Mostly Concurrent Garbage Collection Revisited

Katherine Barabash* Yoav Ossia† Erez Petrank‡

ABSTRACT

The *mostly concurrent garbage collection* was presented in the seminal paper of Boehm et al. With the deployment of Java as a portable, secure and concurrent programming language, the mostly concurrent garbage collector turned out to be an excellent solution for Java's garbage collection task. The use of this collector is reported for several modern production Java Virtual Machines and it has been investigated further in academia.

In this paper, we present a modification of the mostly concurrent collector, which improves the throughput, the memory footprint, and the cache behavior of the collector without foiling the other good qualities (such as short pauses and high scalability). We implemented our solution on the IBM production JVM and obtained a performance improvement of up to 26.7%, a reduction in the heap consumption by up to 13.4%, and no substantial change in the (short) pause times. The modified algorithm was subsequently incorporated into the IBM production JVM.

Categories and Subject Descriptors

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

General Terms

Languages, Performance, Algorithms

Keywords

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

*IBM Haifa Research Laboratory, Mount Carmel, Haifa 31905, ISRAEL. Email: kathy@il.ibm.com

†IBM Haifa Research Laboratory, Mount Carmel, Haifa 31905, ISRAEL. Email: yossia@il.ibm.com

‡Computer Science Department, Technion – Israel Institute of Technology. Email: erez@cs.technion.ac.il. Work done in IBM Haifa Research Laboratory.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

1. INTRODUCTION

Modern SMP servers with multi-gigabyte heaps provide new challenges for garbage collection (GC). GC techniques originally designed for single processor client machines lead to unacceptable pauses when used on large servers. There is a growing need for a GC that is specifically targeted at large server configurations: 64-bit shared-memory running multi-threaded 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 kind of "server-oriented" GC include: ensuring short pause times on a multi-gigabyte heap, while maintaining high throughput and good scaling on multiprocessor hardware.

The *mostly concurrent garbage collector* was presented in the seminal paper of Boehm et al [9]. With the deployment of Java as a portable, secure and concurrent programming language, the mostly concurrent garbage collector turned out to be an excellent solution for Java's garbage collection task. The use of this collector is reported for the production JVM of IBM [27], SUN [29], and BEA WebLogic JRockit [22], and has been investigated further in academic research (e.g., [17]).

In this paper, we present a basic improvement 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).

1.1 The Mostly Concurrent Collector

The basic collector of Boehm et al [9] is a mark-and-sweep collector. It begins by marking the roots and tracing concurrently with the program run. Since the object connectivity graph is modified by the program while the collector traverses the heap, a write barrier is required to ensure correctness. The write barrier uses a set of virtual dirty bits (similar to a card marking scheme) to record locations of references in the heap that are modified by the program. These locations must be retraced by the collector to make sure that all reachable objects are marked. The repeated trace is called *card cleaning* and it may run once (or more), concurrently with the program. However, during concurrent card cleaning, more cards (or pages) may become dirty (via the write barrier or via page protection traps) and so this process may never terminate. To finish the collection, the program is stopped and all cards are cleaned. At this time, all reachable objects are guaranteed to be marked and the sweep phase may start (concurrently or during the program pause). For a more detailed description of the basic collector see Section 2.1.

Printezis and Detlefs [29] have extended this collector to run with generations, noting that the write barrier is similar. They suggest several improvements and report excellent behavior with a modern JVM. Ossia et al [27] have extended the collector to run in several parallel threads (so the pause times are more efficiently used on an SMP) and combined the incremental collector with separate low priority collector threads to better utilize the CPU. Endo et al [17] have investigated how this collector can run in an uncooperative environment while maintaining a low bound on program pauses.

1.2 This Work

In this paper, we present a basic improvement 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 would like to eliminate repetitive collector work as much as possible, and thus improve its efficiency. Second, we would like to 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 3.

1.3 Implementation and Results

We implemented our solution on top of the mostly concurrent collector that is part of the IBM production JVM 1.4.0. We used the SPECjbb2000 benchmark on an IBM 6-way pSeries server and an IBM 4-way Netfinity server. As a sanity check we also measured performance on a client application: the SPECjvm98 benchmark suite on a uniprocessor. Our measurements show 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. To summarize, we improve performance for all relevant rates of the concurrent collector; the biggest improvement is obtained for the most important concurrent rate, in which the collector runs non-intrusively. We stress that our measurements are not taken on a research system; these significant improvements were actually measured on IBM's production JVM.

1.4 Related Work

The study of concurrent (and on-the-fly) garbage collectors was initiated by Steele and Dijkstra, et al [34, 35, 11, 12]. Incremental collection was presented by Baker [6]. There exists vast literature on the development of concurrent collectors, see for example [18, 7, 8, 23, 26, 2, 24]. More recent concurrent collectors implemented on modern systems appear in [14, 13, 15, 16, 5, 20, 25, 4, 3]. Recently, the mostly concurrent collector has been attracting more attention than other collectors in this list. The reason may be that it is both effective and simple. It has been reported to be used in the production JVM of IBM [27], SUN Solaris [29], and BEA WebLogic JRockit [22].

1.5 Organization

In Section 2, we review the mostly concurrent collector algorithm and some of its properties. In Section 3, we present our algorithmic improvement. In Section 4, we specify our implementation details and report measurement results. We conclude in Section 5.

2. THE MOSTLY CONCURRENT COLLECTOR

In this section, we review the basics of the Boehm et al collector [9]. In Section 2.2 we discuss the tracing rate notion that has a major influence on the collector behavior and efficiency.

2.1 The Basic Collector

Boehm et al [9] suggest marking the heap using a separate designated collector thread running concurrently with the program threads. On a multiprocessor, this means that the program may continue to run with only a slight interruption, whereas on a uniprocessor this would mean an incremental collector: whenever the collector thread gets CPU time, it performs more collection work. However, running the trace 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 [31]. 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. 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 this phase, 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.

The card cleaning phase may run concurrently with the program threads or while the program is halted. The common practice is to run one card cleaning phase concurrently (and thus, the program continues to dirty cards during the cleaning phase) and then another phase while the program threads are stopped (and thus, the cleaning is guaranteed to eliminate all dirty cards and finish the trace). During the second (final) card cleaning phase, the roots are rescanned. Since the write barrier is not applied to the roots, we must assume conservatively that they have been modified and scan them as if they were dirty. The second card cleaning phase handles fewer cards (than the first), hence, pause time of the program is short. Sweeping may run lazily and concurrently afterwards. For more details, the reader is referred to the original paper [9].

Note that when the collection cycle terminates, some marked objects are no longer reachable. These objects will not be reclaimed and are called *Floating garbage* [21]. All floating garbage is reclaimed in the next GC cycle.

2.2 The Tracing Rate

An important parameter of the mostly concurrent collector (with respect to performance) is the *tracing rate*. This parameter signifies how fast the concurrent collector works, relative to the execution of the program. The common way to run the mostly concurrent collector is to let the collector run on an additional designated collector thread and let it compete on 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 the implementation we use as a basis for this work, the collector is incremental, i.e., the program threads execute some of the collector work within each allocation [6]. Thus, the tracing rate becomes the ratio between collector work and allocation work and may be determined by the user. Specifically, each mutator, after allocating k bytes of new objects, performs $s \cdot k$ steps of the collector code, where k is a predetermined threshold 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 collection runs for a longer time. The latter mode is more representative of a concurrent collector. 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 rate), 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).

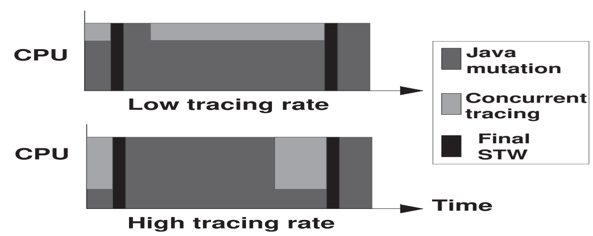

Figure 1: Characteristics of different tracing rates

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.

3. IMPROVING THE COLLECTOR

In this section, we explain our algorithmic improvement to the mostly concurrent collector. 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 marked, 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.

3.1 Reducing Tracing Work

Recall that the objects that get scanned twice 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.

3.2 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[1]. 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 (see [27] for a description of the original 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.

3.2.1 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 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
    }
}
```

[1]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.

Note that we need to 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. 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. It is easy to see that with the above order of instructions, any card is made not dirty only if at the time it is made not dirty no object on it was traced. We remark on platforms that allow instruction reordering in Section 3.3 below. This procedure turned out to be very effective at removing the dirty marks from many dirty cards, and keeping the pause times short.

3.2.2 Undirtying via allocation caches

In typical programs, a lot of the pointer modification activity on the heap happens while newly created objects are initialized. This activity creates a lot of 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 for this special case and provide measurements in Section 4 below.

Allocation caches are used in several modern JVMs [10, 1]. 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 on 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 4.6 below.

3.3 Dealing with Weakly Consistent Platforms

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 mutators 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.

4. RESULTS

In this section, we present the results of implementing our improved mostly concurrent collector. We start by describing our implementation and test environment. We then compare the results of the reference collector to those of our improved implementation. In particular, we present throughput, pause times, and heap consumption measurements on both platforms. We continue with a discussion on 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 reference collector as a stand alone. Finally, we compare the two proposed methods for undoing of the dirty state of cards.

4.1 Implementation of Our Improvements

Our reference collector is the IBM mostly-concurrent collector; this collector is part of the IBM JVM 1.4.0. The IBM JVM is a production level JVM, which employs highly optimized memory management techniques and uses an optimizing JIT compiler to deliver high performance execution of Java server applications. We refer to this reference collector as the *base* collector. A detailed description of this collector appears in [27].

We have incorporated our improvements into this reference collector. We used the same parameter tuning as in the original (highly optimized) collector. In particular, the card size used is 512 bytes. We have reduced tracing work by avoiding the tracing through dirty cards (as described in Section 3.1) and we have reduced the number of dirty cards (as described in Section 3.2). 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 3.2.2). In addition we scan the whole card table and undirty all the cards which were not traced yet (as described in Section 3.2.1) 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 to the mark-stack when it is on a dirty card during the push time. 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 4.5.

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 previous 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 3.2. There, we need a card table signifying which cards contain objects that have been traced already. However, our base 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.

4.2 Platform, Benchmarks, and Methodology

Our measurements were taken primarily on a pSeries server, with six 600 MHz PowerPC RS64 III processors (64 bit) and 5 GB of RAM running AIX 5.1. We refer to this machine as *AIX-6*. We repeated these measurements on an IBM Netfinity 7000 server, with four 550 MHz Pentium III XeonTM processors (32 bit) and 2 GB of RAM, running Windows NT 4.0. We refer to this machine as *NT-4*. The improvements on the IBM Netfinity 7000 server are smaller than those of the pSeries server, and are discussed in Section 4.3.2.

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.

Benchmarks. We used two benchmarks: *SPECjbb2000*, and the *SPECjvm98* benchmark suite.

SPECjbb2000 [32] 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* (transaction per minute). 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.

SPECjvm98 [33] 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). 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.

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.

4.3 Comparing the Base 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 base implementation and improved implementations. We present the results of runs with various tracing rates (1, 2, 4, & 8). Recall that tracing rate 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.

4.3.1 SPECjbb2000 on pSeries server (AIX-6)

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	20.4
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 1: 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.

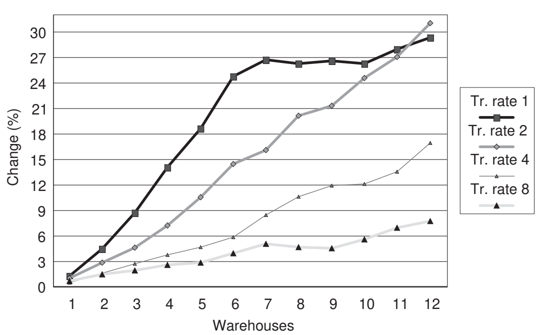

Figure 2: SPECJbb (AIX-6): Throughput change between the base collector and the improved collector, for all tracing rates

Table 1 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. The standard deviation values (in thousands of TPMs) for these results were 0.2 on average (maximum 0.5) for the base collector, and 0.3 on average (maximum 0.7) for the improved collector.

Figure 2 graphically depicts the improvement, for all warehouses and for all tracing rates.

We can see that there is a significant increase in scores from the fourth warehouse and upwards. 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 4.2 above. When switching to higher tracing rates, the improvement becomes smaller. For tracing rate 8 the average improvement is only 5.4%.

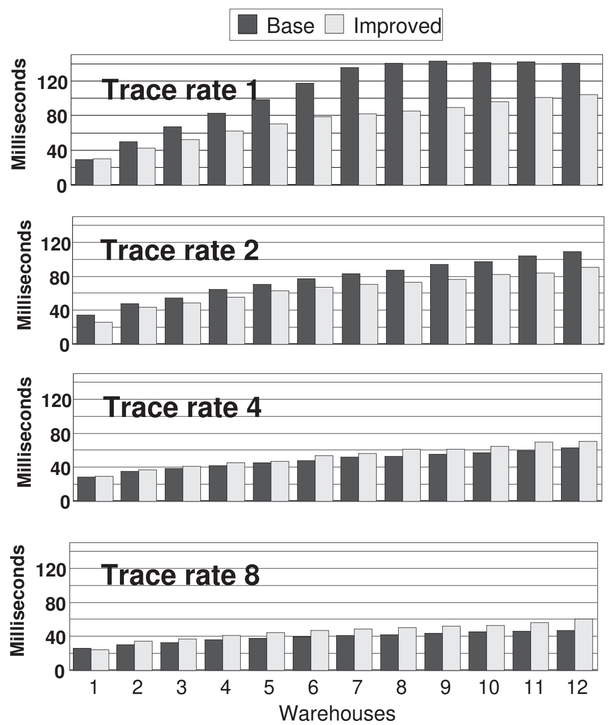

Figure 3: SPECJbb (AIX-6): Average pause time comparison for all tracing rates

In Figure 3, we present the average pause times of the improved collector and the base collector. The standard deviation values (in milliseconds) for these results were 1.5 on average (maximum 5.9) for the base collector, and 2.8 on average (maximum 4.8) for the improved 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. Figure 4 shows the maximal pause times. The pattern of change in the maximal pause times is similar to that of the average pause times. The standard deviation values (in milliseconds) for the maximal pause times results were 3.3 on average (maximum 9.8) for the base collector, and 5.5 on average (maximum 13.9) for the improved collector.

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

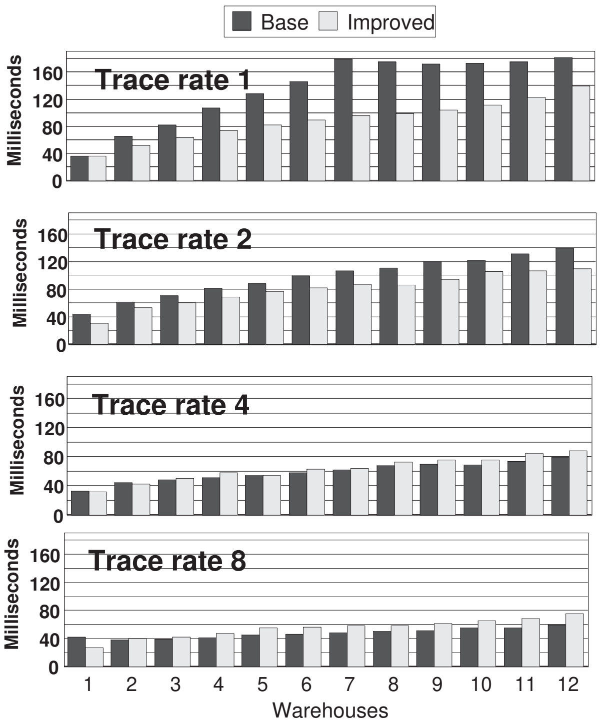

Figure 4: SPECJbb (AIX-6): Maximal pause time comparison for all tracing rates

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 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 2: 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.

Table 2 shows how our improvements reduced the heap

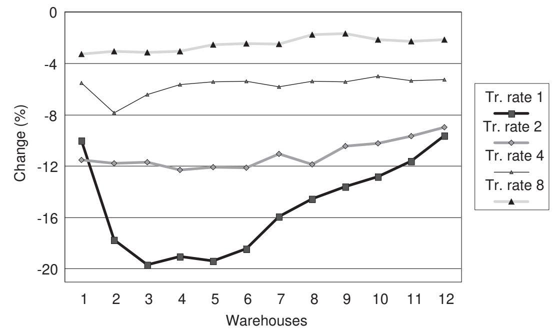

Figure 5: SPECjbb (AIX-6): Heap residency change between the base collector and the improved collector, for all tracing rates.

residency, for warehouses 2, 4, 6, 8, 10, and 12, and for tracing rates 1, 2, 4, and 8. The standard deviation values (in Mbytes) for these results were 1.0 on average (maximum 2.7) for the base collector, and 1.1 on average (maximum 3.0) for the improved collector.

Figure 5 graphically depicts the changes. The average reduction in 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.

Varying the heap size. As discussed in Section 4.1, 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 3 we compare the improvement of our algorithm in both sizes of the heap, and report the reduction in the number of garbage collections. The 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 size. The influence on the overall throughput depends on the percentage of time spent on garbage collection during program run.

4.3.2 SPECjbb2000 on Netfinity 7000 server (NT-4)

In this section, we report measurements for the 4-way IBM Netfinity 7000 server. Figure 6 shows the throughput

	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 3: 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

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% (in tracing rate 1) and the smallest average improvement is 2.8%. We believe the difference between the two machines emanates from different cache behavior.

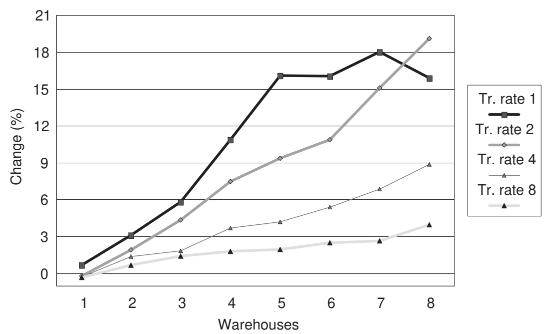

Figure 6: SPECJbb (NT-4): Throughput change between the base collector and the new collector, for all tracing rates

Figures 7 and 8 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 machine; the improved collector reduces the pause times in tracing rates 1 and 2, but slightly increases the pause times in tracing rates 4 and 8. The change in heap residency is presented in Figure 9, and is similar to the AIX-6 results. All standard deviation values of the Netfinity 7000 server results are smaller (as percentage of the actual results) than those of the AIX-6 results.

4.3.3 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 and 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.

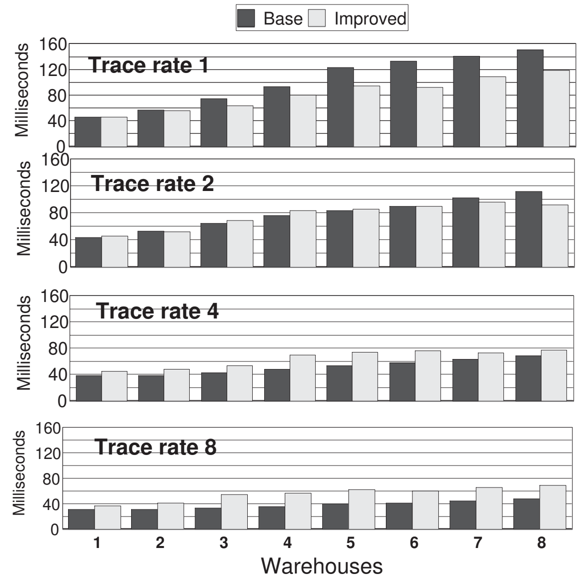

Figure 7: SPECJbb (NT-4): Average pause times comparison for all tracing rates

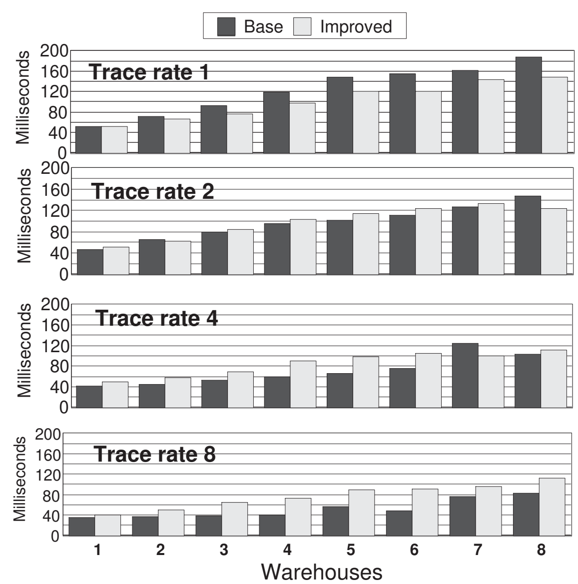

Figure 8: SPECJbb (NT-4): Maximum pause times comparison for all tracing rates

We ran all the suite on a 2.00 GHz Pentium 4 uniprocessor running Windows XP. 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

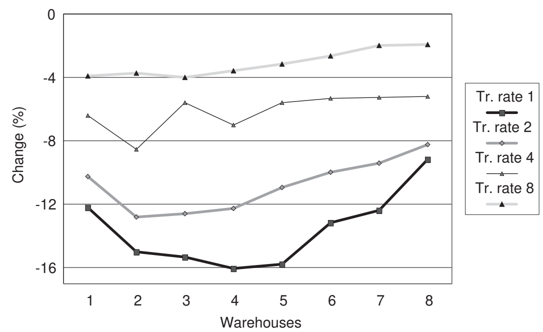

Figure 9: SPECjbb (NT-4): Heap residency change between the base collector and the improved collector, for all tracing rates

substantial allocation activity ([33]) and indeed the modified collector had no effect on the behavior of this benchmark. All results reported below were very steady. For all of them, the standard deviation did not exceed 1% of the result measured.

In Table 4, 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 for up to 6%. At the higher tracing rates, we get almost no influence on the execution time.

Run	compress	jess	raytrace	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 4: SPECjvm98: Execution times in seconds for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

In Table 5, we present the maximal pause times in milliseconds. There is no substantial difference between the collectors in terms of pause times. In Table 6, we present the average heap occupancy in Mbytes for all the benchmarks. Here, also, there is no notable change in the benchmarks behavior.

Finally, in Table 7 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	compress	jess	raytrace	db	javac	mtrt	jack
Tr1 base	3	6	4	3	31	173	9
Tr1 imp.	3	6	5	3	29	172	8
Tr2 base	3	5	5	5	20	169	6
Tr2 imp.	3	6	4	4	23	168	7
Tr4 base	4	6	4	3	14	167	5
Tr4 imp.	6	6	4	3	13	168	5
Tr8 base	6	7	5	3	14	169	4
Tr8 imp.	5	6	4	3	14	172	4

Table 5: SPECjvm98: The maximal pause times in milliseconds for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

Run	compress	jess	raytrace	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 6: SPECjvm98: The average live objects in Mbytes for all the programs, base and improved collectors, tracing rates 1, 2, 4 and 8.

Run	compress	jess	raytrace	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 7: SPECjvm98: The number of GC cycles for all the programs, tracing rates 1, 2, 4 and 8.

4.4 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 [30] 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 rampup 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 metrics we use is cache miss rate, which is the rate between cache misses and data access operations.

Table 8 shows the L2 cache miss rate for the base and improved collectors in 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 [28], 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 done 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.

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

Table 8: 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

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 runs are shown at the bottom of Table 8; 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. The standard deviation was smaller than 0.0042 in all runs, i.e., it was less than 0.32% of the actual results.

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.

4.5 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 4.2. Figure 10 presents the throughput for four different collectors: the original 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. The standard deviation values for *Base* and *Combined* were already given in Section 4.3.1. For both the *UndoDirty* and the *Restrict* collectors, these values (in thousands of TPMs) were 0.3 on average (maximum 0.6).

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.

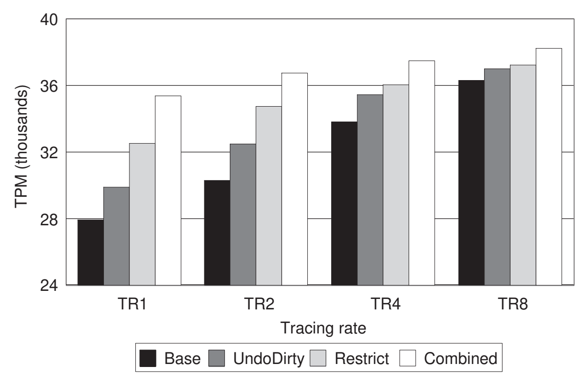

Figure 10: SPECjbb (AIX-6): Throughput comparisons between different configurations of our improvements

Figure 11 shows the average pause time values. The standard deviation values (in milliseconds) for these results were 0.8 on average (maximum 2.1) for the *UndoDirty* collector, and 4.5 on average (maximum 21.3) for the *Restrict* collector. 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 12 shows the average heap residency when using none of our improvements (the *base* version), and when using each of *UndoDirty*, *Restrict*, and *Combined*. The results are displayed for all tracing rates. The standard deviation values (in Mbytes) for these results were 0.9 on average (maximum 1.9) for the *UndoDirty* collector, and 1.0 on average (maximum 2.2) for the *Restrict* collector.

The *UndoDirty* improvement has a small impact on reduc-

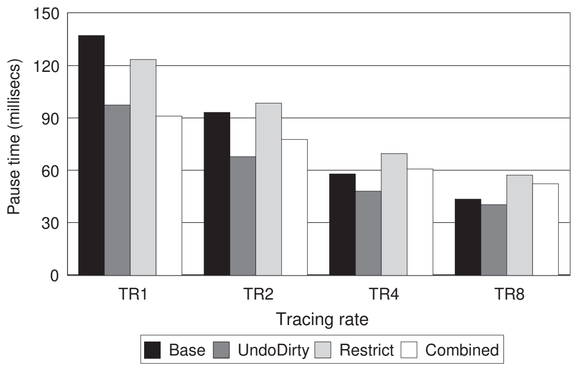

Figure 11: SPECjbb (AIX-6): Pause time comparisons between different configurations of our improvements

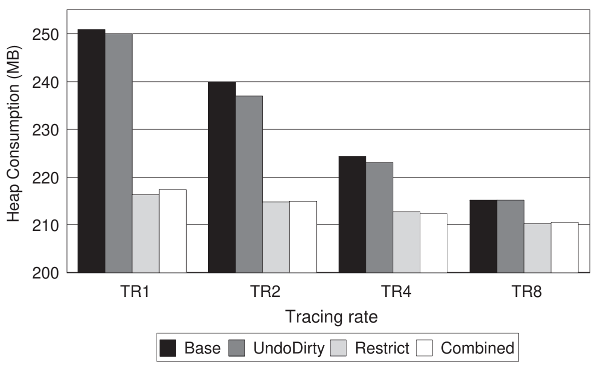

Figure 12: SPECjbb (AIX-6): Heap residency comparisons between different configurations of our improvements

ing the heap residency, whether used alone or when added to *Restrict*. The dominant method in reducing 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 base collector. When the base collector re-traced them, they contained more descendants to trace. The base 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 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: 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 4.4).

4.6 Comparing Card Undirtying Methods

In Section 3.2.2 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 3.2.1 (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.

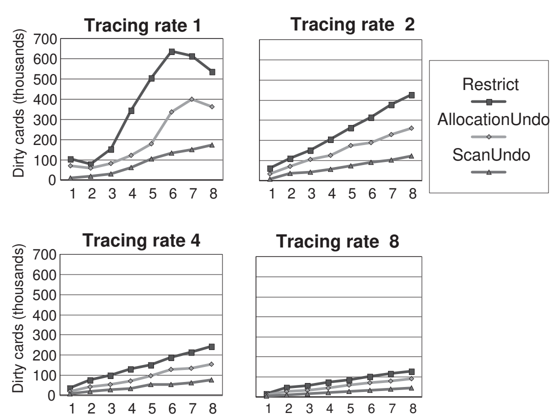

Figure 13: SPECjbb (NT-4): The number of dirty cards traced by the collector when using different Undo methods

Figure 13 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 9, 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 4.1. The change between this collector (denoted *Improved*) and *Restrict* is also presented in Table 9.

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 9: SPECjbb (NT-4): Change in throughput and pause time (relative to the base collector) with different Undo methods and tracing rates

5. CONCLUSION

In this paper, we presented two basic improvements of the mostly concurrent collector that reduce repetitive collector work and the number of dirty cards the collector needs to scan. These improvements significantly increase the throughput. Furthermore, they reduce the heap consumption (by eliminating much of the floating garbage) and they reduce the L2 cache miss rate. We obtained these improvements without impairing the other good benefits of the collector, such as its short pause times and its scalability.

We have implemented our improvements on top of the mostly concurrent collector that is part of the IBM production JVM 1.4.0. We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suite and ran it on both an IBM 6-way pSeries server and an IBM 4-way Netfinity server. Our measurements show a performance improvement of up to 26.7%, a reduction in heap consumption of up to 13.4%, and no substantial change in pause times. The performance improvement of 26.7% is obtained at a low tracing rate, when the collector runs concurrently with the mutator at a slow pace. We believe that this is the more important case, in which the collector runs non-intrusively. Allowing the collector to run at a the high tracing rate (while hindering concurrent program activity), our improvements obtain a smaller performance advantage (of around 4%). Subsequently, the improved algorithm has been incorporated into IBM's production JVM.

6. REFERENCES

[1] Java development kit version 1.2, summary of new features (performance enhancements). http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, Aneheim, CA, 2003.

[4] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the Compiler Construction: 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 185 – 199, Warsaw, Poland, May 2003. Springer-Verlag Heidelberg.

[5] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

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

[7] Mordechai Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programming. Ninth colloquium*, pages 14–22, Aarhus, Denmark, July 12–16 1982. Springer-Verlag.

[8] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

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

[10] Sam Borman. Sensible sanitation - understanding the IBM java garbage collector (part 1: Object allocation). http://www.ibm.com/developerworks/ibm/library/i-garbage1.

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

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

[13] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor

systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

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

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

[16] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Hosking [19].

[17] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Reducing pause time of conservative collectors. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 12–24, Berlin, June 2002. ACM Press.

[18] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

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

[20] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.

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

[22] BEA WebLogic JRockit. The server jvm. http://www.bea.com/products/weblogic/server/jrockit_wp_052303_final.pdf.

[23] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[24] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[25] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

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

[27] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.

[28] David Patterson and John Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufman, 1990.

[29] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [19].

[30] Intel Software Development Products. Vtune performance analyzers. http://www.intel.com/software/products/vtune.

[31] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

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

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

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

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