An object-aware memory architecture

Greg Wright, Matthew L. Seidl
and Mario Wolczko

An object-aware memory architecture

Greg Wright, Matthew L. Seidl, and Mario Wolczko

SMLI TR-2005-143 February 2005

Abstract:

Despite its dominance, object-oriented computation has received scant attention from the architecture community. We propose a novel memory architecture that supports objects and garbage collection (GC). Our architecture is co-designed with a Java Virtual Machine to improve the functionality and efficiency of heap memory management. The architecture is based on an address space for objects accessed using object IDs mapped by a translator to physical addresses. To support this, the system includes object-addressed caches, a hardware GC barrier to allow in-cache GC of objects, and an exposed cache structure cooperatively managed by the JVM. These extend a conventional architecture, without compromising compatibility or performance for legacy binaries.

Our innovations enable various improvements such as: a novel technique for parallel and concurrent garbage collection, without requiring any global synchronization; an in-cache garbage collector, which never accesses main memory; concurrent compaction of objects; and elimination of most GC store barrier overhead. We compare the behavior of our system against that of a conventional generational garbage collector, both with and without an explicit allocate-in-cache operation. Explicit allocation eliminates many write misses; our scheme additionally trades L2 misses for in-cache operations, and provides the mapping indirection required for concurrent compaction.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
greg.wright@sun.com
matthew.seidl@sun.com
mario.wolczko@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

An object-aware memory architecture

Greg Wright, Matthew L. Seidl and Mario Wolczko

1. Introduction and motivation

Object-oriented programming is the dominant software development paradigm, and has been so for the last decade. Object-oriented programming languages, such as Java™ and C#, have converged on a common object model whose roots can be found in Smalltalk [9]. For our purposes we can summarize the model with three properties. Firstly, objects are small, cheap and plentiful. Secondly, object storage is reclaimed automatically through garbage collection (GC). Thirdly, although object references are unforgeable, all permissions are managed through checks done at class-loading or compile time (the Java language combines static type safety with package-based access controls). Although our work focuses on Java it is equally applicable to other languages with a similar model; we note that it may not provide any benefits for some other language models (most notably C++, which allows pointers into the middle of objects).

Computer architecture has mostly ignored this object model. In particular the requirements of, and constraints on, the memory system are quite different from those of a flat, paged virtual memory. For example, small objects reduce the effectiveness of long cache lines and TLBs [22] because inter-object locality is not assured. The primary architectural interface to Java applications is at the virtual machine (JVM™) level, and this gives considerable freedom to innovate across the hardware/software boundary. Memory system performance is important in the commercial server market, where applications based on Java 2 Enterprise Edition (J2EE™) may require many gigabytes of heap space. We are investigating how hardware support for objects, co-designed with the virtual machine, can lead to better memory system performance and also enable new memory management algorithms which cannot currently be implemented efficiently in software.

We assume that changes can be made to the virtual machine, operating system and hardware, but we do not consider a completely 'clean sheet' design: legacy applications must run unchanged on the same system. We therefore extend a conventional instruction set architecture where appropriate, retaining backwards compatibility. The architectural modifications are only support for the software algorithms; we propose hardware structures only where they offer functionality or efficiency that software cannot match, and leave as much flexibility as possible for later software innovation. In particular, we are not proposing hardware implementations of complete garbage collectors [3, 19], bytecode execution [10, 20], or fine-grained protection [14, 17, 31].

Our approach is centered around two ideas: support for objects as first-class entities in the memory system, and collaboration between hardware and software for memory management. The fundamentals of the proposed architecture are presented in §2, and GC-related extensions are described in §3. Comparisons to related work are in §4, then §5 describes our evaluation methodology and §6 presents the results of our initial evaluation. In §7 we present conclusions, and §8 suggests future work.

2. A memory hierarchy for objects

Current production JVMs use a direct pointer representation for object references: each object is referenced by its base virtual address. The conventional virtual memory system maps virtual to physical addresses. This is in contrast to early vir-

tual machines, such as Smalltalk VMs of the '80s [9], which used an indirect representation: objects were referred to by a location-independent Object ID (OID), actually an index into an *object table*. An object's object table entry (OTE) contained the object's virtual memory address. The direct representation saves a (dependent) load instruction on field access by the *mutator* (application code, as distinct from the garbage collector); the indirect representation makes relocation easier because an object's memory address is stored in only one place. Relocation is an important component of a variety of memory management functions, most notably heap compaction which overcomes fragmentation. However, since the late '80s direct access has been the method of choice, because the indirection overhead of the object table has been too high [26].

The temporal properties of memory management algorithms are important when scaling to large heaps: it is desirable to have good average mutator throughput, i.e., low memory management overhead, and also to ensure that pauses, during which mutator work is suspended, are kept within acceptable bounds. Concurrent, or mostly-concurrent [1], garbage collection is very attractive for avoiding lengthy pauses when operating on multi-gigabyte heaps: here the collector runs at the same time as the mutator, with some synchronization mechanisms (barriers, mutator thread suspension, etc.) to ensure correctness. Concurrent compaction is more difficult: an object must be relocated with apparent atomicity to the mutator. With a direct pointer representation this means updating all references to an object ‘‘at once’’ (strictly harder than concurrent GC, which requires only reading a single reference). An indirect representation implemented on a conventional architecture still requires extra synchronization in the mutator because the load–load or load–store indirection through the object table is not atomic.

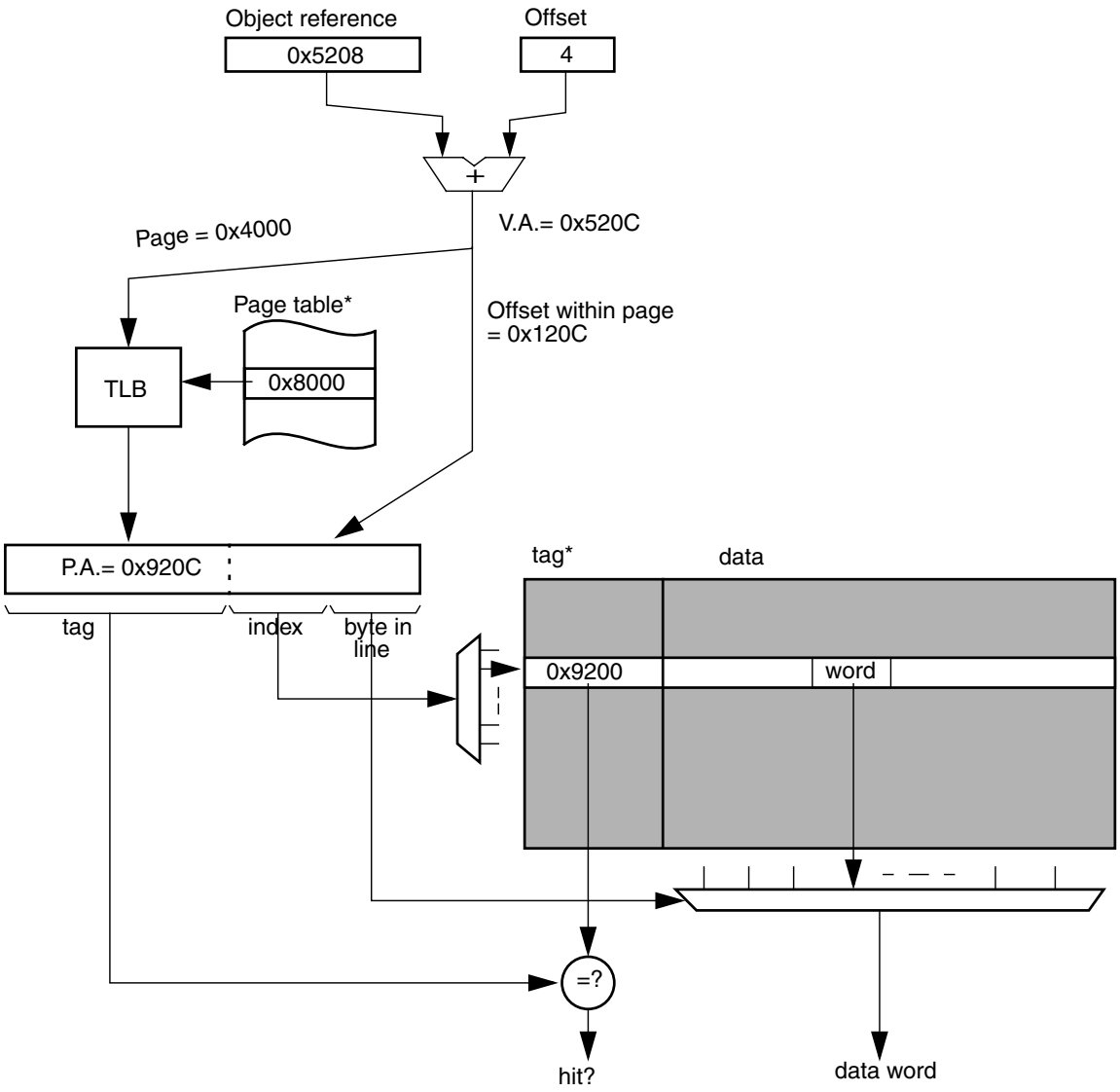
Architectural support for objects can provide the best of both worlds: fast access to objects in the common case, with easy relocation. With little hardware support we can also perform synchronized relocation with no overhead in the mutator: this allows truly concurrent compaction. With concurrent relocation we can move objects while all mutator threads are active, thus allowing various optimizations to be performed on the heap and improving the maximum heap sizes that can be handled without inducing unacceptable garbage collection pauses.

2.1 Object-addressed caches

In a conventional system the instruction to load an object field takes two source operands: the reference to the object (i.e., its base virtual address) and the field offset within the object. The pointer and offset are summed to produce the field's virtual address which is fed to the TLB and (physically-tagged) L1 cache (Fig. 1).

An object-addressed cache [29] instead has cache lines tagged directly with (OID, offset) addresses, and thus contains parts of objects rather than blocks of physical memory (Fig. 2). This organization is similar to a virtually-addressed cache, except that the OID and offset bits are concatenated rather than summed (a slight simplification for now, see more below); there is no aliasing of object addresses. An object-load instruction takes the same two source operands as the conventional load, but rather than being added they bypass the TLB and go directly to the cache index/tag match hardware. The tags of an object-addressed cache must be longer to handle the desired (virtual) size of object space. The maximum object size can be restricted: 10 offset bits (256 fields of 4 bytes each) cover the majority of objects in practice, and longer objects can be composed by the virtual machine. Object addresses and tags may therefore be only a few bits longer than the physical addresses already handled in upcoming systems. The OID address space is a shared (non-virtualized) resource, but multiple application-level processes may reside within a single JVM [5].

Each object cache line contains part of only one object; objects can span multiple cache lines, but most objects will occupy only one (given 64-byte cache lines). Some space at the ends of lines will be unused (external fragmentation). The fragmentation has two potentially negative effects on the system. Firstly, it reduces the effective cache



* Tags, pages, etc., represented as the full addresses: in practice, some of these bits are redundant.

Fig. 1: Finding a physically-tagged datum in a conventional cache

capacity: with denser packing more objects would fit into the cache. Secondly, it removes the possibility that a cache miss will prefetch an adjacent object. Both the packing and prefetching rely on spatial locality between the objects: in its absence the adjacent objects are not used and provide no benefit. In our system, objects are allocated directly in the cache (§2.3); the prefetching effect is only relevant for objects old enough to have been already evicted from the cache. Some studies have demonstrated spatial locality between recently-allocated objects but little for older objects [2], thus we would not expect much benefit from prefetching older objects in any case. Both of these effects are of course reflected in our results: they are costs which we pay in exchange for the other benefits of our system. An additional benefit of an object cache is that it precludes false sharing between different objects.

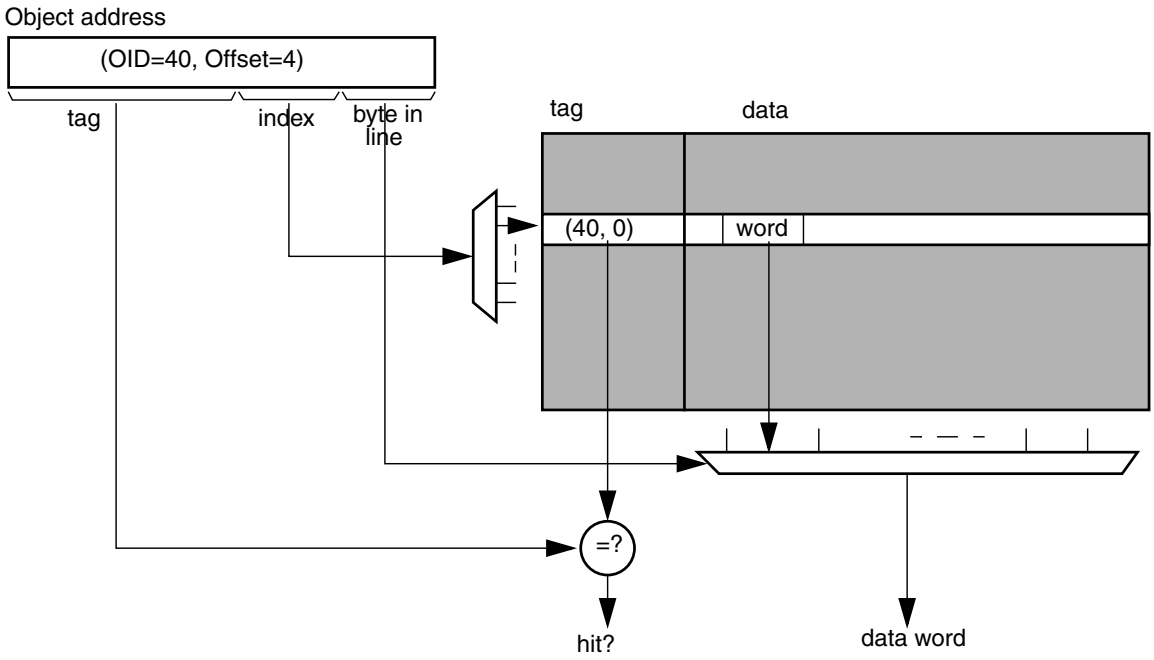


Fig. 2: An object-addressed cache (simplified)

Following our principle that object functionality merely extends an existing system, we allow ordinary physically-tagged cache lines to coexist in the object cache with the object-tagged cache lines. As long as the tags do not collide, ensured for example by embedding the object addresses into a higher address range than physical memory (e.g., a set high-order bit), the cache hardware operates in the usual manner after the address generation stage and is unaware that it is dealing with objects. We will use 'extended physical address' (EPA) to mean an address (bit string) which is either a physical memory location or an encoded (OID, offset) address.

Most cache coherence mechanisms work unchanged when using EPAs rather than straightforward physical addresses; snoops from other CPUs destined for object addresses will match on the encoded tags and change coherence states in the usual manner. The unusual cases are cache misses in all levels of the hierarchy, which necessitate a memory access (see §2.4), and the directories in directory-based protocols. Although directory-based object coherence is possible, details are beyond the scope of this report.

2.2 Object address encoding

A system implemented exactly as described above would have poor placement of objects within the cache. Non-fully-associative caches are conventionally indexed using a subset of the (physical) address bits. This has the effect of distributing consecutive addresses through the cache in the hope of reducing conflicts, and it is cheap to compute. With objects in the cache the index should neither consist solely of OID bits (so that the several cache lines of large objects conflict within one cache set), nor only of offset bits (so that small objects are restricted to a portion of the cache). Instead, we encode (OID, offset) addresses into the EPA range using an invertible mapping which combines part of the OID and offset bits into the index bits. An example using exclusive-or combination is illustrated in Fig. 3. Indexing by bit extraction will have the desired effect, and the 'real' OID and offset (when needed) can be recovered by inverting the encoding.

One other operation is performed as part of the encoding. A bifurcated object layout [8, 33] supports fast identification of references for the garbage collector: the object header is in the middle

of the object with references at negative offsets and primitive values at positive offsets (Java's fields are declared as containing one or the other, unlike Smalltalk's). In a straightforward implementation even small objects would occupy two cache lines, an intolerable overhead. We therefore *skew* objects within cache lines: the OID contains a small number of bits specifying where within its cache line the object header (logical offset zero) should lie. Fig. 4 shows how an object is skewed so that the reference fields fit within the same cache line as the object header. The skew value is

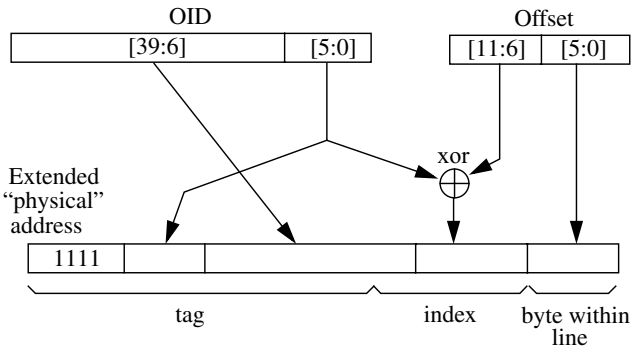


Fig. 3: Object address encoding (example with 40 bit OID + 12 bit offset)

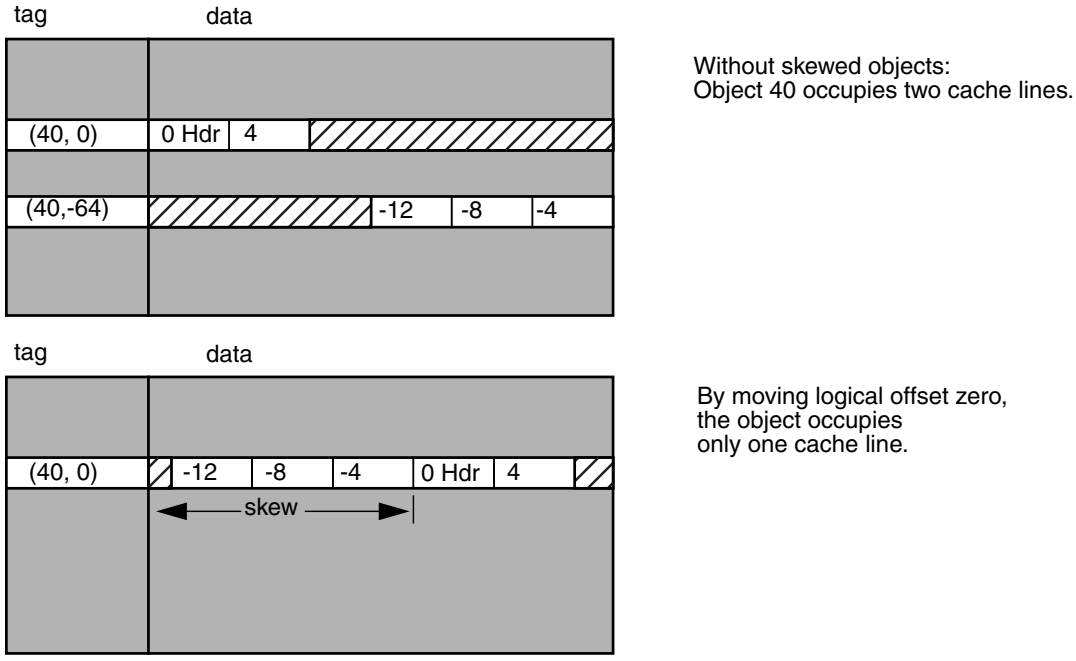


Fig. 4: A small object with and without skewing

added to the requested logical offset as part of the address generation, when the (OID, offset) address is encoded to form the EPA. Fig. 5 shows how Fig. 3 is modified to include skew; here three skew bits are taken from the high-order bits of the OID, allowing skewing to any 8-byte boundary within the 64-byte cache line. The skewing is mostly transparent to software; only the allocator and garbage collector, which must know how fields are organized within cache lines, are aware. The desired skew value is constant for a particular class.

2.3 Cache management: object allocation and deallocation

Caches in conventional systems are designed to operate mostly transparently to application code, although some parts of the operating system may need to be aware of the cache organization. Hints, such as prefetch instructions, may be provided for performance but are semantically neutral. In our case, co-design of the hardware and virtual machine allows better communication of intent. For example, when allocating a new object there is no need to fetch the contents from memory into

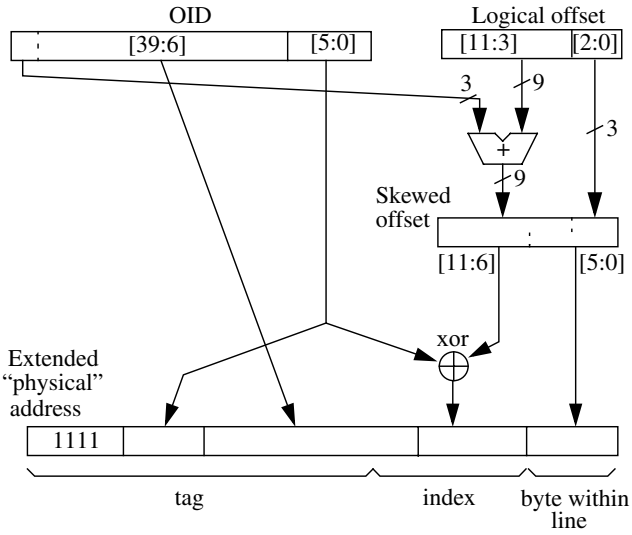


Fig. 5: Object address encoding (example with skew)

cache – the constructor will overwrite all the data. A zero-and-allocate instruction, like the PowerPC DCBZ (data cache block zero) [13], requests both that a cache line be allocated writable in the cache and that the contents be set to zero. Similarly, once an object is known to be garbage there is no need to write back its contents from the cache to memory; freeing its cache lines immediately will prevent unnecessary memory traffic and enable better placement of new objects in the cache. Variants of both of these operations were proposed in [21].

2.4 Translation

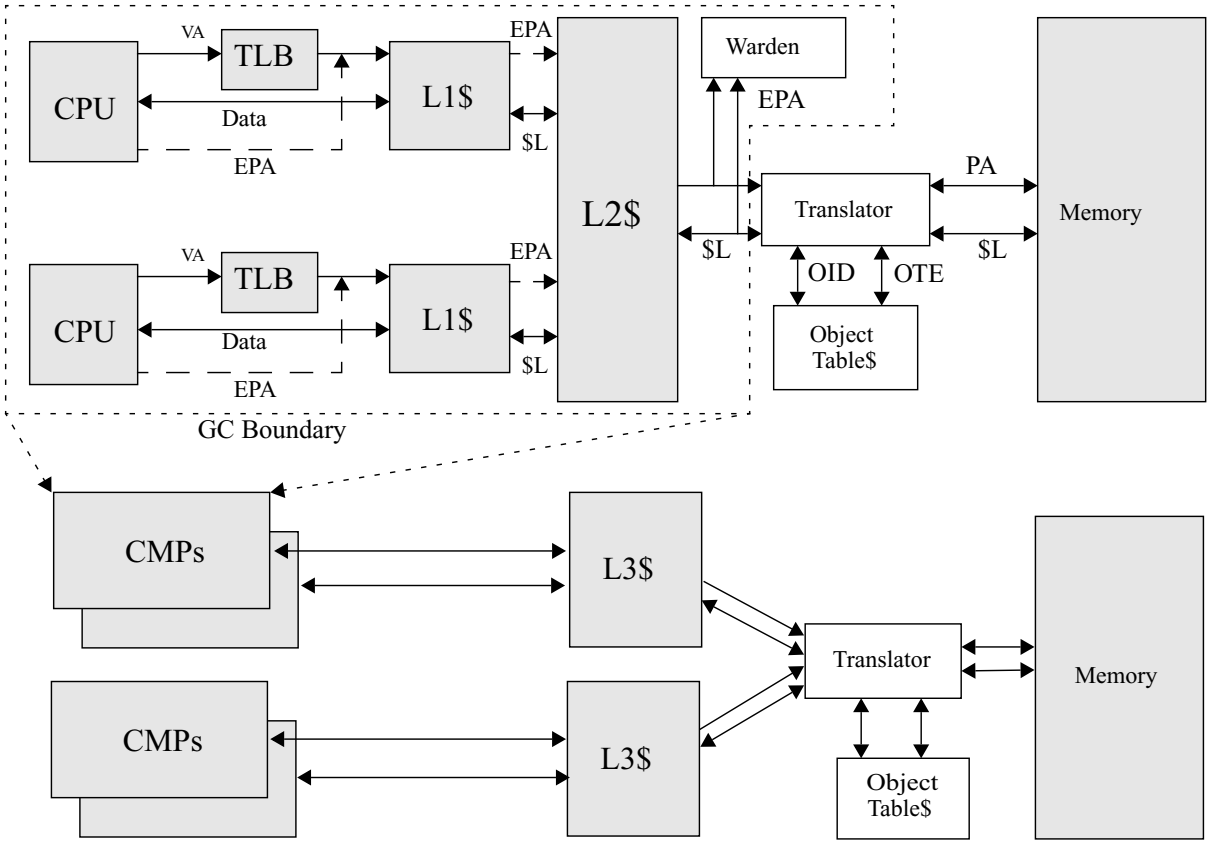
An object cache allows fast access to objects, in the common (hit) case, using only the location-independent object ID. In the case of a cache miss or eviction the system must still provide a means of retrieving or storing the object's state; translation involves looking up the object's physical address in the object table and then reconciling the contents of the object cache line with the in-memory representation of the object. We accomplish this with a hardware mechanism logically interposed between the cache hierarchy and main memory (Fig. 6). The translator intercepts cache misses to object EPAs and, using a simple state machine, reads the object table entry to get the physical address and generates (cache coherent) fetches of the necessary physical cache lines; the object table lookup is similar to a hardware page-table walk. A single object cache line may overlap one or two physical cache lines – objects in memory are word-aligned, not cache-line aligned, to avoid the loss of heap memory to fragmentation. The object table entry also contains size information so that the ends of objects are handled correctly. A logical view of the object table is shown in Fig. 7, but it is not a physically separate structure; as with conventional page tables, the object table is stored in coherent memory and the VM manipulates it with loads and stores.

It should be emphasized that the object cache line representation of an object is not kept coherent with the view of its physical memory representation as seen through conventional memory accesses. Coherence works completely separately on the object cache lines and physically-tagged cache lines; the translator only reconciles the two when an object cache line is created on a cache miss or written back on eviction. The mutator's view of the object is solely through the object cache lines. The virtual machine must take care not to modify an object using conventional loads and stores, which is easily achieved by never mapping the heap into the conventional virtual address space. The CPUs have coherent access to an object's physical storage only for internal VM functions such as relocation.

2.5 Concurrent object relocation

A long-running object system must relocate live objects to avoid heap fragmentation; this heap compaction should be concurrent or incremental to avoid lengthy mutator pauses. As discussed above, indirect references through an object table can make relocation much easier because the physical address of an object is stored in only one place. Here we describe how to synchronize translation and relocation in the object memory hierarchy so that mutation, with its implied translations, can proceed concurrently with compaction.

The basic model for copying relocation is that the compactor reads the contents of the physical memory backing the object, copies those data to the new location, and then updates the object table entry to point to the new area. There are two con-



The unshaded boxes represent new units; dashed lines are new data paths.
Upper diagram represents a 2-CPU single-chip system, as simulated.
Lower diagram shows extensions: multiple-CMP system, caches outside the GC boundary.

Fig. 6: Schematic of the proposed architecture

currency problems with respect to translation: firstly, a write-translation (cache eviction) of part of the object must be reflected in the copy, and, secondly, a read-translation must use the old location until the copy is complete. The read problem is easier: in the absence of write translations the two versions are identical when the copy is done, so apparent atomicity with respect to the OTE update is not critical.

We solve the write-translation problem by using a single 'evicted' bit in each OTE. Whenever the translator performs a write-translation on part of an object it sets this bit atomically with respect to the update of the physical cache lines, for example by holding the cache line containing the OTE until the updates are complete. The compactor clears the bit before starting a relocation, copies the physical memory, and then swaps the new physical address into the OTE using a compare-and-swap (CAS) operation. The CAS will fail and the copy be retried if the translator processed an eviction during the copy; if the CAS succeeds the new location is atomically switched in.

2.6 Summary

We have described a memory hierarchy for objects, offering concurrent relocation and the flexibility of an object table with no code size or speed penalty (in the common case). An object cache holds parts of objects, tagged directly with the location-independent object ID and offset, alongside conventional physically-tagged cache lines. The cache's index function for object addresses is determined by the choice of encoding

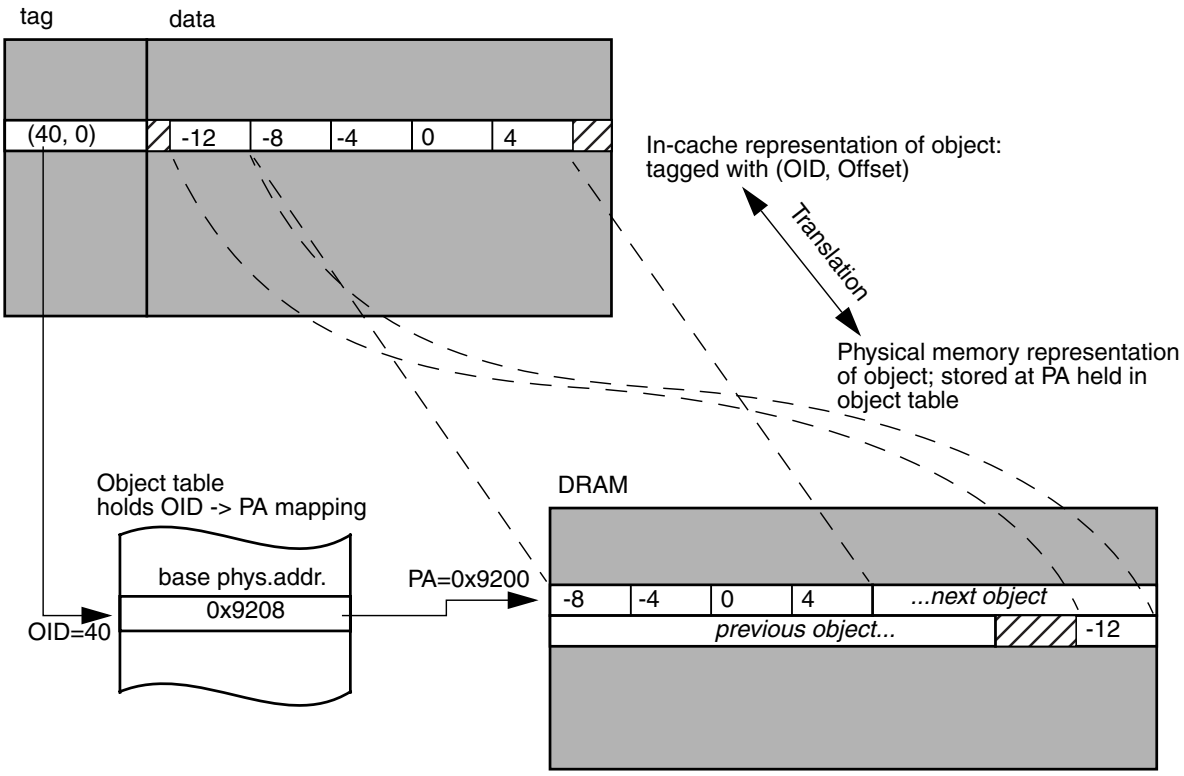


Fig. 7: The organization of an object within the cache and in memory, and its associated object table entry

into the extended address range. Object cache lines are kept coherent using the normal cache coherence mechanism. A hardware translator handles conversion of object cache lines to and from the representation in physical memory in the case of a cache miss or eviction.

The memory hierarchy for objects forms a base for future work on highly scalable, concurrent memory management algorithms. The next section presents one possibility, building on the features of the object cache.

3. In-cache garbage collection

Garbage collection is an inefficient process for caches: typically the object graph is traversed by a mark–sweep collector touching reference fields in each live object exactly once; although there is some spatial locality within objects there is little locality between objects referenced in this pattern [15]. If most accesses are to recently allocated objects, the caches and the youngest generation of a generational garbage collector hold similar portions of the heap. Objects are certainly allocated in the cache; short-lived objects will also die in the cache. With some hardware support, we can build a young generation garbage collector which operates entirely within the object caches, with almost no external memory references required: *in-cache garbage collection*. This gives fast, predictable collections without global memory traffic or synchronization and saves the writing back of garbage objects to memory.

3.1 Garbage collection boundaries

For the purposes of in-cache garbage collection we group the CPUs and their associated caches within one or more *GC boundaries*. A GC boundary delimits a group of CPUs on which mutation will be paused and the garbage collector will run, and the caches within which collection will take place; we expect fast communication within the GC boundary. In the case of a single-chip multiprocessor (CMP) with several on-chip L1s and a shared L2 the natural GC boundary coincides with

the chip boundary and contains all the on-chip CPUs and caches (Fig. 6). A multiprocessor built from several such CMPs would have one GC region per chip. For concreteness we will assume this organization, but others are possible.

3.2 Local objects and the GC barrier

An object is eligible for in-cache collection if it was created inside the GC boundary and a reference to it has never left the boundary, i.e., all references to the object are known to remain in the on-chip caches [34]. The liveness of such a *local* object can be determined without global memory operations by examining the contents of the caches; no thread executing on a remote CPU can get hold of a reference without communication through shared memory. An object is said to be *non-local* if this is no longer true: a reference to it has left the cache. Non-local objects are assumed to be live; any non-local objects in the caches are roots for the in-cache collection.

We track the local state of objects with one extra state bit per cache line. The *non-local bit* of an object's header cache line (the cache line containing offset zero) is set when a reference to that object leaves the GC boundary. To maintain this barrier a piece of hardware, which we call the *warden*, examines outgoing cache lines to find the references. In a *broadcast-nonlocal* request, the warden broadcasts the referenced OIDs to caches within the boundary, rather like a snoop-invalidation; a tag match on the header cache line will set the non-local bit. The warden processes references from other objects (object cache lines), and from VM data structures and stacks in physically-tagged cache lines. The warden also sets the non-local bit of any incoming cache lines: we cannot track references when an object is outside the cache, so objects which have left the cache are conservatively marked non-local.

The set of local objects roughly corresponds to the youngest generation of a generational garbage collector [15]. An object is promoted out of the in-cache generation when a reference to it escapes the GC boundary, and the warden maintains the associated GC barrier.

3.3 Locating references and roots

References inside objects can be identified using knowledge of the bifurcated object layout. References in physical memory (i.e., on the stack or in other VM data structures and hence roots for the in-cache collection) are indistinguishable from integers, and are conventionally located using stack maps, custom data structure parsers, etc., but having the warden process these complex data structures is undesirable. A more general solution involves tagged memory: an extra bit added to each word in the system distinguishes references from integers.

We can get the benefits of tagged memory for in-cache collection but without requiring 33- or 65-bit memory throughout the system using *modified reference* bits only within the caches; the cost is one extra bit per word of cache. This works because the warden and in-cache collector only need to find references to local objects; a cache line outside the GC boundary cannot contain any local references because of the action of the warden. A store-reference instruction sets the modified-reference bit, and a store-integer instruction clears it, whether in object or non-object memory. The warden issues broadcast-nonlocal requests for outgoing words with set modified-reference bits. Similarly, the in-cache collector can find roots outside objects by looking for modified references within the GC boundary.

3.4 The in-cache collection algorithm

The in-cache garbage collector (ICGC) is a simple mark and sweep garbage collector but with the hardware introduced above it is both parallel, using all the CPUs within the GC boundary during a collection, and concurrent, in that mutation may continue on other CPUs outside the GC boundary. The GC uses two extra bits per cache line (Fig. 8), labeling them as the *mark* bit and the *done* bit. The mark bit of the object header line designates a live object and the done bit indicates an object line that has been scanned.

Each CPU within the boundary iterates over any caches private to it and a portion of any shared cache. Every cache line that contains a piece of an object is scanned if that object's header has either

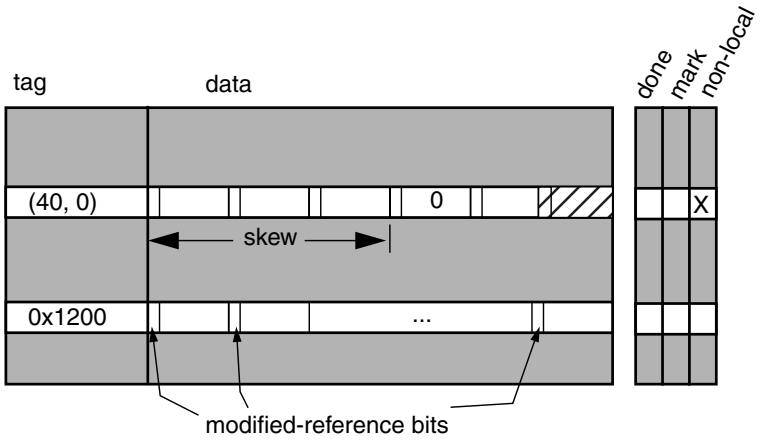


Fig. 8: Extra GC state maintained in an object cache

of the mark or non-local bits set. Any modified references within the object piece are then *broadcast-marked*; this is the same operation triggered by the warden except that it sets the mark bit instead of the non-local bit and is exposed to software control. During the GC phase, any broadcast which hits the unmarked header of an object places the OID on a queue, associated with that cache, for newly marked objects. The responsible CPU then pops the OID from this mark queue and scans that object if it resides in the cache. The queue is an optimization: the alternative is to rescan (parts of) the cache to find the newly marked lines. In the simulations described later we assume an unlimited queue size; a more detailed discussion of queue overflow is beyond the scope of this report.

Roots for the in-cache collection are non-local objects, object fragments (parts of objects whose header is outside the GC boundary), and modified references contained in physically-tagged cache lines. With the modified reference bit we do not need to use stack maps or other more complex techniques for locating references on the stack or locations outside the heap. A single pass through the cache locates all the roots; recursive reachability is handled through the broadcast mechanism.

We expose the contents of the cache for inspection by the collector: a mechanism is provided to iterate over the cache sets returning the tag and state (non-local, marked, etc.). Based on this state the GC issues ordinary loads and stores to read the contents of those cache lines where necessary.

The mark phase ends when all nonlocal and marked objects have been scanned. We can then reclaim the OIDs and cache lines of any objects that do not have either of these bits set. We reclaim the cache lines by simply invalidating their contents during another pass through the cache. Invalidation prevents those contents, which are no longer valid, from being written back to main memory, frees space for new in-cache allocations, and saves bus traffic.

This algorithm is parallel; every CPU within the boundary can be scanning objects simultaneously. GC threads communicate through the mark queues described earlier. A final synchronization phase ensures that all processes are ready to move onto the sweep phase.

The algorithm is also concurrent in that mutation outside the GC boundary can continue without any explicit synchronization with the GC threads. Threads running outside the GC boundary can continue to use objects and even request and obtain objects from within the GC boundary during a collection. The warden handles any objects leaving the GC boundary during a collection just as it would during normal operation, thus ensuring that any escaping reference is accounted for. Additionally, mutator threads that were suspended within the GC boundary can be rescheduled to other CPUs. As each thread accesses objects and stacks it will depopulate the GC boundary being

Table 1: Summary of ISA extensions

Feature	Function
Store reference at virtual address	Allows references to be located within the stack or non-object data structures
Load/store primitive/reference at object address (OID, offset)	Basis of object-addressed memory
Inspect cache tag & state within a particular cache set (for caches within the GC boundary)	In-cache garbage collector scans caches to find root references
Modify cache GC state	Mark cache lines Done for ICGC
Broadcast non-local or GC mark bits	Root finding and recursion within the in-cache GC
Pop OID from mark queue (optimization)	GC recursion from objects found by other CPUs
Zero and allocate object cache line	In-cache object allocation
Invalidate object cache line	In-cache garbage collection

collected, but all the objects leaving will be live – dead objects will be left behind for collection.

3.5 Limitations of in-cache GC

In-cache GC has some limitations not suffered by conventional collectors. The most fundamental is that the size of the collectable region is fixed in hardware. A software young generation may be increased in size to fit the specific properties of the application and improve the efficiency of GC. It is always possible to choose how frequently to run the in-cache GC, or even disable it completely without penalizing mutator performance, but the hardware implementation cost has already been paid.

3.6 Summary

In-cache garbage collection operates on newly-allocated objects with no cache misses and no software barrier overhead. Although mutation is suspended in some partition of the machine the collection is concurrent with other partitions, and collection proceeds in parallel on all CPUs within the partition. Table 1 summarizes the hardware and ISA extensions introduced.

4. Related work

The MUSHROOM project introduced object caches [29] and in-cache GC [34], originating some of the ideas described here. However, MUSHROOM was strictly a single-processor design, and cache filling, evictions and barrier processing (similar to our warden and translator) were managed in software by trap handlers with no concurrency required. Memory was tagged throughout, as a pure Smalltalk system, and mixed object/conventional caches were not needed.

Object caches with delayed translation are closely related to virtually-addressed caches, for example the Berkeley SPUR [11]. The object case is actually easier; we do not have problems with address aliasing or reverse (physical to virtual) translations.

In-cache allocation and the tagging of garbage was proposed in [21]; the authors argued that garbage should be detected whilst still in the cache, but did not couple GC with the contents of the cache as we do.

Various hardware GC schemes have been proposed [3, 4, 6, 19]; our scheme provides hardware assistance for a software-based collector.

Our system shares motivation with the co-designed virtual machines of [23] but the specific memory and GC organization is very different.

Various studies of the interaction of GC and conventional caches have been published [7, 16, 30]. None has related the hardware-level activity back to the high-level activity as we do in §6.

Our cache inspection and manipulation operations are similar in spirit to those described in [12].

Azul Systems has developed a CMP-based architecture for Java which includes hardware GC barriers atop a conventional memory hierarchy; at the time of writing there is no published description.

Unrelated work

Protection in Java is managed completely at the virtual machine level. References in our system are unprotected by the ISA, and the CPU has no special knowledge of objects other than the store-reference instructions. We do not require hardware protection or controlled sharing based on access control [31] or capabilities [17, 28]. The CPU core is minimally changed; it does not process objects,

object layout or object-related behavior other than the load- and store-to-object-space instructions. Unlike the Intel 432 [14], we do not support a full language-level object model in hardware.

5. Simulation methodology

In the next section we present a preliminary evaluation of the proposed architecture. To evaluate the architecture we have constructed a simulation driven by traces obtained from a modified JVM.

Our proposed architecture incorporates new instructions (e.g., loads and stores to object space) and hence we cannot take instruction-level traces from an existing JVM. It would have been possible to modify an existing JVM to emit these instructions, but then we would also have needed a matching instruction-level simulator and an implementation of the new GC algorithm within the JVM. For a preliminary evaluation we decided to adopt a different strategy. Rather than building a complete GC implementation we chose to construct a simulation of both the relevant hardware structures and the proposed GC algorithm. This allows us to finesse many of the details of the implementation and gives a much more malleable framework for experimentation.

5.1 Trace generation and simulation

The simulator is driven by traces of Java applications running on a modified JVM, called the Tracing VM (TVM) [32]. The TVM emits object-level traces of application behavior, including all object creations, accesses to objects, static variables and threads' stacks (as defined in the JVM spec [18]), the loading and resolution of classes, etc. The trace is a complete description of the execution of the program in terms of its effects on the objects, stacks and static data, but is mostly independent of the actual implementation within the TVM. For example, each object named in the trace is identified by a unique ID unrelated to the memory address of the object within the TVM. This allows us to simulate diverse memory management strategies.

With this approach to simulation we are able to attribute hardware level events (such as cache misses) to their high-level causes (application behavior, GC activity, etc.) with relative ease. The hardware-level measurements summarized in the next section are broken out by high-level activity; we believe this is the first time this has been done.

In addition to a simulator of our proposed architecture and JVM, we also constructed a reference simulation of a conventional architecture with a generational heap and a scavenger for the new generation, as used in many production JVMs. This simulation resides in the same simulation framework, and is driven by the same traces, enabling a direct comparison of the two systems.

One drawback of this simulation framework is that we have insufficient information to derive execution time estimates for a plausible implementation. Our simulator models only the memory system activity induced by the virtual machine in manipulating data; it has no notion of instructions, pipelines or latencies and hence we can only compare memory system effects (such as cache misses). However, memory system activity is a major component of the time spent executing most Java applications, and, since our work is focused on this area alone, we believe it is reasonable to limit the study to this area for now. In §8 we will describe some possible next steps.

A consequence of our simplified simulation framework is that it does not model a variety of effects which are important in determining real-world performance but which we believe are neutral in our comparison (or, if anything, penalize our proposed architecture):

1. The TVM traces describe execution in terms of simple Java bytecode interpretation: e.g., an “add” bytecode results in two pops and a push to a thread's stack, whereas a production JVM would use a dynamic compiler to optimize this into a register-based operation. Hence there are many more memory operations to the stacks in our simulation than would be present in a production system. However, these operations are the same in both simulated models; we have subtracted out stack activity so that other activity can be compared without being polluted by the stack data.
2. Our simulator does not model any instruction-side activity (I-caches, or memory system activity due to fetching instructions) or just-in-time

Table 2: Allocation statistics

Program	Object Bytes $x10^6$	Array Bytes $x10^6$	Static Bytes	Objects $x10^3$	Arrays $x10^3$	Static Objects	Mean object size (bytes)	Mean array size (bytes)	Object Classes	Array Classes
201_compress	0.143	110.3	4,936	6.2	3.2	238	23.1	34,469	215	23
202_jess	145.9	128.9	7,024	5316	2618	390	27.4	49.2	360	20
209_db	50.2	27.7	4,882	3068	146	234	16.4	189.7	210	24
213_javac	100.0	104.3	9,156	4036	2185	412	24.8	47.7	379	33
222_mpegaudio	0.168	0.779	5,824	7.7	4.7	284	21.8	165.7	255	29
227_mtrt	108.7	36.8	5,212	5319	1323	259	20.4	27.8	230	29
228_jack	114.3	101.0	6,044	3888	2954	289	29.4	34.2	264	25
JBB	21.2	82.9	15,484	656.6	884.0	621	28.1	93.7	569	52
Telco app	11.3	35.0	62,520	419	261	2467	27.0	134.1	2352	115
Create	120.0	0.148	3,428	10000	0.7	187	12.0	211.4	163	24

compilation. The instruction sequences in the mutator are shorter in our proposed architecture due to elimination of a software GC barrier, so this effect favors the reference system. The only part of the virtual machine which would be significantly different is the GC code, which tends to be rather small and has good locality; our proposed modifications should make it more cache-friendly.

3. Our simulator does not model memory system activity outside the heap, stacks and static data, i.e., to JVM data structures or native code. Our GC techniques will reduce the memory traffic to JVM data structures (e.g., there are no data structures required to locate references on stacks or within objects). Our modifications are neutral with respect to native code.

5.2 Workloads used

To drive our simulator we took TVM traces from the following programs:

1. The SPEC JVM98 suite, using size 100 runs [24].
2. SPEC JBB2000 [25], using 4 warehouses, a half-size item table (10,000 entries), and an extended measurement interval to counteract the slowdown caused by tracing.
3. A telecommunications soft-switching application, driven by a synthetic workload.
4. Create, a micro-benchmark (described later).

Table 2 contains measurements of the allocation behavior of each trace: the number of allocated bytes and objects in the heap (for arrays and non-array objects) and static areas, average object and array sizes, and the number of classes loaded (object and array classes). Object sizes are uniformly small; array sizes vary more. Table 3 lists the length of each trace (measured by the number of TVM events), and the number of load and store events for objects, arrays, stacks and static areas.

5.3 Simulation Parameters

Our simulated machine (illustrated in Fig. 6 on page 7) consists of two identical CPUs each with 64KB L1 caches (8-way associative with 64 byte cache lines). These CPUs share an 8MB L2 cache (also 8-way associative). The cache replacement policy used is pseudo-LRU. We present results only from this configuration as we believe it representative of possible implementations several years out.

The reference garbage collector used two 16MB semi-spaces with an unlimited tenured generation. Objects were tenured after they survived three collection cycles. Smaller semi-spaces (4MB and 8MB) were also tried; we report results only for 16MB as it performed the best and is also a typical setting for production JVMs. Larger semi-spaces lead to less frequent garbage collections, thus potentially reclaiming more garbage for less work, but lead to a bigger total footprint and longer indi-

Table 3: Trace event statistics

Program	Event Count x 10^9	Object		Array		Stack		Static	
		Loads x10^6	Stores x10^6	Loads x10^6	Stores x10^6	Loads x10^6	Stores x10^6	Loads x10^3	Stores x10^3
201_compress	43.6	2796.5	391.7	450.7	247.1	14998.9	11538.7	33429	2.3
202_jess	6.61	491.1	37.1	103.6	22.9	2125.6	1626.7	2309	46
209_db	13.9	1051.7	18.1	256.8	32.4	4676.3	3458.7	81.6	73
213_javac	6.89	451.0	89.3	70.1	43.0	2244.1	1677.8	4650	82
222_mpegaudio	41.0	2516.9	148.3	1422.4	258.2	13888.4	10820.1	155637	25
227_mtrt	8.67	654.7	53.8	77.7	8.69	2709.9	2122.4	74.5	5.3
228_jack	7.00	348.9	51.4	152.5	24.6	2359.7	1722.1	1612	1454
JBB	4.15	171.4	21.8	66.4	55.6	1349.1	1090.2	14839	1.9
Telco app	1.65	88.1	8.99	31.3	14.9	558.8	427.0	1530	9.1
Create	0.642	0.085	20.0	0.024	0.021	180.5	140.4	10006	246

vidual GC pauses. All threads allocate into the same semi-space, but to prevent false sharing on allocation they claim large (64KB) blocks which are then subdivided into objects.

The object cache's zero-and-allocate operation prevents cache misses when allocating objects. As originally proposed [21], such an operation could also be used in a conventional system such as our reference. We isolate its effect by also simulating our reference system using zero-and-allocate during both object allocation and GC (copying objects into the new semi-space).

The ICGC is initiated more frequently than the reference collector, so that allocations do not fill the cache and begin to evict potentially reclaimable objects. We chose to have the ICGC collect every 4MB of allocation, as that was infrequent enough to allow garbage to accrue, but often enough that the garbage objects were not forced from the cache by new allocations. In general, we found that, for caches of 8MB or more, it worked well to collect after new allocations used roughly half of the cache. As cache sizes increase the ICGC runs less often and less garbage eludes it.

6. Preliminary results

We took traces from the workloads and simulated the reference system (hereafter known as the "conventional system"), with and without zero-and-allocate, and the object cache system. For each simulation we counted cache hits and misses, reads and writes, coherence traffic, and translations. Each event was ascribed to either the collector, the mutator's stack activity (which we do not report, for the reasons enumerated earlier), the mutator's heap activity, and, for the object cache, the extra operations involved in managing the object table ("OID management"). Fig. 9 shows the results at each level of the memory hierarchy, normalized by the number of events in each trace (analogous to time). For each workload the left bar is for the conventional system (C), the middle for the conventional system with zero-and-allocate (Z), and the right for the object system (O). Each bar is further decomposed into work performed during mutation and GC, and in the object system an additional component is due to OID management. We have included GC Boundary (GCB) operations, which represent zero-and-allocate, cache inspection and object management operations, in the chart with L2 hits: our GCB encloses the L2, so we expect GCB operations to have about the same cost as L2 hits. In the case of zero-and-allocate this may be unduly generous, depending on the implementation; we later explore a range of costs. The raw data for each operation are broken out in tables in Appendix B.

Examining the graphs from top to bottom, we see that the object system typically has slightly fewer L1 hits in the mutator. The number of muta-

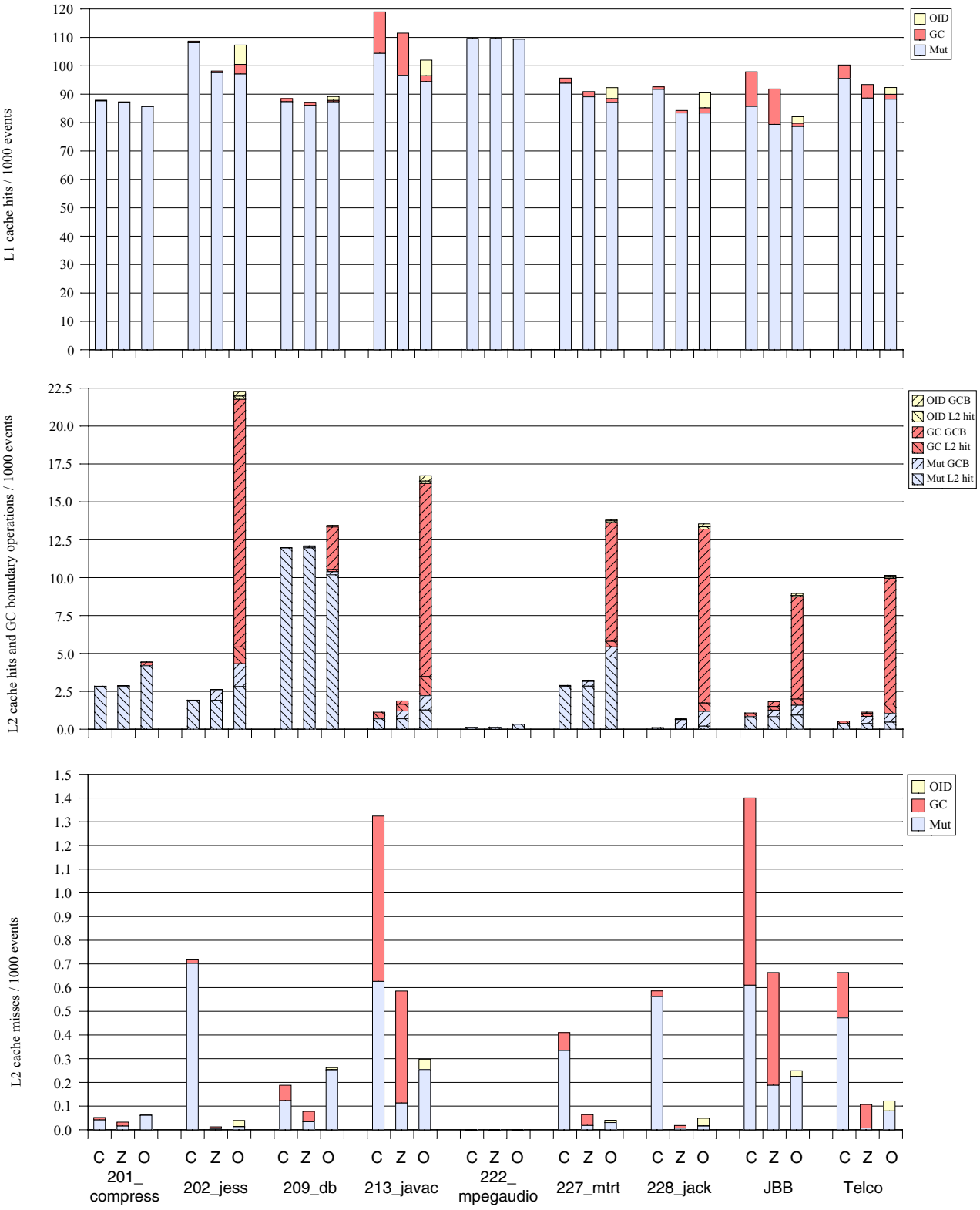


Fig. 9: Cache activity for each workload on conventional (C), conventional with zero-and-allocate (Z), and object (O) caches, normalized to trace events.

Top: L1 cache hits; Middle: L2 cache hits and GC boundary operations; Bottom: L2 cache misses. Each bar is further decomposed by software activity (mutation, garbage collection and OID management).

tor accesses is lower because zero-and-allocate replaces writes on allocation (seen going from C to Z), and also because card table writes are eliminated; the L1 hit ratio is slightly worse in O than Z because of the less dense packing of objects (hence the slightly greater number of mutator L2 hits and, in some cases, misses in the object system). The geometric mean of the ratios of L1 hits in the object system to L1 hits in the conventional system is 0.98.

The ICGC effectively trades more L2 or GCB operations for a reduced number of L2 misses (except for 201_compress and 209_db). The geometric mean of the ratios of object L2 hits and GCB operations to conventional L2 hits is 7.5, while for L2 misses it is 0.24. The in-cache collector takes essentially no L2 cache misses, leading to a dramatic reduction in total L2 misses on the programs where they were significant (213_javac, JBB). Zero-and-allocate also helps the reference system by reducing the L2 mutator (allocation) misses, but the ICGC reduces the total misses even further. As the relative cost of L2 misses increases we expect this to result in a net performance gain for the object system.

During collection most of the activity in the object cache is due to GCB operations (as the collector is doing considerable cache inspection and invalidation work). Note that 222_mpegaudio does not allocate enough objects to trigger a single conventional collection.

6.1 The Create micro-benchmark

The results for Create, our micro-benchmark designed to stress ICGC, are in Table 4 (broken out separately, as the bars would not fit comfortably on Fig. 9). This benchmark creates java.lang.Integers as fast as possible, each becoming garbage immediately. This results in the highest possible object allocation rate. In contrast, a generation scavenger has much less work to do and reclaims the garbage in the least possible time (because it does not visit each reclaimed object, unlike our collector). The mutator in the conventional system is performing more operations than the object mutator because allocation of the three-word objects incurs three zeroing writes and three initializing writes, whereas the object system performs a single zero-and-allocate and three initializing writes. The results show the conventional scavenger doing very little work but the mutator incurring many L2 cache misses. In contrast, our collector does much more work (within the caches), but the mutator incurs many fewer cache misses.

Table 4: Operation counts (per 1000 events) for Create

Operation	Conventional	Object
L1 hit (Mutator)	105.5	47.0
L1 hit (GC)	0.3	64.9
L1 hit (OID management)		73.0
Total L1 hits	**105.8**	**184.8**
L2 hit (Mutator)	0.0	0.0
GCB op (Mutator)		15.6
L2 hit (GC)	0.0	0.1
GCB op (GC)		127.5
L2 hit (OID management)		3.2
GCB op (OID management)		0.3
Total L2 hits and GCB ops	**0.0**	**146.8**
L2 miss (Mutator)	3.9	0.0
L2 miss (GC)	0.0	0.0
L2 miss (OID management)		0.1
Total L2 misses	**3.9**	**0.1**

```
public class Create extends Thread {
  static int n;
  public static void main(String args[]) {
    int nObjs= 10000000;
    int nThreads= 2;
    n= nObjs/nThreads;
    Create workers[]= new Create[nThreads];
    for (int i= 0; i < nThreads; i++)
      workers[i]= new Create();
    for (int i= 0; i < nThreads; i++)
      workers[i].start(); // each thread executes run() method
    // join with workers (not shown)
  }

  public void run() {
    for (int j= 0; j < n; j++)
      new Integer(42); // create and discard java.lang.Integer
  }
}
```

Fig. 10: The Create benchmark

The results for Create can be understood in terms of the operation of its inner loop (Fig. 10). The inner loop consists of 19 bytecodes whose execution in the TVM is captured by 64 events (Fig. 11). All but four of these events are stack

```
1   GetBytecode iload1
2   GetStack I -2(1) => 375
3   PutStack I 2(0), 375
4   GetBytecode getstatic
5   GetStatic I 184(2) => 5000000
6   PutStack I 2(1), 5000000
7   GetBytecode if_icmplt
8   GetStack I 2(0) => 375
9   GetStack I 2(1) => 5000000
10  GetBytecode new
11  NewObject R c=72 => 2712
12  PutField C 2712(-1), 72
13  PutStack R 2(0), 2712
14  GetBytecode dup
15  GetStack R 2(0) =>2712
16  PutStack R 2(1), 2712
17  GetBytecode bipush
18  PutStack I 2(2), 42
19  GetBytecode invoke_special
20  GetStack R 2(1) => 2712
21  PutLocalRoot R 2712
22  PushFrame 2, 2 =>1125
23  GetStack R 2(1) => 2712
24  PutStack R -1125(0), 2712
25  GetStack I 2(2) => 42
26  PutStack I -1125(1), 42
27  GetBytecode aload0
28  GetStack R -1125(0) => 2712
29  PutStack R 1125(0), 2712
30  GetBytecode invoke_special
31  GetStack R 1125(0) => 2712
32  PutLocalRoot R 2712
33  PushFrame 1, 1 => 1126
34  GetStack R 1125(0) => 2712
35  PutStack R -1126(0), 2712
36  GetBytecode aload0
37  GetStack R -1126(0) => 2712
38  PutStack R 1126(0), 2712
39  GetBytecode invoke_special
40  GetStack R 1126(0) => 2712
41  PutLocalRoot R 2712
42  PushFrame 1, 0 => 1127
43  GetStack R 1126(0) => 2712
44  PutStack R -1127(0), 2712
45  GetBytecode return
46  PopFrame 1127
47  GetBytecode return
48  PopFrame 1126
49  GetBytecode aload0
50  GetStack R -1125(0) => 2712
51  PutStack R 1125(0), 2712
52  GetBytecode iload1
53  GetStack I -1125(1) => 42
54  PutStack I 1125(1), 42
55  GetBytecode putfield
56  GetStack R 1125(0) => 2712
57  GetStack I 1125(1) => 42
58  PutField I 2712(1), 42
59  GetBytecode return
60  PopFrame 1125
61  GetBytecode pop
62  GetBytecode iinc
63  GetStack I -2(1) => 374
64  PutStack I -2(1), 375
```

Fig. 11: Events from the inner loop of Create

manipulation operations or bytecode fetches. The remaining four create an object, set its class and single instance variable, and load a static (the loop bound). In the conventional model, object creation results in four zeroing writes (the object size, which is three words, is rounded up to a double-word size), and two initializing writes. Thus there is a total of six writes and one read. In the object model, object creation results in a single zero-and-allocate and the two initializing writes, for a total of three reads/writes in the mutator. Additionally, there are three reads and two writes required to obtain an OID.

Referring to Table 4, each 1000 events of the inner loop will encompass n=15.625 (i.e., 1000/64) iterations. The conventional system will perform $7n$=109.375 mutator memory operations per 1000 events (measured = 109.4, the sum of L1 hits, L2 hits and L2 misses). Every fourth object creation will most likely result in an L2 miss ($n/4$ = 3.90625 L2 misses predicted per 1000 events, actual = 3.9), the rest of the mutator operations resulting in L1 hits.

For the object system, each iteration results in a mutator GCB operation (zero-and-allocate), hence we would expect 15.625 GCB ops per 1000 events (actual = 15.6). We also expect $3n$ = 46.875 reads/writes in the mutator (actual = 47.0), and $5n$ = 78.125 reads/writes for OID management (actual = 76.3).

The fact that the predicted and observed data match well for this small easily-understood program helps to validate the simulator.

6.2 Cache Line Utilization

As expected, our cache miss rates (in particular in the L1) are slightly worse because we do not pack objects densely in cache lines. The worst benchmark in this respect is 209_db, which uses on average only 34.1% of each cache line (32-bit references, 64-byte cache lines); in this case it interacts badly with the application's reference pattern, explaining the increase in L2 misses for the object system. The mean over all benchmarks is 55.5%. These numbers improve by 10–15% with a move to 64-bit references. Table 5 contains the complete utilization results.

This effect will be more significant for small benchmarks (such as the SPEC JVM98 suite), where even the relatively long-lived data structures may still reside in new space. We expect less inter-object spatial locality in the tenured data of large, long-running applications, where this effect should be less significant (research is continuing in this direction).

We do not believe that the fragmentation in the cache is a serious issue. Clearly, it negatively affects the cache miss rates for some applications. However, our results include the costs of this effect and still show an improvement in overall performance. The cost of fragmentation (fewer objects fit in the cache) is more than compensated for by the improvement in GC efficiency. In §6.4 we further investigate the effect of using cache lines which are significantly larger than objects.

6.3 Detailed Study – 213_javac

Table 9 on page 26 presents a detailed look at the results for one of our benchmark programs, 213_javac. The table presents cache read/write counts and miss ratios for the two L1 caches

Table 5: Object cache line utilization

Benchmark	32-bit references			64-bit references		
	Words allocated	Object cache lines	Object cache line utilization	Words allocated	Object cache lines	Object cache line utilization
201_compress	27,608,523	1,731,333	99.7%	27,632,787	1,731,898	99.7%
202_jess	68,710,003	8,179,072	52.5%	105,270,870	9,911,440	66.4%
209_db	19,486,391	3,575,779	34.1%	31,243,806	3,925,003	49.8%
213_javac	51,038,971	6,739,117	47.3%	68,210,751	7,110,693	60.0%
222_mpegaudio	236,777	22,555	65.6%	267,229	23,348	71.5%
227_mtrt	36,357,334	6,670,505	34.1%	45,754,612	6,742,354	42.4%
228_jack	53,835,899	7,393,164	45.5%	72,050,778	7,682,244	58.6%
JBB	26,042,033	2,291,054	71.0%	29,747,369	2,914,978	63.8%
Telco app	11,855,703	1,141,336	64.9%	14,491,698	1,241,535	73.0%
Mean			55.5%			65.2%

(aggregated) and the L2 cache, as well as the number of translations and other cache statistics. Each metric is broken down by activity (GC vs. mutator vs. OID management). Results for other benchmarks are also in Appendix B.

Overall, the number of L2 cache misses is considerably smaller with the ICGC. The conventional collector makes very poor use of the L2 cache, with a 52% read miss ratio and over 99% write miss ratio (caused by copying objects into the new semi-space or promoting them), while the ICGC read miss ratio is close to zero and there are no write misses at all. The mutator thread in the conventional system sees high write miss rates as well, as it fetches cache lines from memory during object allocation. The use of zero-and-allocate in the conventional system eliminates most of the write misses, and in fact saves 56M allocation writes (28%), but the GC still takes many read misses. The in-cache GC in the object system successfully eliminates all L2 misses during GC, resulting in a total of half the L2 misses of the conventional system with zero-and-allocate, or less than one quarter without.

6.4 Sensitivity studies

The results reported above, for the configuration described in §5.3, explore only a single point in a very large parameter space. We have performed some limited sensitivity studies to determine how our results may be affected by different choices of cache size, cache line size, and collection interval. To save space we do not include the raw data for these experiments.

Collection interval

The object system reclaims more garbage when it collects more frequently; the conventional system, given the tenuring policy in use, reclaims more garbage by collecting less frequently (i.e., by using larger semi-spaces). For the conventional system, our results show a small but consistent improvement in garbage reclaimed as semi-space size is increased from 2 to 4, 8, and then 16MB. On some benchmarks the L2 cache misses rise to a peak with 8MB semi-spaces and then start to improve, although this is not universal. When the semi-spaces do not fit in the cache larger semi-spaces should only improve matters, and the standard tradeoff of throughput versus heap footprint must determine the choice of semi-space size.

In the object system the number of L2 misses is very insensitive to the collection interval. The cost/benefit decision is therefore between reclaimed garbage and the number of in-cache operations. Collecting every 2MB of allocation reclaims about 10–20% more garbage than our default choice of 4MB, at the cost of about twice as many in-cache operations. On the other hand, collecting every 8MB (the size of the L2 cache) results in a similar decrease in reclamation for an approximately 40% reduction in in-cache operations. Given the sensitivity to this parameter, we conclude that our choice of 4MB is fair but that

more research is needed into in-cache GC initiation policies, including adaptive techniques.

Cache size

A smaller, 4MB, L2 cache results in the reclamation of only around 75% of the garbage that is collected in the 8MB object cache. This number could be improved by collecting more frequently, but the high cost was discussed above and the smaller cache will result in more garbage escaping in any case. The cache size should certainly be factored into the GC initiation policy.

Cache line size

The most interesting result is the variation with respect to cache line size. We do not see a penalty, either in cache misses or in garbage reclaimed, for 128-byte instead of 64-byte cache lines in the object system, keeping the cache size constant at 8MB in each case. As might be expected, the short non-array objects suffer a few more cache misses, but overall a decrease in misses for arrays more than compensates. This is further evidence that some fragmentation in the object cache, which is worse with the longer cache lines, is acceptable. Cache lines of 256 bytes and longer start to cause significant increases in cache misses on some benchmarks.

In the conventional system, without the zero-and-allocate instruction, a substantial fraction of misses are due to allocation, and very long cache lines, up to 512 bytes, continue to show improvements in L2 misses. Note that in this case, although the number of misses is reduced, more bytes are in fact being moved around through the memory system.

7. Summary & conclusions

We have described a novel memory architecture for objects which extends a classical system. The architecture is based on a hybrid object–conventional cache, a warden to police the GC barrier, and a translator to map object names to physical addresses using an object table. Objects are addressed using an object ID and offset; the object cache is also indexed and tagged using encoded forms of these. An in-cache garbage collector exploits the architecture to provide cheap, low-pause collections, entirely within the caches in a GC boundary, without requiring explicit synchronization with processing outside the GC boundary. A bifurcated object layout allows exact GC without requiring additional mapping structures. Skewing results in bifurcated objects residing (usually) in the minimal number of cache lines. The architecture enables concurrent relocation of objects and hence concurrent compaction of the heap.

Our results indicate that the performance of the object cache and in-cache garbage collection compares favorably with a classical system, and yet the object cache allows us to perform incremental compaction and relocation. The object cache and in-cache GC reduce cache misses considerably, at the cost of extra operations to manage the cache.

We conclude that object caching is a promising technique for scaling systems to very large heaps. The data for in-cache GC are less conclusive: whether a performance advantage would be seen depends on the costs and latencies in a particular implementation, and our simulations are not intended to model an implementation at that level of accuracy. The conventional semi-space scavenging collector has two advantages: pointer-bumping allocation is cheaper than management of an OID free list, and the time efficiency of GC can be improved by using larger semi-spaces because the scavenging GC does not touch the dead objects. The in-cache GC offers hardware-supported concurrency and reclamation without L2 cache misses, but whether there is an advantage in throughput depends on the ‘‘constant factors’’. Our simulations show a rough parity in performance if the cost of an L2 cache miss is equated to about 10 cache hits.

8. Future work

Many details remain to be completed, and there is much scope for further investigation.

Our results show that the object cache does not adversely penalize small-scale applications in their cache and new-space GC behavior. We have yet to demonstrate the performance and scaling advantages that the object cache should facilitate for large applications; this will require the modeling of long-running applications with large heaps.

Our results show that a large component of GC activity in the object cache is due to OID management. This code is not heavily optimized and it should be possible to reduce these overheads.

We are investigating GC beyond the caches ("old-space GC"), and the contribution the warden and translator can make to this. Unlike other generational schemes, the initiation of in-cache collection affects only performance and not correctness, so we plan to investigate heuristics for when to initiate in-cache GC.

A more complete evaluation to obtain performance estimates requires the implementation of a realistic processor model and the porting and modification of a state-of-the-art JVM to this model.

In addition, we plan to explore the new applications that these additions can enable or simplify in the virtual machine. Some of the applications we have been considering are support for transparent persistent object stores and allowing objects to be relocated within a cluster of servers. Both of these techniques may be enabled at a much lower overhead through the translator hardware.

Acknowledgements

We thank Dave Ungar, Gary Lauterbach, Adam Talcott and Bernd Mathiske for comments on the draft.

References

[1] H.–J. Boehm, A. J. Demers, S. Shenker. *Mostly parallel garbage collection*. Proc. ACM SIGPLAN 1991 Conf. Programming Language Design and Implementation, pp. 157–164.

[2] S. M. Blackburn, P. Cheng, K. S. McKinley. *Myths and Realities: The Performance Impact of Garbage Collection*. Proc. SIGMETRICS–Performance '04, June 2004.

[3] J. M. Chang, W. Srisa-an, C.D. Lo, E. F. Gehringer, *DMMX: Dynamic Memory Management Extensions*, The Journal of Systems and Software, 63(3), September 2002.

[4] T. Chiueh, *An architectural technique for cache-level garbage collection*, Proc. Fifth Conf. Functional Languages and Computer Architecture, pp.520–537, 1991.

[5] G. Czajkowski, L. Daynès, *Multitasking without compromise: a virtual machine evolution*, Proc 16th OOPSLA, pp. 125–138, 2001.

[6] S. Dieckmann, U. Hölzle, *A Case for Using Active Memory to Support Garbage Collection*, Proc. First Workshop on Hardware Support for Objects and Microarchitectures in Java, 1999.

[7] A. Diwan, D. Tarditi, E. Moss, *Memory subsystem performance of programs using copying garbage collection*, Proc 21st ACM Symp. Principles of Programming Languages, pp. 1–14, 1994.

[8] E. Gagnon, L. Hendren, *SableVM: A Research Framework for the Efficient Execution of Java Bytecode*, Proc. JVM '01, 2001.

[9] A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[10] D. S. Hardin, A. P. Mass, M. H. Masters, N. M. Mykris. *An efficient hardware implementation of Java bytecodes, threads and processes for embedded and real-time applications*. In [27], pp. 41–54.

[11] M. Hill et al., *SPUR: A VLSI Multiprocessor Workstation*, IEEE Computer, 19(11), pp. 8–22, Nov. 1986.

[12] M. Horowitz, M. Martonosi, T. C. Mowry, M. D. Smith, *Informing Memory Operations: Memory Performance Feedback Mechanisms and their Applications*, ACM Transactions on Computer Systems, May 1998.

[13] IBM Corp., *Assembler language reference – AIX 5L for POWER-based Systems*, April 2001.

[14] Intel Corp., *iAPX 432 GDP architecture reference manual*, 171860-001, 1981.

[15] R. Jones, R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[16] M. Karlsson, K. E. Moore, E. Hagersten, D. A. Wood, *Memory System Behavior of Java-Based Middleware*, Proc. 9th Intl. Symp. High Performance Computer Architecture, pp. 217–228, 2003

[17] H. Levy. *Capability-based computer systems*. Digital Press, 1984.

[18] T. Lindhom, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[19] K. D. Nilsen, W. J. Schmidt, *A high-performance hardware-assisted real time garbage collection system*, Journal of Programming Languages, 2(1), 1994.

[20] M. O'Connor, M. Tremblay. *picoJava–1: The Java virtual machine in hardware*. IEEE Micro, pp. 45–53, March/April 1997.

[21] C. J. Peng, G. S. Sohi, *Cache memory design considerations to support languages with dynamic heap allocation*, Tech Rep. Univ. Wisconsin CS Dept 860, Jul 1989.

[22] Y. Shuf, M. J. Serrano, M. Gupta, J. P. Singh. *A study of memory behavior of Java workloads*. In [27], pp. 19–39.

[23] J. E. Smith, S. Sastry, T. Heil, T. Bezenek, *Achieving High Performance via Co-Designed Virtual Machines,* International Workshop on Innovative Architecture, 1998.

[24] Standard Performance Evaluation Corporation, *SPEC JVM98*, http://www.spec.org/jvm98/, 1998.

[25] Standard Performance Evaluation Corporation, *SPEC JBB2000 Java Business Benchmark*, http://www.spec.org/jbb2000/, 2000.

[26] D. M. Ungar, *The Design and Evaluation of a High-Performance Smalltalk System*, MIT Press, 1987.

[27] Vijaykrishnan N., M. I. Wolczko, eds. *Java Microarchitectures*, Kluwer, 2002.

[28] M. V. Wilkes, *Hardware Support for Memory Protection: Capability Implementation*, Proc. First Intl. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 107–116, 1982.

[29] I. W. Williams, M. I. Wolczko, *An object-based memory architecture*, Proc. Fourth Intl. Workshop on Persistent Object Systems, Morgan Kaufmann, pp. 114–130, 1990.

[30] P. R. Wilson, M. S. Lam, T. G. Moher, *Caching considerations for generational garbage collection: a case for large and set-associative caches*,

Tech. Rep. Univ. Illinois Chicago EECS-90-5, 1990.

[31] E. Witchel, J. Cates, and K. Asanovic, *Mondrian Memory Protection*, Proc. 10th Intl. Conf. Architectural Support for Programming Languages and Operating Systems, pp. 304–316, October 2002.

[32] M. Wolczko, *Using a Tracing Java Virtual Machine to gather data on the behavior of Java programs*, http://research.sun.com/people/mario/tracing-jvm/, 1999.

[33] M. Wolczko, D. Ungar, *Method and apparatus for optimizing exact garbage collection using a bifurcated data structure*, US Patent 5,900,001.

[34] M. I. Wolczko, I. W. Williams, *Multi-level Garbage Collection in a High-Performance Persistent Object System*, Proc. Sixth Intl. Workshop on Persistent Object Systems, Springer-Verlag, pp. 396–418, 1992.

A. GLOSSARY

Bifurcated object layout A scheme for assigning offsets to an object's fields using both negative and positive offsets. We assign reference fields at negative offsets and primitive fields at positive offsets; the object header is at offset 0. See §2.2.

Card table A common write barrier implementation. The heap is divided into equally-sized 'cards' of, say, 128 bytes each; each card has a corresponding flag in the card table. The mutator sets the flag when it stores a reference into the corresponding card; the collector examines the card table to find the cards containing updated references.

Compaction The relocation of objects to contiguous storage, thereby removing fragmentation. See *Fragmentation*.

Extended physical address (EPA) An address used for cache coherence within the CPU caches and cache coherence interconnect. An EPA may be either the encoded object address of an object cache line or the physical address of a physical cache line. See *Object address encoding*, and §2.2.

Field Contains a single datum within an object. A Java field is statically declared to contain a primitive value or a reference.

Fragmentation Unusable small chunks of free space existing between objects. See *Compaction*.

Garbage collector (GC) A component of the Java virtual machine which reclaims the storage space and/or OIDs of objects which are unreachable by the mutator.

GC barrier An algorithm (with perhaps an associated data structure) performed by the mutator in order to synchronize with the garbage collector.

GC boundary Delimits a collection of CPUs and caches which will cooperate to perform in-cache GC. The GC barrier at the boundary is maintained by a warden. See §3.1.

Header cache line An object cache line containing offset 0 (the object header).

Heap A logical space in which objects are allocated.

In-cache GC A garbage collector which only uses information available inside a GC boundary, and therefore collects only local objects. See §3.

Local object An object to which no reference exists outside its containing GC boundary. Such objects can be collected in-cache because no CPU outside the GC boundary can possess a reference. Our system tracks localness conservatively. See §3.2.

Modified reference A reference stored into an object cache line after the cache line was brought inside the GC boundary. Only modified references may refer to local objects.

Mutator In the language of garbage collection, that part of the system performing the "real work" of a program (as distinct from the garbage collector). So called because it mutates the contents of the heap.

Non-local object An object which is not known to be local.

Object address An OID and offset, identifying a field within an object.

Object address encoding An OID and offset are combined into an EPA using an invertible encoding function. The choice of the function is determined with consideration for the arrangement of objects within the caches. See *Skew*, §2.2 and Fig. 3.

Object cache A cache which can hold object cache lines; our proposed caches can simultaneously contain physical cache lines because both species of cache line are tagged with EPAs. See *Extended physical address (EPA)*.

Object cache line A cache line containing a cached copy of part of an object. Object cache lines are tagged with encoded object addresses containing an object ID and offset.

Object header Information about an object which is stored alongside the object's fields; it represents overhead added by the virtual machine for its internal use and is not directly addressable from Java code. Such information may include the size, class, lock state, GC state, etc., and may be stored in one or several words. Header information common to all objects is stored at offset 0.

Object ID (OID) A location-independent name for an object. That is, an OID refers to an object but does not directly specify where it is stored. See "Object Addressed Caches" in the text. cf. *Pointer*.

Object table A system data structure containing an object table entry for each object. In our proposed system the object table is built recursively out of objects. See *Object table entry (OTE)*, *Recursion*.

Object table entry (OTE) In our proposed system every object has a corresponding OTE; the OTE holds information about the object including its physical address and size.

Offset The (byte) index of a field within an object.

Physical address An address identifying a location in physical memory.

Physical cache line A cache line containing a cached copy of part of physical memory.

Physical memory Storage (typically DRAM) physically present in the machine.

Pointer (as applied to object addressing) The memory address of an object used as a reference, i.e., a reference which indicates where the referenced object is stored. See §2.1 on page 2. cf. *Object ID (OID)*.

Primitive Not a reference to an object. Java's primitive types are byte, char, short, int, long, float and double.

Reachable Property of an object which is in the transitive closure of references from the roots.

Read barrier A GC barrier in which the mutator checks data which it has read to ensure that they are consistent with respect to the garbage collector.

Recursion See *Recursion*.

Reference The canonical name of an object. In a direct representation a reference is a pointer; our system uses in an indirect representation where a reference is an OID.

Root A reference which is not contained in another object; e.g., a reference stored in a CPU register or on a thread's stack. Objects referenced by a root are always reachable.

Single-chip multiprocessor (CMP) Several cooperating processors integrated onto a single silicon chip.

Skew The shifting of an object's fields within its cache lines so that offset 0 (the header) may be somewhere in the middle of a cache line. This allows a bifurcated object layout within a single object cache line. Skewing is implemented as part of the object address encoding. See *Bifurcated object layout*, *Object address encoding*, §2.2, Fig. 4 and Fig. 5.

Translator A hardware structure interposed between the CPU's caches and the physical memory. The translator handles cache misses or evictions for object cache lines, converting the cache line's EPA to a physical address (using the object table entry) and then reading or writing the corresponding physical cache line(s). See *Extended physical address (EPA)*, *Object table entry (OTE)*.

Warden A hardware structure which implements a write barrier for in-cache GC by examining cache lines leaving its GC boundary and marking objects which have become non-local. See §3.2.

Write barrier A GC barrier in which the mutator informs the collector when a relevant modification has been made to the heap.

Zero-and-allocate Cache operation which allocates a cache line (with a specified address) into a cache and initializes the contents to zero.

B. ADDITIONAL RESULTS

Table 6: 201_compress detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	43,610,084,573						
Bytes allocated (collectable)	110,434,092						
Reads $\times 10^6$	6.01	3,280	6.01	3,280	0.080	3,280	0.029
L1 read miss ratio	6.38%	3.46%	6.38%	3.47%	13.9%	5.3%	4.6%
Writes $\times 10^6$	5.90	667	5.90	639	0.0045	639	0.090
L1 write miss ratio	6.22%	1.74%	6.10%	1.55%	10.5%	1.73%	5.79%
L1 explicit tag checks $\times 10^6$					0.12	0.00	0.00
L2 reads $\times 10^6$	0.38	114	0.38	114	0.01	174	0.00
L2 read miss ratio (local/global) %	21.9 / 1.39	0.00 / 0.00	97.0 / 6.19	0.01 / 0.00	0.09 / 0.012	0.41 / 0.02	18.3 / 0.84
L2 writes $\times 10^6$	0.37	11.6	0.36	9.89	0.00	11.1	0.01
L2 write miss ratio (local/global) %	100 / 6.22	15.6 / 0.27	96.9 / 5.91	6.88 / 0.11	0 / 0	17.6 / 0.30	63.3 / 3.7
Combined L2 misses $\times 10^6$	0.45	1.81	0.72	0.69	0.00	2.66	0.004
Zero & allocate $\times 10^6$			0.37	1.73	0.00	0.013	0.010
L2 modify mark $\times 10^6$					2.49	0	0
L2 explicit tag checks $\times 10^6$					7.48	0	0.005
Translations (reads) $\times 10^6$					0.00	2.66	0.003
Translations (evictions) $\times 10^6$					0.00	1.77	0.005

Table 7: 202_jess detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	6,608,547,740						
Bytes allocated (collectable)	274,810,792						
Reads $\times 10^6$	2.53	597	2.53	597	23.1	597	23.8
L1 read miss ratio	7.33%	2.10%	7.33%	2.10%	28.3%	3.10%	4.60%
Writes $\times 10^6$	1.45	135	1.45	60.5	6.71	63.3	22.8
L1 write miss ratio	4.07%	3.46%	0.29%	0.04%	12.5%	0.16%	2.19%
L1 explicit tag checks $\times 10^6$					1.93	0.00	1.13
L2 reads $\times 10^6$	0.19	12.5	0.19	12.5	6.52	18.5	1.10
L2 read miss ratio (local/global) %	32.1 / 2.35	0.18 / 0.00	32.3 / 2.37	0.18 / 0.00	0.00 / 0.00	0.38 / 0.01	0.65 / 0.03
L2 writes $\times 10^6$	0.059	4.66	0.004	0.025	0.84	0.10	0.50
L2 write miss ratio (local/global) %	99.6 / 4.06	3.43 / 4.64	2.33 / 0.01	4.53 / 0.00	0 / 0	16.2 / 0.03	33.4 / 0.73
Combined L2 misses $\times 10^6$	0.12	4.64	0.06	0.024	0.00	0.086	0.17
Zero & allocate $\times 10^6$			0.06	4.64	0.00	9.97	1.01
L2 modify mark $\times 10^6$					17.0	0	0
L2 explicit tag checks $\times 10^6$					58.1	0	0.54
Translations (reads) $\times 10^6$					0.00	0.09	0.16
Translations (evictions) $\times 10^6$					0.01	3.16	0.80

Table 8: 209_db detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	13,908,455,679						
Bytes allocated (collectable)	77,945,564						
Reads $\times 10^6$	9.97	1,309	9.97	1,309	6.56	1,309	9.65
L1 read miss ratio	9.45%	12.7%	9.45%	12.72%	25.3%	11.1%	4.21%
Writes $\times 10^6$	6.62	74.4	6.62	54.7	2.39	51.1	9.92
L1 write miss ratio	4.49%	2.07%	0.17%	0.65%	12.5%	0.70%	2.44%
L1 explicit tag checks $\times 10^6$					0.79	0.00	0.46
L2 reads $\times 10^6$	0.94	166	0.94	166	1.66	145	0.41
L2 read miss ratio (local/global) %	64.4 / 6.08	0.28 / 0.04	64.4 / 6.08	0.28 / 0.04	0.00 / 0.00	2.19 / 0.24	1.91 / 0.08
L2 writes $\times 10^6$	0.30	1.54	0.01	0.35	0.30	0.36	0.24
L2 write miss ratio (local/global) %	100 / 4.49	80.5 / 1.66	2.29 / 0.00	0.71 / 0.01	0 / 0	99.1 / 0.67	50.9 / 1.24
Combined L2 misses $\times 10^6$	0.90	1.71	0.61	0.48	0.00	3.52	0.13
Zero & allocate $\times 10^6$			0.30	1.23	0.00	2.81	0.49
L2 modify mark $\times 10^6$					7.08	0	0
L2 explicit tag checks $\times 10^6$					22.8	0	0.26
Translations (reads) $\times 10^6$					0.00	3.52	0.12
Translations (evictions) $\times 10^6$					0.02	1.31	0.41

Table 9: 213_javac detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	6,886,315,454						
Bytes allocated (collectable)	204,219,612						
Reads $\times 10^6$	70.4	526	70.4	526	19.5	526	18.7
L1 read miss ratio	8.87%	1.01%	8.87%	1.03%	42.9%	1.94%	4.15%
Writes $\times 10^6$	38.1	202	38.1	146	3.50	135	20.9
L1 write miss ratio	4.13%	1.83%	0.06%	0.13%	12.4%	0.21%	3.10%
L1 explicit tag checks $\times 10^6$					1.53	0.00	0.89
L2 reads $\times 10^6$	6.25	5.33	6.25	5.40	8.34	16.6	2.41
L2 read miss ratio (local/global) %	51.8 / 4.59	14.2 / 0.14	52.07 / 4.62	14.2 / 0.14	0.00 / 0.00	16.6 / 0.32	0.10 / 0.65
L2 writes $\times 10^6$	1.57	3.71	0.02	0.19	0.43	0.29	0.65
L2 write miss ratio (local/global) %	99.95 / 4.13	95.97 / 1.76	0.51 / 0.00	9.99 / 0.01	0.00 / 0.00	20.6 / 0.04	43.3 / 1.34
Combined L2 misses $\times 10^6$	4.81	4.31	3.25	0.78	0.00	1.75	0.30
Zero & allocate $\times 10^6$			1.57	3.54	0.00	6.51	1.16
L2 modify mark $\times 10^6$					13.6	0	0
L2 explicit tag checks $\times 10^6$					45.8	0	0.62
Translations (reads) $\times 10^6$					0.00	1.75	0.27
Translations (evictions) $\times 10^6$					0.01	4.41	1.14

Table 10: 222_mpegaudio detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	41,035,818,512						
Bytes allocated (collectable)	947,108						
Reads $\times 10^6$	0	4,095	0	4,095	0	4,095	0.04
L1 read miss ratio	–	0.06%	–	0.06%	–	0.21%	4.74%
Writes $\times 10^6$	0	407	0	407	0	406	0.10
L1 write miss ratio	–	0.72%	–	0.72%	–	1.14%	5.24%
L1 explicit tag checks $\times 10^6$					0.00	0.00	0.00
L2 reads $\times 10^6$	0	2.28	0	2.28	0	8.80	0.00
L2 read miss ratio (local/global) %	– / –	0.00 / 0.00	– / –	0.00 / 0.00	– / –	0.00 / 0.00	0.17 / 0.01
L2 writes $\times 10^6$	0	2.92	0	2.92	0	4.65	0.00
L2 write miss ratio (local/global) %	– / –	0.55 / 0.00	– / –	0.00 / 0.00	– / –	0.08 / 0.00	67.3 / 3.52
Combined L2 misses $\times 10^6$	0	0.02	0	0.00	0	0.00	0.00
Zero & allocate $\times 10^6$			0	0.02	0	0.02	0.01
L2 modify mark $\times 10^6$					0	0	0
L2 explicit tag checks $\times 10^6$					0	0	0.00
Translations (reads) $\times 10^6$					0	0.00	0.00
Translations (evictions) $\times 10^6$					0	0.00	0.01

Table 11: 227_mtrt detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	8,669,286,380						
Bytes allocated (collectable)	145,429,336						
Reads $\times 10^6$	10.7	732	10.7	732	7.55	732	19.9
L1 read miss ratio	7.37%	3.38%	7.37%	3.38%	31.2%	5.67%	4.70%
Writes $\times 10^6$	6.18	109	6.18	64.6	6.59	64.6	15.0
L1 write miss ratio	4.22%	2.58%	0.14%	0.08%	12.5%	0.02%	1.09%
L1 explicit tag checks $\times 10^6$					1.57	0.00	0.95
L2 reads $\times 10^6$	0.79	24.7	0.79	24.7	2.35	41.5	0.94
L2 read miss ratio (local/global) %	49.9 / 3.68	0.62 / 0.02	50.1 / 3.69	0.62 / 0.02	0.00 / 0.00	0.63 / 0.04	0.59 / 0.03
L2 writes $\times 10^6$	0.26	2.80	0.01	0.05	0.82	0.01	0.16
L2 write miss ratio (local/global) %	100 / 4.22	98.0 / 2.53	0.97 / 0.00	4.54 / 0.00	0.00 / 0.00	22.4 / 0.00	49.3 / 0.53
Combined L2 misses $\times 10^6$	0.66	2.90	0.40	0.16	0.00	0.26	0.09
Zero & allocate $\times 10^6$			0.26	2.75	0.00	5.82	0.33
L2 modify mark $\times 10^6$					13.2	0	0
L2 explicit tag checks $\times 10^6$					42.7	0	0.18
Translations (reads) $\times 10^6$					0.00	0.26	0.08
Translations (evictions) $\times 10^6$					0.00	0.79	0.29

Table 12: 228_jack detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	7,003,140,485						
Bytes allocated (collectable)	215,343,596						
Reads $\times 10^6$	4.39	503	4.39	503	10.05	503	20.5
L1 read miss ratio	6.34%	0.11%	6.34%	0.11%	30.1%	0.28%	4.19%
Writes $\times 10^6$	2.16	144	2.16	81.6	6.39	82.4	18.0
L1 write miss ratio	5.48	2.72	0.65%	0.03%	12.4	0.17	2.36
L1 explicit tag checks $\times 10^6$					1.67	0.00	0.98
L2 reads $\times 10^6$	0.29	0.55	0.28	0.54	3.03	1.38	0.86
L2 read miss ratio (local/global) %	15.7 / 0.99	9.0 / 0.01	28.1 / 1.78	8.93 / 0.01	0.00 / 0.00	7.51 / 0.02	0.77 / 0.03
L2 writes $\times 10^6$	0.12	3.91	0.01	0.02	0.79	0.14	0.42
L2 write miss ratio (local/global) %	99.9 / 5.48	99.5 / 2.71	3.56 / 0.02	5.47 / 0.00	0.00 / 0.00	5.03 / 0.01	52.8 / 1.24
Combined L2 misses $\times 10^6$	0.16	3.94	0.08	0.05	0.00	0.11	0.23
Zero & allocate $\times 10^6$			0.19	3.89	0.00	6.85	0.65
L2 modify mark $\times 10^6$					14.8	0	0
L2 explicit tag checks $\times 10^6$					49.0	0	0.35
Translations (reads) $\times 10^6$					0.00	0.11	0.20
Translations (evictions) $\times 10^6$					0.00	1.79	0.67

Table 13: JBB detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	4,155,935,320						
Bytes allocated (collectable)	104,168,132						
Reads $\times 10^6$	30.8	253	30.8	253	5.53	253	4.93
L1 read miss ratio	9.61%	1.48%	9.61%	1.48%	29.0%	1.64%	4.43%
Writes $\times 10^6$	24.2	109	24.2	81.0	0.81	78.5	5.26
L1 write miss ratio	5.51%	2.06%	0.13%	0.63%	12.2%	0.83%	4.58%
L1 explicit tag checks $\times 10^6$					0.47	0.00	0.23
L2 reads $\times 10^6$	2.96	3.75	2.96	2.75	1.60	4.15	0.22
L2 read miss ratio (local/global) %	65.9 / 6.33	20.5 / 0.30	66.8 / 6.42	0.30 / 0.51	0.00 / 0.00	21.7 / 0.36	9.52 / 0.42
L2 writes $\times 10^6$	1.33	2.25	0.03	0.51	0.10	0.65	0.24
L2 write miss ratio (local/global) %	100 / 5.51	78.5 / 1.62	2.27 / 0.00	2.26 / 0.01	0.00 / 0.00	4.36 / 0.04	35.0 / 1.60
Combined L2 misses $\times 10^6$	3.28	2.54	1.98	0.78	0.00	0.93	0.10
Zero & allocate $\times 10^6$			1.33	1.76	0.00	2.69	0.26
L2 modify mark $\times 10^6$					5.11	0	0
L2 explicit tag checks $\times 10^6$					16.5	0	0.14
Translations (reads) $\times 10^6$					0.00	0.93	0.09
Translations (evictions) $\times 10^6$					0.01	1.69	0.27

Table 14: Telco app detailed results

Metric	Conventional (C)		With zero-and-allocate (Z)		Object system (O)		
	GC	Mutator (excl. Stack)	GC	Mutator (excl. Stack)	ICGC	Mutator (excl. Stack)	OID Overhead
Trace events	168,918,959						
Bytes allocated (collectable)	47,422,812						
Reads $\times 10^6$	5.39	125	5.39	125	3.54	125	2.13
L1 read miss ratio	8.35%	0.46%	8.35%	0.46%	28.5%	0.61%	4.36%
Writes $\times 10^6$	3.16	38.2	3.16	25.6	0.40	25.3	2.17
L1 write miss ratio	5.12	2.20	0.95%	0.35%	11.7%	0.64%	5.33%
L1 explicit tag checks $\times 10^6$					0.21	0.01	0.10
L2 reads $\times 10^6$	0.45	0.58	0.45	0.57	1.01	0.76	0.09
L2 read miss ratio (local/global) %	35.9 / 3.00	1.99 / 0.01	37.0 / 3.07	1.85 / 0.01	0.00 / 0.00	4.89 / 0.03	8.21 / 0.36
L2 writes $\times 10^6$	0.16	0.84	0.03	0.09	0.05	0.16	0.12
L2 write miss ratio (local/global) %	99.9 / 5.12	93.5 / 2.06	2.36 / 0.02	2.87 / 0.01	0.00 / 0.00	60.2 / 0.39	54.8 / 2.93
Combined L2 misses $\times 10^6$	0.32	0.80	0.17	0.01	0.00	0.13	0.07
Zero & allocate $\times 10^6$			0.16	0.78	0.00	0.95	0.10
L2 modify mark $\times 10^6$					2.23	0	0
L2 explicit tag checks $\times 10^6$					7.51	0	0.05
Translations (reads) $\times 10^6$					0.00	0.13	0.06
Translations (evictions) $\times 10^6$					0.00	0.50	0.13

About the authors

Greg Wright is a Staff Engineer at Sun Microsystems Laboratories. His research interests include processor and memory system architecture and simulation, virtual machines, and garbage collection. He holds an M.A. from the University of Cambridge and a Ph.D. in Computer Science from the University of Manchester.

Matthew Seidl is a researcher at Sun Microsystems Laboratories. His primary current research involves hardware/software codesign for improving Java performance on a range of systems. His other research interests include application introspection (profiling, monitoring, debugging), Distributed Systems, Garbage Collection, Hardware/Software Codesign, Java, JVM, Mobile Computing, Object Oriented Programming, open source, Programming Languages and Software. He holds a B.S. in Computer Science and Electrical Engineering from the University of California, Berkeley, and an M.S. and Ph.D. in Computer Science from the University of Colorado, Boulder.

Mario Wolczko is a Distinguished Engineer at Sun Microsystems Laboratories. His current projects include computer architectures for object-based systems and performance instrumentation hardware design and usage.

He joined Sun Labs in 1993, and has worked on the Self system, various research and production JVMs, and two SPARC microprocessors.

His research interests include object-oriented programming language design and usage, the implementation of virtual machines (in both hardware and software), memory system design and management (including garbage collection) and the co-design of virtual machines and hardware architectures. He holds B.Sc, M.Sc., and Ph.D. degrees in Computer Science from the University of Manchester.