A Parallel, Incremental and Concurrent GC for Servers

Yoav Ossia Ori Ben-Yitzhak Irit Goft Elliot K. Kolodner
Victor Leikehman Avi Owshanko

IBM Haifa Research Laboratory
Mount Carmel
Haifa 31905, ISRAEL
{yossia,orib,girit,kolodner,lei,avshash}@il.ibm.com

ABSTRACT

Multithreaded applications with multi-gigabyte heaps running on modern servers provide new challenges for garbage collection (GC). The challenges for "server-oriented" GC include: ensuring short pause times on a multi-gigabyte heap, while minimizing throughput penalty, good scaling on multiprocessor hardware, and keeping the number of expensive multi-cycle fence instructions required by weak ordering to a minimum. We designed and implemented a fully parallel, incremental, mostly concurrent collector, which employs several novel techniques to meet these challenges. First, it combines incremental GC, to ensure short pause times, with concurrent low-priority background GC threads, to take advantage of processor idle time. Second, it employs a low-overhead work packet mechanism to enable full parallelism among the incremental and concurrent collecting threads and ensure load balancing. Third, it reduces memory fence instructions by using batching techniques: one fence for each block of small objects allocated, one fence for each group of objects marked, and no fence at all in the write barrier. When compared to the mature well-optimized parallel stop-the-world mark-sweep collector already in the IBM JVM, our collector prototype reduces the maximum pause time from 284 ms to 101 ms, and the average pause time from 266 ms to 66 ms while losing only 10% throughput when running the SPECjbb2000 benchmark on a 256 MB heap on a 4-way 550 MHz Pentium multiprocessor.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Languages, Performance, Algorithms

PLDI'02, June 17-19, 2002, Berlin, Germany.

Keywords

Garbage collection, Java, JVM, concurrent garbage collection, incremental garbage collection, weak ordering.

1. INTRODUCTION

Modern SMP servers with multi-gigabyte heaps provide new challenges for garbage collection (GC). GC techniques originally designed for single processor client machines lead to unacceptable pauses when used on large servers.

There is a growing need for a GC that is especially targeted to large server configurations: 64-bit shared-memory multiprocessors implementing a weak-ordering memory access model running multithreaded applications on a multi-gigabyte heap. Such applications include web application servers, which must provide relatively fast responses to client requests and scale to support thousands of clients.

The requirements for this "server-oriented" GC include: ensuring short pause times on a multi-gigabyte heap, good scaling on multiprocessor hardware, and utilizing idle processor time (if available), while minimizing the throughput penalty.

In this paper we present the design and implementation of a server-oriented GC for the IBM Java Virtual Machine (JVM). This collector is fully *parallel*, *incremental*, and *mostly concurrent*. By fully *parallel*, we mean that all collection work is done simultaneously by multiple threads. By *incremental*, we mean that a small amount of collection work is done on behalf of mutators with each allocation [5]. By *mostly concurrent*, we mean that the heap is collected using the algorithm first published by Boehm *et al* [7][1] and later extended by Printezis and Detlefs [31].

The new collector is based on the existing parallel mark-sweep collector of the IBM JVM [12]. This collector employs sophisticated compaction avoidance techniques [12], so compaction is rarely executed. However, to avoid the long pauses when compaction does occur, we also designed and implemented a parallel incremental compactor [6]. In the future, we expect to combine our collector with a generational collector in a manner similar to Printezis and Detlefs [31]. However, we devote this paper to the mostly concurrent aspect of the collector.

[1]Boehm *et al* called the algorithm "mostly parallel". Unfortunately this name is inconsistent with our usage of the word "parallel", so following both local tradition [15, 16] and other authors [31, 8], we call it "mostly concurrent".

1.1 Our Contribution

The IBM JVM is a mature product, which uses an optimizing JIT compiler to deliver high performance execution of Java programs. Much previous effort has been spent on improvement and fine tuning of the JIT [34, 35], the JVM [4] and its existing GC module [12]. As a result, even the slightest overhead introduced by the new algorithms could degrade overall JVM performance.

We added a parallel mostly concurrent collector to a highly tuned mature parallel stop-the-world mark-sweep collector to achieve shorter and even response times, while sacrificing little throughput. Measurements of our prototype on a 4-way 550 MHz Pentium processor using a 256 MB heap and the default collection parameters show that our maximum pause time for SPECjbb2000 at 8 warehouses is 101 ms, the average pause is 66 ms, and that we lose only 10% throughput with respect to the stop-the-world collector. On average, the application runs at 47% of its normal speed during the concurrent collection phase. We also measured our prototype on larger multiprocessors with good results; however, these machines were not available to us while preparing this paper.

We use two novel techniques to achieve these results. First, we combine incremental and concurrent collection to ensure short pause times and take advantage of idle processing time. Marking work is carefully paced as part of object allocation by the mutator threads, while low-priority collector threads in the background soak up idle processing cycles. Second, in place of mark stacks, we employ a work packet mechanism in order to enable the collector to be fully parallel (i.e., multiple mutator threads and collector threads can do collection work at the same time, when the number of participants can be large and dynamic) and to ensure load balancing.

In addition, we built the collector to run on a multiprocessor with a weak-ordering memory access system. A straightforward implementation would require memory fences on every object allocation, as part of every write barrier and for every object marked by the concurrent collector. As fences are expensive multi-cycle instructions, we designed our collector to use significantly fewer fences: a fence for each block of small objects allocated, no fences at all in the write barriers, and a fence for each block of objects marked.

We provide experimental results and performance analysis of the mostly concurrent collector on a multiprocessor. In the past, many authors have provided all performance measurements [31, 8] (or at least pause time measurements [3]) for concurrent collectors on multiprocessors where there are less mutator threads than processors, so the collector thread gets a dedicated processor. Our results and analysis are for programs where there are more mutator threads (we measured with as many as 2000) than processors. These are the conditions that we expect for realistic server programs running on a multiprocessor.

In particular, our results show that our collector has done an excellent job at reducing the portion of the pause time due to mark, so that a large proportion of the remaining pause time is due to sweep, and that we should implement lazy sweep [7] in order to obtain a significant additional reduction in pause time.

To the best of our knowledge ours is the first implementation of a parallel incremental mostly concurrent mark sweep collector reported in the literature.

1.2 Related Work

The need for short pause times is not new, and many concurrent and incremental techniques have been introduced over the years. We choose Boehm *et al*'s mostly concurrent mark-sweep collector [7] as extended by Printezis and Detlefs [31] as our base, because it is simpler than the other algorithms, easy to parallelize, expected to have lower overhead (e.g., its write barrier is much cheaper), and it can be easily integrated in a system where stacks are scanned conservatively.

The first concurrent collectors were those of Steele [33] and Dijkstra [11] and were based on mark-sweep. Baker [5] introduced the first incremental copying collector, which required a read barrier, but ensured a bound on the garbage collection pauses. Replicating collectors [28, 8, 20] replace the read barrier with a write barrier, but their write barrier must record every object mutation during garbage collection, including those to object cells not holding references. Train [19] achieves incrementality by dividing the heap into independently collectible areas, but it must keep track of inter-area references through a write barrier and it imposes an ordering on the collection of the areas in order to ensure collection of cycles. Fully concurrent on-the-fly mark sweep collectors [14, 13, 16, 15] can provide low pause times, but they are complex to implement, require relatively expensive write barriers, and are not able to move objects. Concurrent [10] and on-the-fly [3, 25] reference counting collectors are also complex to implement. They also require an additional scheme to collect cycles and are not able to move objects.

Generational collectors [37, 26, 27] are also employed to reduce pause times. However, though they exhibit short pauses during collections of the new area, they still require an incremental or concurrent technique to collect the old area.

Additional related work is discussed in the relevant sections.

1.3 Organization

We start with an overview of our parallel, incremental and mostly concurrent collector. Next, we describe in detail our contributions. In Section 3, we discuss our unification of incremental and concurrent GC, and describe our mechanism to ensure short pause times. In Section 4, we present our new work packet mechanism used for load balancing. In Section 5, we show how to reduce the number of fence instructions required on weak ordering hardware. In Section 6, we present our experimental results. Finally, we conclude and suggest future work in Section 7.

2. OVERVIEW OF THE COLLECTOR

Mostly concurrent collection divides tracing work into two phases. During the first (concurrent) phase, objects are marked (traced) concurrently with the mutators either on behalf of mutators or by specialized low-priority background threads. A *card marking write barrier* is used to keep track of objects modified after the concurrent phase has started. These objects have to be retraced later, either during the concurrent or stop-the-world phases. During the second (stop-the-world) phase, mutator threads are stopped, and objects which were not reached during the concurrent phase are traced.

2.1 Concurrent Phase

A mostly concurrent collector releases all memory that was unreachable at the beginning of the concurrent phase. Objects that become unreachable after the initiation of the concurrent phase and before the stop-the-world phase, may or may not be released during the current GC cycle. Memory occupied by such objects, and not released in this collection cycle, is called *floating garbage*, and it is important to keep its amount to a minimum.

We initialize and start a new collection cycle when the amount of free space drops below a certain threshold (see Section 3). The collector employs a mark bit vector, one bit per 8 bytes, in order to keep track of marked objects. During initialization, the card table is cleared, the mark bits are cleared, and the low-priority background GC threads are woken up.

For allocation of small objects, the JVM employs a cache allocation scheme. Each thread obtains its own allocation cache from which it allocates small objects; when its allocation cache empties, a thread obtains a new allocation cache. Incremental marking work occurs when servicing allocation requests for allocation caches or large objects.

After initialization, the first allocation request (per thread) scans the thread's stack for roots, thus avoiding stopping these threads for root scanning. Threads that never allocate are stopped for stack scanning when no other tracing work remains to be done. Each thread stack is scanned once during the concurrent phase. As a policy, we put off scanning thread stacks as much as possible to reduce the amount of floating garbage.

Subsequent allocation requests do some tracing, as described in Section 3. Depending on the availability of idle processor time, low priority background threads also do tracing work. Tracing mutators and the background threads use a work packet mechanism to keep track of marked objects, that require tracing, to distribute work and to ensure load balancing. We describe the work packet mechanism in detail in Section 4.

During the concurrent tracing phase, mutators continue working and may modify already-marked objects. These objects must be retraced because they may now point to objects that were not marked before. The only objects that need to be retraced are the marked objects on the cards marked dirty by the write barrier.

The process of scanning dirty cards and collecting roots for further tracing is called *card cleaning*. It could be argued that all card cleaning should be put off to the stop-the-world phase, since a card cleaned during the concurrent phase may become dirty again. If that happens, work spent on the cleaning of that card might be wasted. Our experience is that dirty cards contain roots for many live objects not found elsewhere, and if all card cleaning is put off to the stop-the-world phase, the concurrent phase terminates prematurely without tracing large numbers of live objects. Moreover, about 10% of the heap may be dirty at the beginning of stop-the-world phase. For a large heap, the scanning of such an amount of memory is time consuming and must be avoided during the stop-the-world phase to guarantee short pause times.

Card cleaning work is distributed among mutators and background threads similarly to tracing work. We decided to clean each card at most once[2], and defer card cleaning as long as there is other tracing work available. The rationale for this heuristic is that we have to clean as many cards as possible during the concurrent phase to guarantee short pause time, but we want to avoid scanning the same cards many times. We also want to clean a card as late as possible, to reduce the risk of it being dirtied again. Using this heuristic, the amount of dirty memory which remains to be scanned during the stop-the-world phase is about 2% for the benchmarks we measured.

When an allocation request cannot be satisfied, or when the concurrent phase is finished (all thread stacks scanned, each card cleaned once, and no marked objects left to trace), the stop-the-world phase begins. Detecting the termination of concurrent tracing is described in Section 4.

2.2 Stop-the-world Phase

The stop-the-world phase is fully parallel. It completes the marking of live objects and sweeps. It starts by stopping all mutator and background marking threads. No safe points are required, neither for scanning the stacks, nor for the correct execution of the write barrier. The collector scans stacks conservatively, and the order of write barrier operations allows stopping at any point[3].

After all threads are stopped, we clean dirty cards as described earlier in this section. We rescan all thread stacks, and complete the marking of live objects. The parallel marker is similar to that of Endo *et al* [17].

Next, the sweep phase reclaims all unused storage employing a straight forward parallelization of bitwise sweep [12]. Bitwise sweep frees memory in time essentially proportional to the number of live objects by finding ranges of unmarked memory in the mark bit vector.

2.3 Incremental Compaction

Considering that we require short pause times, full compaction of a multi-gigabyte heap is not an option, but it is possible [24] to "evacuate" a part of the heap when we stop mutators for the stop-the-world phase. In particular, we choose an area to be evacuated before the start of the concurrent mark phase and keep track of all pointers into the area during marking (both during the concurrent and stop-the-world phases). After sweep we evacuate the objects from the area and fix up the references to the evacuated objects. Further discussion of incremental compaction can be found in [6].

3. CONCURRENT AND INCREMENTAL

There are two approaches to concurrent tracing. One approach, often called *incremental collection*, was introduced by Baker [5] in the context of copying collection. Baker's idea was to link garbage collection work (copying in his case, marking in our case) to allocation, so that a small amount of GC work is done on behalf of mutators with each allocation.

The advantage of the incremental approach is that the tracing (marking) work is distributed between all the mu-

[2]Our recent research shows that adding, when possible, a second card cleaning pass yields a further reduction in pause time, without a noticeable impact on throughput.

[3]The write barrier is activated when a reference is written into an object cell. It first makes the new reference accessible as a root (e.g., from a register), then modifies the referencing cell, and finally marks the card dirty.

tator threads performing allocations, and that the amount of tracing is simple to calculate, using what we call the allocator *tracing rate*, which is the amount of tracing work to be done for each byte of memory allocated. The disadvantage of this approach is that processor idle time, e.g., while waiting for IO to complete, is not utilized.

The other common approach to concurrent tracing is to assign specialized GC background threads to do the work [7, 31, 16, 3]. The amount of tracing done by the background threads depends on external factors, e.g., competition with other programs running on the system, and cannot be easily controlled to achieve performance goals.

We combine the two approaches to obtain the advantages of both. The background threads run at low priority and make whatever progress is possible without burdening the system, while incremental tracing ensures progress.

To achieve the goals of short GC pause times and low GC overhead, the concurrent phase must be started at the proper time and tracing must be done at the proper pace. If concurrent tracing is started too early, it must either run at a slower tracing rate, or terminate before the heap is full. The former will result in accumulated floating garbage, and the latter in more frequent GC cycles. Both behaviors increase GC overhead. If concurrent tracing is started too late or done too slowly, not enough tracing is done before memory is exhausted. The remaining tracing must be done during the stop-the-world phase and short pause times are not achieved. Furthermore, if the tracing rate is too high, the collector may steal too many processing cycles from the mutators, thereby slowing their progress.

In the remainder of this section, we present the formulas used to decide when to start a new GC cycle and to control its progress. We also show how to account for the work of the background threads.

3.1 Kickoff and Progress Formulas

We present two formulas: The "kickoff formula" calculates when to start the concurrent collection, and is calculated once per collection cycle. The "progress formula" calculates how much tracing work to assign to a mutator. It is recalculated at the beginning of each increment of concurrent work, i.e., on allocations of large objects and allocation caches for small objects, thus introducing dynamic adaptation of the tracing rate.

Let K_0 be the desired value for the allocator's tracing rate (typically 5 to 10). Let L be a prediction of the amount of memory to be traced in the concurrent phase. Let M be the prediction of the amount of memory on dirty cards, that also needs to be scanned in the concurrent phase. The "kickoff formula"

$$\frac{L+M}{K_0}$$

provides the threshold for the amount of remaining free memory that triggers a new collection cycle.

To estimate L and M, we collect their actual values in past cycles and compute new values using an exponential smoothing average.

Let T be the total amount of memory traced since the beginning of the concurrent phase (both by mutators and by the concurrent threads). Let F be the current amount of free memory. We assume that T and F are known. The "progress formula" calculates the current tracing rate K:

$$K = \frac{M+L-T}{F}$$

A negative value for K means that the values of M or L were underestimated, and K is then assigned its maximum allowed value K_{max}, which is typically $2K_0$.

3.2 Accounting for Background Tracing

The kickoff formula intentionally does not take into account the work done by the background threads, which depends on external factors such as scheduling policy, workload, etc. Background threads may not trace at all, and the concurrent phase is still expected to finish tracing on time. If background threads do some of the work, tracing should finish on time, but the tracing rates of mutators should be less than K_0. When there is a plenty of idle time, all tracing will be done by background threads without any mutator tracing overhead.

We require a measure of the background threads' tracing speed. Let B be the ratio between the amount of tracing work done by all background threads and the amount of allocation done by all mutators, for a given window of time. We occasionally calculate B, and reevaluate B_{est}, the exponential smoothing average of B. B_{est} is used as a prediction for the "near future" tracing rate of the background threads.

If $K < B_{est}$ then the background threads are successfully taking care of the tracing, and there is no need to do any tracing, otherwise:

$$K = K - B_{est}$$

If $K > K_0$, tracing is behind schedule, due to imperfect load balancing or other implementation specific reasons. K reflects the lag in tracing, but using it "as is" may not be enough to correct the situation, since any future problem would cause K to increase again, until it reaches K_{max}. To eliminate this risk, when $K > K_0$, we increase K using a corrective term C, so that the actual value used is

$$K + (K - K_0)C$$

4. PARALLEL LOAD BALANCING

Parallel stop-the-world collectors require a load balancing mechanism, which is responsible for "fair" sharing of tracing work between participating threads, thus preventing starvation and reducing mark stack overflow. Unlike parallel stop-the-world collectors, a parallel incremental collector does not use a fixed number of tracing threads; incremental tracing is done by mutators during allocation, so the number of simultaneously tracing threads may be as large as the number of mutators. We believe that solutions for parallel stop-the-world tracing are not optimal when used for parallel incremental collection.

We describe a new mechanism, which we call *work packet management*. Its basic data structure is a *work packet*, which contains a small mark stack. Work packet management differs from existing solutions on three key points: (1) it ensures load balancing by separating a collector thread's input from its output and forcing threads to compete for input, (2) its synchronization overhead is low, and (3) it provides an efficient way to determine the tracing state, e.g., overflow, underflow or termination.

In the remainder of this section, we describe these features in more detail, followed by a comparison of work packet management with previous load balancing solutions.

4.1 Separating Input from Output

When a thread starts tracing work, it obtains two work packets from a global shared pool: *pop* operations are done only on the *input* packet, and *push* operations are done only on the *output* packet. There is no swapping of the packets' roles (with one exception noted in Section 4.3). An empty input packet is replaced by a non-empty packet from the global pool. A full output packet is replaced by a (preferably) empty packet from the pool. When a thread completes its increment of tracing work, both packets are returned to the global pool.

Work packets deliver same functionality as traditional mark stacks, but ensure that the volume of marked objects is distributed evenly between all threads that are currently participating in the tracing. This is because they compete for the input work packets in a "fair" manner, so that a thread does not keep the new work that it generates (its output packet) to itself; rather, it puts it in a pool where any thread can obtain it.

In addition to "fair" sharing of tracing work, work packet management allows prefetching of the next object to be traced, because the next object to be traced is always known in advance. This is unlike the traditional mark stacks, where the next object to be traced is not necessarily on the top of the mark stack, so it is not always known.

Finally, separation between input and output helps in resolving weak ordering problems, as described in Section 5.

4.2 Low Synchronization Cost

The global pool consists of several separate sub-pools that are accessed directly by threads performing collection work. Each sub-pool holds packets within its occupancy range. Our implementation uses three sub-pools:

- The *Empty Packet Pool* contains empty packets.
- The *Non-empty Pool* contains packets that are less than 50% full.
- The *Almost Full Pool* contains packets that are at least 50% full, including totally full packets.

The granularity, classification and number of these sub-pools could vary. For example, an implementation of work packets might put totally full work packets in a separate sub-pool.

The sub-pools are implemented as linked lists. The get/put operations on a sub-pool can be done simultaneously (there is no need for mutually exclusive sequences of puts and gets). Multithread safety is achieved by using compare-and-swap on the head of the list[4].

Packets are returned to the proper sub-pool, according to their occupancy. When getting an output packet, the packet is taken from the sub-pool of the lowest possible occupancy range, which contains packets. When getting an input packet, it is taken from the sub-pool of the highest possible occupancy range that still contains work packets[5].

The only synchronization mechanism needed for our load balancing method is of little cost. Having sub-pools further reduces this cost, by reducing the possible contention from competing compare-and-swap operations. The sub-pools also ensure that the tracing threads can easily find the packets with the most suitable capacities.

4.3 Identifying the Parallel Trace State

In this section, we describe our CPU-effective method for detecting the state of the concurrent collection.

Each sub-pool has an associated *packet counter*, which is updated (using compare-and-swap) after each put or get operation. The packet counter is not necessarily accurate at any point of time (for example, when sampled after a put, but before the counter update), but can serve as an upper limit and a rough estimate of the number of packets in the sub-pool.

Tracing work is complete and termination detected when the Empty Pool's packet counter equals the total number of packets. This means that either all packets are empty and not owned by any thread, or that some threads are in the middle of getting an empty packet. However, these threads will not find any objects to trace in any case. For this to be correct, a thread that wishes to replace a packet must first try to get the needed new packet and only then, if successful, return the old packet. This guarantees that an attempt to replace work packets will not create a temporary state that may be mistaken for tracing termination. When starting to trace, a thread must first get an input work packet and only if successful, an output work packet. This ensures that attempts to acquire work packets when no tracing work remains will not prevent termination detection.

A temporary shortage of tracing work occurs when a thread fails to get (or replace) a non-empty input packet, and the termination criteria does not hold. In this case, a thread does other concurrent tracing tasks (e.g., card cleaning, see Section 2.1). If these tasks are not available, the thread may quit the tracing task, i.e., if it is an application thread, or yield and try again, i.e., if it is a background thread.

If a thread fails to obtain a non-full output packet, when trying to mark and push an object, it may try to swap its input and output work packets. If both packets are full, a temporary overflow state has been detected and is treated by marking the object and activating a write barrier for it. (In our implementation, the card holding the newly marked object is marked dirty). As tracing algorithms have been designed to reduce the risk of overflow, this is expected to be rare, and should not add many additional dirty cards.

4.4 Comparison with Other Solutions

Several solutions to the problem of load balancing for parallel stop-the-world tracing have been published. These solutions assign a private (usually very big) mark stack to each thread, and add synchronized sharing mechanisms such as *stealing*, where each tracing thread exposes some of its excessive objects in a separate attached queue, so they may be stolen by other "starved" threads [17]. Another solution is to expose the mark stacks themselves to the other threads [18]. The internal stack access operations (*PushBottom* and *Pop-*

[4]There is a subtle race condition when using compare-and-swap for linked list manipulation (the ABA problem). We add a unique ID to the head of list pointer to avoid this (see [21] page A-48 for details).

[5]There are some subtle conditions in which this order is reversed that are not covered in this paper.

Bottom) require no synchronization. Stealing is done with a synchronized *PopTop* operation, and excessive objects are treated by adding them to overflow queues.

A solution for concurrent parallel tracing was published by Cheng and Blelloch [8]. They used a private stack for each thread and a single shared stack to exchange objects between threads; they reported a load balancing overhead of 14%.

These previously published solutions either create bottlenecks accessing the shared pool of excessive objects [8], or face difficulties in finding the right thread with objects to "steal" [17, 18]. Work packet management provides fast access to a packet of tracing work with minimal synchronization overhead. Furthermore, the solution of sharing work using attached (per mark stack) queues is not efficient with incremental collection, where the number of participating tracing threads may vary. Work packet management was designed for such cases.

Finally, the previously published methods require a complex mechanism for the threads to agree on tracing termination. This is mentioned as the principal synchronization problem [17, 8]. Work packet management detects termination naturally, with little cost.

The main advantage of traditional load balancing techniques is that they are activated only when needed, while work packets management is always activated. However, the extra cost should be negligible because work packet management adds little overhead and the synchronization mechanism is cheap.

An additional difference between the traditional use of mark stacks and the work packet mechanism is that the former performs a depth-first (DFS) traversal of the object graph, while the latter is more breadth-first (BFS) in nature. There are many implications of these differences, such as cache locality, object access contention, etc. One may speculate about the advantages or disadvantages of each of these differences, yet they seem of secondary importance. The major difference that must be mentioned is that BFS graph traversal tends to put more entries (objects) on the stacks, and therefore is more vulnerable to overflow. The work packets mechanism is not fully BFS, since the level of BFS is limited by the capacity of a work packet. Our results show that the amount of memory needed for the work packets is insignificant, relative to the size of the heap (see Section 6.3).

5. WEAK ORDERING ISSUES

Garbage collectors designed for large server configurations must take into account processor architecture design based on a weak ordering memory consistency model, e.g., IBM's PowerPC [9] and Intel's IA-64 [22]. Weak ordering is a part of the wider issue of *Relaxed Consistency* memory models, which are explained in detail in [1].

In a weak ordering memory model, there is no guarantee of the order in which writes, issued on one processor, become visible to other processors. For example, suppose a thread running on processor A stores x_1 to location X, replacing its previous value x_0, and then stores y_1 to location Y. If another thread on processor B first loads Y and then loads X, it may see the new value of Y (y_1), but the old value of X (x_0).

To solve such problems, weak ordering architectures provide memory synchronization operations, also called *fence* operations. A fence operation guarantees that the execution of all preceding store and load operations complete before any subsequent store or load operation.

We use the same example as above, but now processor A issues a fence between the two stores, and processor B issues a fence between the two loads. In this case, if B loads y_1 from Y, it is bound to load x_1 from X. Notice that both fences are needed, since a reordering of memory access by either processor could cause the anomalous behavior described earlier.

These fence operations are expensive multi-cycle instructions. Because a garbage collector is a performance-sensitive component, it is important to avoid the use of fences as much as possible.

There are three primary weak ordering problems for parallel and concurrent collectors: (1) when parallel or concurrent collector threads communicate work between them, e.g., mark stack entries, (2) when a mutator thread allocates and initializes an object, and a collector thread traces it, and (3) when a mutator thread updates an object slot and performs a write barrier, e.g., dirties its card, and a collector thread retraces the object and cleans the barrier indicator, e.g., the card.

Cheng and Blelloch [8] address some of the problems, but in the context of a replicating collector. Hudson and Moss [20] discuss order-critical accesses due to Java volatile variables and monitor locks. They also claim that their collector does not introduce any new weak ordering issues; however, they do not provide solutions for the problems outlined above. Domani *et al* [16] address the problems of reversing the order of store and load operations in the DLG collector [14, 13], but do not address the more general weak ordering problems. Levanoni and Petrank [25] remove many fences from their write barrier by relying on the fact that most objects are small so that for most updates, the dirty bit set by the write barrier and the modified slot reside in the same coherency granule (usually a cache line). A fence operation is not needed in this case. Furthermore, for objects that do require fences, the fence need only be done the first time the object is updated during each collection cycle.

In the remainder of this section we elaborate on the weak ordering problems mentioned above and present our solutions to keep fence overhead low.

5.1 Load balancing

A load balancing mechanism, which is a part of any parallel collector, enables a thread on one processor to access a data structure (e.g., shared mark stack), containing objects to trace, which was updated by a thread on another processor. Synchronization to access the data structure is typically handled through a cheap mechanism such as compare-and-swap (see Section 4). However, there still remains the problem of the ordering of memory accesses, which modify the contents of the data structure.

A simple but expensive solution would be to insert a fence after every store of an object to a shared mark stack. Our work packet solution provides an easy way to reduce the number of fences by performing the fence for groups of objects. In particular, the collector performs a fence before returning an output work packet to a pool. This prevents the stores to the packet from being reordered with respect to the store of the packet pointer inserting the packet in the pool. Notice that the thread that gets a packet from

the pool does not need to perform a fence. This is due to the data dependency between the load of the pointer to the packet and access to its content, which the hardware cannot reorder. A similar batching mechanism was proposed for the write barrier of a replicating collector [2].

5.2 Tracing a Newly Allocated Object

A second weak ordering problem could allow a concurrent tracing thread to begin tracing an object, but see the uninitialized memory that preceded the creation and initialization of the object. Incorrect behavior and a memory access violation could result. We describe the scenario that could produce the problem and the solution below.

Suppose that mutator A, executing on one processor, creates and stores an initial value in object O_2, and then stores a reference to O_2 in object O_1. Further, suppose that the processor reverses the order of the stores. Now a tracing thread B, running on another processor, loads O_1, containing the reference to O_2, and attempts to trace into O_2 before the initial value of O_2 has been stored into memory. Thread B has accessed uninitialized memory in O_2.

A simple but inefficient solution would be to add a fence right after the creation and initialization of every object. Thus, a tracing thread would always see initialized objects. Instead, we employ a batching mechanism as before to reduce the number of fences, so that we execute one fence for each group of objects allocated and one fence for each group of objects marked.

As mentioned in Section 2.1, our collector employs a cache allocation scheme for small objects. This is the natural unit for batching. We also need a bit vector, one bit for each unit of memory, where the size of the unit is a power of 2 and we restrict allocation caches to begin on a unit boundary. Our collector already has such a vector, one bit per 8 bytes, called the *allocation bit* vector for marking the first byte of each allocated object. Thus, this solution does not require any extra space for our collector. (Our collector uses allocation bits during the conservative scan of a mutator stack to determine whether to treat a stack slot as a reference.) All bits for free space are initially zero. For convenience we refer to the allocation bits in the description of our solution.

Here is the solution on the mutator side. The mutator:

1. Allocates and initializes objects from its allocation cache until the cache is full.
2. Performs a fence.
3. Sets the allocation bits for the allocated objects.

The mutator's fence ensures that the stores to allocate and initialize the objects cannot precede the store of the allocation bits.

Here is the concurrent tracer side. A tracer thread:

1. Gets an input work packet to be traced.
2. Tests the allocation bits of all the objects in the work packet and marks as "safe" (in some private data structure) those objects whose allocation bit is set, and marks as "unsafe" those whose bit is not yet set.
3. Performs a fence.
4. Pops objects from the work packet, tracing those objects marked "safe", and deferring the tracing of objects marked "unsafe" to a later time by storing them in another work packet.

The tracer's fence ensures that the tracing of an object cannot precede the load of its allocation bit. Together with the fence on the mutator side, this ensures that the tracer never sees uninitialized objects.

The mechanisms for deferring object tracing may be implemented in many ways. We chose to implement it by adding another work packet sub-pool, the *Deferred Pool*. In this sub-pool we store packets containing deferred objects. Periodically, we return all packets in the Deferred Pool to the other sub-pools, so these objects are given another chance to be traced.

Future revisions to the Java memory model [36] could require a fence operation after every object allocation In that case the solution described in this section would no longer be needed.

5.3 Cleaning Dirty Cards

A third weak ordering problem could allow a tracing thread to clean a card, but miss tracing an object that had been updated by a mutator thread. This could lead to a reachable object that is incorrectly collected. Below we describe the scenario, which could produce the problem, and its solution.

A mutator updates a slot of a marked object O_1 to reference an as yet unmarked object O_2 and then sets the dirty indicator for the card of O_1. Suppose the processor reorders these two stores. Meanwhile, a tracing thread notices that the card is dirty, erases the dirty indicator, rescans the card including O_1 and misses the new reference to O_2 since it has not yet been written to memory. Suppose further, that no further update to an object on the card occurs during this collection cycle. The card will not be rescanned further during the current GC cycle. Object O_2 could remain unmarked and be incorrectly collected.

A simple but inefficient solution would add a fence between the update to the object slot and the setting of its card dirty indicator on the mutator side. On the collector side, a tracing thread would execute a fence just before starting to scan the card. This solves the problem as it ensures that the tracing thread will see the updated object. However, this is too expensive as it requires a fence instruction as part of every write barrier.

Our solution avoids a fence in each write barrier by adding cost to the collector's card cleaning mechanism. It is based on making a copy of the card table and before using it, forcing all mutators to execute a fence. The full algorithm is:

1. Scan the card table, registering all found dirty cards (in some other data structure) and clearing the dirty indicators in the card table for the cards that were registered.
2. Force all mutators to execute a fence, e.g., stop each one individually.
3. Clean the cards that were registered.

The cleaning, which is done in Step 3, is guaranteed to clean object reference slots that were fully written by a mutator before Step 1. If the slot changes later, the mutator will perform another write barrier, dirtying the card again.

6. RESULTS

We compare the performance of our implementation of the parallel, incremental, and mostly-concurrent collector (hereafter CGC) with the existing mature and optimized stop-the-world parallel mark-sweep collector (STW) on an IBM prototype of JDK 1.3.1. We also measure and analyze various aspects of CGC, including the effect of the tracing rate on performance, and the effectiveness and cost of load balancing. Except where otherwise noted, measurements were done on an IBM Netfinity 7000, a server with four 550 MHz Pentium III XeonTM processors and 2 GB of RAM running Windows NT 4.0.

All measurements were run with a fixed heap size. We chose the heap size so the heap would reach 60% occupancy. Bigger heap sizes would have produced fewer collection cycles; thus, showing a lower overhead for CGC. However, we chose this more challenging setup to exercise the collector. There was no compaction in any of the runs.

Except where noted, we ran CGC with a tracing rate of 8.0 (see Section 3), used 1000 work packets (each packet holds up to 493 entries), used 4 low priority background threads, and performed a single pass of concurrent card cleaning.

We used three benchmarks in our measurements: *SPECjbb*, *pBOB*, and *javac*. *SPECjbb2000* [32] is a Java business benchmark inspired by TPC-C. It emulates a 3-tier system, concentrating on the middle tier. SPECjbb is throughput oriented: it measures the amount of work (number of "transactions") done during a given time. On a 4-way multiprocessor, SPECjbb runs at 1 through 8 warehouses (threads). To achieve a 60% heap residency at 8 warehouses we used a 256 MB heap.

pBOB is an internal IBM benchmark on which SPECjbb is based; it is more tunable. We use pBOB in its "autoserver" mode, because it reaches a high level of parallelism, can effectively use a large heap, and can simulate processor idle time by adding think times to its transactions.

javac is part of SPECjvm98 [32]. It is a single-threaded Java compiler. We ran it with a 25 MB heap to achieve 70% heap occupancy. We show the results of javac to provide some insight into the performance of our collector for small applications.

6.1 Comparison with STW Collector

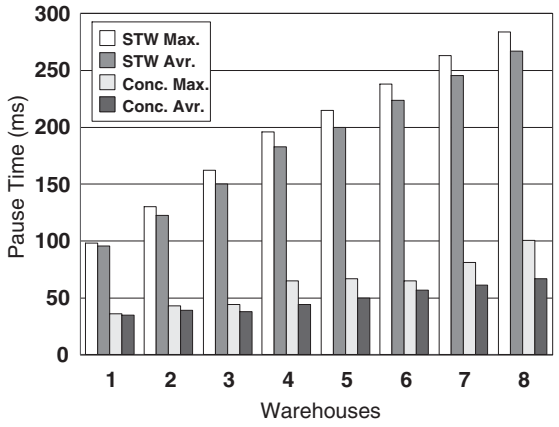

Figure 1: SPECjbb run with tracing rate 8.0

Figure 1 compares running SPECjbb, from 1 to 8 warehouses, using the concurrent collector and the existing STW collector. It shows the average and the maximum pause times of both collectors. We see that there is a significant reduction in both maximum and average pause times. At 8 warehouses CGC cuts the average pause time from 266 ms to 66 ms (a 75% reduction), while losing 14% in throughput. Considering just the mark component of the pause time, CGC cuts the average mark time from 235 ms to 34 ms (an 86% reduction). To put these results in perspective, 11% of the time is spent in GC when using the STW collector at 8 warehouses. Finally, the reduction in the overall SPECjbb throughput score for the concurrent GC is 10%.

The same SPECjbb test was done on a weak ordering memory access system. We used an Intel Itanium server, with four 667 MHz IA-64 processors and 8 GB of RAM running Windows XP 5.1. In order to achieve the same 60% heap residency on this 64 bit machine (where objects are bigger), we used a 320 MB heap. Both the reduction in pause times, and the reduction in the overall SPECjbb throughput score, are similar to those presented in Figure 1 and discussed above.

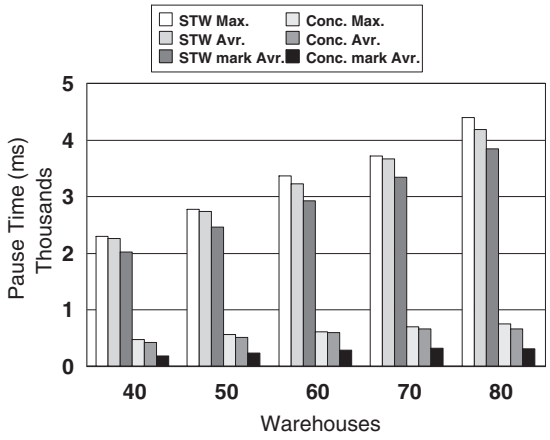

Figure 2: pBOB run with tracing rate 8.0

In order to test our collector on multi-gigabyte heap, we ran pBOB, using a 2.5 GB heap and 3000 work packets, from 30 through 80 warehouses at 25 terminals (threads) per warehouse. We used a pSeries server, with four 600 MHz PowerPC RS64 III processors (64 bit) and 5 GB of RAM running AIX 5.1.

Figure 2 shows average pause times, maximum pause times, and average mark time for pBOB. A measurement of throughput for pBOB (in autoserver mode) is not significant due to the large amount of CPU idle time. We present results starting from 40 warehouses, because few GCs occur prior to that point. At 80 warehouses our test uses 2000 threads, and reaches 85% heap occupancy.

In this test the heap size is much larger, so the reduction in pause time (84%) is even bigger than for SPECjbb. We found that sweep time becomes a dominant part of the remaining pause time; at 80 warehouses the average sweep time is 279 ms or 42% of the total pause time. Thus, implementing lazy sweep [7], i.e., deferring the sweep to after the stop-the-world phase (see Section 7), would reduce the pause time close to that required for mark, and we would

obtain a large additional reduction in pause times. Another interesting result is that the average mark time grows much slower than heap occupancy. Moving from 50 to 80 warehouses the heap occupancy increases by 58% (from 57% to 91%), while the mark times increases by 35% (from 232 ms to 314 ms).

The javac runs were done on a 550 MHz Pentium uniprocessor running Windows NT 4.0 with a single background collector thread. Using an SMP machine, with more background collector threads, seems like an unfair advantage for CGC. CGC achieves a maximum pause time of 41 ms and an average pause time of 34 ms, while the STW collector's pause times were 167 ms and 138 ms. The throughput reduction for CGC was 12%.

6.2 Varying the Tracing Rate

In this section we present results showing how the performance of CGC changes as we vary the tracing rate. We consider performance items such as throughput and pause time, we check the effectiveness of the metering of tracing work, and we measure the processor utilization available for mutator work during the concurrent phase.

Table 1 compares the execution of the STW collector with CGC, while varying the tracing rates for SPECjbb. All results in the table, except throughput, were measured at 8 warehouses. The throughput measurement is the overall SPECjbb throughput score. In the table, TRx denotes a tracing rate of x. At tracing rate 1, CGC attempts to trace a KB of objects for each KB of allocation. Using the formulas from Section 3, at tracing rate 1 CGC will start immediately after the stop-the-world phase is terminated. At the other extreme, tracing rate 10, CGC is activated only when the heap is close to full.

Varying the tracing rate from 1 to 10, we measured the changes in throughput (using the SPECjbb scores), floating garbage (measured by comparing average heap occupancy measured at the end of GC cycles for CGC versus STW), the average number of cards cleaned in the stop-the-world phase of CGC, and the average and maximum pause times.

Measurement	STW	TR 1	TR 4	TR 8	TR 10
Throughput	19904	15511	16984	17970	18177
Floating Garbage	0.0%	18.0%	14.2%	5.3%	4.2%
Average Final Card Cleaning	—	93627	40147	11772	8394
Average Pause Time	267	177	115	67	61
Max Pause Time	284	233	134	101	126

Table 1: The effects of different tracing rates

As expected, a higher tracing rate produces less floating garbage. The number of cards cleaned in the final stop-the-world phase decreases as the tracing rate increases, since concurrent card cleaning starts later, and less new cards stand the chance of becoming dirty while the card table is being scanned. The card size is 512 bytes, so there are 524288 cards in the 256 MB heap. Therefore, at tracing level 8, only 2.2% of the cards need to be cleaned in the stop-the-world phase. The pause times also tend to be shorter, given that there are less cards to clean. The throughput improvement at higher tracing rates is probably due to lower overheads for card cleaning (less cards are dirtied during the concurrent phase) and less floating garbage.

In order to check the effectiveness of the metering of concurrent collection work as the tracing rate varies, we define three criteria:

Card Cleaning Ratio (CC Rate) : the ratio between the number of cards cleaned in the concurrent phase to the stop-the-world phase.

Premature GC Free Space (Free Space): the amount of free space left in the heap, when the concurrent phase manages to complete all its work.

Cards Left (Cards Left): when the concurrent phase does not complete all its work, but is halted due to an allocation failure; this measures how many cards the collector has left to clean. This is a good criteria for how much work is left, since card cleaning occurs last in the concurrent phase.

We believe that the CC Rate should be less than 20% (i.e., our single scan of the card table for concurrent card cleaning should leave little card cleaning to the stop-the-world phase), the Free Space should be less then 5% of the heap size, and Cards Left should be 0.

Criterion	TR 1	TR 4	TR 8	TR 10
CC Rate fails	76%	61%	23%	21%
Free Space fails	26.6%	3.2%	0.4%	0.4%
Cards Left	0%	0%	0%	0%

Table 2: Effectiveness of metering

Table 2 shows the percentage of collections that failed the above criteria, when running SPECjbb with various tracing rates. Free Space failures are a problem only for tracing rate 1. The CC Rate failures are very high at the lower tracing rates, where the concurrent cleaning of dirty cards starts earlier and is spread out over more time. As a result, more cards (in the part already scanned) become dirty before the stop-the-world phase. CC Rate failures are also higher than expected at the higher tracing rates.

Both Tables 1 and 2 indicate that lower tracing rates are currently unattractive and will demand repetitive concurrent card cleaning as in earlier mostly concurrent collectors [7, 31], and further research on reduction of floating garbage (see Section 7).

Another important outcome, of any selection of a tracing rate, is the effect it has on mutator performance. An interesting measurement of *minimum mutator utilization* (MMU) was proposed by Cheng and Blelloch [8], but is very difficult to measure when the number of threads exceeds the number of processors. Instead, we choose to measure the average processor utilization available to mutators while CGC is active (i.e., during the concurrent phase). We calculate the utilization as the ratio of the application allocation rate during the concurrent phase to the application allocation rate during the pre-concurrent phase (the period between the end of the last stop-the-world phase and the next concurrent phase). This is based on the assumption that over long enough periods, the allocation rate changes similarly to the mutator execution rate. This assumption holds for SPECjbb.

Measurement	TR 1	TR 4	TR 8	TR 10
pre-concurrent	—	48.4	48.7	49.1
concurrent	37.9	30.6	23.1	21.1
utilization	78%	63%	47%	43%

Table 3: Mutator utilization

Table 3 presents the concurrent and pre-concurrent allocation rates (in KB/ms), and the mutator utilization[6]. We see that even at tracing rate 8, where we start the concurrent phase quite late in the cycle, the mutators are able to use close to 50% of the processing power while CGC is active.

6.3 Evaluation of Work Packets

In order to evaluate the work packet mechanism, we define load balancing measures and check how the work packet mechanism stands up to them as we increase the number of mutator threads. We also measure the space needed for work packets due to the mostly breadth-first traversal they impose on tracing.

We define three parameters to evaluate load balancing:

tracing factor – the ratio of the amount of tracing actually done by a mutator thread during its tracing increment to the amount it was assigned. The tracing factor shows the amount of starvation (not enough work to do) encountered during the concurrent phase.

fairness – the standard deviation of the tracing factors over a collection cycle. It demonstrates how the load was distributed between the mutators doing increments of tracing.

cost – the number of synchronization operations (compare-and-swap) done for all the get/put operations of work packets over a collection cycle. This number indicates the actual cost of our load balancing. However, it cannot be compared between different numbers of warehouses, since as the number of warehouses increase, the tracing volume, and therefore the work packet usage, increases. Thus, we normalize the cost by the size of the live memory at the end of the collection cycle.

Table 4 presents our load balancing results for pBOB, running without CPU idle time (idle time would decrease contention for work packets and improve load balancing). We used a 1.2 GB heap and 1000 work packets. The measurements were also done without background threads; due to their low priority they might produce irrelevant data. We measured results in finer granularity on the higher number of warehouses, where the interesting data resides. At each warehouses level, we show the average tracing factor (for tracing increments), the average fairness (over all GC cycles), and the average and maximal normalized costs.

We see that the average tracing factor remains stable as the number of threads increases, showing no increase in "starvation". The fairness declines as the number of threads increase at a reasonable rate until 900 threads, when it starts plummeting. The reason is that our prototype has a total of 1000 work packets; since every tracer holds at least two packets (and while replacing, even more), there were simply not enough packets for 1000 threads. The normalized cost

[6]There is no pre-concurrent allocation rate for tracing rate 1 (no pre-concurrent phase). As the pre-concurrent allocation rate stays fairly constant, we used the allocation rate for tracing rate 4.

Warehouses	25	30	34	36	38	40
Threads	625	750	850	900	950	1000
Average tracing factor	.961	.958	.953	.952	.949	.950
Fairness	.038	.039	.045	.049	1.97	2.79
Average Cost	251	280	306	325	341	361
Max Cost	272	294	316	337	353	376

Table 4: The Quality of Load Balancing

increases with the number of threads, but at a moderate level. The reason for its stable behavior, relative to the fairness values, is that in our mechanism, a tracer which fails to get work packets, simply quits the tracing task. Although there is work to be done with respect to adapting the size and number of packets to heap size and number of threads, the interesting finding here is that poor load balancing does not drastically increase the synchronization costs.

In Section 4.4, we discussed the amount of memory needed to manage the work packet mechanism, and predicted that it will need more space than needed for traditional mark stacks. In order to measure this, we instrumented our JVM to include a high-level watermark for the highest number of work packet slots in use at one time. This serves as a lower limit on the amount of memory needed, since this number does not indicate how these slots are distributed over the work packets in use. We added a second high-level watermark on the number of work packets used simultaneously. This serves as an upper limit on the amount of memory needed, since the work packet mechanism always takes an empty packet (if available) for output, yet it could have also worked well with less work packets (by taking non-empty packets for output).

Using this instrumentation, we found that the memory requirements for the work packet mechanism, are bounded between 0.11% and 0.25% of the heap size. We believe that 0.15% of the heap size is a realistic estimation of the needed space for work packets.

7. CONCLUSIONS

We present a parallel, incremental and mostly concurrent garbage collector for modern shared-memory multiprocessor servers. Our collector design aims at supporting highly multithreaded applications, such as web application servers, which must provide relatively fast responses to client requests and scale to support thousands of clients.

We implement a prototype of our collector on a highly mature and tuned parallel stop-the-world mark-sweep base. For pBOB autoserver with 2000 threads, our prototype achieves a large reduction in overall pause time, from 4192 ms to 657 ms, especially for the component due to marking, from 3849 ms to 314 ms, so that a large proportion of the remaining pause time (42%) is due to sweep.

To achieve these results we apply two novel techniques. First, we combine incremental and concurrent GC, so as to both take advantage of processor idle time and ensure short pause times. Second, we introduce a work packet mechanism designed to provide good load balancing for the situation where there are many more mutator threads than processors and the mutators all compete for collector work.

We also show how to reduce the number of expensive memory fence instructions required when implementing parallel and concurrent collectors on weak ordering multiprocessor hardware.

Analysis of our results suggests several directions for future work. First, the pause time results show that we should implement lazy sweep [7]. We can use techniques similar to those used for concurrent tracing to delay sweeping until needed and spread sweeping work between mutator threads and idle low priority background threads. Second, the experiments with lower tracing rates, where the mutators get to utilize more processor time while the concurrent collector is active, show that there is a problem with floating garbage. Floating garbage leads to both unnecessary tracing effort and more frequent GC cycles. We plan to research techniques to reduce floating garbage, so as to reduce the overhead for concurrent collection and enable the collector to be less intrusive in its concurrent phase. Finally, our results show that the work packet mechanism does a good job of load balancing. Given its termination detection scheme, which is much simpler than the schemes of earlier load balancing techniques, we plan to experiment with work packets for parallel collectors.

8. ACKNOWLEDGMENTS

We would like to thank Katherine Barabash, Tamar Domany, Ethan Lewis, Erez Petrank, Shlomit Pinter, and Ronny Sivan from the IBM GC group in Haifa for their advice and support. We thank Bob Dimpsey, Mike Collins, and Kean Kuiper from the IBM Java Performance Group in Austin, for their helpful analysis and ideas. We are particularly grateful to Robert Berry, Sam Borman, and Martin Trotter from the IBM Java Technology Center in Hursley, for their collaboration, support, and advice. We are also grateful to Andy Wharmby and Oliver Dineen from the IBM Java Technology Center in Hursley and Hong Hua from the IBM AIX Performance Group in Austin for their generous help in performing some of the experiments included in this paper.

9. REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Research Report 95/7, Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA, September 1995.

[2] Alain Azagury, Elliot K. Kolodner, and Erez Petrank. A note on the implementation of replication-based garbage collection for multithreaded applications and multiprocessor environments. *Parallel Processing Letters*, 1999.

[3] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI [30].

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 258–268, Montreal, June 1998. ACM Press.

[5] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[6] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *ISMM 2002 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[7] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[8] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In PLDI [30].

[9] F. Corella, J.M. Stone, and C.M. Barton. Specification of the PowerPC shared memory architecture. Technical Report 18638, IBM Thomas J. Watson Research Center, January 1993.

[10] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[11] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[12] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable Jvms. *IBM System Journal*, 39(1):151–174, 2000.

[13] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[14] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[15] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[16] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In ISMM [23].

[17] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[18] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium*

(JVM '01), Monterey, CA, April 2001.

[19] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[20] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying gc without stopping the world. In *ISCOPE Conference on ACM 2001 Java Grande*, pages 48–57, Palo Alto CA USA, 2001. ACM Press.

[21] *Z/Architecture Principles of Operation (SA22-7832-01). Appendix A*, 2000. Available at www.ibm.com.

[22] *IA-64 Application Developer's Architecture Guide*, May 1999. Available at http://developer.intel.com/design/itanium.

[23] *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[24] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *ACM SIGPLAN Notices*, pages 253–263. ACM Press, 1987.

[25] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In OOPSLA [29].

[26] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[27] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[28] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[29] *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[30] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[31] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In ISMM [23].

[32] *Standard Performance Evaluation Corporation*. Available at www.spec.org.

[33] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[34] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 29(1), February 2000.

[35] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java Just-In-Time compiler. In OOPSLA [29], pages 180–194.

[36] *JSRs: Java Specification Requests*, 2001. JSR 133: Java Memory Model and Thread Specification Revision. Available at http://jcp.org/jsr/detail/133.jsp.

[37] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.