A Parallel, Incremental, Mostly Concurrent Garbage Collector for Servers

KATHERINE BARABASH, ORI BEN-YITZHAK, IRIT GOFT, ELLIOT K. KOLODNER, VICTOR LEIKEHMAN, YOAV OSSIA, and AVI OWSHANKO
IBM Haifa Research Lab
EREZ PETRANK
Technion, Israel

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 collector facing these demands building on the mostly concurrent garbage collector proposed by Boehm et al. Our collector incorporates new ideas into the original collector. We make it parallel and incremental; we employ concurrent low-priority background GC threads to take advantage of processor idle time; we propose novel algorithmic improvements to the basic mostly concurrent algorithm improving its efficiency and shortening its pause times; and finally, we use advanced techniques, such as a low-overhead work packet mechanism to enable full parallelism among the incremental and concurrent collecting threads and ensure load balancing.

We compared the new collector to the mature, well-optimized, parallel, stop-the-world mark-sweep collector already in the IBM JVM. When allowed to run aggressively, using 72% of the CPU utilization during a short concurrent phase, our collector prototype reduces the maximum pause time from 161ms to 46ms while only losing 11.5% throughput when running the SPECjbb2000 benchmark on a 600 MB heap on an 8-way PowerPC 1.1 GHz processors. When the collector is limited to a non-intrusive operation using only 29% of the CPU utilization, the maximum pause time obtained is 79ms and the loss in throughput is 15.4%.

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

General Terms: Languages, Performance, Algorithms

Additional Key Words and Phrases: Garbage collection, JVM, concurrent garbage collection.

Author's address: K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and Y. Ossia, IBM Haifa Research Lab, Mount Carmel, Haifa 31905, ISRAEL. Email: {kathy,orib,girit,kolodner,lei,yossia}@il.ibm.com. A. Owshanko and E. Petrank, Computer Science Department, Technion – Israel Institute of Technology. Email: {avshash,erez}@cs.technion.ac.il. Work done in IBM Haifa Research Laboratory.

Preliminary versions of this work have appeared in [Ossia et al. 2002] and [Barabash et al. 2003].

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 can 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 [Baker 1978]. By *mostly concurrent* we mean that the heap is collected using the algorithm first published by Boehm et al Boehm et al. [1991][1] and later extended by Printezis and Detlefs Printezis and Detlefs [2000]. In this collector we have incorporated two novel algorithmic modifications of the mostly concurrent collector that provide substantial improvements in pause times and pause time reductions.

The new collector is based on the existing parallel mark-sweep collector of the IBM JVM [Dimpsey et al. 2000]. This collector employs sophisticated compaction avoidance techniques, so compaction is rarely executed. However, to avoid the long pauses when compaction does occur, we also designed and implemented a parallel incremental compactor [Ben-Yitzhak et al. 2002]. In the future, we expect to combine our collector with a generational collector in a manner similar to Printezis and Detlefs Printezis and Detlefs [2000]. However, we devote this paper to the mostly concurrent aspect of the collector.

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 [Suganuma et al. 2000; Suganuma et al. 2001], the JVM [Bacon et al. 1998] and its existing GC module [Dimpsey et al. 2000]. 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 pa-

[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 [Domani et al. 2000; Domani et al. 2000] and other authors [Printezis and Detlefs 2000; Cheng and Belloch 2001], we call it "mostly concurrent".

rameters 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 have also measured our prototype on larger multiprocessors with good results; however, these machines were not available to us while preparing this paper.

We use several novel techniques to achieve these results. First, we combine incremental and concurrent collection so as to ensure short pause times and also 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. Although these techniques have been used before, the combination that allows the number of collecting threads to change dynamically and work in parallel poses a few challenges on the design of the collector. These include pacing the work, load balancing, and termination detection. Also, some issues in weak ordering have arisen that were not dealt with previously. Second, we modify the mostly original concurrent algorithm with two novel algorithmic modifications that reduce repetitive tracing work and reduce the amount of work during the pause times. Using these two modifications (to be discussed below) we obtained improved performance, reduced pause times, and reduced amount of floating garbage. Next, 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. Finally, 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 many 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 [Printezis and Detlefs 2000; Cheng and Belloch 2001] (or at least pause time measurements [Bacon et al. 2001]) for concurrent collectors on multiprocessors where the number of running threads (mutators and collectors) do not exceed the number of processors. Our results and analysis are for programs where there are more mutator threads (we measure with as many as 2000) than processors, 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 [Boehm et al. 1991] 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.

1.1.1 *Algorithmic improvements.* Our algorithmic contribution is mainly concentrated on two specific algorithmic improvements to the mostly concurrent collector aimed at improving the throughput and heap consumption, without interfering with the other good qualities (such as low pauses and high scalability). The idea is to study the collector algorithm and improve it in two ways. First, we eliminate repetitive collector work as much as possible, and thus improve its efficiency. Second, we reduce the number of dirty cards as much as possible in order to keep the pause times low.

The optimization potential stems from the fact that the collector inherently does repetitive work. The heap is scanned, and then marked objects on dirty cards are re-scanned. In particular, the (mark-bits of the) children are read again to check whether a new unmarked child object has been linked to the marked parent object by the program threads. At first glance, it seems that correctness dictates this amount of repetitive work. However, in this paper we claim that a substantial fraction of this work may be eliminated. The idea is to avoid the initial tracing through dirty cards. If the collector traces through an object on a dirty card, the same object will be traced again when the card gets cleaned. Thus, the first trace can be spared. This simple idea buys a large improvement in the throughput, a substantial reduction of the memory footprint, and a significant reduction in the cache miss rate. However, this idea moves some of the tracing work from the tracing phase to the card cleaning phase, and that may increase the pause times.

An additional simple idea is used to reduce the number of dirty cards and keep the pause times low. We assert that there is no point in marking a card dirty if, at the time of dirtying the card, it contains no traced objects. If the collector has not yet traced through this card, the modification of a pointer by the program does not interfere with trace correctness and dirtying the card is redundant. Indeed adding such a check to the write barrier significantly decreases the number of dirty cards, but also adds a high cost to the write barrier. We choose an implementation which approximates this idea and gets good results. The details are presented in Section 4.

Checking these algorithmic improvements on top of the "naive" mostly concurrent collector, we see a performance improvement of up to 26.7%, a reduction in heap consumption of up to 13.4%, a reduction in the cache miss rate of up to 6.4%, and no substantial change in pause times. The performance improvement of 26.7% is obtained when the collector runs concurrently with the mutator at a low rate so that it interferes minimally with mutator work. If we let the collector use a lot of CPU time and finish the collection very quickly (while hindering program activity), the number of dirty cards and the amount of repetitive work decrease significantly. In this case, our methods improve performance by only 5.4%, on the 6-way pSeries server. We stress that our measurements are not taken on a research system; these significant improvements were measured on IBM's production JVM.

1.2 Server Requirements

It is interesting to ask whether the characteristics of this collector are useful for servers running Java. In particular, are customers willing to pay in throughput in order to shorten the pause times? Furthermore, have we shortened the pause times enough to satisfy these customers? This is not a scientific question, but a question

about the use of such JVMs in the real world. From our experience, customers who want to reduce the pause times are willing to pay in throughput. The new pauses are much more bearable, especially for applications such as telecommunication. Ofcourse, there will be customers who will not care about long pauses and will prefer to get the maximum throughput. Those customers may use the original parallel collector.

1.3 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 [Boehm et al. 1991] as extended by Printezis and Detlefs Printezis and Detlefs [2000] 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.

In parallel to this work, Endo and Taura Endo et al. [2002] have worked out a mostly concurrent collector that does not use compiler support (such as a write barrier) but uses virtual memory primitives. As compiler support is not provided, the algorithmic challenges they faced were very different from ours. They use repeated card cleaning to bound the pause times. Their collector is concurrent and parallel but not incremental and it was implemented for the C programming language. The measurement were run on (typical) C benchmarks that are lightweight compared to the benchmarks we used. Thus, the results are not comparable. However, in their settings, they report excellent results with short pause times and a small hit on the throughput. Although there is no way to directly compare the collectors, we note that we obtain similar results with the SPECjvm98 Java client benchmarks.

The first concurrent collectors were those of Steele Steele [1975; Steele [1976] and Dijkstra et al Dijkstra et al. [1976; Dijkstra et al. [1978] and were based on mark-sweep. Baker Baker [1978] introduced the first incremental copying collector, which required a read barrier, but ensured a bound on the garbage collection pauses. Replicating collectors [Nettles and O'Toole 1993; Cheng and Belloch 2001; Hudson and Moss 2001a] 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. The Train algorithm [Hudson and Moss 1992] 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 [Doligez and Leroy 1993; Doligez and Gonthier 1994; Domani et al. 2000; Domani et al. 2000; Azatchi et al. 2003] can provide low pause times, but they are complex to implement, require relatively expensive write barriers and are not able to move objects. Concurrent [DeTreville 1990] and on-the-fly [Bacon et al. 2001; Levanoni and Petrank 2001] 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 [Ungar 1984; Lieberman and Hewitt 1983; Moon 1984] 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 in order to avoid longer pauses.

Additional related work is considered at the relevant sections.

1.4 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 algorithmic improvement. In Section 5 we present our new work packet mechanism used for load balancing. In Section 6 we show how to reduce the number of fence instructions required on weak ordering hardware. In Sections 7 and 8, we specify our implementation details and report measurement results. Finally, we conclude in Section 9.

2. OVERVIEW OF THE COLLECTOR

2.1 The Basic Mostly Concurrent Collector

Let us briefly review the original mostly concurrent collector as proposed by Boehm et al Boehm et al. [1991]. The Boehm et al collector marks the heap using a separate designated collector thread running concurrently with the program threads. Running the trace concurrently with the program with no cooperation from the program threads does not guarantee that all reachable objects will be traced and it may cause a reclamation of a reachable object. The problem is that when the collector has finished tracing and marking object O, having traced all its children, object O is not traced again. However, the program thread may modify the pointers in O, making O the only parent of a reachable object A, which has not been traced. In this case, object A will not be noticed by the collector and it may be reclaimed. To solve this correctness problem, the mostly concurrent collector requires mutator cooperation in the form of a card-marking write-barrier [Sobalvarro 1988]. Whenever the mutators modify a pointer, they also mark the card on which the pointer resides as dirty. Now, if the collector retraces all pointers of marked objects on all dirty cards, all reachable objects are guaranteed to be properly traced.

Using the above observations, the mostly concurrent collector works as follows. First, roots are marked and dirty bits of all cards are cleared. Then, a tracing phase is executed concurrently. We denote this *the concurrent phase*. While the collector runs, the program threads use the card-marking write-barrier (or page protection traps) to record any pointer modification. When tracing is done, the collector starts a card cleaning phase. During card cleaning, the collector goes through all dirty cards. For each dirty card, the card is marked not dirty and then the collector scans all marked objects on the card. If one of them points to an object that has not been marked, then the unmarked object and all its descendants are traced.

Card cleaning is run once concurrently with the program threads and then a second time while the program is halted. When the program is halted, the cleaning is guaranteed to eliminate all dirty cards and finish the trace. During the second (final) card cleaning phase, the roots are re-scanned. Since the write barrier is not applied to the roots, it must be assumed conservatively that the roots have been modified and they must be scanned as if they were dirty. The second card cleaning phase handles fewer cards (than the first), hence, pause time of the program is short.

We denote the phase in which the program is halted *the stop-the-world phase*. For more details, the reader is referred to the original paper [Boehm et al. 1991].

Floating garbage. The 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* [Jones 1996]. All floating garbage is reclaimed in the next GC cycle. It is important to keep its amount to a minimum, to allow good use of the heap free space, and reduce the number of collections.

We now turn to specific details of our implementation. We elaborate on more substantial issues in subsequent sections.

2.2 Our Concurrent Phase

Initializing a collection. 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 low-priority background GC threads are woken up. These threads will help marking the heap when the program is idle.

Allocation and incremental work. For allocation of small objects, the JVM employs a *cache allocation scheme* – each thread obtains its own allocation block denoted *allocation cache* in which it allocates small objects. When the allocation cache is exhausted and cannot be used for more allocations, the thread obtains a new allocation cache. Incremental marking work occurs when servicing allocation requests for allocation caches or large objects.

Marking the roots. When a collection cycle begins, 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.

Tracing. 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 requiring tracing, to distribute work and to ensure load balancing. We describe the work packet mechanism in detail in Section 5.

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.

Card cleaning. 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.

Concurrent card cleaning work is distributed among mutators and background threads similarly to tracing work. During one card cleaning phase, we decided to clean each card at most once, 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 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 reduced to 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 5.

2.3 The 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[2].

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 Endo et al. [1997].

Then, the sweep phase reclaims all unused storage employing a straightforward parallelization of bitwise sweep [Dimpsey et al. 2000]. 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.

2.4 Incremental Compaction

Considering that we require short pause times, full compaction of a multi-gigabyte heap is not an option, but it is possible (as shown in [Lang and Dupont 1987]) 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

[2]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.

the concurrent and stop-the-world phases). After the 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 [Ben-Yitzhak et al. 2002].

3. CONCURRENT AND INCREMENTAL

There are two approaches to concurrent tracing. One approach, often called *incremental collection*, was introduced by Baker Baker [1978] 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 mutator 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 [Boehm et al. 1991; Printezis and Detlefs 2000; Domani et al. 2000; Levanoni and Petrank 2001]. 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 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, slowing their progress. Let us first discuss the tracing rate and then present the formulas we use to decide when to start a new GC cycle and to control its progress.

3.1 The Tracing Rate

The main tuning parameter of the collector is the *tracing rate*. This parameter signifies how fast the concurrent collector works, relative to the execution of the program. Previously, the mostly concurrent collector was run by letting the collector run on an additional designated collector thread and letitng it compete for CPU time with the program threads. If there is a small number of program threads, the collector runs faster, whereas with a large number of mutators, the collector runs slower.

In our implementation, the collector is incremental, and the tracing rate determines the ratio between collector work and allocation work. Specifically, each mutator, after allocating a new local cache, performs $s \cdot k$ steps of the collector code, where k is a predetermined constant and s is the tracing rate parameter specified by the user. If s is large, the collector runs fast. In this case, when the collector is on, the program runs much slower (than when running without the collector). However, this happens for a short while. The behavior of the collector in this case is closer to a stop-the-world collector. When s is small, the collector runs slowly and the program runs with little interference. However, the collector spreads the work it has to do over a longer period of concurrent execution, thereby giving the mutators a greater share of the CPU. The latter mode is more representative of a concurrent collector as the mutators get high CPU utilization and may work in a non-intrusive manner. In this mode (where s is small), our new algorithmic improvement gives its best performance.

The two different modes are depicted in Figure 1. The y-axis represents sharing of CPU resources and the x-axis represents time. The black area represents the time during which the program is stopped and the CPU is devoted to the final stage of the collection. It is denoted STW (the stop-the-world phase). During the rest of the time period the CPU is utilized by both the collector (the lighter gray color), which is running concurrently with the program, and the Java mutators (the darker gray color). The concurrent collection can be run fast as depicted in the lower picture (higher tracing rates), but it then uses a large share of the CPU utilization. Alternatively, it may run concurrently with the program for a long time, requiring less of the CPU resource and allowing the program to run non-intrusively as depicted in the upper picture (lower tracing rate).

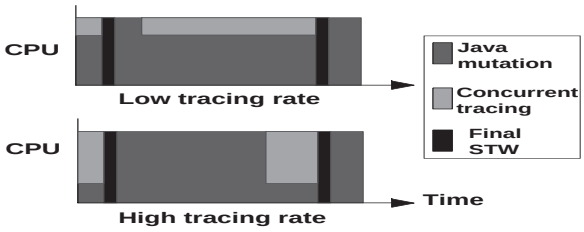

Fig. 1. Characteristics of different tracing rates

Note that our tracing rate specifies the ratio between collection work and *allocation* work. The ratio between collector work and overall program work (which include allocations, but also other computational tasks) is not determined by the tracing rate and depends on how frequent allocations are in the program. For a typical benchmark, such as SPECjbb2000, tracing rates translate to the following behavior: at tracing rate 8, 4, 2 and 1, the collector gets 72%, 58%, 42%, and 29% of the CPU, respectively. Thus, tracing rate 8 may be thought of as running one program thread and three collector threads, and letting the four threads share the CPU time equally. This fraction of CPU given to the collector is quite large, and may not be appropriate for applications. Tracing rate 1, in which we let the

collector use 29% percent of the CPU, is a more reasonable choice for a system. It is for this realistic choice of the tracing rate parameter that our improvement does best. We do not consider tracing rates higher than 8, since they are unlikely to be used in practice.

Next, we present the formulas we use 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.

3.1.1 *Discussion.* How can one choose the correct tracing rate? If one is only interested in pause times, then it seems like a high tracing rate is good. During the short time that the collector is active, not many cards are dirtied and cleaning them incurs a short pause. However, if one chooses a high tracing rate, then during the collection, the mutators get a small percentage of the CPU time. Thus, although there is some activity in the system, it is low. Low tracing rates are less intrusive, but they increase the time during which the collector is active and also the pause times, i.e., the final phases in which the program is halted for the final card cleaning. How small can the tracing rate be? Roughly speaking, the collection should be able to trace all live objects before allocation space is exhausted. (This is not completely accurate as it neglects the additional work of cleaning the cards and the saved work by the background threads, see next subsection.) If the live space is stable at, say, 40% of the heap, then the maximum space that cab be allocated between two collection cycles is 60% of the heap. On the other hand, the amount of tracing required is the amount of live objects in the heap, which is 40% of its space. In this case, the minimum tracing rate is $\frac{40}{60} = \frac{2}{3}$. We usually run the experiments with heap occupancy being at most 60%. With SPECjbb that heap occupancy occurs with the maximum number of warehouses. Thus, towards the end of the run, we expect that tracing rate 1 not to be enough. Our collector automatically increases the pace to finish the collection on time as described in Subsection 3.2 below.

Another interesting issue is the possibility to run additional concurrent card cleaning phases. The mostly concurrent collector may run more concurrent card cleaning phases in order to reduce pause times even further. Our experience has taught us that additional concurrent cleaning is not useful with a low tracing rate. The reason is that with slow collection work, the number of cards that are cleaned in additional concurrent phases equals the number of cards that get dirty during these phases. So the pause times do not decrease much. On the other hand, with high tracing rates additional card cleaning phases may help, but not a lot as the number of cards scanned is small. The percentage of cards scanned during the final stop-the-world phase for tracing rate 8 is as small as 2%. (See Table III in Section 8.)

3.2 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 1 to 8). 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 on the amount of remaining free memory that triggers a new collection cycle.

To estimate L and M, we collect their actual values on 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, so K is assigned its maximum allowed value K_{max}, which is typically $2K_0$.

3.3 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: 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. ALGORITHMIC IMPROVEMENTS

In this section, we present two simple improvements to the mostly concurrent collector increasing the throughput, reducing the heap consumption, and reducing the cache miss rate, without impairing the other good qualities (such as short pauses and high scalability). We start by pointing out repetitive work that seems necessary for the correctness of the collection. However, we are able to eliminate a substantial fraction of the repetitive work while preserving correctness. Next, we study ways to reduce the number of dirty cards as much as possible, in order to keep the pause times low. Here, again, correctness is the main difficulty; however, we are able to reduce a substantial fraction of the dirty cards without foiling the correctness of the collector.

Repetitive work. We start by pointing out the extra work done by the collector to achieve correctness. After executing the tracing phase, the collector moves to the card cleaning phase, in which it goes through all the dirty cards and scans each marked object. The scanning of a marked object on a dirty card consists of reading all references in the scanned object and checking the mark bit of each referent object (each child). If the mark bit of a child is clear, the child object and its descendants must be traced. If the mark bit of the child is set, no further operation is required. Note that when a child is discovered with its mark bit set, the collection does not really gain anything from this discovery. This operation is only required to assure correctness. Furthermore, the scanned references get scanned twice (once during the tracing phase and once again while cleaning the cards). This extra work has a performance cost that we would like to save. We do not see how it may be possible to refrain from scanning the marked objects on all dirty cards, since correctness dictates that we check all modified pointers. However, we suggest an improvement to the tracing phase, which reduces the amount of repeated work. Instead of cutting the work during card cleaning, we cut some of the tracing work so most of the card cleaning work is no longer repetitive.

4.1 Reducing Tracing Work

Recall that the objects that get scanned twice (or more) are marked objects on dirty cards. These objects are first scanned (and marked) by the collector during the tracing phase and then re-scanned during card cleaning, since they reside on dirty cards. Our first idea is to avoid tracing through dirty cards. When the collector traces an object and discovers that this object resides on a dirty card, scanning it is redundant. It is enough to mark the object. Since the card is dirty, we know that this (marked) object will be scanned during the card cleaning phase. This way, we avoid much of the repeated work, although not all of it. Objects that are scanned before a card is made dirty will still go through double scanning. However, we eliminate double scanning for objects that are traced after the card got dirty. Recall that the card cleaning phase also involves tracing operations. The same rule (i.e., not tracing through dirty cards) also applies for the card cleaning phase (for the same reason). It turns out that this method reduces a substantial amount of collector work. Furthermore, it yields a significant reduction in the memory footprint and a significant reduction in the cache miss rate.

4.2 Reducing the Number of Dirty Cards

Having made this improvement with respect to collector efficiency, we now turn our attention to the pause times. The pause times are dominated by the number of dirty cards that need to be cleaned when the program is stopped. Thus, we would like to keep only the minimum number of dirty cards necessary, those which are required for correctness. Let us say a few words on the correctness analysis and then observe what is actually required to make the collector correct. A simple way to claim correctness for the mostly concurrent collector is to claim that all objects that are reachable at the time the program is stopped for the final stop-the-world card cleaning must have been marked by the end of the stop-the-world card cleaning phase. This can be shown by induction on the distance of the objects from the roots. The base follows from the fact that the roots are scanned during the final (stop-the-world) phase of the collection. Now, consider an object A that is reachable at the stop-the-world phase and let B be a reachable parent of A that is closer to the roots (one must exist). By induction, B is marked by the end of the stop-the-world phase. We need to show that the reference of B to A is scanned at some point in the collection. If the reference of B to A existed when B was first traced, then it must have been scanned at that time[3]. Otherwise, this reference has been written to B after it has been traced. Consider the last time this reference was written to B. At that time, the card associated with B was marked dirty with B already being marked. Thus, A must have been traced during the following card cleaning phase.

Note that what is really needed for the above proof is that the program marks a card dirty *if the modified reference has already been traced by the collector*. However, if the modified reference has not yet been traced, then the collector will notice the new child in any case and there is no need to mark the card dirty. This is the main idea behind the second modification.

The straightforward implementation of this idea requires a modification of the write barrier. Instead of simply marking a card dirty upon a modification of a reference on the card, the write barrier first checks if the modified object has been traced (marked). If not, no card marking is required. Such a check may hinder the efficient write barrier. We chose two different implementations of this idea, keeping the write barrier unchanged. These two implementations are described in the following two subsections. One may choose to run one of them or both.

4.2.1 *Undirtying via scanning.* Instead of avoiding marking a card dirty when the modified object has not yet been traced, we do mark the card dirty, but return to check this mark at a later time and clear the mark if possible. To this end, we keep a second card table signifying for each card if it contains an object that has been traced. The collector marks a card in this table as *traced*, before tracing an object on the card. "Once in a while" (e.g., after each m allocations for some parameter m) we run the procedure below. In this procedure, we say that a card is *dirty* if it was marked dirty by the write barrier when a mutator modified a pointer

[3]This claim should be modified when the collector is modified so that it does not trace through dirty cards. However, if B resides on a dirty card when first traced, then it is marked and a later card cleaning phase should spot A.

on the card, and we say that a card is *traced* if it was marked by the collector (in the second card table) when the collector traced an object on the card.

```
for each dirty card C {
    if C is not traced {
        clear dirty mark of C
        check C again
        if C is traced, mark C as dirty
    }
}
```

Note that no synchronization is required in the code above, even though other mutators and collectors may run concurrently with this collector code. We assume, though, that a card dirty bit resides on a separately accessible byte so that it can be written without affecting the marks of other cards. Recall that concurrently with this procedure, the mutator may dirty a card when it modifies a pointer, and the collector may mark a card traced, when starting to trace it. The mutator dirtying is not an issue here if the collector has not traced this card yet. To deal with the collector tracing the card, this procedure repeats the check of the traced bit. We must run the two checks of the traced bit, since a collector thread might be tracing concurrently with this procedure and setting the trace bit of card C, right before the dirty mark of C is cleared. With the additional check, we can be sure that the tracing must have occurred only after we have erased the dirty mark from the card. Without the additional check, a correctness problem may occur if (in addition to the collector tracing into this card) a mutator modifies an object on this card. We remark on platforms that allow instruction reordering in Section 6.4 below. This procedure turned out to be very effective at removing the dirty marks from many dirty cards, and keeping the pause times short.

4.2.2 *Undirtying via allocation caches.* In typical programs, much the pointer modification activity on the heap happens while newly created objects are initialized. This activity creates many dirty cards. Also, most new objects are not traced immediately upon creation. Thus, a great opportunity to undo dirty marks on cards occurs just after new objects are initialized. At the JVM level it is not possible to tell when an object has been initialized. However, in the special case where allocation caches are used, an approximation of this idea is possible. In this section, we elaborate on this special case, explaining what allocation caches are and how they can be used. We also implemented our ideas with allocation caches and provide measurements in Section 8 below.

Allocation caches are used in several modern JVMs [Borman ; White and Garthwaite 1998] as well as in ours. The idea is to reduce synchronization in heap access, by letting each thread hold a local *allocation cache* in which it can allocate small objects. When the program thread has finished allocating in an allocation cache, it requests another allocation cache. We claim that this is an ideal time to try to undo all dirty cards that are contained in the allocation cache. Usually, at this time, all the new objects have been initialized and there is only a small probability that the collector has made its way to these new objects during the short time in which the allocation cache was active.

Furthermore, instead of keeping the additional card table (in which the collector marks traced cards) it is possible to keep a bit for each object to signify whether it is part of an active allocation cache. If this bit is set, then the collector refrains from tracing the object immediately, and it defers the tracing of this object to a later time. A designated list may be used to remember objects whose trace was deferred. When the allocation cache is filled and becomes inactive, the mutator clears the dirty bits for all cards in the allocation cache and then clears also these "defer" bits for all objects contained in the allocation cache. We observed that it seldom happens that the collector actually reaches an object on an active allocation cache, i.e., our designated list for postponed tracing is almost empty. In particular, in our SPECjbb2000 runs, out of millions of traced objects we saw only 18 deferred objects on average in each GC cycle (the maximal number was 287). The guarantee that objects on active allocation caches are not traced ensures that all the dirty cards (that are contained in an allocation cache) may be undirtied when a thread stops using this allocation cache.

We have implemented both these methods and compare their results in Section 8.5.7 below.

4.3 Tricolor Abstraction

We would like to shed more light on the algorithmic modifications by relating them to the well-known tricolor abstraction. In this abstraction all objects are white in the beginning of the collection. An object becomes gray when the (tracing) collector first visits it. It turns black when the collector has visited all of its direct descendants (its children).

A standard tracing procedure starts by coloring objects directly reachable from the roots as gray. Then, it iterates by choosing a gray object, marking it black and marking all its white children gray. Tracing ends when no more gray objects exist. In the implementation of our collector, an object is white if it is not marked and it is not listed in the work packets (the mark stack). An object is gray if it is listed in an entry of a work packet. An object is black if it is marked and it does not appear in the work packets. The use of card marking in the write barrier of the mostly concurrent collector complicates the picture slightly. Black objects residing on dirty cards are considered gray, as their children will be traced again. Namely, the operation of marking a card dirty immediately makes all its gray objects black. Thus black objects are redefined to be marked objects not residing on dirty cards and not listed in a work packet. Gray objects are redefined to be all objects listed in the work packets and also all marked objects on dirty cards. The employed card marking write barrier is an incremental-update, and retreating (i.e. it grays the source of the pointer a la Steele rather than the new target a la Dijkstra).

We first use this abstraction to explain how we can avoid tracing through dirty cards. The goal of the collector is to finish tracing all gray objects making them black and terminating the collection. It thus traces each gray object to make it black. However, when the mostly concurrent collector traces a gray object on a dirty card, it does not make it black. Recall that all marked objects on dirty cards are gray. Thus, it is useless to trace gray objects on dirty cards, and our first algorithmic modification avoids tracing such objects.

Now consider marking a card in which no object has been traced. The purpose of

dirtying this card is to make all black objects on this card gray. However, dirtying a card in which all objects are white is useless and only create additional useless work for the collector. Thus, if a card is dirty and all objects on it are white, we propose to undirty it. Correctness is preserved because there is no object on the card that needs to be traversed a second time.

5. 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. Although this mechanism has not been published before, we have been told that a related method has been previously used in Insignia's Jeode JVM, where it was denoted *grey packets*. See [Thomas et al. 1998] for a related patent application.

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.

5.1 Separating Input from Output

When a thread starts a tracing work, it acquires from a global shared pool an *input* packet that contains a number of objects that need to be traced. From the same global shared pool it also acquires a second work packet, as empty as possible, which serves as an *output* packet. The thread then traces objects in the input packet and it fills the output packet with children of those objects that need to be further traced. Obtaining objects from the packets via *pop* operations is 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 5.3). 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 a traditional mark stack, 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 determined dynamically according to the most recent object scan, and it is not always known early enough to be used for prefetching.

Finally, separation between input and output helps in resolving weak ordering problems, as described in Section 6.

5.2 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 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 packets that are as empty as possible. When getting an input packet, it is taken from the sub-pool of packets that are as full as possible.

The only synchronization mechanism needed for our load balancing method is of little cost. Having sub-pools reduces this cost even more, 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.

5.3 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.

[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 it (see [IBM 2000] page A-48 for details).

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 with 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 criterion does not hold. In this case, a thread does other concurrent tracing tasks (e.g., card cleaning, see 2.2). If these tasks are not available, the thread may quit the incremental 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, then 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.

5.4 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 that they may be stolen by other "starved" threads [Endo et al. 1997]. Another solution is to expose the mark stacks themselves to the other threads [Flood et al. 2001]. The internal stack access operations (*PushBottom* and *PopBottom*) 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 Cheng and Belloch [2001]. They used a private stack for each thread and a single shared stack to exchange objects between threads, and reported a load balancing overhead of 14%. This work, as well as the work of Endo et al Endo et al. [1997], provided measurements on a large system of 32 or 64 processors. The overhead is thus incomparable to ours. We feel that separating inputs from outputs is adequate for load balancing when the number of collecting threads is dynamically changing during the collection.

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 [Endo et al. 1997; Cheng and Belloch 2001]. 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 8.4).

6. 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 [Corella et al. 1993] and Intel's IA-64 [Intel 1999]. Weak ordering is a part of the wider issue of *Relaxed Consistency* memory models, which are explained in detail in [Adve and Gharachorloo 1995].

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. As 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 Cheng and Belloch [2001] address some of the problems, but in the context of a replicating collector. Hudson and Moss Hudson and Moss [2001b] 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 Domani et al. [2000] address the problems of reversing the order of store and load operations in the DLG collector [Doligez and Leroy 1993; Doligez and Gonthier 1994], but do not address the more general weak ordering problems. Levanoni and Petrank Levanoni and Petrank [2001] 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 reminder of this section we elaborate on the weak ordering problems mentioned above and present our solutions to keep fence overhead low.

6.1 Shared mark stack accesses

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., a 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 5). 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 [Azagury et al. 1999].

6.2 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 immediately after the JVM has created and initialized each single 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 2.2, 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.[5]) All bits for free space are initially zero. For convenience we refer to allocation bits below 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.

[5]Here we deal with a seldom race between allocation and tracing and we want to defer tracing objects that the JVM has not had enough time to initialize. In section 4.2.2 above, allocation caches were used in a different scenario to undirty cards when there is no race (which is usually the case).

Future revisions to the Java memory model [SUN 2003] could require a fence operation after every object allocation In that case the solution described in this section would no longer be needed.

6.3 The Cleaning of 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 the solution to it.

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, it 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.

6.4 Algorithmic improvements

Our first method, not tracing through dirty cards, is not vulnerable to a change in the order of memory access. If the collector reads a wrong value from the card table showing that a card is not dirty, then, at worst, this only means that the collector does not gain the advantage of not tracing through a dirty card. We expect this to happen infrequently. On the other hand, if a card appears to be dirty while it has not yet been modified, then we know that the modification will be seen by the collector when all program threads are stopped for the final stop-the-world phase. At that time, the dirty card will be scanned properly.

Next, we note that it is easy to deal with weakly consistent platforms in the special case in which the allocator uses local caches. In this case, whenever the mu-

tators fill an allocation cache, they perform a synchronization barrier after undoing the dirty cards and before allowing the collector to trace through the objects on the allocation cache (i.e., before resetting the "in active cache" bits of the objects).

The generic case does require a modification that has a (small) performance cost. There, the order of operation is used to synchronize the operations of the collector (marking a card traced) and the operations of the undirtying procedure (checking whether the card has already been traced). The naive approach is to let the collector run a synchronization barrier after it marks a card traced (and before actually tracing an object in the card), and to let the undirtying procedure run a synchronization barrier on each dirty card (before checking whether it is traced). This solution is not so bad, since a synchronization barrier is run at most once by the collector and at most once for each invocation of the undirtying routine. (Note that the mutator is not involved in these costs.) We also propose an alternative method that runs only a few synchronization barriers, but requires a handshake between the collector and the undirtying procedure for each invocation of the undirtying procedure.

The alternative method is as follows. The undirtying procedure starts by running sequentially on the card table and marking all the dirty-and-not-yet-traced cards as not dirty. The procedure records the cards whose dirty bits were cleared (by making a list or using an additional card table). Next, the procedure needs to cooperate on a synchronization barrier with the collector. To this end, a handshake with the collector is used: the procedure runs a synchronization barrier and requests the concurrent collector to run a synchronization barrier in its code. When both have run the handshake, the undirtying procedure runs again over all cards whose dirty bit was cleared. For each such card that is now marked traced, the undirtying procedure makes the card dirty again.

7. PLATFORM, BENCHMARKS, IMPLEMENTATION

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. The measurements of the algorithmic improvement were separated from the other measurements so that their impact may be well noticed. The collector that was finally incorporated into the IBM production JVM includes the algorithmic improvements.

7.1 Platform

Measurements were mostly done on several platfroms which we now detail and name for future reference.

(1) **NT-4**: this is an IBM Netfinity 7000, a server with four 550 MHz Pentium III XeonTM processors and 2GB of RAM running Windows NT 4.0. Unless otherwise mentioned, this is the platform we use.

(2) **AIX-6**: some measurements were also run on a pSeries server, with six 600 MHz PowerPC RS64 III processors (64 bit) and 5 GB of RAM running AIX 5.1.

(3) **XP-1**: the single-threaded client-side benchmarks were measured on an IBM ThinkPad A31p, with a 2.00 GHz Pentium 4 processor and 512 MB of RAM, running Windows XP Professional.

(4) **XP-4**: as a representative of a weak ordering memory access system we used the Intel Itanium server, with four 667 MHz IA-64 processors and 8 GB of RAM running Windows XP 5.1.

(5) **AIX-4**: this is a pSeries server, with four 600 MHz PowerPC RS64 III processors (64-bit) and 5 GB of RAM running AIX 5.1.

(6) **AIX-8**: we added a final measurements for this journal version on a modern 8-way 64 bit PowerPC Power4 machine with 1.1 GHz processors, a 10 Gbytes memory and 1.44 Mbytes cache, running AIX 5.2.

All measurements were run with a fixed heap size. We chose the heap size so that the heap would reach 60% occupancy. Bigger heap sizes would have produced fewer collection cycles; thus, showing a lower overhead for the collector. However, we chose this more challenging setup to exercise the collector.

We start by running the CGC with tracing rate 8.0 (see Section 3 for a discussion on the tracing rate parameter). At this level the collector performs quite well. However, at lower tracing rates, the "naive" collector's performance deteriorates substantially. Our algorithmic improvements ameliorate the collector's behavior and make it efficient at the lower tracing rates, which are likely to be used in practice.

In the runs we used 1000 work packets (each packet is of size 512 and holds up to 493 entries). Note that the size of the packets imposes a trade-off. Smaller packets imply better load balancing but more synchronization overhead whereas larger packets imply lower overhead on synchronization but worse load balancing. We tried increasing the size by a factor of 2 and found out that performance has not changed and we tried reducing the size by a factor of 2, but that did not influence the load balancing. In the runs we also used four low priority background threads, and performed a single pass of concurrent card cleaning.

7.2 Benchmarks

We used three benchmarks in our measurements: *SPECjbb*, *pBOB*, and *SPECjvm98*.

SPECjbb2000 [SPECjbb2000 1998] is a Java business benchmark inspired by TPC-C. It emulates a 3-tier transaction system, concentrating on the middle tier. SPECjbb is throughput oriented; it measures the amount of work done during a given time. The result is given in *TPM* (transactions per minute). Since the amount of work done in a fixed amount of time differs for each machine tested, the overall allocation amount changes. In general, SPECjbb allocates tens of Gbytes in the experiments we ran. On a 6-way multiprocessor, an official run of SPECjbb includes twelve short cycles of two minutes each. The first cycle creates a single warehouse (thread), and each successive cycle increases the number of warehouses by one, ending with twelve warehouses (on an N-way machine the benchmark is usually run from one warehouse to twice the number of processors). Each warehouse is represented by a separate thread, and thus, the number of program threads equals the number of warehouses. Adding warehouses increases the amount of live objects,

the object allocation rate, and the level of GC activity. SPECjbb issues a score for each cycle, and a total score for the entire run.

We will sometimes use the SPECjbb averaging convention for computing the total score to produce a similar single score for memory consumption, pauses, etc. Specifically, on a 6-way multiprocessor, a simplified method to approximate the total score out of the twelve separate scores, is to average over the scores from six warehouses and upwards. We use exactly the same method to calculate a single value from the set of per-warehouse results. For example, when we quote the average heap residency of a set of tests, we mean that we have calculated the average heap residency for six warehouses and up, over all these tests. On the 4-way NT machine, we average from four warehouses and up.

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.

SPECjvm98 [SPECJVM98 1998] is a benchmark suite that measures computer system performance for Java Virtual Machine (JVM) client platforms. It consists mostly of single-threaded benchmarks that use relatively small heaps (typically, less than 50 MB). The overall allocated space during the benchmark run is: 334MB for compress, 748MB for jess, 224MB for db, 518MB for javac, 355MB for mtrt, and 481MB for jack. We measured this benchmark suite in order to provide some insight into the behavior of our improvements for small applications. The performance measure of SPECjvm98 is execution time. These benchmarks were run with the default input size parameter 100.

As the applications of *SPECjvm98* are quite small, they do not really test the target setting for our collector, and sometimes, with such small applications we do not see differences in behaviors between compared collectors. Therefore, we sometimes included only measurements of the largest benchmark in the suite, the *javac* benchmark. It is a single-threaded Java compiler.

7.3 Test rules

In order to test the garbage collection mechanism under a reasonably heavy load when running SPECjbb, we aimed at achieving a 60% heap residency at the maximal number of warehouses, and therefore used a 448 MB heap for the 6-way (64-bit) machine, and 256 MB for the 4-way (32-bit) machine. Note that the heap size is also influenced by the architecture used, as 64-bit objects are bigger. For the same reasons, we used a smaller heap when measuring the SPECjvm98 benchmarks. The entire suite was run with the 32 MB heap. Except where noted otherwise, results are averaged over five runs.

7.4 Implementation of the algorithmic improvements

We have reduced tracing work by avoiding the tracing through dirty cards (as described in Section 4.1) and we have reduced the number of dirty cards (as described in Section 4.2). We refer to the resulting collector as the *improved* collector. In the latter improvement, we combined the two proposed implementations: all through the concurrent collection cycle we undirty cards via allocation caches (as described in Section 4.2.2). In addition we scan the whole card table and undirty all the cards

which were not traced yet (as described in Section 4.2.1) once during each collection: after the concurrent tracing phase is over and before starting the concurrent card cleaning phase.

For the first improvement (not tracing through dirty pages) there are two possibilities regarding when to check if the object resides on a dirty card. The first option is to avoid *pushing* an object onto the mark-stack if it is on a dirty card. The second option is to avoid tracing a *popped* object if it resides on a dirty card when it is popped from the mark-stack. The advantage in checking the dirty card before pushing the object into the stack is that we do not need to spend time on pushing and popping objects that will later not be traced. However, checking the dirty card upon popping the object occurs at a later time after more pages have been marked dirty and so may spare double tracing of more objects. We have implemented both options and did not see any difference in performance. Our reported results are for the version that checks the dirty card when popping the object from the mark-stack.

We also created implementations of each improvement separately in order to analyze their specific impact. These partial implementations are discussed in Section 8.5.6.

In some parts of this discussion, we compare various results to those of the mark and sweep stop-the-world collector, which is also part of the IBM JVM. We refer to this collector as *MS STW*.

We stress that our algorithm did not require any extra auxiliary data structures in addition to the ones already existing in the naive algorithm. The same card table has been used. The only part of the algorithm that seems to require an additional structure is the second part of the algorithm as described in Subsection 4.2. There, we need a card table signifying which cards contain objects that have been traced already. However, our base ("naive") collector keeps a mark-bit table for which a card is reflected by two long words. Instead of creating a new table, we use the existing mark-bit table and for each card we checked that its corresponding two long words contained no set mark-bits.

8. RESULTS

8.1 Comparison with STW Collector

We first provide the main results, presenting the throughput and pause times of the new collector and the stop-the-world collector. We later provide more measurements on more platforms and checking specific characteristics of the collector. The measurements provided here were measured on the AIX-8 platform (see section 7.1 for the platforms description). As always, the size of the heap was chosen to 600Mbytes so that at the end of the run (with 16 warehouses), the heap occupancy will be 60% of the heap size. The results are reported in Table I.

When using the collector aggressively with tracing rate 8, the maximum pause time is reduced from 161ms to 46ms and the average pause time is reduced from 115.2ms to 34ms, with less than 12% throughput hit. When using the collector non-intrusively with tracing rate 1, the maximum pause time obtained is 79.2ms and the average pause time is 54ms. The throughput hit in this case is less than 16%. The algorithmic improvements are most effective with the lower tracing rates. Without them, the throughput hit is much higher for low tracing rates.

Measurement	STW	TR 1	TR 2	TR 4	TR 8
Average Pause Time (ms)	115.2	54.0	47.5	39.4	34.0
Max Pause Time (ms)	161.0	79.2	68.8	56.4	46.0
Throughput	72627	61417	62550	63326	64244
throughput Percentage	100.0%	84.6%	86.1%	87.2%	88.5%

Table I. SPECjbb (AIX-8): the overall performance (throughput and pauses) of the proposed collector compared to the stop-the-world collector.

We next report the percentage of times spent in each mode: *mutator only* is the fraction of time the mutator runs without any collector activity, *concurrent* is the fraction of time the concurrent collector runs concurrently with the program threads, and *stw* is the fraction of time the mutators are stopped and the collector is active alone. The results are provided in Table II for each of the tracing rates. This gives the information on how much the mutators are interrupted with each tracing rate. Finally, we measured the overhead of the write barrier, by using the (card marking) write barrier with the stop-the-world collector (and not using the information recorded). The average write-barrier overhead on the AIX-8 platform is 2.3%.

Activity	STW	TR 1	TR 2	TR 4	TR 8
percent mutator only	92.2	15.5	45.7	64.6	74.9
percent concurrent	0.0	81.6	51.9	33.5	23.5
percent stw	7.8	2.9	2.4	1.9	1.6

Table II. SPECjbb (AIX-8): the fraction of time spent on each of the collection phases on average, for each tracing rate and for the STW collector.

We now turn to checking the naive collector that does not use the algorithmic improvements.

8.2 Properties of the naive collector

In this section we provide measurements of the naive collector without use of the algorithmic improvements (of Section 4).

Figure 2 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%.

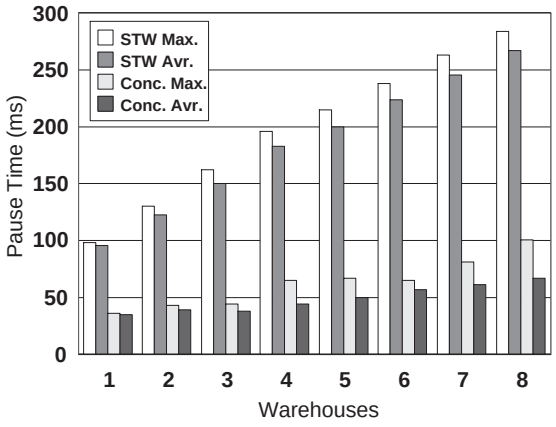

Fig. 2. SPECjbb (NT-4): Pause times for SPECjbb with tracing rate 8.0

The same SPECjbb test was also run on the *XP-4* weak ordering memory access platform. 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 2 and discussed above.

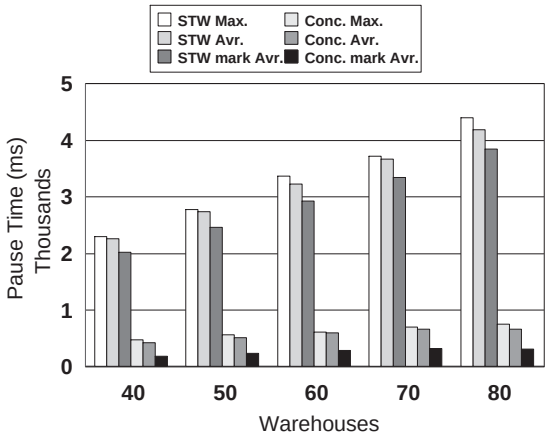

Fig. 3. pBOB (AIX-4): Pause times and concurrent mark times for pBOB with tracing rate 8.0

In order to test our collector on a 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 the *AIX-4* platform described in Section 7.1.

Figure 3 shows average pause times, maximum pause times, and average mark time for pBOB. For the STW collector, this is the time spent on marking the live objects during the pause. For the concurrent collector, this is the time spent on

marking while clearing the dirty cards. 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 [Boehm et al. 1991], i.e., deferring the sweep to after the stop-the-world phase (see Section 9), 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 more slowly 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).

8.3 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 III compares the execution of the STW collector with CGC, while varying the tracing rates for SPECjbb. This reports the characteristics of the mostly concurrent collector without the algorithmic improvements proposed in section 4. 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 final stop-the-world phase of CGC, and the average and maximum pause times.

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 therefore fewer new cards stand the chance to become dirty while the card table is being scanned. The card size is 512 bytes, so there are 524288 cards in the 256 MB heap, so that 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. Average pause times are more stable and better reflect pause time values. Maximum pause times fluctuate and this instability shows in the unexpected increase in the maximum pause times at tracing rate 10. The throughput improvement at higher

Measurement	STW	TR 1	TR 4	TR 8	TR 10
Average Pause Time (ms)	267	177	115	67	61
Max Pause Time (ms)	284	233	134	101	126
Average Final Card Cleaning	—	93627	40147	11772	8394
Throughput	19904	15511	16984	17970	18177
throughput Percentage	100.0%	77.9%	85.3%	90.3%	91.3%
Floating Garbage	0.0%	18.0%	14.2%	5.3%	4.2%

Table III. SPECjbb (NT-4): the effects of different tracing rates (8 warehouses) (for the basic mostly concurrent collector without the algorithmic improvements).

tracing rates is probably due to lower overheads for card cleaning (less cards are dirtied during the concurrent phase) and less floating garbage.

8.4 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 allocations 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 IV 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

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 IV. SPECjbb (NT-4): the Quality of Load Balancing

increases, showing no increase in "starvation". The fairness declines as the number of threads increases 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 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 5.4 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 0.15% of the heap size is a realistic estimation of the needed space for work packets.

8.5 The algorithmic extensions

In the previous subsections, we have compared the naive collector (denoted CGC) with the stop-the-world parallel collector. In this section, we measure the improvement of the algorithmic modifications over the "naive" CGC mostly concurrent collector. We stress that the CGC "naive" version is the one that contains all other ingredients of the design described in this paper. In particular, it is parallel, incremental, it employs work packets to obtain load balancing, etc. In particular, we measure the improvement in throughput, pause times, and heap consumption on the Netfinity Intel platform as well as on an AIX platform. We then discuss the effect of our improvements on L2 cache behavior. Next, we provide measurements for each of our improvements separately (i.e., the improvement of not tracing through

dirty cards and the improvement of undoing dirty cards). These measurements show how each improvement impacts the naive base collector as a stand alone. Finally, we compare the two proposed methods for undoing the dirty state of cards.

8.5.1 *Comparing the naive and improved collectors.* When evaluating a mostly concurrent garbage collector, a major concern is its effect on the **performance**, **pause time**, and **heap residency** of the application. In this section, we present detailed comparisons of the naive implementation and improved implementations. We present the results of runs with various tracing rates. Recall that tracing rates 8, 4, 2 and 1 mean that the collector gets 72%, 58%, 42%, and 29% of the CPU, respectively. We feel that tracing rate 1 is more interesting for a concurrent collector since it lets the program run non-disruptively. Our improvement works best at the lower tracing rates.

Run	W 2	W 4	W 6	W 8	W 10	W 12
Base Tr1	20.0	31.2	34.2	29.3	26.0	22.8
Imp. Tr1	20.9	35.6	42.7	37.1	32.8	29.4
Base Tr2	20.5	33.6	38.6	32.2	27.8	23.2
Imp. Tr2	21.0	36.0	44.2	38.7	34.7	30.4
Base Tr4	20.8	35.0	41.9	35.6	31.5	26.8
Imp. Tr4	21.1	36.3	44.4	39.3	35.3	31.4
Base Tr8	20.9	35.8	43.6	38.2	34.4	29.9
Imp. Tr8	21.2	36.7	45.3	40.0	36.3	32.2

Table V. SPECjbb (AIX-6): Throughput comparison of the base and improved collectors, for all tracing rates and for 2, 4, 6, 8, 10 and 12 warehouses. Values in thousands of TPMs.

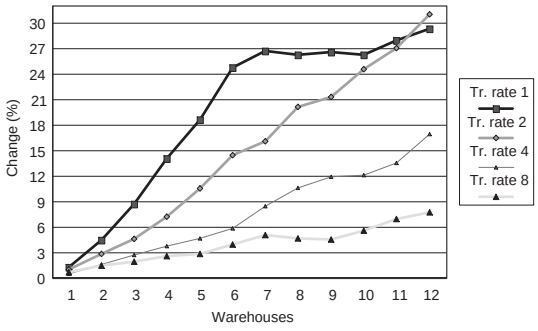

Fig. 4. SPECjbb (AIX-6): Throughput change between the base collector and the improved collector, for all tracing rates

8.5.2 *SPECjbb2000 on pSeries server (AIX-6).* Table V shows the throughput scores of SPECjbb for both the base collector and our improved one. The results are shown for 2, 4, 6, 8, 10, and 12 warehouses and for tracing rates 1, 2, 4, and 8.

Figure 4 graphically depicts the improvement, for all warehouses and for all tracing rates.

We can see that there is a significant increase in scores. Note that our improved collector achieves an average throughput increase of 26.7% at tracing rate 1. The scoring convention by which we average the twelve numbers into one is explained in Section 7.2 above. When switching to higher tracing rates, the improvement becomes smaller. For tracing rate 8 the average improvement is only 5.4%.

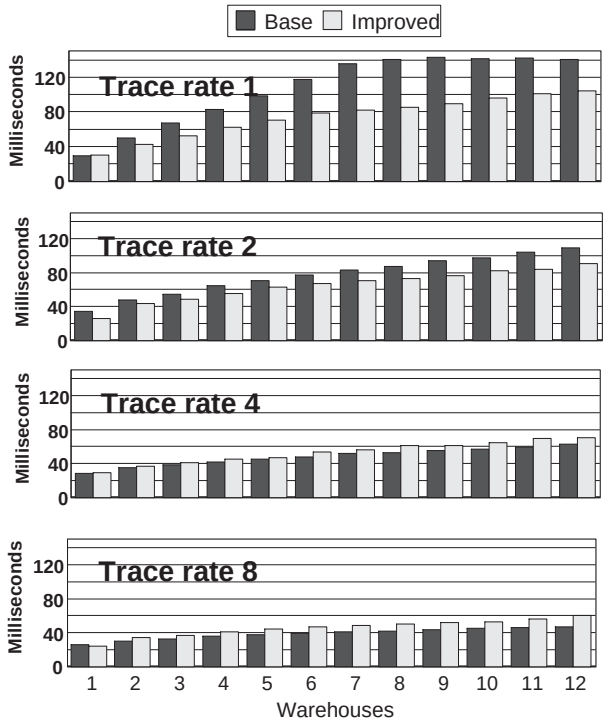

Fig. 5. SPECjbb (AIX-6): Average pause time comparison for all tracing rates

In Figure 5, we present the average pause times of the improved collector and the base collector. The improved collector managed to reduce the pause times for tracing rates 1 and 2. For higher tracing rates the pause times increased slightly. Recall that our first method (of not tracing through dirty pages) increases the pause times whereas the second method (of undirtying dirty cards) reduces the pause times. The effects of each method is investigated in subsection 8.5.6 below. What we see in Figure 5 is the combined effect. Figure 6 shows the maximal pause times. The pattern of change in the maximal pause times is similar to that of the average pause times.

Finally, we checked the impact of our improvements on the heap residency, which is defined as the total amount memory on the Java heap which is not reclaimed. Measuring the heap residency is important, since it influences the required size for the Java heap and therefore the footprint of the JVM. In addition to all the objects that are reachable, heap residency consists of two other elements: objects that were

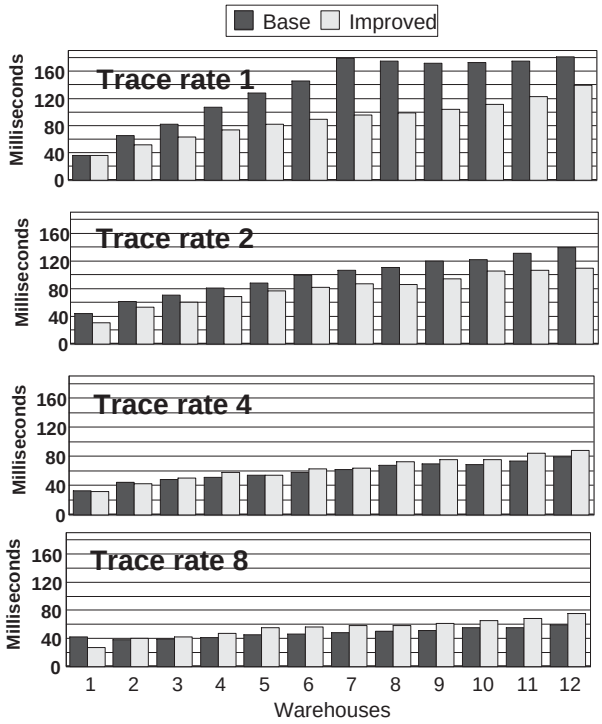

Fig. 6. SPECjbb (AIX-6): Maximal pause time comparison for all tracing rates

traced by the concurrent collector, but became unreachable later in the collection cycle, and gaps between objects that were too small to be reused. The former is called *floating garbage*, and the latter is the fragmentation inside the heap, and is also known as *dark matter*. As the amount of reachable objects (in a stable state of a SPECjbb execution) does not depend on the nature of the garbage collector, it follows that the differences in heap residency come from the collector's influence on the amount of floating garbage and fragmentation.

Run	W 2	W 4	W 6	W 8	W 10	W 12
Base Tr1	80.0	139.6	193.4	231.4	271.8	305.4
Imp. Tr1	65.8	113.0	157.8	197.8	237.0	276.0
Base Tr2	74.8	127.0	176.6	220.6	262.0	300.8
Imp. Tr2	66.0	111.4	155.2	194.4	235.2	273.8
Base Tr4	69.0	116.8	163.0	204.0	244.6	285.2
Imp. Tr4	63.6	110.2	154.2	193.0	232.4	270.2
Base Tr8	65.4	112.4	156.6	195.8	235.0	273.2
Imp. Tr8	63.4	109.0	152.8	192.4	230.0	267.4

Table VI. SPECjbb (AIX-6): Heap residency comparison of the base and improved collectors, for all tracing rates. Values in Mbytes, for warehouses 2, 4, 6, 8, 10 and 12.

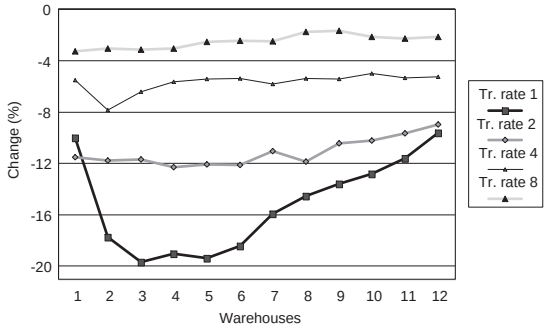

Fig. 7. SPECjbb (AIX-6): Heap residency change between the base collector and the improved collector, for all tracing rates.

Table VI shows how our improvements reduced the heap residency, for warehouses 2, 4, 6, 8, 10, and 12, and for tracing rates 1, 2, 4, and 8.

Figure 7 graphically depicts the changes. The average reduction at tracing rate 1 is 13.4%. As the tracing rate increases, we see a gradual decline in this reduction, until the average drops to 2.1% when using tracing rate 8. We compared the heap residency when using a *MS STW* collector (which has no floating garbage) to our results. Relative to the *MS STW* collector, the base mostly concurrent collector added 20.1% to the heap residency, with tracing rate 1; and 3.0% with tracing rate 8. Our improved collector added only 4.0% to the heap residency with tracing rate 1, and 0.7% with tracing rate 8. We conclude that our improvement eliminates most of the floating garbage created by the concurrent collector.

The behavior of the collector with tracing rate 1 is a bit different than with other tracing rate. The difference is noticeable with a small number of threads and with a large number of threads. The behavior with a small number of threads is not very interesting and it arises from the fact that there are too few garbage collections and the adaptive triggering mechanism does not get to a stable point. To explain the behavior with a large number of threads, we note that when many threads run, the heap occupancy grows. Since the size of the live objects gets larger, tracing rate 1 cannot be used to complete the collection on time. Therefore, the dynamic triggering increases the tracing rate automatically and the behavior starts to look more and more similar to the behavior with tracing rate 2. This is the reason why the line representing tracing rate 1, comes close to the line of tracing rate 2 when the number of warehouses increases.

Varying the heap size. As discussed in Section 7.4, most of our measurements were run with a heap of size 448 MB, in order to set the heap residency to 60% at the maximal number of warehouses. As a sanity check, we also measured performance on a larger heap to verify that our improvements do not misbehave in a different environment. In particular, we ran the base and improved collector using a heap of (double) size 896 MB. Our expectations were met by these measurements: when the heap gets larger, the number of collections is reduced, and so our improvement has less effect on the overall throughput.

In Table VII we compare the improvement of our algorithm in both sizes of

the heap, and report the reduction in the number of garbage collections. The

	Tr1	Tr2	Tr4	Tr8
Throughput improvement in 448 MB	26.7%	21.2%	10.9%	5.4%
Throughput improvement in 896 MB	13.9%	7.9%	4.5%	2.9%
Drop in throughput improvement	47.9%	62.9%	58.9%	45.2%
Drop in number of GC cycles	55.9 %	55.6%	56.9%	57.3%

Table VII. SPECjbb (AIX-6): Throughput improvement when using a 448 MB heap and an 896 MB heap, and the drop in the number of GC cycles, when switching to the 896 MB heap, for all tracing rates

improvements in the pause times (in the lower tracing rates) and the reduction in heap residency were not substantially affected by the move to a larger heap, and are not reported. These findings confirm the expected. Our ability to improve the efficiency of the collection and eliminate most of the floating garbage is not restricted to smaller heap sizes. The influence on the overall throughput depends on the percentage of time spent on garbage collection during program run.

8.5.3 *SPECjbb2000 on Netfinity 7000 server (NT-4).* In this subsection, we report measurements for the 4-way IBM Netfinity 7000 server. Figure 8 shows the throughput improvement (in scores) of the improved collector over the base collector for each warehouse and for all tracing rates. The overall pattern resembles the results on the AIX-6, but the improvement is smaller; the best average throughput increase is 16.5% (at tracing rate 1) and the smallest average improvement is 2.8%. We believe the difference between the two machines emanates from different cache behavior.

Fig. 8. SPECjbb (NT-4): Throughput change between the base collector and the new collector, for all tracing rates

Figures 9 and 10 present the average and maximum pause times of the improved collector and the base collector. The results are similar to those on the AIX-6

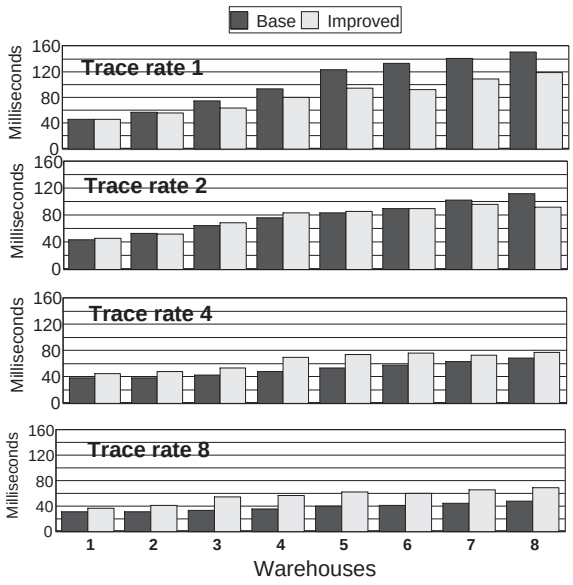

Fig. 9. SPECjbb (NT-4): Average pause times comparison for all tracing rates

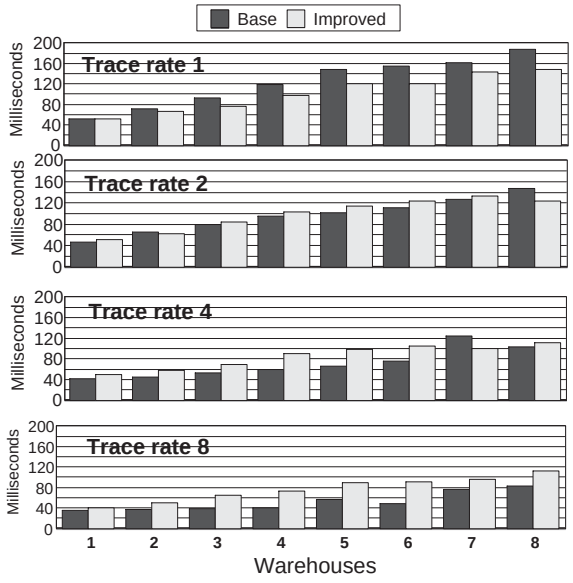

Fig. 10. SPECjbb (NT-4): Maximum pause times comparison for all tracing rates

machine; the improved collector reduces the pause times at tracing rates 1 and 2, but slightly increases the pause times at tracing rates 4 and 8. The change in heap

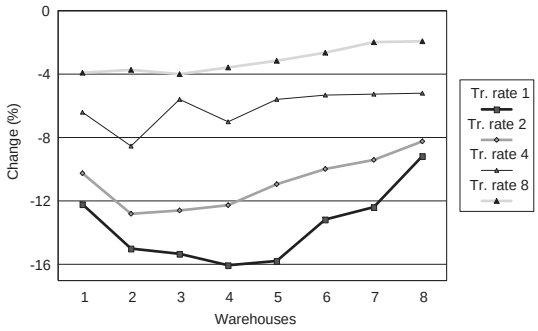

Fig. 11. SPECjbb (NT-4): Heap residency change between the base collector and the improved collector, for all tracing rates

residency is presented in Figure 11, and is similar to the AIX-6 results.

8.5.4 *Measurements of SPECjvm98.* In this section, we present the results of the base collector and our improved collector running the SPECjvm98 benchmarks suite. Our modifications are not expected to help small client applications. The reason is that such small applications do not create many dirty cards or much floating garbage. However, as a sanity check we measured our improvement also on a client setting. Indeed it turns out that our improvement does not cause harm for client benchmarks. We ran all the suite on the *XP-1* platform (see section 7.1). This is a customary setting for client side Java applications. As required for a legal SPECjvm98 run, we used the same set of the runtime parameters for all the benchmarks in the suite, including the heap size parameter which was set to 32 MB. We have chosen to use 32 MB heaps in order to accommodate the largest of the benchmarks (**javac**) with a live heap occupancy of about 60% of the total heap. We do not report results for the **mpegaudio** benchmark. This benchmark is known to have no substantial allocation activity ([SPECJVM98 1998]) and indeed the modified collector had no effect on the behavior of this benchmark.

In Table VIII, we present the execution times of the SPECjvm98 test programs for the base and the improved collectors for different tracing rates. The numbers are reported in seconds as measured by the benchmarks. At the low tracing rate, the improved collector runs faster than the base collector by up to 6%. At the higher tracing rates, we get almost no influence on the execution time.

In Table IX, we present the maximal pause times in milliseconds. There is no substantial difference between the collectors in terms of pause times. The pause times results for mtrt are a bit odd. The reason is that mtrt has a small number of collections (around 5) out of which the first one does a lot of work (due to the characteristic behavior of mtrt). The first pause is around 35ms whereas the other pauses are around 10ms. As the triggering mechanism is learning the behavior of the application from cycle to cycle, its behavior in the first cycle is not optimal. Therefore, allocation fails before the concurrent collector is done and the stop-the-world phase starts early, creating a long first pause.

In Table X, we present the average heap occupancy in Mbytes for all the benchmarks. Here, also, there is no notable change in the benchmarks behavior.

Run	com press	jess	ray trace	db	javac	mtrt	jack
Tr1							
Base	6.1	4.7	2.4	12.6	11.1	3.2	4.7
Imp.	5.8	4.4	2.3	12.5	10.5	3.1	4.6
Tr2							
Base	5.9	4.6	2.4	12.8	9.9	3.1	4.5
Imp.	5.9	4.5	2.3	12.5	9.8	3.0	4.7
Tr4							
Base	5.8	4.6	2.3	12.5	9.3	3.0	4.5
Imp.	5.9	4.6	2.3	12.9	9.4	3.4	4.5
Tr8							
Base	5.8	4.6	2.3	12.5	8.9	3.0	4.5
Imp.	5.8	4.5	2.3	12.5	9.0	3.0	4.5

Table VIII. SPECjvm98 (XP-1): Execution times in seconds for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

Run	com press	jess	ray trace	db	javac	mtrt	jack
Tr1 base	3	6	4	3	31	32	9
Tr1 imp.	3	6	5	3	29	33	8
Tr2 base	3	5	5	5	20	36	6
Tr2 imp.	3	6	4	4	23	34	7
Tr4 base	4	6	4	3	14	35	5
Tr4 imp.	6	6	4	3	13	36	5
Tr8 base	6	7	5	3	14	39	4
Tr8 imp.	5	6	4	3	14	39	4

Table IX. SPECjvm98 (XP-1): The maximal pause times in milliseconds for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

Run	com press	jess	ray trace	db	javac	mtrt	jack
Tr1							
Base	9.8	6.5	5.7	10.4	20.1	14.6	4.7
Imp.	10.0	6.3	5.7	10.4	19.3	14.8	4.7
Tr2							
Base	6.3	6.4	5.7	10.1	18.9	14.3	4.8
Imp.	6.3	6.4	5.7	10.0	18.9	14.3	4.8
Tr4							
Base	7.1	6.5	5.7	10.1	18.1	14.0	3.6
Imp.	5.5	6.4	5.7	10.0	18.4	14.3	3.7
Tr8							
Base	5.9	6.5	5.7	10.0	18.1	14.6	3.7
Imp.	5.9	6.5	5.7	10.1	17.9	14.5	3.7

Table X. SPECjvm98 (XP-1): The average live objects in MBytes for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

Finally, in Table XI we report the number of GC cycles executed during the run of each benchmark. This number was not affected by the improvement, and

therefore we only report it for the base collector. Note that although these are client (small) benchmarks, all of them execute at least four collections, and thus, the results are meaningful.

Run	com press	jess	ray trace	db	javac	mtrt	jack
Tr1	5	12	4	4	11	7	6
Tr2	4	12	4	4	9	7	6
Tr4	4	12	4	4	9	7	5
Tr8	4	12	4	4	8	7	5

Table XI. SPECjvm98 (XP-1): The number of GC cycles for all the programs, tracing rates 1, 2, 4 and 8.

8.5.5 *Cache misses.* Cache behavior has a significant impact on performance. It is therefore interesting to investigate the influence of our improvements on the cache. We concentrated on L2 cache misses, measured on the 4-way IBM Netfinity 7000 server. This machine has a 2 MB 4-way associative L2 cache. We used VTune [Products] to measure data accesses and cache misses on SPECjbb.

To focus on the cache misses inside the measured part of the cycle, we reduced the relative amount of terminal ramp-up by running a single long cycle of five minutes with six warehouses. We chose six warehouses, as this is the middle warehouse in the range of warehouses used for calculating the official score. We concentrate on tracing rate 1, as this rate is the more interesting one for a concurrent collector and the impact of our improvements is more noticeable. The metric we use is cache miss rate, which is the rate between cache misses and data access operations.

Table XII shows the L2 cache miss rate for the base and improved collectors at tracing rate 1. As one can see, the improved collector reduces the cache miss rate by 6.4%.

The improvement may come from two factors. First, collector's work is more likely to increase the cache misses. Thus, reducing the collector's work and letting mutators get more CPU time should result in reduced cache miss rate. However, there is an interesting additional factor: elimination of coherency cache misses.

As described in Patterson and Hennessy Patterson and Hennessy [1990], there are cache misses which relate to cache line traffic (i.e., *Compulsory*, *Capacity*, and *Conflict*), and cache misses which result from maintaining cache coherency on multiple caches. We refer to the former type of cache misses as *Access* misses and to the latter as *Coherency* misses. In MS STW collection, object tracing does not generate coherency misses, as the parallel tracing is finished while the Java mutators are suspended, and is a read-only operation. However, a concurrent tracing thread may trace into objects while they are modified by mutators executing on another processor. This will create coherency misses.

In order to check the effect of our improvements on coherency misses, we repeated the measurement on the same machine, when the SPECjbb run was restricted to a single processor. As only a single cache was used, all coherency misses are eliminated and only access misses remain. Obviously, the benchmark runs much slower, but this is filtered out by the cache miss rate metrics. The results of these

Run	4 processors	single processor
Base	1.38%	1.18%
Improved	1.30%	1.15%
Change	-6.43%	-2.73%

Table XII. SPECjbb (NT-4): L2 cache miss rates comparisons between the base collector and the improved collector, for tracing rate 1. Done both when using all processors and only a single processor

runs are shown at the right of Table XII; once again, the improved collector reduces the cache miss rate. However, the reduction in cache miss rate introduced by our improvements is much greater when running on all the processors of a 4-way machine than when running on a single processor, where no coherency misses occur. When comparing *Base* and *Improved*, there is a 6.43% reduction; but when running on a single processor, the reduction is only 2.73%. These results may imply that our improvements are also effective in reducing coherency misses. The measurements were extremely steady over the five runs.

Not tracing through dirty cards may be effective in reducing the cost of maintaining cache coherency on multiple caches for the following reason. Dirty cards tend to be "hot" in the cache. These are cards that are more likely to be modified by the mutators. Thus, when the collector reads them (to trace marked objects), the program is likely to be writing them and coherency misses occur more frequently.

8.5.6 *Impact of each of the improvements.* In this work, we present two novel techniques for improving the mostly concurrent collector: the elimination of repetitive collector work by restricting the trace through dirty cards, and the reduction in the number of dirty cards by undoing the dirty marks. In this section, we study the characteristics of each of these methods by running each of them separately and comparing the results to running both of them, or none of them.

We start with the throughput measurements, using the scoring convention described in Section 7.2. Figure 12 presents the throughput for four different collectors: the original naive collector using none of our improvements (denoted *Base*); a collector that runs only undoing of dirty cards (denoted *UndoDirty*); a collector that only restricts tracing through dirty cards (denoted *Restrict*); and the improved collector, that runs both improvements (denoted *Combined*). The results are provided for the tracing rates 1, 2, 4, and 8.

One can see that the impact of *UndoDirty* on the throughput is smaller than that of *Restrict*. However, both methods are beneficial to the throughput, even when they stand alone.

Figure 13 shows the average pause time values. Here we get the complementary results. *UndoDirty* has a clear positive impact with all tracing rates and is doing better than *Restrict*. *Restrict* has a positive impact only with the lower tracing rate, but a negative impact on tracing rates 2 and above.

Figure 14 shows the average heap residency when using none of our improvements (the naive *base* version), and when using each of *UndoDirty*, *Restrict*, and *Combined*. The results are displayed for all tracing rates.

The *UndoDirty* improvement has a small impact on reducing the heap residency, whether used alone or when added to *Restrict*. The dominant method in reducing

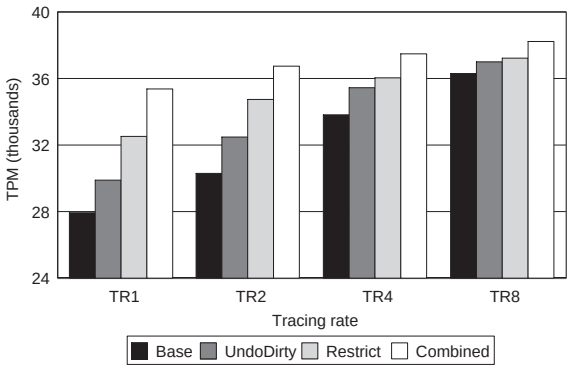

Fig. 12. SPECjbb (AIX-6): Throughput comparisons between different configurations of our improvements

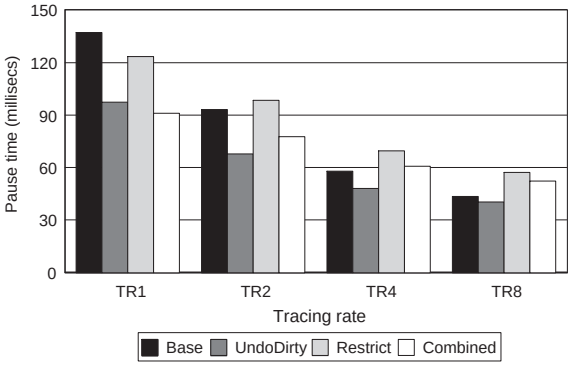

Fig. 13. SPECjbb (AIX-6): Average pause time comparisons between different configurations of our improvements

heap residency (and thus also floating garbage and dark matter) is *Restrict*.

Why is this impact so large? Our proposed explanation is as follows. The fact that the floating garbage was reduced by not tracing through dirty cards means that marked objects on dirty cards were modified after they were traced by the naive collector. When the naive collector re-traced them, they contained more descendants to trace. The naive collector had to trace the descendants of the original children as well as the descendants of the newly assigned children, whereas the improved collector traced only the latter. It turned out that many of the descendants of the original children became unreachable before the end of the cycle. An implication of this observation is that objects that reside on dirty cards are likely to be modified and furthermore, are likely to have their descendants become unreachable soon. This may explain why deferring the trace of objects on dirty cards eliminates much of the floating garbage.

Another interesting point to note in these measurements is that lower tracing rates cause the heap residency to increase. This is because the collector runs for

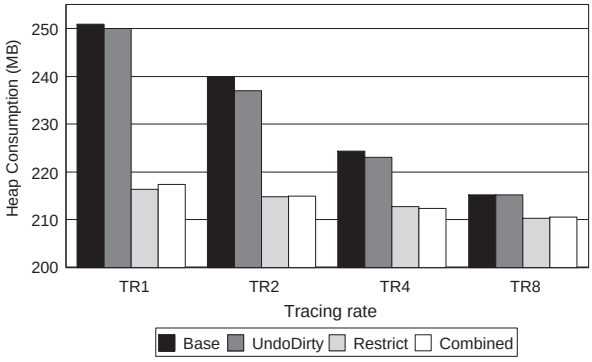

Fig. 14. SPECjbb (AIX-6): Heap residency comparisons between different configurations of our improvements

a longer time, so it may produce more floating garbage and fragmentation. Our improvement significantly reduces this effect.

Finally, reducing the floating garbage is a meaningful advantage. Our collector gains from eliminating repeated scans. But it also gains from the fact that it traces fewer objects: it avoids the (unreachable) floating garbage objects that the original collector has to trace. It is not clear (and it is not easy to measure) which of these benefits provides a higher improvement for the throughput of the improved collector. An additional benefit of reducing floating garbage is that the heap is better utilized with live objects and thus less collections are necessary during the run.

We can deduce that the method of not tracing through dirty cards has multiple effects on throughput:

(1) It does a more efficient tracing by eliminating the double handling of reachable objects in dirty cards.
(2) It reduces the amount of objects that are traced, by reducing the floating garbage.
(3) It produces more free space, and thus reduces the number of garbage collections.
(4) It reduces the rate of L2 cache misses, especially those that are introduced in order to maintain cache coherency (as described in Section 8.5.5).

8.5.7 *Comparing card undirtying methods.* In Section 4.2.2 above, we described how to undo dirty cards when allocation caches are in use by the allocator. Recall that with this method, all cards in a full local allocation cache are marked as not dirty, before any tracing into their objects is allowed. We implemented this method, together with restricting the trace through dirty cards. This implementation is denoted *AllocationUndo*. We also implemented the undirtying via scanning method described in Section 4.2.1 (which is also appropriate for collectors that do not use allocation caches), again with restricting the trace through dirty cards. This implementation is denoted *ScanUndo*. We compared *AllocationUndo*, and *ScanUndo*, with the collector that only restricts tracing through dirty cards (denoted *Restrict*). The measurements were done on the IBM Netfinity 7000 server.

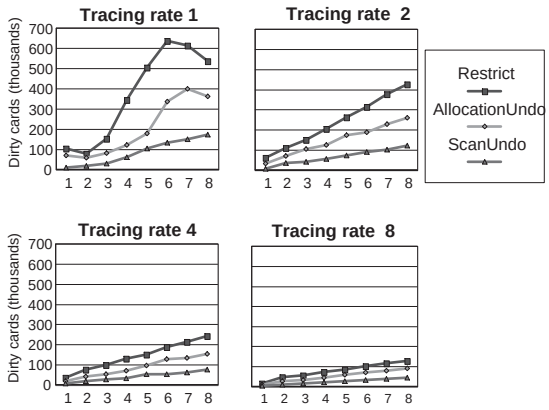

Fig. 15. SPECjbb (NT-4): The number of dirty cards traced by the collector when using different Undo methods for 1–8 warehouses.

Figure 15 shows, for all methods and all tracing rates, the number of dirty cards that are cleaned by the collector through the collection cycle. One can see that *AllocationUndo* reduces the number of dirty cards by roughly 40% compared to the *Restrict* collector, whereas *ScanUndo* reduces it much more, by roughly 70%. Clearly, both methods do very well. The reason that *ScanUndo* does better is probably because it is not limited to new objects and it runs repeatedly. Yet when comparing the performance of these two methods, we could not measure a significant difference in throughput or in pause time. This can be seen in Table XIII, which shows the change between the *Restrict* collector and both methods of undoing dirty cards. This relative change is shown for all tracing rates. The likely reason for the throughput similarity is that the gain from less dirty cards (with *ScanUndo*) is balanced by the cost of the repetitive scans of the card tables.

In our improved collector we combined the use of these two methods, as described in Section 7.4. The change between this collector (denoted *Improved*) and *Restrict* is also presented in Table XIII.

Change in	Tr1	Tr2	Tr4	Tr8
Throughput				
ScanUndo	2.1%	2.3%	1.8%	1.0%
AllocationUndo	2.3%	2.4%	1.8%	1.5%
Improved	4.0%	3.0%	2.0%	1.5%
Pause time				
ScanUndo	-8.9%	-12.8%	-7.7%	-11.4%
AllocationUndo	-2.4%	-13.9%	-9.4%	-4.2%
Improved	-6.0%	-15.6%	-6.2%	-4.1%

Table XIII. SPECjbb (NT-4): Change in throughput and average pause time (relative to the naive collector) with different Undo methods and tracing rates

9. CONCLUSIONS

We presented 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 implemented 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. To achieve these results we apply several 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 propose two novel algorithmic improvements to the mostly concurrent algorithm that improve the throughput and shorten the pause times. Finally, 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.

10. ACKNOWLEDGMENTS

We thank the anonymous referees for their deep and helpful remarks. We would like to thank Tamar Domani, Ethan Lewis, 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.

REFERENCES

Adve, S. V. and Gharachorloo, K. 1995. Shared memory consistency models: A tutorial. Research Report 95/7, Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA. Sept.

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

Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E. 2003. An on-the-fly mark and sweep garbage collector based on sliding view. See OOPSLA [2003].

Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, V. T., and Smith, S. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. See PLDI [2001].

Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. 1998. Thin locks: Featherweight synchronization for java. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices. ACM Press, Montreal, 258–268.

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

Barabash, K., Ossia, Y., and Petrank, E. 2003. Mostly concurrent garbage collection revisited. See OOPSLA [2003].

Ben-Yitzhak, O., Goft, I., Kolodner, E., Kuiper, K., and Leikehman, V. 2002. An algorithm for parallel incremental compaction. See Detlefs [2002], 100–105.

Boehm, H.-J., Demers, A. J., and Shenker, S. 1991. Mostly parallel garbage collection. *ACM SIGPLAN Notices 26,* 6, 157–164.

Borman, S. Sensible sanitation - understanding the ibm java garbage collector (part 1: Object allocation). http://www.ibm.com/developerworks/ibm/library/i-garbage1.

Cheng, P. and Belloch, G. 2001. A parallel, real-time garbage collector. See PLDI [2001], 125–136.

Corella, F., Stone, J., and Barton, C. 1993. Specification of the PowerPC shared memory architecture. Tech. Rep. 18638, IBM Thomas J. Watson Research Center. Jan.

Detlefs, D., Ed. 2002. *ISMM'02 Proceedings of the Third International Symposium on Memory Management.* ACM SIGPLAN Notices. ACM Press, Berlin.

DeTreville, J. 1990. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, DEC Systems Research Center, Palo Alto, CA. Aug.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. 1976. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46.* Springer-Verlag, New York.

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

Dimpsey, R., Arora, R., and Kuiper, K. 2000. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal 39,* 1, 151–174.

Doligez, D. and Gonthier, G. 1994. 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.

Doligez, D. and Leroy, X. 1993. 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. ACM Press, 113–123.

Domani, T., Kolodner, E., and Petrank, E. 2000. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation.* ACM SIGPLAN Notices. ACM Press, Vancouver.

Domani, T., Kolodner, E. K., Lewis, E., Salant, E. E., Barabash, K., Lahan, I., Petrank, E., Yanover, I., and Levanoni, Y. 2000. Implementing an on-the-fly garbage collector for Java. See Hosking [2000].

Endo, T., Taura, K., and Yonezawa, A. 1997. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97).*

Endo, T., Taura, K., and Yonezawa, A. 2002. Reducing pause time of conservative collectors. See Detlefs [2002], 12–24.

Flood, C., Detlefs, D., Shavit, N., and Zhang, C. 2001. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01).* Monterey, CA.

Hosking, T., Ed. 2000. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management.* ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN.

Hudson, R. L. and Moss, J. E. B. 1992. Incremental garbage collection for mature objects. In *Proceedings of International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, vol. 637. Springer-Verlag, University of Massachusetts, USA.

Hudson, R. L. and Moss, J. E. B. 2001a. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference.* Stanford University, CA.

Hudson, R. L. and Moss, J. E. B. 2001b. Sapphire: Copying gc without stopping the world. In *ISCOPE Conference on ACM 2001 Java Grande.* ACM Press, Palo Alto CA USA, 48–57.

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

Intel 1999. *IA-64 Application Developer's Architecture Guide.* Available at http://developer.intel.com/design/itanium.

Jones, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley. With a chapter on Distributed Garbage Collection by R. Lins.

Lang, B. and Dupont, F. 1987. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques.* ACM SIGPLAN Notices, vol. 22(7). ACM Press, 253–263.

Levanoni, Y. and Petrank, E. 2001. An on-the-fly reference counting garbage collector for Java. See OOPSLA [2001].

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

Moon, D. A. 1984. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, G. L. Steele, Ed. ACM Press, Austin, TX, 235–245.

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

OOPSLA 2001. *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications.* ACM SIGPLAN Notices, vol. 36(10). ACM Press, Tampa, FL.

OOPSLA 2003. *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications.* ACM SIGPLAN Notices. ACM Press, Anaheim, CA.

Ossia, Y., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V., and Owshanko, A. 2002. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation.* ACM SIGPLAN Notices. ACM Press, Berlin, 129–140.

Patterson, D. and Hennessy, J. 1990. *Computer architecture: a quantitative approach.* Morgan Kaufman.

PLDI 2001. *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation.* ACM SIGPLAN Notices. ACM Press, Snowbird, Utah.

Printezis, T. and Detlefs, D. 2000. A generational mostly-concurrent garbage collector. See Hosking [2000].

Products, I. S. D. Vtune performance analyzers. http://www.intel.com/software/products/vtune.

Sobalvarro, P. 1988. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Tech. Rep. AITR-1417, MIT AI Lab. Feb. Bachelor of Science thesis.

SPECjbb2000 1998. SPECjbb2000 Java Business Benchmark. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. Available at http://www.spec.org/osg/jbb2000/.

SPECJVM98 1998. SPECjvm98 Benchmarks. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. Available at http://www.spec.org/osg/jvm98/.

Steele, G. L. 1975. Multiprocessing compactifying garbage collection. *Communications of the ACM 18,* 9 (Sept.), 495–508.

Steele, G. L. 1976. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM 19,* 6 (June), 354.

Suganuma, T., Ogasawara, T., Takeuchi, M., Kawahito, T. Y. M., Ishizaki, K., Komatsu, H., and Nakatani, T. 2000. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal 29,* 1 (Feb.).

Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. 2001. A dynamic optimization framework for a Java Just-In-Time compiler. See OOPSLA [2001], 180–194.

SUN 2003. *JSRs: Java Specification Requests. JSR 133: Java Memory Model and Thread Specification Revision.* Available at hhttp://jcp.org/jsr/detail/133.jsp.

Thomas, S., Charnell, W., Darnell, S., Dias, B. A. A., Kramskoy, J. G. P., Sextonand, J., Wynn, J., Rautenbach, K., and Plummer, W. 1998. Low-contention grey object sets for concurrent, marking garbage collection. United States Patent Application, 20020042807.

Ungar, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19,* 5 (Apr.), 157–167. 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.

White, D. and Garthwaite, A. 1998. The GC interface in the EVM. Tech. Rep. SML TR–98–67, Sun Microsystems Laboratories. Dec.