
	256 B	128 B	2 way	54.6 %	0.91	5.45	4.31	27.00
	512 B	256 B	2 way	54.6 %	0.91	5.45	4.31	27.00
	256 B	64 B	4 way	78.4 %	0.43	2.59	2.05	12.86
	512 B	128 B	4 way	78.4 %	0.43	2.59	2.05	12.86
	1 KB	256 B	4 way	78.4 %	0.43	2.59	2.05	12.86
	512 B	64 B	8 way	83.0 %	0.34	2.04	1.62	10.13
	1 KB	128 B	8 way	83.0 %	0.34	2.04	1.62	10.12
	2 KB	256 B	8 way	83.0 %	0.34	2.04	1.62	10.12
	4 KB	256 B	16 way	83.0 %	0.34	2.04	1.62	10.12
	8 KB	256 B	32 way	83.0 %	0.34	2.04	1.62	10.12
	16 KB	256 B	64 way	83.0 %	0.34	2.04	1.62	10.12
	32 KB	256 B	128 way	83.0 %	0.34	2.04	1.62	10.12
64 KB	256 B	256 way	83.0 %	0.34	2.04	1.62	10.12	
Flush Line	64 B	64 B	1 way	66.6 %	10.69	24.06	53.32	70.04
	128 B	128 B	1 way	66.6 %	21.40	48.14	106.82	123.54
	256 B	256 B	1 way	66.6 %	42.79	96.29	213.81	230.52
	128 B	64 B	2 way	81.6 %	5.86	13.21	29.29	38.47
	256 B	128 B	2 way	81.6 %	11.75	26.45	58.68	67.87
	512 B	256 B	2 way	81.6 %	23.51	52.89	117.45	126.63
	256 B	64 B	4 way	91.8 %	2.62	5.90	13.10	17.21
	512 B	128 B	4 way	91.8 %	5.25	11.81	26.20	30.30
	1 KB	256 B	4 way	91.8 %	10.50	23.62	52.44	56.54
	512 B	64 B	8 way	93.9 %	1.92	4.35	9.65	12.68
	1 KB	128 B	8 way	94.0 %	3.86	8.68	19.26	22.28
	2 KB	256 B	8 way	94.0 %	7.71	17.36	38.54	41.56
	4 KB	256 B	16 way	94.0 %	7.71	17.35	38.52	41.54
	8 KB	256 B	32 way	94.0 %	7.71	17.35	38.52	41.53
	16 KB	256 B	64 way	94.0 %	7.71	17.35	38.52	41.53
	32 KB	256 B	128 way	94.0 %	7.71	17.35	38.52	41.53
64 KB	256 B	256 way	94.0 %	7.71	17.35	38.52	41.53	

TABLE XVII
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE BLOAT BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	31.0 %	1.38	8.28	6.56	41.07
	128 B	128 B	1 way	30.9 %	1.38	8.29	6.56	41.10
	256 B	256 B	1 way	30.9 %	1.38	8.29	6.56	41.10
	128 B	64 B	2 way	55.7 %	0.89	5.32	4.21	26.37
	256 B	128 B	2 way	55.7 %	0.89	5.32	4.21	26.37
	512 B	256 B	2 way	55.7 %	0.89	5.32	4.21	26.37
	256 B	64 B	4 way	64.6 %	0.71	4.25	3.37	21.08
	512 B	128 B	4 way	64.6 %	0.71	4.25	3.37	21.08
	1 KB	256 B	4 way	64.6 %	0.71	4.25	3.37	21.08
	512 B	64 B	8 way	73.9 %	0.52	3.13	2.48	15.51
	1 KB	128 B	8 way	73.9 %	0.52	3.13	2.48	15.51
	2 KB	256 B	8 way	73.9 %	0.52	3.13	2.48	15.51
	4 KB	256 B	16 way	77.3 %	0.45	2.72	2.16	13.51
	8 KB	256 B	32 way	79.7 %	0.41	2.43	1.93	12.07
	16 KB	256 B	64 way	82.3 %	0.36	2.13	1.69	10.56
	32 KB	256 B	128 way	84.3 %	0.31	1.88	1.49	9.35
64 KB	256 B	256 way	85.4 %	0.29	1.75	1.39	8.69	
CC Line	64 B	64 B	1 way	50.6 %	15.78	35.55	78.81	103.51
	128 B	128 B	1 way	50.6 %	31.65	71.20	157.98	182.70
	256 B	256 B	1 way	50.6 %	63.29	142.40	316.21	340.93
	128 B	64 B	2 way	69.2 %	9.81	22.12	49.05	64.43
	256 B	128 B	2 way	69.3 %	19.62	44.15	97.97	113.30
	512 B	256 B	2 way	69.3 %	39.25	88.31	196.09	211.42
	256 B	64 B	4 way	80.5 %	6.21	14.02	31.11	40.87
	512 B	128 B	4 way	80.6 %	12.43	27.96	62.03	71.74
	1 KB	256 B	4 way	80.6 %	24.85	55.91	124.15	133.86
	512 B	64 B	8 way	86.9 %	4.16	9.39	20.87	27.41
	1 KB	128 B	8 way	87.0 %	8.31	18.71	41.50	48.00
	2 KB	256 B	8 way	87.0 %	16.63	37.41	83.07	89.56
	4 KB	256 B	16 way	89.0 %	14.08	31.69	70.37	75.87
	8 KB	256 B	32 way	90.3 %	12.38	27.85	61.85	66.69
	16 KB	256 B	64 way	91.9 %	10.38	23.35	51.86	55.91
	32 KB	256 B	128 way	93.1 %	8.80	19.79	43.95	47.39
64 KB	256 B	256 way	93.8 %	7.99	17.97	39.90	43.02	
Flush Single	64 B	64 B	1 way	25.3 %	1.49	8.96	7.09	44.44
	128 B	128 B	1 way	25.3 %	1.49	8.97	7.10	44.46
	256 B	256 B	1 way	25.3 %	1.49	8.97	7.10	44.46
	128 B	64 B	2 way	42.1 %	1.16	6.95	5.50	34.47
	256 B	128 B	2 way	42.1 %	1.16	6.95	5.50	34.47
	512 B	256 B	2 way	42.1 %	1.16	6.95	5.50	34.47
	256 B	64 B	4 way	50.2 %	1.00	5.97	4.73	29.62
	512 B	128 B	4 way	50.2 %	1.00	5.97	4.73	29.62
	1 KB	256 B	4 way	50.2 %	1.00	5.97	4.73	29.62
	512 B	64 B	8 way	59.0 %	0.82	4.92	3.89	24.38
	1 KB	128 B	8 way	59.0 %	0.82	4.92	3.89	24.38
	2 KB	256 B	8 way	59.0 %	0.82	4.92	3.89	24.38
	4 KB	256 B	16 way	61.8 %	0.76	4.59	3.63	22.75
	8 KB	256 B	32 way	63.5 %	0.73	4.38	3.47	21.72
	16 KB	256 B	64 way	64.5 %	0.71	4.26	3.37	21.13
	32 KB	256 B	128 way	65.3 %	0.69	4.17	3.30	20.66
64 KB	256 B	256 way	65.7 %	0.69	4.11	3.26	20.39	
Flush Line	64 B	64 B	1 way	46.8 %	16.98	38.24	84.78	111.36
	128 B	128 B	1 way	46.8 %	34.04	76.60	169.96	196.55
	256 B	256 B	1 way	46.8 %	68.09	153.20	340.18	366.78
	128 B	64 B	2 way	64.9 %	11.22	25.28	56.06	73.63
	256 B	128 B	2 way	64.9 %	22.43	50.48	112.00	129.52
	512 B	256 B	2 way	64.9 %	44.87	100.95	224.16	241.69
	256 B	64 B	4 way	72.3 %	8.82	19.88	44.12	57.95
	512 B	128 B	4 way	72.4 %	17.64	39.69	88.07	101.85
	1 KB	256 B	4 way	72.4 %	35.28	79.39	176.27	190.06
	512 B	64 B	8 way	76.3 %	7.56	17.05	37.83	49.69
	1 KB	128 B	8 way	76.4 %	15.12	34.02	75.47	87.29
	2 KB	256 B	8 way	76.4 %	30.24	68.03	151.07	162.88
	4 KB	256 B	16 way	77.7 %	28.52	64.17	142.48	153.62
	8 KB	256 B	32 way	78.6 %	27.41	61.66	136.92	147.63
	16 KB	256 B	64 way	79.1 %	26.78	60.26	133.81	144.27
	32 KB	256 B	128 way	79.5 %	26.24	59.03	131.08	141.33
64 KB	256 B	256 way	79.7 %	25.95	58.38	129.64	139.77	

TABLE XVIII
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE CHART BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	43.9 %	1.24	7.43	5.88	36.86
	128 B	128 B	1 way	43.9 %	1.24	7.43	5.88	36.85
	256 B	256 B	1 way	43.9 %	1.24	7.43	5.88	36.85
	128 B	64 B	2 way	59.5 %	0.92	5.53	4.38	27.43
	256 B	128 B	2 way	59.4 %	0.92	5.54	4.39	27.48
	512 B	256 B	2 way	59.5 %	0.92	5.53	4.38	27.42
	256 B	64 B	4 way	67.6 %	0.74	4.47	3.54	22.15
	512 B	128 B	4 way	66.9 %	0.77	4.63	3.67	22.98
	1 KB	256 B	4 way	67.0 %	0.77	4.62	3.66	22.92
	512 B	64 B	8 way	76.2 %	0.55	3.32	2.63	16.47
	1 KB	128 B	8 way	76.5 %	0.55	3.29	2.60	16.29
	2 KB	256 B	8 way	76.6 %	0.55	3.28	2.59	16.24
	4 KB	256 B	16 way	80.0 %	0.47	2.80	2.21	13.86
	8 KB	256 B	32 way	87.0 %	0.30	1.80	1.43	8.93
	16 KB	256 B	64 way	89.7 %	0.24	1.43	1.13	7.07
	32 KB	256 B	128 way	90.4 %	0.22	1.34	1.06	6.62
64 KB	256 B	256 way	90.7 %	0.22	1.29	1.02	6.40	
CC Line	64 B	64 B	1 way	70.4 %	8.59	20.44	47.26	62.08
	128 B	128 B	1 way	71.0 %	17.94	41.13	92.62	107.11
	256 B	256 B	1 way	71.1 %	36.65	82.87	184.76	199.20
	128 B	64 B	2 way	77.6 %	6.27	15.21	35.68	46.86
	256 B	128 B	2 way	78.1 %	13.39	30.90	69.90	80.84
	512 B	256 B	2 way	78.7 %	26.98	61.12	136.46	147.13
	256 B	64 B	4 way	80.5 %	5.36	13.17	31.14	40.90
	512 B	128 B	4 way	81.7 %	11.12	25.78	58.56	67.72
	1 KB	256 B	4 way	82.3 %	22.32	50.64	113.19	122.04
	512 B	64 B	8 way	86.0 %	3.59	9.19	22.34	29.34
	1 KB	128 B	8 way	87.6 %	7.35	17.31	39.75	45.97
	2 KB	256 B	8 way	88.3 %	14.67	33.43	74.98	80.84
	4 KB	256 B	16 way	90.1 %	12.34	28.18	63.31	68.26
	8 KB	256 B	32 way	92.9 %	8.70	19.98	45.11	48.64
	16 KB	256 B	64 way	94.0 %	7.36	16.99	38.46	41.47
	32 KB	256 B	128 way	94.3 %	7.01	16.20	36.71	39.58
64 KB	256 B	256 way	94.4 %	6.81	15.74	35.68	38.47	
Flush Single	64 B	64 B	1 way	41.9 %	1.28	7.67	6.08	38.05
	128 B	128 B	1 way	41.9 %	1.28	7.67	6.08	38.05
	256 B	256 B	1 way	41.9 %	1.28	7.67	6.08	38.05
	128 B	64 B	2 way	52.8 %	1.06	6.35	5.02	31.47
	256 B	128 B	2 way	52.7 %	1.06	6.35	5.03	31.51
	512 B	256 B	2 way	52.8 %	1.06	6.34	5.02	31.46
	256 B	64 B	4 way	56.0 %	0.98	5.87	4.65	29.10
	512 B	128 B	4 way	55.4 %	1.00	6.02	4.77	29.87
	1 KB	256 B	4 way	55.4 %	1.00	6.01	4.76	29.82
	512 B	64 B	8 way	62.6 %	0.83	4.98	3.94	24.71
	1 KB	128 B	8 way	62.8 %	0.82	4.95	3.92	24.54
	2 KB	256 B	8 way	62.9 %	0.82	4.94	3.91	24.49
	4 KB	256 B	16 way	65.4 %	0.76	4.57	3.62	22.67
	8 KB	256 B	32 way	71.4 %	0.62	3.71	2.94	18.40
	16 KB	256 B	64 way	73.6 %	0.57	3.39	2.69	16.83
	32 KB	256 B	128 way	74.1 %	0.56	3.33	2.64	16.51
64 KB	256 B	256 way	74.2 %	0.55	3.30	2.62	16.38	
Flush Line	64 B	64 B	1 way	66.0 %	9.98	23.58	54.21	71.20
	128 B	128 B	1 way	66.6 %	20.75	47.45	106.64	123.32
	256 B	256 B	1 way	66.7 %	42.27	95.52	212.84	229.48
	128 B	64 B	2 way	72.5 %	7.90	18.89	43.83	57.57
	256 B	128 B	2 way	73.0 %	16.69	38.30	86.34	99.85
	512 B	256 B	2 way	73.5 %	33.62	76.06	169.63	182.89
	256 B	64 B	4 way	74.8 %	7.16	17.23	40.15	52.73
	512 B	128 B	4 way	75.9 %	14.79	34.03	76.86	88.89
	1 KB	256 B	4 way	76.5 %	29.71	67.27	150.10	161.84
	512 B	64 B	8 way	79.7 %	5.59	13.70	32.32	42.45
	1 KB	128 B	8 way	81.2 %	11.41	26.44	60.02	69.41
	2 KB	256 B	8 way	81.9 %	22.86	51.86	115.90	124.96
	4 KB	256 B	16 way	83.4 %	20.89	47.42	106.02	114.31
	8 KB	256 B	32 way	85.9 %	17.69	40.21	90.02	97.06
	16 KB	256 B	64 way	86.8 %	16.54	37.63	84.29	90.88
	32 KB	256 B	128 way	87.0 %	16.30	37.08	83.08	89.57
64 KB	256 B	256 way	87.1 %	16.20	36.86	82.58	89.04	

TABLE XIX
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE FOP BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	48.2 %	1.05	6.30	4.99	31.25
	128 B	128 B	1 way	49.2 %	1.03	6.19	4.90	30.67
	256 B	256 B	1 way	50.5 %	1.00	6.03	4.77	29.90
	128 B	64 B	2 way	63.7 %	0.74	4.44	3.52	22.02
	256 B	128 B	2 way	65.3 %	0.71	4.25	3.37	21.09
	512 B	256 B	2 way	68.7 %	0.64	3.84	3.04	19.06
	256 B	64 B	4 way	68.9 %	0.64	3.82	3.02	18.93
	512 B	128 B	4 way	71.5 %	0.58	3.50	2.77	17.38
	1 KB	256 B	4 way	75.2 %	0.51	3.07	2.43	15.21
	512 B	64 B	8 way	71.2 %	0.59	3.54	2.80	17.55
	1 KB	128 B	8 way	74.1 %	0.53	3.20	2.53	15.84
	2 KB	256 B	8 way	78.1 %	0.45	2.71	2.15	13.44
	4 KB	256 B	16 way	79.8 %	0.42	2.51	1.98	12.43
	8 KB	256 B	32 way	81.2 %	0.39	2.32	1.84	11.51
	16 KB	256 B	64 way	81.9 %	0.37	2.23	1.77	11.08
	32 KB	256 B	128 way	82.3 %	0.36	2.18	1.72	10.80
64 KB	256 B	256 way	82.7 %	0.35	2.13	1.68	10.55	
CC Line	64 B	64 B	1 way	72.5 %	6.05	17.04	43.83	57.57
	128 B	128 B	1 way	75.5 %	12.83	32.45	78.36	90.62
	256 B	256 B	1 way	79.8 %	25.83	58.12	129.06	139.15
	128 B	64 B	2 way	80.8 %	3.40	11.08	30.63	40.24
	256 B	128 B	2 way	84.5 %	7.09	19.52	49.66	57.43
	512 B	256 B	2 way	89.0 %	14.13	31.78	70.57	76.09
	256 B	64 B	4 way	83.4 %	2.56	9.20	26.46	34.75
	512 B	128 B	4 way	87.7 %	5.04	14.91	39.44	45.62
	1 KB	256 B	4 way	92.3 %	9.90	22.27	49.45	53.32
	512 B	64 B	8 way	84.3 %	2.28	8.57	25.06	32.91
	1 KB	128 B	8 way	88.6 %	4.43	13.55	36.42	42.12
	2 KB	256 B	8 way	93.2 %	8.68	19.54	43.39	46.78
	4 KB	256 B	16 way	93.9 %	7.86	17.68	39.25	42.32
	8 KB	256 B	32 way	94.4 %	7.12	16.02	35.56	38.34
	16 KB	256 B	64 way	94.7 %	6.77	15.24	33.84	36.48
	32 KB	256 B	128 way	94.9 %	6.53	14.70	32.63	35.18
64 KB	256 B	256 way	95.1 %	6.31	14.19	31.51	33.97	
Flush Single	64 B	64 B	1 way	42.3 %	1.17	7.02	5.56	34.80
	128 B	128 B	1 way	43.3 %	1.15	6.90	5.46	34.22
	256 B	256 B	1 way	44.6 %	1.12	6.75	5.34	33.45
	128 B	64 B	2 way	57.4 %	0.87	5.21	4.12	25.83
	256 B	128 B	2 way	58.9 %	0.84	5.02	3.98	24.90
	512 B	256 B	2 way	62.3 %	0.77	4.61	3.65	22.88
	256 B	64 B	4 way	59.4 %	0.83	4.97	3.94	24.65
	512 B	128 B	4 way	61.9 %	0.78	4.67	3.70	23.15
	1 KB	256 B	4 way	65.5 %	0.71	4.23	3.35	20.99
	512 B	64 B	8 way	60.2 %	0.81	4.87	3.85	24.14
	1 KB	128 B	8 way	63.0 %	0.76	4.53	3.59	22.47
	2 KB	256 B	8 way	67.1 %	0.67	4.05	3.20	20.07
	4 KB	256 B	16 way	67.8 %	0.66	3.96	3.13	19.63
	8 KB	256 B	32 way	68.1 %	0.65	3.93	3.11	19.48
	16 KB	256 B	64 way	68.2 %	0.65	3.91	3.10	19.40
	32 KB	256 B	128 way	68.2 %	0.65	3.91	3.09	19.37
64 KB	256 B	256 way	68.2 %	0.65	3.91	3.09	19.37	
Flush Line	64 B	64 B	1 way	69.4 %	7.06	19.32	48.88	64.20
	128 B	128 B	1 way	72.2 %	14.90	37.11	88.68	102.56
	256 B	256 B	1 way	76.6 %	29.97	67.42	149.72	161.42
	128 B	64 B	2 way	77.2 %	4.54	13.65	36.33	47.72
	256 B	128 B	2 way	80.8 %	9.43	24.80	61.37	70.97
	512 B	256 B	2 way	85.3 %	18.81	42.33	94.00	101.34
	256 B	64 B	4 way	78.2 %	4.23	12.95	34.78	45.68
	512 B	128 B	4 way	82.3 %	8.45	22.60	56.49	65.33
	1 KB	256 B	4 way	86.9 %	16.73	37.63	83.57	90.10
	512 B	64 B	8 way	78.5 %	4.12	12.70	34.22	44.94
	1 KB	128 B	8 way	82.7 %	8.18	21.99	55.14	63.77
	2 KB	256 B	8 way	87.4 %	16.18	36.41	80.86	87.18
	4 KB	256 B	16 way	87.5 %	15.97	35.94	79.81	86.05
	8 KB	256 B	32 way	87.6 %	15.89	35.75	79.38	85.58
	16 KB	256 B	64 way	87.6 %	15.85	35.66	79.19	85.38
	32 KB	256 B	128 way	87.6 %	15.84	35.63	79.13	85.31
64 KB	256 B	256 way	87.6 %	15.83	35.63	79.11	85.30	

TABLE XX
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE HSQLDB BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	10.5 %	1.80	10.79	8.54	53.48
	128 B	128 B	1 way	10.7 %	1.79	10.77	8.53	53.40
	256 B	256 B	1 way	10.7 %	1.79	10.77	8.52	53.38
	128 B	64 B	2 way	19.8 %	1.61	9.67	7.66	47.96
	256 B	128 B	2 way	20.0 %	1.61	9.65	7.64	47.84
	512 B	256 B	2 way	20.0 %	1.61	9.65	7.64	47.84
	256 B	64 B	4 way	26.0 %	1.49	8.93	7.07	44.27
	512 B	128 B	4 way	26.3 %	1.48	8.89	7.04	44.07
	1 KB	256 B	4 way	26.4 %	1.48	8.88	7.03	44.03
	512 B	64 B	8 way	39.4 %	1.22	7.32	5.80	36.31
	1 KB	128 B	8 way	39.9 %	1.21	7.27	5.75	36.03
	2 KB	256 B	8 way	39.9 %	1.21	7.26	5.74	35.98
	4 KB	256 B	16 way	44.4 %	1.12	6.72	5.32	33.34
	8 KB	256 B	32 way	49.1 %	1.03	6.16	4.87	30.53
	16 KB	256 B	64 way	85.4 %	0.30	1.79	1.42	8.89
	32 KB	256 B	128 way	87.7 %	0.25	1.51	1.20	7.51
64 KB	256 B	256 way	88.4 %	0.24	1.43	1.13	7.09	
CC Line	64 B	64 B	1 way	23.3 %	24.12	54.82	122.41	160.78
	128 B	128 B	1 way	23.9 %	48.52	109.40	243.14	281.19
	256 B	256 B	1 way	24.0 %	97.28	218.87	486.00	524.00
	128 B	64 B	2 way	34.2 %	20.62	46.94	104.97	137.88
	256 B	128 B	2 way	35.1 %	41.33	93.22	207.25	239.68
	512 B	256 B	2 way	35.3 %	82.87	186.46	414.02	446.39
	256 B	64 B	4 way	39.0 %	19.08	43.48	97.30	127.80
	512 B	128 B	4 way	40.1 %	38.15	86.08	191.40	221.35
	1 KB	256 B	4 way	40.3 %	76.38	171.86	381.62	411.46
	512 B	64 B	8 way	60.3 %	12.25	28.11	63.24	83.07
	1 KB	128 B	8 way	61.5 %	24.46	55.26	123.03	142.28
	2 KB	256 B	8 way	61.8 %	48.94	110.12	244.51	263.63
	4 KB	256 B	16 way	66.5 %	42.88	96.48	214.24	230.99
	8 KB	256 B	32 way	70.1 %	38.26	86.08	191.15	206.09
	16 KB	256 B	64 way	94.4 %	7.19	16.19	35.94	38.75
	32 KB	256 B	128 way	95.6 %	5.61	12.62	28.01	30.20
64 KB	256 B	256 way	96.0 %	5.17	11.63	25.82	27.84	
Flush Single	64 B	64 B	1 way	10.0 %	1.81	10.85	8.59	53.82
	128 B	128 B	1 way	10.1 %	1.81	10.84	8.58	53.73
	256 B	256 B	1 way	10.2 %	1.81	10.83	8.58	53.71
	128 B	64 B	2 way	18.7 %	1.63	9.81	7.76	48.63
	256 B	128 B	2 way	18.9 %	1.63	9.78	7.74	48.51
	512 B	256 B	2 way	18.9 %	1.63	9.78	7.74	48.50
	256 B	64 B	4 way	24.3 %	1.52	9.14	7.23	45.31
	512 B	128 B	4 way	24.6 %	1.52	9.10	7.20	45.12
	1 KB	256 B	4 way	24.6 %	1.52	9.10	7.20	45.11
	512 B	64 B	8 way	37.0 %	1.27	7.61	6.02	37.72
	1 KB	128 B	8 way	37.5 %	1.26	7.55	5.98	37.45
	2 KB	256 B	8 way	37.5 %	1.26	7.55	5.98	37.43
	4 KB	256 B	16 way	41.4 %	1.18	7.09	5.61	35.14
	8 KB	256 B	32 way	43.0 %	1.15	6.89	5.45	34.14
	16 KB	256 B	64 way	44.3 %	1.12	6.73	5.33	33.37
	32 KB	256 B	128 way	44.3 %	1.12	6.73	5.33	33.37
64 KB	256 B	256 way	44.3 %	1.12	6.73	5.33	33.37	
Flush Line	64 B	64 B	1 way	22.0 %	24.50	55.69	124.34	163.32
	128 B	128 B	1 way	22.6 %	49.32	111.20	247.15	285.82
	256 B	256 B	1 way	22.7 %	98.88	222.49	494.03	532.66
	128 B	64 B	2 way	32.5 %	21.15	48.15	107.64	141.39
	256 B	128 B	2 way	33.4 %	42.45	95.75	212.86	246.17
	512 B	256 B	2 way	33.5 %	85.12	191.51	425.25	458.50
	256 B	64 B	4 way	36.9 %	19.74	44.97	100.60	132.14
	512 B	128 B	4 way	37.9 %	39.55	89.21	198.35	229.40
	1 KB	256 B	4 way	38.1 %	79.24	178.30	395.90	426.86
	512 B	64 B	8 way	57.9 %	13.02	29.86	67.12	88.16
	1 KB	128 B	8 way	59.0 %	26.07	58.89	131.08	151.60
	2 KB	256 B	8 way	59.2 %	52.24	117.54	261.00	281.40
	4 KB	256 B	16 way	63.6 %	46.57	104.79	232.69	250.88
	8 KB	256 B	32 way	65.4 %	44.29	99.66	221.28	238.59
	16 KB	256 B	64 way	66.7 %	42.67	96.00	213.17	229.84
	32 KB	256 B	128 way	66.7 %	42.67	96.00	213.17	229.83
64 KB	256 B	256 way	66.7 %	42.67	96.00	213.17	229.83	

TABLE XXI
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE JYTHON BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	28.7 %	1.46	8.78	6.95	43.53
	128 B	128 B	1 way	28.9 %	1.46	8.76	6.94	43.44
	256 B	256 B	1 way	28.9 %	1.46	8.76	6.94	43.44
	128 B	64 B	2 way	45.3 %	1.13	6.79	5.38	33.67
	256 B	128 B	2 way	45.5 %	1.13	6.77	5.36	33.55
	512 B	256 B	2 way	45.5 %	1.13	6.77	5.36	33.55
	256 B	64 B	4 way	76.0 %	0.50	3.00	2.38	14.88
	512 B	128 B	4 way	76.2 %	0.50	2.97	2.35	14.74
	1 KB	256 B	4 way	76.2 %	0.50	2.97	2.35	14.74
	512 B	64 B	8 way	95.7 %	0.09	0.53	0.42	2.62
	1 KB	128 B	8 way	95.9 %	0.08	0.50	0.40	2.48
	2 KB	256 B	8 way	95.9 %	0.08	0.50	0.40	2.48
	4 KB	256 B	16 way	97.3 %	0.06	0.34	0.27	1.66
	8 KB	256 B	32 way	97.8 %	0.05	0.28	0.22	1.39
	16 KB	256 B	64 way	98.0 %	0.04	0.25	0.20	1.26
	32 KB	256 B	128 way	98.1 %	0.04	0.24	0.19	1.19
64 KB	256 B	256 way	98.2 %	0.04	0.23	0.18	1.13	
CC Line	64 B	64 B	1 way	52.8 %	15.01	33.87	75.21	98.78
	128 B	128 B	1 way	53.1 %	30.02	67.54	149.86	173.31
	256 B	256 B	1 way	53.1 %	60.03	135.08	299.94	323.40
	128 B	64 B	2 way	71.3 %	9.10	20.58	45.78	60.13
	256 B	128 B	2 way	71.6 %	18.20	40.95	90.87	105.09
	512 B	256 B	2 way	71.6 %	36.40	81.90	181.87	196.09
	256 B	64 B	4 way	88.6 %	3.58	8.15	18.24	23.95
	512 B	128 B	4 way	88.8 %	7.15	16.08	35.68	41.27
	1 KB	256 B	4 way	88.8 %	14.29	32.16	71.42	77.00
	512 B	64 B	8 way	97.4 %	0.77	1.82	4.22	5.54
	1 KB	128 B	8 way	97.6 %	1.52	3.42	7.60	8.79
	2 KB	256 B	8 way	97.6 %	3.04	6.85	15.21	16.40
	4 KB	256 B	16 way	98.4 %	2.07	4.66	10.36	11.17
	8 KB	256 B	32 way	98.6 %	1.78	4.02	8.92	9.62
	16 KB	256 B	64 way	98.7 %	1.65	3.71	8.25	8.90
	32 KB	256 B	128 way	98.8 %	1.57	3.55	7.88	8.49
64 KB	256 B	256 way	98.9 %	1.42	3.19	7.09	7.64	
Flush Single	64 B	64 B	1 way	28.1 %	1.48	8.86	7.01	43.93
	128 B	128 B	1 way	28.2 %	1.47	8.84	7.00	43.84
	256 B	256 B	1 way	28.2 %	1.47	8.84	7.00	43.84
	128 B	64 B	2 way	33.5 %	1.37	8.21	6.50	40.73
	256 B	128 B	2 way	33.7 %	1.36	8.19	6.48	40.60
	512 B	256 B	2 way	33.7 %	1.36	8.19	6.48	40.60
	256 B	64 B	4 way	38.7 %	1.25	7.47	5.92	37.06
	512 B	128 B	4 way	38.9 %	1.24	7.45	5.90	36.93
	1 KB	256 B	4 way	38.9 %	1.24	7.45	5.90	36.93
	512 B	64 B	8 way	38.9 %	1.24	7.45	5.90	36.95
	1 KB	128 B	8 way	39.1 %	1.24	7.43	5.88	36.82
	2 KB	256 B	8 way	39.1 %	1.24	7.43	5.88	36.82
	4 KB	256 B	16 way	39.2 %	1.24	7.42	5.88	36.80
	8 KB	256 B	32 way	39.2 %	1.24	7.42	5.88	36.80
	16 KB	256 B	64 way	39.2 %	1.24	7.42	5.88	36.80
	32 KB	256 B	128 way	39.2 %	1.24	7.42	5.88	36.80
64 KB	256 B	256 way	39.2 %	1.24	7.42	5.88	36.80	
Flush Line	64 B	64 B	1 way	49.1 %	16.22	36.60	81.25	106.72
	128 B	128 B	1 way	49.3 %	32.44	72.99	161.96	187.30
	256 B	256 B	1 way	49.3 %	64.88	145.99	324.17	349.51
	128 B	64 B	2 way	56.7 %	13.79	31.12	69.12	90.79
	256 B	128 B	2 way	56.9 %	27.57	62.03	137.63	159.17
	512 B	256 B	2 way	56.9 %	55.14	124.06	275.48	297.02
	256 B	64 B	4 way	63.4 %	11.62	26.25	58.33	76.62
	512 B	128 B	4 way	63.7 %	23.24	52.28	116.01	134.16
	1 KB	256 B	4 way	63.7 %	46.47	104.56	232.19	250.34
	512 B	64 B	8 way	63.5 %	11.59	26.18	58.17	76.40
	1 KB	128 B	8 way	63.8 %	23.17	52.13	115.68	133.78
	2 KB	256 B	8 way	63.8 %	46.34	104.27	231.53	249.63
	4 KB	256 B	16 way	63.8 %	46.33	104.24	231.47	249.57
	8 KB	256 B	32 way	63.8 %	46.33	104.24	231.47	249.56
	16 KB	256 B	64 way	63.8 %	46.33	104.24	231.46	249.56
	32 KB	256 B	128 way	63.8 %	46.33	104.24	231.46	249.56
64 KB	256 B	256 way	63.8 %	46.33	104.24	231.46	249.56	

TABLE XXII
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE LUINDEX BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	66.9 %	0.70	4.22	3.34	20.90
	128 B	128 B	1 way	66.9 %	0.70	4.22	3.34	20.90
	256 B	256 B	1 way	66.9 %	0.70	4.22	3.34	20.90
	128 B	64 B	2 way	77.8 %	0.48	2.88	2.28	14.29
	256 B	128 B	2 way	77.6 %	0.49	2.92	2.31	14.47
	512 B	256 B	2 way	77.6 %	0.49	2.92	2.31	14.47
	256 B	64 B	4 way	81.5 %	0.40	2.41	1.91	11.95
	512 B	128 B	4 way	81.9 %	0.39	2.37	1.87	11.73
	1 KB	256 B	4 way	81.9 %	0.39	2.37	1.87	11.73
	512 B	64 B	8 way	84.6 %	0.33	1.99	1.58	9.87
	1 KB	128 B	8 way	85.1 %	0.32	1.93	1.53	9.58
	2 KB	256 B	8 way	85.1 %	0.32	1.93	1.53	9.58
	4 KB	256 B	16 way	88.4 %	0.25	1.48	1.17	7.34
	8 KB	256 B	32 way	90.8 %	0.19	1.17	0.93	5.80
	16 KB	256 B	64 way	93.2 %	0.14	0.85	0.67	4.20
	32 KB	256 B	128 way	94.0 %	0.12	0.74	0.59	3.68
64 KB	256 B	256 way	94.4 %	0.12	0.69	0.55	3.43	
CC Line	64 B	64 B	1 way	80.8 %	5.96	13.65	30.67	40.28
	128 B	128 B	1 way	81.3 %	11.99	26.98	59.86	69.23
	256 B	256 B	1 way	81.3 %	23.98	53.96	119.82	129.18
	128 B	64 B	2 way	88.5 %	3.47	8.05	18.27	23.99
	256 B	128 B	2 way	88.8 %	7.17	16.14	35.81	41.41
	512 B	256 B	2 way	88.8 %	14.35	32.28	71.68	77.28
	256 B	64 B	4 way	91.8 %	2.44	5.73	13.13	17.25
	512 B	128 B	4 way	92.4 %	4.85	10.91	24.21	28.00
	1 KB	256 B	4 way	92.4 %	9.70	21.82	48.46	52.25
	512 B	64 B	8 way	93.7 %	1.83	4.37	10.11	13.28
	1 KB	128 B	8 way	94.3 %	3.64	8.18	18.16	21.00
	2 KB	256 B	8 way	94.3 %	7.27	16.37	36.34	39.18
	4 KB	256 B	16 way	95.9 %	5.25	11.80	26.21	28.25
	8 KB	256 B	32 way	96.8 %	4.05	9.11	20.24	21.82
	16 KB	256 B	64 way	97.7 %	3.00	6.75	15.00	16.17
	32 KB	256 B	128 way	97.9 %	2.65	5.96	13.23	14.26
64 KB	256 B	256 way	98.1 %	2.43	5.48	12.16	13.11	
Flush Single	64 B	64 B	1 way	65.7 %	0.73	4.35	3.45	21.59
	128 B	128 B	1 way	65.7 %	0.73	4.35	3.45	21.59
	256 B	256 B	1 way	65.7 %	0.73	4.35	3.45	21.59
	128 B	64 B	2 way	76.0 %	0.52	3.10	2.45	15.37
	256 B	128 B	2 way	75.8 %	0.52	3.13	2.48	15.53
	512 B	256 B	2 way	75.8 %	0.52	3.13	2.48	15.53
	256 B	64 B	4 way	79.6 %	0.44	2.65	2.10	13.13
	512 B	128 B	4 way	79.9 %	0.43	2.60	2.06	12.91
	1 KB	256 B	4 way	79.9 %	0.43	2.60	2.06	12.91
	512 B	64 B	8 way	82.4 %	0.38	2.26	1.79	11.21
	1 KB	128 B	8 way	82.9 %	0.37	2.20	1.74	10.93
	2 KB	256 B	8 way	82.9 %	0.37	2.20	1.74	10.93
	4 KB	256 B	16 way	85.7 %	0.30	1.81	1.43	8.98
	8 KB	256 B	32 way	87.1 %	0.27	1.62	1.28	8.01
	16 KB	256 B	64 way	89.1 %	0.22	1.34	1.06	6.65
	32 KB	256 B	128 way	89.7 %	0.21	1.26	1.00	6.24
64 KB	256 B	256 way	90.0 %	0.20	1.22	0.97	6.07	
Flush Line	64 B	64 B	1 way	79.6 %	6.32	14.47	32.50	42.69
	128 B	128 B	1 way	80.1 %	12.73	28.65	63.57	73.52
	256 B	256 B	1 way	80.1 %	25.47	57.30	127.24	137.19
	128 B	64 B	2 way	87.0 %	3.96	9.15	20.70	27.19
	256 B	128 B	2 way	87.3 %	8.15	18.34	40.70	47.07
	512 B	256 B	2 way	87.3 %	16.31	36.69	81.46	87.83
	256 B	64 B	4 way	90.1 %	2.97	6.93	15.79	20.74
	512 B	128 B	4 way	90.7 %	5.93	13.34	29.59	34.23
	1 KB	256 B	4 way	90.7 %	11.86	26.68	59.24	63.87
	512 B	64 B	8 way	91.7 %	2.45	5.76	13.21	17.34
	1 KB	128 B	8 way	92.4 %	4.89	11.01	24.42	28.24
	2 KB	256 B	8 way	92.4 %	9.78	22.01	48.88	52.70
	4 KB	256 B	16 way	93.8 %	7.95	17.89	39.73	42.83
	8 KB	256 B	32 way	94.4 %	7.19	16.18	35.94	38.75
	16 KB	256 B	64 way	95.1 %	6.31	14.20	31.52	33.98
	32 KB	256 B	128 way	95.3 %	6.03	13.56	30.12	32.47
64 KB	256 B	256 way	95.4 %	5.87	13.21	29.33	31.62	

TABLE XXIII
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE LUSEARCH BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
					Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	22.0 %	1.56	9.36	7.41	46.40
	128 B	128 B	1 way	21.9 %	1.56	9.38	7.42	46.50
	256 B	256 B	1 way	21.9 %	1.56	9.38	7.42	46.50
	128 B	64 B	2 way	73.1 %	0.54	3.24	2.56	16.04
	256 B	128 B	2 way	73.9 %	0.52	3.14	2.48	15.55
	512 B	256 B	2 way	73.9 %	0.52	3.14	2.48	15.55
	256 B	64 B	4 way	84.7 %	0.31	1.84	1.46	9.12
	512 B	128 B	4 way	85.6 %	0.29	1.73	1.37	8.60
	1 KB	256 B	4 way	85.6 %	0.29	1.73	1.37	8.60
	512 B	64 B	8 way	88.8 %	0.22	1.35	1.07	6.68
	1 KB	128 B	8 way	89.8 %	0.20	1.23	0.97	6.09
	2 KB	256 B	8 way	89.8 %	0.20	1.23	0.97	6.09
	4 KB	256 B	16 way	92.3 %	0.15	0.92	0.73	4.57
	8 KB	256 B	32 way	93.4 %	0.13	0.79	0.62	3.91
	16 KB	256 B	64 way	94.0 %	0.12	0.72	0.57	3.55
	32 KB	256 B	128 way	94.5 %	0.11	0.66	0.52	3.26
64 KB	256 B	256 way	94.8 %	0.10	0.62	0.49	3.09	
CC Line	64 B	64 B	1 way	30.9 %	21.71	49.36	110.23	144.78
	128 B	128 B	1 way	30.5 %	44.46	100.04	221.97	256.71
	256 B	256 B	1 way	30.5 %	88.93	200.09	444.29	479.03
	128 B	64 B	2 way	87.6 %	3.57	8.54	19.81	26.02
	256 B	128 B	2 way	88.8 %	7.16	16.11	35.76	41.35
	512 B	256 B	2 way	88.8 %	14.32	32.23	71.56	77.16
	256 B	64 B	4 way	93.4 %	1.71	4.34	10.49	13.78
	512 B	128 B	4 way	94.7 %	3.37	7.57	16.80	19.43
	1 KB	256 B	4 way	94.7 %	6.73	15.15	33.63	36.26
	512 B	64 B	8 way	95.1 %	1.16	3.11	7.77	10.21
	1 KB	128 B	8 way	96.5 %	2.27	5.10	11.32	13.09
	2 KB	256 B	8 way	96.5 %	4.53	10.20	22.65	24.42
	4 KB	256 B	16 way	97.6 %	3.11	6.99	15.53	16.75
	8 KB	256 B	32 way	97.9 %	2.63	5.91	13.13	14.15
	16 KB	256 B	64 way	98.2 %	2.37	5.32	11.82	12.74
	32 KB	256 B	128 way	98.3 %	2.16	4.86	10.78	11.63
64 KB	256 B	256 way	98.4 %	2.02	4.55	10.10	10.89	
Flush Single	64 B	64 B	1 way	18.5 %	1.63	9.79	7.75	48.52
	128 B	128 B	1 way	18.3 %	1.63	9.81	7.76	48.63
	256 B	256 B	1 way	18.3 %	1.63	9.81	7.76	48.63
	128 B	64 B	2 way	68.6 %	0.63	3.77	2.98	18.69
	256 B	128 B	2 way	69.4 %	0.61	3.67	2.91	18.22
	512 B	256 B	2 way	69.4 %	0.61	3.67	2.91	18.22
	256 B	64 B	4 way	78.9 %	0.42	2.54	2.01	12.59
	512 B	128 B	4 way	79.7 %	0.41	2.44	1.93	12.09
	1 KB	256 B	4 way	79.7 %	0.41	2.44	1.93	12.09
	512 B	64 B	8 way	81.1 %	0.38	2.27	1.80	11.26
	1 KB	128 B	8 way	82.0 %	0.36	2.17	1.71	10.74
	2 KB	256 B	8 way	82.0 %	0.36	2.17	1.71	10.74
	4 KB	256 B	16 way	82.6 %	0.35	2.09	1.65	10.35
	8 KB	256 B	32 way	82.8 %	0.35	2.07	1.64	10.27
	16 KB	256 B	64 way	82.8 %	0.34	2.07	1.64	10.26
	32 KB	256 B	128 way	82.8 %	0.34	2.07	1.64	10.26
64 KB	256 B	256 way	82.8 %	0.34	2.07	1.64	10.26	
Flush Line	64 B	64 B	1 way	32.3 %	21.27	48.36	108.03	141.89
	128 B	128 B	1 way	32.0 %	43.52	97.92	217.27	251.27
	256 B	256 B	1 way	32.0 %	87.04	195.84	434.87	468.87
	128 B	64 B	2 way	84.1 %	4.70	11.08	25.42	33.39
	256 B	128 B	2 way	85.3 %	9.44	21.24	47.12	54.50
	512 B	256 B	2 way	85.3 %	18.88	42.48	94.32	101.69
	256 B	64 B	4 way	88.9 %	3.15	7.58	17.68	23.22
	512 B	128 B	4 way	90.2 %	6.28	14.12	31.33	36.23
	1 KB	256 B	4 way	90.2 %	12.55	28.24	62.71	67.61
	512 B	64 B	8 way	89.8 %	2.86	6.93	16.24	21.33
	1 KB	128 B	8 way	91.1 %	5.69	12.81	28.42	32.86
	2 KB	256 B	8 way	91.1 %	11.38	25.61	56.88	61.32
	4 KB	256 B	16 way	91.3 %	11.11	25.00	55.50	59.84
	8 KB	256 B	32 way	91.4 %	11.05	24.86	55.20	59.51
	16 KB	256 B	64 way	91.4 %	11.04	24.84	55.15	59.46
	32 KB	256 B	128 way	91.4 %	11.04	24.83	55.14	59.45
64 KB	256 B	256 way	91.4 %	11.03	24.83	55.13	59.44	

TABLE XXIV
OBJECT CACHE HIT RATE AND MISS PENALTY FOR THE XALAN BENCHMARK

Type	Cache			Hit rate	Miss cycles per field read			
	Size	Line	Assoc.		Uniprocessor		8 core CMP	
					SRAM	SDRAM	SRAM	SDRAM
CC Single	64 B	64 B	1 way	7.1 %	1.86	11.15	8.83	55.28
	128 B	128 B	1 way	5.7 %	1.89	11.32	8.96	56.14
	256 B	256 B	1 way	5.6 %	1.89	11.33	8.97	56.19
	128 B	64 B	2 way	25.3 %	1.49	8.97	7.10	44.45
	256 B	128 B	2 way	22.8 %	1.54	9.26	7.33	45.93
	512 B	256 B	2 way	22.7 %	1.55	9.28	7.35	46.02
	256 B	64 B	4 way	48.3 %	1.03	6.21	4.91	30.77
	512 B	128 B	4 way	49.8 %	1.00	6.02	4.77	29.87
	1 KB	256 B	4 way	49.8 %	1.01	6.03	4.77	29.90
	512 B	64 B	8 way	60.2 %	0.80	4.78	3.78	23.70
	1 KB	128 B	8 way	63.6 %	0.73	4.37	3.46	21.66
	2 KB	256 B	8 way	63.6 %	0.73	4.37	3.46	21.66
	4 KB	256 B	16 way	72.9 %	0.54	3.26	2.58	16.14
	8 KB	256 B	32 way	79.6 %	0.41	2.45	1.94	12.13
	16 KB	256 B	64 way	85.6 %	0.29	1.73	1.37	8.57
	32 KB	256 B	128 way	89.2 %	0.22	1.30	1.03	6.42
64 KB	256 B	256 way	91.1 %	0.18	1.07	0.85	5.30	
CC Line	64 B	64 B	1 way	16.2 %	24.13	57.63	133.62	175.50
	128 B	128 B	1 way	13.5 %	54.79	124.02	276.48	319.75
	256 B	256 B	1 way	13.3 %	111.03	249.81	554.71	598.08
	128 B	64 B	2 way	49.9 %	13.33	33.36	79.83	104.86
	256 B	128 B	2 way	49.3 %	31.86	72.43	162.01	187.37
	512 B	256 B	2 way	49.7 %	64.42	144.95	321.85	347.02
	256 B	64 B	4 way	72.8 %	6.02	16.90	43.38	56.98
	512 B	128 B	4 way	78.7 %	13.01	30.02	67.92	78.55
	1 KB	256 B	4 way	79.5 %	26.23	59.02	131.06	141.31
	512 B	64 B	8 way	79.5 %	3.88	12.08	32.70	42.95
	1 KB	128 B	8 way	86.6 %	7.96	18.64	42.68	49.36
	2 KB	256 B	8 way	87.5 %	16.02	36.05	80.05	86.31
	4 KB	256 B	16 way	91.7 %	10.68	24.02	53.34	57.51
	8 KB	256 B	32 way	94.3 %	7.25	16.31	36.22	39.05
	16 KB	256 B	64 way	96.4 %	4.66	10.50	23.31	25.14
	32 KB	256 B	128 way	97.4 %	3.27	7.35	16.32	17.60
64 KB	256 B	256 way	98.0 %	2.58	5.82	12.92	13.93	
Flush Single	64 B	64 B	1 way	5.6 %	1.89	11.33	8.97	56.16
	128 B	128 B	1 way	4.4 %	1.91	11.47	9.08	56.89
	256 B	256 B	1 way	4.3 %	1.91	11.48	9.09	56.94
	128 B	64 B	2 way	21.9 %	1.56	9.38	7.42	46.50
	256 B	128 B	2 way	19.3 %	1.61	9.68	7.67	48.01
	512 B	256 B	2 way	19.2 %	1.62	9.70	7.68	48.08
	256 B	64 B	4 way	43.2 %	1.14	6.82	5.40	33.80
	512 B	128 B	4 way	44.6 %	1.11	6.65	5.26	32.96
	1 KB	256 B	4 way	44.6 %	1.11	6.65	5.26	32.97
	512 B	64 B	8 way	49.7 %	1.01	6.04	4.78	29.95
	1 KB	128 B	8 way	52.5 %	0.95	5.70	4.51	28.26
	2 KB	256 B	8 way	52.6 %	0.95	5.69	4.50	28.21
	4 KB	256 B	16 way	54.4 %	0.91	5.47	4.33	27.12
	8 KB	256 B	32 way	54.8 %	0.90	5.42	4.29	26.89
	16 KB	256 B	64 way	54.8 %	0.90	5.42	4.29	26.88
	32 KB	256 B	128 way	54.8 %	0.90	5.42	4.29	26.87
64 KB	256 B	256 way	54.8 %	0.90	5.42	4.29	26.87	
Flush Line	64 B	64 B	1 way	19.1 %	23.19	55.55	129.01	169.45
	128 B	128 B	1 way	16.6 %	52.76	119.46	266.36	308.05
	256 B	256 B	1 way	16.5 %	106.85	240.43	533.86	575.61
	128 B	64 B	2 way	50.0 %	13.31	33.30	79.72	104.71
	256 B	128 B	2 way	50.2 %	31.31	71.19	159.27	184.20
	512 B	256 B	2 way	50.6 %	63.25	142.31	316.00	340.70
	256 B	64 B	4 way	68.7 %	7.33	19.86	49.94	65.60
	512 B	128 B	4 way	74.5 %	15.72	36.12	81.45	94.20
	1 KB	256 B	4 way	75.2 %	31.70	71.32	158.37	170.75
	512 B	64 B	8 way	72.3 %	6.18	17.25	44.18	58.03
	1 KB	128 B	8 way	79.0 %	12.88	29.71	67.25	77.77
	2 KB	256 B	8 way	79.8 %	25.88	58.24	129.33	139.44
	4 KB	256 B	16 way	80.8 %	24.62	55.40	123.03	132.65
	8 KB	256 B	32 way	80.9 %	24.41	54.93	121.97	131.50
	16 KB	256 B	64 way	80.9 %	24.40	54.90	121.90	131.43
	32 KB	256 B	128 way	80.9 %	24.40	54.89	121.89	131.42
64 KB	256 B	256 way	80.9 %	24.39	54.89	121.88	131.41	