An on-the-fly Reference Counting Garbage Collector for Java

Yossi Levanoni [*] Erez Petrank [†]

ABSTRACT
Reference counting is not naturally suitable for running on multiprocessors. The update of pointers and reference counts requires atomic and synchronized operations. We present a novel reference counting algorithm suitable for a multiprocessor that does not require any synchronized operation in its write barrier (not even a compare-and-swap type of synchronization). The algorithm is efficient and may compete with any tracing algorithm.

We have implemented our algorithm on SUN's Java Virtual Machine 1.2.2 and ran it on a 4-way IBM Netfinity 8500R server with 550MHz Intel Pentium III Xeon and 2GB of physical memory. It turns out that our algorithm has an extremely low latency and throughput that is comparable to the mark and sweep algorithm used in the original JVM.
Keywords: Runtime systems, Memory management, Garbage collection, Reference counting.

1. INTRODUCTION
Automatic memory management is well acknowledged as an important tool for a fast development of large reliable software. It turns out that the garbage collection process has an important impact on the overall runtime performance. The amount of time it takes to handle allocation and reclamation of memory spaces may reach as high as 30% of the overall running time for realistic benchmarks. Thus, a clever design of efficient memory management and garbage collector is an important goal in today's technology.

[*]Microsoft Corporation. Most of this work was done while the author was at the Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: ylevanon@microsoft.com.

[†]Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: erez@cs.technion.ac.il. This research was supported by the Coleman Cohen Academic Lecturship Fund, by the Technion V.P.R. Fund - Steiner Research Fund, and by the Fund for the Promotion of Research at the Technion.

1.1 Automatic memory management on a multiprocessor
In this work, we concentrate on garbage collection for multiprocessor machines. Multiprocessor platforms have become quite standard for server machines and are also beginning to gain popularity as high performance desktop machines. Many well studied garbage collection algorithms are not suitable for a multiprocessor. In particular, many collectors (Among them the collector supplied with Javasoft's Java Virtual Machine) run on a single thread after all program threads have all been stopped (the so-called *stop-the-world* concept). This causes bad processor utilization, and hinders scalability.

In order to make better use of a multiprocessor, concurrent collectors have been presented and studied (see for example, [5, 35, 36, 16, 2, 12, 13, 7, 18, 30, 17]). A concurrent collector is a collector that does most of its collection work concurrently with the program without stopping the program threads. Most of the concurrent collectors need to stop all program threads at some point during the collection, in order to initiate and/or finish the collection, but the time the mutators must be in a halt is short.

Stopping all the threads for the collection is an expensive operation by itself. Usually, the program threads cannot be stopped at any point. Rather, they should be stopped at *safe points* at which the collector can safely determine the reachability graph and properly reclaim unreachable objects. Thus, each thread must wait until the last of all threads cooperate and come to a halt. This hinders the scalability of the system, as the more threads there are the more delay the system suffers. Furthermore, if the collector is not running in parallel (which is usually the case), then during the time the program threads are stopped, only one of the processors is utilized.

Therefore, it is advantageous to use *on-the-fly* collectors [16, 18, 17]. On-the-fly collectors never stop the program threads simultaneously. Instead, each thread cooperates with the collector at its own pace through a mechanism called (soft) handshakes.

We remark that another alternative for an adequate garbage collection on a multiprocessor is to perform the collection in parallel (see for example [24, 11, 29, 25, 20, 27]. We do not explore this avenue further in this work.

1.2 Reference counting on a multiprocessor
Reference counting is a most intuitive method for automatic storage management. As such, systems using reference count-

ing were implemented starting from the sixties (c.f. [10].) The main idea is that we keep for each object a count of the number of references that reference the object. When this number becomes zero for an object o, we know that o can be reclaimed. At that point, o is added to the free list and the counter of all its predecessors (i.e., the objects that are referenced by the object o) are decremented, initiating perhaps more reclamations.

Reference counting seems very promising to future garbage collected systems. Especially with the spread of the 64 bit architectures and the increase in usage of very large heaps. Tracing collectors must traverse all live objects, and thus, the bigger the usage of the heap (i.e., the amount of live objects in the heap), the more work the collector must perform. Reference counting is different. The amount of work is proportional to the amount of work done by the user program between collections plus the amount of space that is actually reclaimed. But it does not depend on the space consumed by live objects in the heap.

The study and use of reference counting on a multiprocessor has not been extensive and thorough as the study of concurrent and parallel tracing collectors. The reason is that reference counting has a seemingly inherent problem with respect to concurrency: the update of the reference counts must be atomic since they are being updated by all program threads. Furthermore, when updating a pointer, a thread must know the previous value of the pointer-slot being updated, in spite of many such writes occuring in parallel. Otherwise, a confusion occurs in the bookkeeping of the reference counts. Thus, the naive solution requires a lock on any update operation. More advanced solutions have recently reduced this overhead to a compare-and-swap operation, which is still a time consuming write-barrier.

1.3 This work

In this work, we present a new on-the-fly reference counting garbage collector with extremely fine synchronization. In particular, we avoid any synchronization in the write barrier. We proceed with an overview on the novel ideas in our algorithm, with which we could obtain this advantage. A detailed precise description of these ideas is given in the rest of the paper.

Our algorithm, following Deutsch and Bobrow's *Deferred Reference Counting* [14], does not keep account of changes to local pointers (in stack and registers) since keeping this account is too expensive. Instead, it only keeps account of pointers in the heap (Denoted *heap reference count*). Once in a while (when garbage collection is required), the collector inspects all objects with heap reference count zero. Those not referenced by the roots may be reclaimed. Our first observation is that many more updates of the reference counts are redundant and may be avoided. Consider a pointer slot that, between two garbage collections is assigned the values $o_0, o_1, o_2, \ldots, o_n$ for objects $o_0, \ldots, o_n$ in the heap. There are $2n$ updates of reference counts made for these assignments: RC(o_0)- -, RC(o_1)++, RC(o_1)- -, RC(o_2)++, ..., RC(o_n)++. However, only two are required: RC(o_0)- - and RC(o_n)++. Building on this observation, we note that in order to update all reference counts of all objects before a garbage collection, it is enough to know which pointer slots have been modified between the collections, and for each such slot, we must be able to tell what its value in the previous garbage collection was, and what its current value is.

In our algorithm, we keep a record of all pointer slots that have been modified. We also keep the "old" value that existed in the slot before it was first modified. It may seem that we have a problem to obtain this value in a concurrent setting, and indeed, special care must be used to make sure that this value is properly registered. However, we do that without any synchronization operation. We denote the algorithm resulting from the discussion so far as *the snapshot algorithm*. The exact details of the snapshot algorithm are presented in Section 3 below.

Next, we look at the collection itself. The naive implementation of the above approach is to stop all the threads and read the values currently kept in all modified slots. This translates to taking a snapshot of the heap (or only a snapshot of the interesting fields in the heap). Such an approach does not allow full concurrency of the collector (it is not on-the-fly), although it is sound. In order to make the collector on-the-fly, we borrow ideas from the world of distributed computing. When taking a snapshot in a distributed environment, one does not stop all the computers over the distributed environment. Instead, one takes a snapshot of each computer at a time, but as the snapshots are being recorded, special care is taken to avoid confusion due to the non-instantaneous view. For example, all messages between computers are recorded as well. In our case, we will use a similar solution. We will take a non-instantaneous view of the interesting pointer slots in the heap, but while checking these slots, we will use a special mechanism to avoid confusion. We denote this view of the heap *the sliding view*. The sliding view algorithm is briefly described in Section 4. Due to lack of space, we provide the details of the sliding view algorithm as well as implementation details, the allocator mechanism, and a proof of correctness for the algorithm in the full paper [28].

1.4 Cycle collection

A major disadvantage of reference counting is that it does not collect cycles. We have chosen to collect cycles with an on-the-fly mark-and-sweep collector. The Mark-and-sweep algorithm is run seldom to collect cycles and restore stuck reference counts. (Like in [33, 42, 38, 8, 8], we use only two bits for the reference count and thus, stuck counters are created on the fly, and are restored by the mark-and-sweep algorithm.)

We use a novel on-the-fly mark-and-sweep collector that we have designed especially for our reference counting algorithm. Note that it is quite natural to base a mark-and-sweep collector on a snapshot of the heap. The marking can be done on the snapshot view of the heap, and since unreachable objects remain unreachable, changes in the heap do not foil the collection of garbage. We adapt this basic idea to the sliding view notion, thus obtaining a tracing collector perfectly fitting into our setting.

We do not elaborate on the mark-and-sweep collector in this paper. The algorithm is described in our full paper [28]. All measurements of throughput and latency in this paper are reported for the reference count collector run most of the times and the on-the-fly mark-and-sweep run seldom.

1.5 Memory consistency

The algorithm presented in the paper requires a sequentially consistent memory. However, three simple modifications can make the algorithm suitable for platforms that do not provide sequentially consistent memory. We remark that we did not encounter problems in the runs we made on the Intel platform. We list the modifications required and discuss their cost in Section 5 below.

1.6 Implementation

We have implemented our algorithm on SUN's Java Virtual Machine 1.2.2 and ran it on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB of physical memory. We used the two standard Java multithreaded benchmarks: SPECjbb2000 and the mtrt benchmark from SPECjvm98. These benchmarks are described in detail in SPEC's Web site[37]. It turns out that our algorithm has an extremely low latency. It improves over the original JVM by two orders of magnitude. As for efficiency, the JVM with our reference counting collector bits the original JVM by up to 10% improvement in the overall running time with the mtrt benchmark. As for SPECjbb, if we allow a large maximum heap (which is the target of our collector), then our collector slightly improves over the running time of the original JVM. With smaller heaps the original JVM does better than ours for SPECjbb.

1.7 Results

In Section 6 we report the measurements we ran with our collector. We note that the throughput is basically the same as the original tracing collector. For the multithreaded mtrt benchmark our collector improved the throughput up to a 10% difference in the overall running time. For SPECjbb00 our collector had a throughput similar to the original collector. In terms of latency, we got one of the best reported results in the literature. The measure we report is the one reported by SPECjbb00: the maximum time it takes to complete a transaction. It is the same measure that was reported by Domani et. al. [19]. But whereas Domani et. al. ran an IBM JVM with a JIT compiler, our collector gave similar maximal transaction times (20-120ms depending on the number of threads) with the assembler loop (and no compilation). Incorporating our collector with a JIT compiler would no doubt improve the transaction time substantially. We remark that our result and the result of Domani et. al. are incomparable with those of Bacon et. al. [3] since Bacon et. al. report the exact pause times, measured by the Jalapeno JVM.

1.8 Related work

The traditional method of reference counting, was first developed for Lisp by Collins [10]. It was later used in Small talk-80 [22], the AWK [1] and Perl [39] programs. Improvements to the naive algorithm were suggested in several subsequent papers. Weizman [40] studied ameliorating the delay introduced by recursive deletion. Deutsch and Bobrow [14] eliminated the need for a write barrier on local references (in stack and registers). This method was later adapted for Modula-2+ [12]. Further study on reducing work for local variables can be found in [6] and [31]. Several works [33, 42, 38, 8, 8] use a single bit for each reference counter with a mechanism to handle overflows. The idea being that most objects are singly-referenced, except for the duration of short transitions.

DeTreville [12] describes a concurrent multiprocessor reference counting collector for Modula-2+. This algorithm adapts Deutsch and Bobrow's ideas of deferred reference counting and transaction log for a multiprocessor system. However, the update operation is done inside a critical section that uses a single central lock. This implies that only a single update can occur simultaneously in the system, placing a hard bound on its scalability.

Plakal and Fischer in [32] propose a collection method based on reference counting for architectures that support explicit multi-threading on the processor level. Their method requires co-routine type of cooperation between the program thread and corresponding "shadow" collector threads and therefore is probably not suitable for stock SMPs, as SMP architectures do not support this kind of interaction in a natural and efficient manner.

Algorithms that perform garbage collection using a snapshot of the heap appear in [21, 43]. In terms of synchronization requirements and characteristics our work is similar to that of Doligez-Leroy-Gonthier [18, 17] in the sense that we use only fine synchronization, we never require a full halt of the system (the mutators are required to cooperate a few times per collection cycle). In our tracing algorithm we have used an object sweeping method similar to that presented in [18, 17].

1.9 The work of Bacon et. al.

Independently of this work, Bacon et. al. [3, 4] have built an on-the-fly reference counting algorithm appropriate for a multiprocessor. Their work presents a big step towards making reference counting practical for servers. Since our work and theirs are closely related (both introduce an on-the-fly reference counting collector with extremely low pause times), we would like to elaborate on the relations between these two collectors.

Reducing synchronization. A naive approach to multiprocessor reference counting requires at least three compare-and-swaps in the write barrier. One for the update of the pointer and two for the updates of the two reference count. DeTreville [12] has used a lock on each update to make sure that no two pointer updates are executed concurrently. Bacon et. al. [3] made a significant step into exploiting multiprocessor concurrency by reducing the number of synchronizing operations to a single compare-and-swap. While significantly reducing the cost of synchronization, their write barrier still contains a compare-and-swap for each pointer update. In our work, using our novel sliding view idea, we have managed to completely eliminate synchronization in the write barrier. This major improvement is one of the more important contributions of this work.

Improving throughput: It is not possible to compare the throughput of the two collectors since they have been run on different platforms and compared against different base JVM's. Our collector demonstrate a throughput which is comparable to the original SUN JVM. Bacon et. al. [3] report a reduction of around 10% in the throughput of their JVM compared to the original Jalapeno JVM.

Improving latency. With respect to pause times, the measured results provided by Bacon et. al. are incomparable with ours. Bacon et. al. used the Jalapeno JVM to measure the exact pause times. Unfortunately, we did not have the means to get such a measure, which is provided by the Jalapeno JVM. Instead, like in [19], we use the report output by the SPECjbb00 benchmark. It reports the maximum time it takes to complete a transaction. Our results show excellent latency with respect to previous reports of this nature. We believe that measuring the maximum time for a transaction is more meaningful than the shortest garbage collection pause, because it takes into account the slowdown imposed by the collector also. E.g., a collector with very frequent but short pause times might be less good than a collector with slightly longer but much less frequent pause times. This is an important issue in concurrent and incremental collection.

Sequential memory consistency vs. floating garbage. Our algorithm requires sequential memory consistency. However, as explained in section 5 below, this limitation can be overcome at a negligible cost. The algorithm in [3] can run on any platform. But the cost of this robustness is floating garbage. In their algorithm, an unreachable object cannot be collected unless it has been unreachable for two consecutive collections. Thus, although memory coherence is not an issue, the drag time of objects [34] increases significantly, resulting in a substantial amount of floating garbage compared to our collector.

Collecting cycles. Finally, the two papers take different avenues for collecting cycles. Bacon and Rajan [4] provide a novel on-the-fly cycle detection. Their algorithm can be run with any algorithm and in particular with ours, and it demonstrates that the entire collection can be run with a pure reference counting algorithm. In contrast to their approach, we have chosen to develop an on-the-fly mark-and-sweep collector that exploits the sliding view mechanism and uses the same data structure as the reference counting algorithm. This mark-and-sweep collector is run seldom in order to collect cycles and restore stuck reference counts (see below). Our on-the-fly mark-and-sweep collector is a stand-alone collector that can be run also without the reference counting algorithm, and is interesting on its own. It is difficult to compare the efficiency of the two approaches, and since the algorithm is run seldom, such a comparison is not so interesting. However, we note that the use of a tracing collector allows saving space. We use two bits for the reference counts. Reference counts that exceed the value of 2, get stuck and are restored by the tracing collector. In Bacon et. al., it is necessary to keep the counters correct, since there is no mechanism to restore corrupted counts. Thus, more space is used for the counter, and a cache is used to place objects whose reference counts exceed the maximum allowed value.

1.10 Organization

In Section 2 we present definitions and terminology to be used in the rest of the paper. In Section 3 we present our Snapshot Algorithm. In Section 4 we present the sliding view algorithm. In Section 5 we discuss adaptation of the algorithm to platforms that do not provide sequentially consistent memory and in Section 6 we present performance results. We conclude in 7.

2. SYSTEM MODEL, DEFINITIONS

For an introduction on garbage collection and memory management the reader is referred to [26]. We assume the reader is familiar with the concepts such as heap, object, roots, reachability, etc. Note that in a multithreaded environment each thread has its own roots on top of the global roots.

Fields in objects in the heap that hold references are called *heap reference slots* but most of the time we will just call them slots. We will count references to objects by summing over all slots in the heap. We will not consider the threads local stack and registers for the count. We assume all slots are initialized with a **null** pointer. We denote the reference count associated with an object o by $o.rc$.

Coordination of threads. We assume that the garbage collector thread may *suspend* and subsequently *resume* user threads. When a thread is suspended, the collector may inspect and change its local state with the effects taking place after the thread is resumed. In our algorithm, we assume threads are not stopped during execution of *protected* code. In particular, in our algorithm, the only pieces of code which are protected are procedures **Update** and **New**, which are in charge of updating heap-slots and allocating new objects, respectively.

3. THE SNAPSHOT ALGORITHM

For clarity of presentation, we start with an intermediate algorithm called *the snapshot algorithm*. Here, we present the ideas required for an efficient write barrier with no synchronization. In this intermediate algorithm, the threads are stopped for part of the collection. The length of this pause is not too long (the bottle neck is clearing a bitmap with dirty flags for all objects in the heap), but it is long enough to hinder scalability on a multiprocessor. In Section 4, we extend this intermediate algorithm making it on-the-fly.

The idea, as presented in Section 1.3, is based on computing differences between heap snapshots. The algorithm operates in cycles. A cycle begins with a collection and ends with another. Let us describe the collector actions during cycle k (throughout the paper we let the subscript k denote the number of a garbage collection cycle).

Our first goal is to record all pointer slots in the heap that have changed since the previous collection cycle $k-1$. We let the mutators do the recording with a write-barrier. In order to avoid recording slots again and again, we keep a dirty flag for each such slot. When a mutator updates a pointer, it checks the dirty bit. If it is clear, the mutator sets the dirty bit and records the slot into a local buffer. The recorded information is the address of the slot and its value before the current modification. Recording is done in a local buffer with no synchronization.

When a collection begins, the collector starts by stopping all threads and marking as local all objects referenced directly by the threads' stack at the time of the pause. Next, it reads all the threads' local buffers (in which modified slots are recorded), it clears all the dirty bits and it lets the mutators resume. After the mutators resume, the collector updates all the heap reference counts to reflect their values at the time of the pause. (Recall that the heap reference count is the number of references to the object from other objects in the heap.) The algorithm for this update is presented and justified in the remainder of this section. However, assuming that the heap reference counts are properly updated, the collector may reclaim all objects whose reference counts drop to

```
Procedure New(size: Integer) : Object
begin
1.      Obtain an object $o$ from the allocator,
        according to the specified $size$.
        // add $o$ to the thread local ZCT.
2.      $New_i := New_i \cup \{o\}$
3.      return $o$
end
```

Figure 1: Mutator Code: for Allocation

```
Procedure Update(s: Slot, new: Object)
begin
1.      local $old$ := read($s$)
        // was $s$ written to since the last cycle ?
2.      if $\neg Dirty(s)$ then
            // ... no; keep a record of the old value.
3.          $Buffer_i[CurrPos_i] := \langle s, old \rangle$
4.          $CurrPos_i := CurrPos_i + 1$
5.          $Dirty(s)$ := true
6.      write($s$, $new$)
end
```

Figure 2: Mutator Code: Update Operation

zero by this update and are not marked local. As usual, the reference counts of objects referenced by reclaimed objects are decremented and the reclamation proceeds recursively. A standard zero-count table (ZCT) [14] keeps track of all objects whose reference count drops to zero at any time. These objects are candidates for reclamation. We remark that whenever an object is created it has a zero heap reference count. Thus, all created objects are put in (a local) ZCT upon creation. The code for the create routine appears in Figure 1.

It remains to discuss updating the reference counts according to all modified slots between collection $k-1$ and k. As explained in Section 1.3, for each such slot s, we need to know the object O_1 that s pointed to at the pause of collection $k-1$ and the object O_2 that s points to at the pause of collection k. Once these values are known, the collector decrements the reference count of O_1 and increments the reference count of O_2. When this operation is done for all modified slots, the reference counts are updated and match the state of the heap at the kth collection pause.

We go on now and describe how to obtain the addresses of objects O_1 and O_2. We start with obtaining O_1. If no race occured when the slot s was first modified during this cycle, then the write barrier recorded the address of O_1 in the local buffer. It is the value that s held before that (first) modification. But suppose a race did occur between two (or more) threads trying to modify s. The code of the write barrier appears in Figure 2. If one of the updating threads sets the dirty flag of s before any other thread reads the dirty flag, then only one thread records this address and the recording will properly reflect the value of s at the $k-1$ pause. Otherwise, more than one thread finds the dirty bit clear. Looking at the code, each thread starts by recording the old value of the slot, and only then it checks the dirty bit. On the other hand, the actual update of s occurs after the dirty bit is set. Thus, if a thread detects a clear dirty bit, then it is guaranteed that the value it records is the value of s before any of the threads has modified it. So while several threads may record the slot s in their buffers, all of them must record the same (correct) information. To summarize,

```
Procedure Collection-Cycle
begin
1.      Read-Current-State
2.      Update-Reference-Counters
3.      Read-Buffers
4.      Fix-Undetermined-Slots
5.      Reclaim-Garbage
end
```

Figure 3: Collector Code

in case a race occurs, it is possible that several threads record the slot s in their local buffers. However, all of them record the same correct value of s at the $k-1$st pause. When collecting the local buffers from all threads, care is taken to avoid multiple records of a slot. (For implementation details, see the full paper [28]). We conclude that the address of object O_1 can be properly obtained.

We now explain how the collector obtains the address of O_2, the object that s references at the pause of collection k. Note that at the time the collector tries to obtain this value the threads are already running after the k pause. The collector starts by reading the current value of s. It then reads s's dirty flag. If the flag is clear then s has not been modified since the pause of collection k and we are done. If the dirty bit of s is set, then it has been modified. But if it has been modified, then the value of s at pause k is currently recorded in one of the threads local buffers. This value can be obtained by searching the local buffers of all threads. Note that the threads need not be stopped for peeking at their buffers. We know that this slot has a record somewhere and it will not be changed until the next $(k+1)$ collection.

The collector operation is given in Figure 3. In Read-Current-State the collector stops the threads, takes their buffers, mark objects directly referenced from the roots as local, takes all local ZCT's (including records of newly created objects), and clears all the dirty marks. The threads are then resumed. While the threads run, the collector updates the reference counts as much as it can (excluding slots that were modified since the pause). It then reads the current buffers of the threads (without stopping them) to get information on all slots modified since the last pause, and finish updating the reference counts. Finally, it (recursively) reclaims all objects with zero reference count that are not marked local.

We remark that for the correctness of the algorithm, it is required that the stop of all mutators does not stop any of the mutators in the middle of a pointer modification. We do not elaborate on implementation issues. More discussion of this intermediate algorithm together with full code is given in the full paper [28]. The main goal of this section is to explain the write barrier that avoids synchronization. We now turn to the on-the-fly algorithm.

4. THE SLIDING VIEW ALGORITHM

In the snapshot algorithm we have managed to execute a major part of the collection while the mutators run concurrently with the collector. The main disadvantage of this algorithm is the halting of the mutators in the beginning of the collection. During this halt all threads are stopped while the collector clears the dirty flags and receives the mutators' buffers and local ZCTs. This halt hinders both efficiency, since only one processor executes the work and the rest are idle, and scalability, since more threads will cause more delays. While efficiency can be enhanced by parallelizing the flags' clearing phase, scalability calls for eliminating complete halts from the algorithm. This is indeed the case with our second algorithm, which avoids grinding halts completely.

A handshake [18, 17] is a synchronization mechanism in which each thread stops at a time to perform some transaction with the collector. Our algorithm uses four handshakes. Thus, mutators are only stopped one at a time, and only for a short interval, its duration depends on the size of mutators' local states.

In the snapshot algorithm we had a fixed point of time, namely, when all mutators were stopped, for which we computed the reference counts of all objects. Thus, it was easy to claim that if an object has a zero heap reference count at that time, and it is not local at that time, then it can be reclaimed. By dispensing with the complete halting of threads we no longer have this fixed point of time. Rather, we have a fuzzier picture of the system, formalized by the notion of a *sliding view* which is essentially a non-atomic picture of the heap. We show how sliding views can be used instead of atomic snapshots in order to devise a collection algorithm. This approach is similar to the way snapshots are taken in a distributed setting. Each mutator at a time will provide its view of the heap, and special care will be taken by the system to make sure that while the information is gathered, modifications of the heap do not foil the collection.

4.1 Scans and sliding views

Pictorially, a scan σ and the corresponding sliding view V_σ can be thought of as the process of traversing the heap along with the advance of time. Each pointer slot s in the heap is probed at time $\sigma(s)$; $V_\sigma(s)$ is set to the value of the probed pointer. For an object o and a sliding view V_σ we define the *Asynchronous Reference Count of o with respect to* V_σ to be the number of slots in V_σ referring to o: $ARC(V_\sigma; o) \overset{\text{def}}{=} |V_\sigma^{-1}(o)|$

Sliding views can be obtained incrementally, which will get us the benefit of not having to stop all mutators simultaneously in order to compute the view. But in order to use this information to safely collect garbage we need to be careful. Trying to use the snapshot algorithm when we are only guaranteed that logging and determining reflects some sliding view is bound to fail. For example, the only reference to object o may "move" from slot s_1 to slot s_2, but a sliding view might miss the value of o in both s_1 (reading it after modification) and s_2 (reading it before modification).

We avoid these problems via a *snooping* mechanism. While the view is being read from the heap, we let the write-barrier mark any object that is assigned a new reference in the heap. We mark these objects as local, thus, preventing them from being collected in this collection cycle. (Recall that objects directly referenced by the roots are marked local to prevent collecting them because of a zero heap reference count.) We remark that there is nothing preventing the collection of these snooped objects in the next cycle. Assuming this *snooping* mechanism throughout the scan of the heap, we observe the following.

Observation: If object o has $ARC(V_\sigma; o) = 0$, i.e., it is not referenced by any pointer slot in the heap as reflected by the sliding view, and if object o is not referenced directly by the roots of the threads after the scan was completed, and if object o has not been marked local by the snooping mechanism while the heap (and the roots) were being scanned, then at the time the heap scan is completed, object o is unreachable and may be reclaimed.

Proof idea: If the object is referenced by a heap slot in the end of the scan, then this slot has either been pointing to this object when the scan of the heap read it, or it has been written to that slot later. Both cases do not fall in the criteria of unreachable objects in the observation. Finally, if no reference is written into the heap while the roots are scanned, and there is no reference from the roots to this object, then it is unreachable. Here we rely on the fact that a mutator is stopped while reading its stack, so no pointer may move while the thread stack is being read; and furthermore, in Java, a reference cannot be moved from the stack of one thread to another without being written to the heap. The full argument is given in the full paper.

Keeping this observation in mind, we are ready to present the sliding view algorithm. We break the description into two. We first describe (in Section 4.2 below) how a sliding view of the heap may be used to reclaim unreachable objects. We call it a *generic* algorithm since it may use any mechanism for obtaining the sliding view. Then, we describe how the reference counts of all objects can be updated according to a sliding view that is not actually taken. This is an extension of the ideas in the snapshot algorithm, still preserving the light write barrier.

4.2 Using sliding views to reclaim objects

Based on the above observation we present a generic garbage collection algorithm:

1. Each thread T_i has a flag, denoted $Snoop_i$ which signifies whether the collector is in the midst of constructing a sliding view.

2. Mutator T_i executes a write barrier in order to perform a heap slot update. The generic algorithm requires that after the store proper to the slot is performed, i.e., the reference to o is written into slot s, the thread would probe its $Snoop_i$ flag and, if the flag is set, would mark o as *local.* We call this probing of the $Snoop_i$ flag and the subsequent marking *snooping.* Any specific implementation of the generic algorithm may require additional steps to be taken as part of the write barrier.

3. As usual, threads may not be suspended in the midst of an update.

4. A collection cycle contains the following stages:

1. the collector raises the $Snoop_i$ flag of each thread. This indicates to the mutators that they should start snooping.

2. the collector computes, using an implementation-specific mechanism, a scan σ and a corresponding sliding view, V_σ, concurrently with mutators' computations. The actual manner using which the collector computes V_σ

is immaterial, it's just important that it arrives at a sliding view.

3. each thread is then suspended (one at a time), its $Snoop_i$ flag is turned off and every object directly reachable from it is marked *local*. The thread is then resumed.

4. now, for each object o we let $o.rc := ARC(V_\sigma; o)$.

5. at that point, we can deduce that any object o that has $o.rc = 0$ and that was not marked *local* is garbage.

Consider an object o with $ARC(V_\sigma; o) = 0$ and which is not marked *local*. Since for each thread the $Snoop_i$ flag is set for the entire duration of the sliding view computation we conclude that o's true reference count at the end of the heap scan is zero as well. It may be, however, that o is *directly reachable* from some thread at that time. Nevertheless, since no local reference to o was observed by any thread when its state was scanned (in stage (3) of the collector) and it was not "snooped" prior to it, any thread which possessed such a local reference must have discarded it prior to responding to the handshake of stage (3) without ever raising the heap reference count of o above zero. We conclude that by the time the handshake of stage (3) ends, o is garbage.

The snooping mechanism may lead to some floating garbage as we conservatively do not collect objects which are marked *local*, although such objects may become garbage before the cycle ends. However, such objects are bound to be collected in the next cycle.

We have termed this algorithm "generic" since the mechanism for computing the sliding view is unspecified. We next present an algorithm for updating the reference counts for an implicitly defined sliding view of the heap. When the algorithm is done, it holds for each object that $o.rc = ARC(V; o)$, where V is the sliding view that was constructed implicitly. Since we are not interested in the sliding view itself but rather on its manifestation through the rc fields, this implicit computation suffices for collection purposes.

4.3 Obtaining the sliding view

We use four handshakes during the collection cycle. The sliding view associated with a cycle spans from the beginning of the first handshake up to the end of the third handshake. The "sampling" timing of each individual slot in the scan is determined by mutators' logging regarding the slot. The snooping flags are raised prior to the first handshake and are turned off at the forth handshake. Thus, they are set for the entire duration of the scan, adhering to the snooping requirement of the generic sliding view algorithm.

Any slot which is changed between cycles is logged along with its value in the most recent sliding view, hence there is no loss of information regarding "old" values. It turns out from our analysis that inconsistent logging of slots is only possible between responding to the first and third handshakes of a cycle. Just after the fourth handshake, the collector employs a consolidation mechanism to consolidate any inconsistently logged slot into a fixed value. No thread would log a conflicting value after responding to the fourth handshake, hence no inconsistencies will be visible in the history for the next cycle.

```
Procedure Update(s: Slot, new: Object)
begin
1.     Object $old$ := read($s$)
2.     if $\neg Dirty(s)$ then
3.         $Buffer_i[CurrPos_i] := \langle s, old \rangle$
4.         $CurrPos_i := CurrPos_i + 1$
5.         $Dirty(s) :=$ true
6.     write( $s$, $new$)
7.     if $Snoop_i$ then
8.         $Locals_i := Locals_i \cup \{new\}$
end
```

Figure 4: Sliding View Algorithm: Update Operation

In addition, the collector always consolidates any slot which has been logged between the first and third handshakes, so there is no risk that the collector would use one value of a slot (before it is consolidated), and that value will be modified later by the consolidation mechanism. Hence the collector and mutators always "agree" on the values of slots in the sliding view.

We refer the reader to the full version in [28] where we systematically define the sliding view associated with a cycle and prove its properties. We now present the algorithm and the code.

4.4 Mutator's code

Mutators use the write barrier of the snapshot algorithm with the additional snooping and marking added after the store proper (see procedure **Update** in figure 4). Object creation is unchanged from the snapshot algorithm.

4.5 Collector's code

We now go over the main steps of the collection cycle. The code for each step is provided as well.

1. Signaling snooping. The collection starts with the collector raising the $Snoop_i$ flag of each thread, signaling to the mutators that it is about to start computing a sliding view.

2. Reading buffers (first handshake). during the handshake threads' buffers are retrieved and then are cleared. (These are the same thread buffers as in the first (snapshot) algorithm.) The slots which are listed in the buffers are exactly those slots that have been changed since the last cycle. However, in the sliding view scenario this notion requires more care. The meaning of "changing" in this asynchronous setting is defined as follows. A slot is changed during cycle k if some thread changed it after responding to the first handshake of cycle k and before responding to the first handshake of cycle $k+1$.

Steps (1) and (2) are carried out by procedure **Initiate-Collection-Cycle** (figure 5).

3. Clearing. The dirty flags of the slots listed in the buffers are cleared. Note that the clearing occurs *while the mutators are running*. This step is carried out by procedure **Clear-Dirty-Marks** (figure 6). This step may clear dirty marks that have been concurrently set by the running mutators. Since we want to keep these dirty bits set, we will use the logging in the buffers (which contain all objects that have been marked dirty since the first handshake) to set these

```
Procedure Initiate-Collection-Cycle
begin
1.    for each thread $T_i$ do
2.        $Snoop_i$ := true
3.    for each thread $T_i$ do
4.        suspend thread $T_i$
          // copy (without duplicates).
5.        $Hist_k := Hist_k \cup$
              $Buffer_i[1 \ldots CurrPos_i - 1]$
          // clear buffer.
6.        $CurrPos_i := 1$
7.        resume $T_i$
end
```

Figure 5: Sliding View Algorithm: Procedure Initiate-Collection-Cycle

```
Procedure Clear-Dirty-Marks
begin
1.    for each $\langle s, o \rangle \in Hist_k$ do
2.        $Dirty(s)$ := false
end
```

Figure 6: Sliding View Algorithm: Procedure Clear-Dirty-Marks

dirty bits on again.

4. Reinforcing dirty marks (second handshake). during the handshake the collector reads the contents of the threads' buffers (which contain slots that were logged since the first handshake). The collector then *reinforces*, i.e., sets, the flags of the slots listed in the buffers.

5. Assuring reinforcement is visible to all mutators (third handshake). The third handshake is carried out. Each thread is suspended and resumed with no further action. By the time all threads resume, we know that they view correctly all dirty bits. Namely, a slot is dirty iff it was modified by a thread that responded to the first handshake.

Steps (4) and (5) are executed by procedure **Reinforce-Clearing-Conflict-Set** (figure 7).

6. Consolidation (fourth handshake). During the fourth handshake thread local states are scanned and objects directly reachable from the roots are marked *local*. Threads' buffers are retrieved once more and are *consolidated*.

```
Procedure Reinforce-Clearing-Conflict-Set
begin
1.    $ClearingConflictSet_k := \emptyset$
2.    for each thread $T_i$ do
3.        suspend thread $T_i$
4.        $ClearingConflictSet_k :=$
              $ClearingConflictSet_k \cup$
              $Buffer_i[1 \ldots CurrPos_i - 1]$
5.        resume thread $T_i$
6.    for each $s \in ClearingConflictSet_k$ do
7.        $Dirty(s)$ := true
8.    for each thread $T_i$ do
9.        suspend thread $T_i$
10.       nop
11.       resume $T_i$
end
```

Figure 7: Sliding View Algorithm: Procedure Reinforce-Clearing-Conflict-Set

```
Procedure Consolidate
begin
1.    local $Temp := \emptyset$
2.    $Locals_k := \emptyset$
3.    for each thread $T_i$ do
4.        suspend thread $T_i$
5.        $Snoop_i$ := false
          // copy and clear snooped objects set
6.        $Locals_k := Locals_k \cup Locals_i$
7.        $Locals_i := \emptyset$
          // copy thread local state and ZCT.
8.        $Locals_k := Locals_k \cup State_i$
9.        $ZCT_k := ZCT_k \cup New_i$
10.       $New_i := \emptyset$
          // copy local buffer for consolidation.
11.       $Temp := Temp \cup$
              $Buffer_i[1 \ldots CurrPos_i - 1]$
          // clear local buffer.
12.       $CurrPos_i := 1$
13.       resume thread $T_i$
      // consolidate $Temp$ into $Hist_{k+1}$.
14.   $Hist_{k+1} := \emptyset$
15.   local $Handled := \emptyset$
16.   for each $\langle s, v \rangle \in Temp$
17.       if $s \notin Handled$ then
18.           $Handled := Handled \cup \{s\}$
19.           $Hist_{k+1} := Hist_{k+1} \cup \{\langle s, v \rangle\}$
end
```

Figure 8: Sliding View Algorithm: Procedure Consolidate

```
Procedure Update-Reference-Counters
begin
1.    $Undetermined_k := \emptyset$
2.    for each $\langle s, v \rangle$ pair in $Hist_k$ do
3.        $curr$ := read($s$)
4.        if $\neg Dirty(s)$ then
5.            $curr.rc := curr.rc + 1$
6.        else
7.            $Undetermined_k :=$
                  $Undetermined_k \cup \{s\}$
8.        $v.rc := v.rc - 1$
9.        if $v.rc = 0 \wedge v \notin Locals_k$ then
10.           $ZCT_k := ZCT_k \cup \{v\}$
```

Figure 9: Collector Code: ProcedureUpdate-Reference-Counters

Consolidating threads' buffers amounts to the following. For any slot that appears in the threads' buffers accumulated between the first and fourth handshakes, pick *any* occurrence of the slot and copy it to a digested consistent history. All other occurrences of the slot are discarded.

The digested history replaces the accumulated threads' buffers. i.e., the history for the next cycle is comprised of the digested history of threads' logging between the first and fourth handshakes of the current cycle, unified with threads' buffers representing updates that will occur after the fourth handshake of the current cycle but before the first handshake of the next cycle. Consolidation is carried out by procedure **Consolidate** of figure 8.

7. Updating. The collector proceeds to adjust *rc* fields due to differences between the sliding views of the previous and current cycle. This is done exactly as in the snapshot algorithm (see figure 9). The collector fails to determine the "current" value of all slots that were modified (i.e., are

```
Procedure Read-Buffers
begin
1.    $Peek_k := \emptyset$
2.    for each thread $T_i$ do
3.        local $ProbedPos := CurrPos_i$
          // copy buffer (without duplicates.)
4.        $Peek_k := Peek_k \cup$
              $Buffer_i[1 \dots ProbedPos - 1]$
end
```

Figure 10: Collector Code: Procedure Read-Buffers

```
Procedure Merge-Fix-Sets
begin
1.    $Peek_k := Peek_k \cup Hist_{k+1}$
end
```

Figure 11: Sliding View Algorithm: Procedure Merge-Fix-Sets

dirty). These slots will be treated later and are now marked as *undetermined.*

8. Gathering information on undetermined slots. The collector asynchronously reads mutators' buffers (using the procedure **Read-Buffers** of figure 10). Then, in procedure **Merge-Fix-Sets** (figure 11) it unifies the set of read pairs with the digested history computed in the consolidation step. The set of undetermined slots is a subset of the slots appearing in the unified set so the collector may now proceed to look up the values of these undetermined slots.

9. Incrementing *rc* fields of objects referenced by undetermined slots. In procedure **Fix-Undetermined-Slots** (figure 12) any undetermined slot is looked up in the unified set and the *rc* field of the associated object is incremented.

10. Reclamation. Reclamation generally proceeds as in the previous algorithm, i.e., recursively freeing any object with zero *rc* field which is not marked *local.* We should be careful, however, not to reclaim objects whose slots appear in the digested history. i.e., objects which were modified since the cycle commenced but became garbage before it ended. The reclamation of such objects is deferred to the next cycle. Reclamation is carried out using the procedures **Reclaim-Garbage** (figure 13 and **Collect** (figure 14).

5. MEMORY COHERENCE

As mentioned in the introduction, two simple modifications can make the algorithm suitable for platforms that do not guarantee sequential memory consistency. We list these modifications here and discuss their cost.

```
Procedure Fix-Undetermined-Slots
begin
1.    for each pair $\langle s, v \rangle$ pair in $Peek_k$
2.        if $s \in Undetermined_k$ do
3.            $v.rc := v.rc + 1$
end
```

Figure 12: Collector Code: Procedure Fix-Undetermined-Slots

```
Procedure Reclaim-Garbage
begin
1.    $ZCT_{k+1} := \emptyset$
2.    for each object $o \in ZCT_k$ do
3.        if $o.rc > 0$ then
4.            $ZCT_k := ZCT_k - \{o\}$
5.        else if $o.rc = 0 \land o \in Locals_k$ then
6.            $ZCT_k := ZCT_k - \{o\}$
7.            $ZCT_{k+1} := ZCT_{k+1} \cup \{o\}$
8.    for each object $o \in ZCT_k$ do
9.        Collect($o$)
end
```

Figure 13: Collector Code: Procedure Reclaim-Garbage

```
Procedure Collect($o$: Object)
begin
1.    local $DeferCollection$ := false
2.    foreach slot $s$ in $o$ do
3.        if $Dirty(s)$ then
4.            $DeferCollection$ := true
5.        else
6.            $val$ := read($s$)
7.            $val.rc := val.rc - 1$
8.            write($s$, null)
9.            if $val.rc = 0$ then
10.               if $val \notin Locals_k$ then
11.                   Collect($val$)
12.               else
13.                   $ZCT_{k+1} := ZCT_{k+1} \cup \{val\}$
14.   if $\neg DeferCollection$ then
15.       return $o$ to the general purpose allocator.
16.   else
17.       $ZCT_{k+1} := ZCT_{k+1} \cup \{o\}$
end
```

Figure 14: Sliding View Algorithm: Procedure Collect

We first note that most platforms provide a sequentially consistent view of the memory for reads and writes made to the same word. Furthermore, this guarantee is provided more generally within the *coherence granule*. Namely, a guarantee on a sequential consistency is made for an entity that is larger than one word of memory. It is made on the coherence granule of the platform, which is usually of the size of the cache line. We keep this guarantee in mind and turn to the algorithm. There are three dependencies on instruction ordering in the algorithm.

Dependency 1: in the write barrier, the reads and writes of the dirty flag and the pointer slot must be executed in the order stated in the algorithm. To solve this dependency, we note that in most cases the dirty bit and the pointer slot reside on the same coherence granule. In our implementation, we keep the dirty bit in the header of the object. Thus, we only need to perform a memory synchronization barrier for objects whose dirty bit does not reside on the same coherence granule with the modified slot. Furthermore, the write barrier begins with a check whether the object is not dirty. The synchronization barrier is required only if the check is validated, i.e., the object is not dirty.

Cost: As reported in a study of the SPECjvm98 benchmarks [15] and is implied by the results of Chilimbi and Larus [9], most objects are small. For example, the median of the object size runs between 12 to 24 [15]. The size of the cache line ranges between 32 to 128 depending on the platform. Furthermore, our measures show that objects tested in the write barrier rarely turn out not dirty. For the javac benchmark this happens less than once in a hundred, and for the SPECjbb benchmark and all the other SPECjvm98 benchmarks this happens less than once in a thousand. So the vast majority of the pointer updates require no cost for handling memory coherence. To summarize, the number of actual pointer modifications whose write barriers require a synchronization overhead (i.e., large objects that are not dirty) is tiny and we expect to see negligible impact on the running time.

Dependency 2: the modification of the snoop flag. We assume that the modification of the snoop flag is visible to all threads before we actually start the first handshake. To make sure this is the case, we can add a preliminary handshake in the beginning of the cycle in which the snoop flags are raised (currently this is done without stopping the threads).

Cost: this is done once per collection cycle and is thus negligible compared to the overall running time of the collection cycle (and to the running time of the program).

We remark that we have not implemented these two modifications, yet, we have not witnessed any problem caused by reordering instructions by the Intel platform.

6. AN IMPLEMENTATION FOR JAVA

We have implemented our algorithm on SUN's Java Virtual Machine 1.2.2. The implementation was done for the interpreter (no JIT). In both the original and modified JVM we used the assembler loop (it was modified to take into account the write barrier and a modified object layout). We ran the measurements on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB

		Heap Size (MB)	
		600	1200
score in JBB's throughput units	Original	1,131.3	1,101.0
	RC	1,101.7	1,108.3
Change in JBB score		-2.6%	0.7%
Maximal response time (milliseconds)	Original	7763	16,100
	RC	115	110
Times RC is more responsive		×67.5	×146.4

Figure 15: Throughput and latency of the reference counting collector and the original collector in standard SPECjbb runs, with 600 MB and 1200 MB heaps.

Threads	1	2	4	6	8	10	15
Original	7433	8037	8463	6923	7857	7536	6593
RC	16	16	47	78	110	146	250
Times RC is more responsive	×464	×502	×180	×88	×71	×51	×26

Figure 16: Maximal response time, in milliseconds, of the original JVM and our reference counting collectors in a series of SPECjbb2000 runs with a fixed number of threads per run and a 600 MB heap.

of physical memory. We measured our algorithm's performance characteristics compared to the original algorithm used in the JVM. We also measured the run of our collector on a client machine: a single Pentium III at 500Mhz with 256MB of physical memory.

We used two standard testing suites: SPECjbb2000 and JPECjvm98. These benchmarks are described in detail in SPEC's Web site[37]. We target our reference counting algorithm for use with big heaps. Thus, we used a 1,200MB Java heap for the JBB server benchmark. We also present the (bit worse) results for a Java heap of size 600MB. For the jvm98 client benchmarks we used a 64MB heap.

6.1 Server performance

A standard execution of SPECjbb requires a multi-phased run with increasing number of threads. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. Prior to the beginning of each phase a synchronous GC cycle may or may not occur, at the discretion of the tester. We decided not to perform this synchronous garbage collection as we believe it defeats capturing real world scenarios in which the server is not given a chance for this "offline" behavior so often. The results presented here are averaged over three standard runs.

Figure 15 shows throughput and latency of the reference counting algorithm compared to the original JVM: while we essentially retain the throughput attained by the original JVM, we improve the maximal response time by two orders of magnitude. To illustrate, the original JVM may take as long as 16 seconds to complete a JBB transaction while we never require more than 130 milliseconds. To get some more measurements of the latency, we also checked the latency as a function of the number of threads ran by SPECjbb00. We compared the response time of the original JVM with our reference counting collector. This non-standard run of the benchmark is reported in Figure 16.

Threads	Time to completion (seconds)		% Improvement
	Original	RC	
1	93	88.6	4.9%
2	71.9	68.5	5.0%
3	56.3	52.5	7.2%
4	57.2	54.2	5.6%
8	58.2	52.3	11.4%
12	58	57.9	0.2%
16	59	59.1	-0.1%

Figure 17: Time to completion, in seconds, of the MTRT benchmark, with varying number of threads.

Threads	1	2	4	6	8	10	15	20
Original	24	39	70	100	139	160	236	312
RC	27	44	77	108	170	171	251	329

Figure 18: MB allocated at the end of a SPECjbb run with a fixed number of threads and a heap of 600 MB.

The second benchmark that we have used is MTRT (multi-threaded ray tracer). This benchmark does not measure response time, only elapsed running time, which corresponds to the JVM's throughput. As can be seen from figure 17 the reference counting collector outperforms the original JVM with an improvement of up to 10% in the total running time.

In Figure 18 we present measurements of the heap consumption. The reason for increased consumption in the reference counting algorithm is the lack of compaction yielding more fragmentation and the space required for the dirty bit (which is implemented as an extra pointer per object, refer to the full paper [28] for details). We remark that since we do not move objects for compaction, we can get most of this waste back by joining the object and its handle and getting rid of the handle pointer to the object.

In addition to the space occupied directly by objects we also allocate memory from the operating system for the following data structures:

ZCT: we implement the ZCT as a bitmap with each potential object address having an associated bit in the bitmap. Since the object alignment is eight bytes the ZCT requires a sixteenth of the heap.

Snoop/local marks: similarly, we mark objects as snooped using a bitmap which requires an additional sixteenth of the heap.

Reference counters: the reference counters are implemented in a bitmap which associates each potential object address with a two bit counter. Hence this bitmap requires an additional eighth of the heap.

Buffers: local ZCTs, update buffers and snoop buffers are all allocated from the same pool of buffers. We have used a fixed buffer size of 64 KB and usually the "working set" of buffers didn't exceed twenty simultaneously allocated buffers, which amounts to less than two MB of additional memory.

6.2 Client performance

Benchmark	Time to completion (seconds)	
	Original	RC
Total	2582.2	2676.0
compress	720.8	723.3
db	374.0	383.7
jack	264.6	299.7
javac	225.0	235.2
jess	181.7	209.7
mpegaudio	607.1	610.6

Figure 19: Elapsed time for the execution of the entire SPECjvm98 suite and intermediate execution time of a double-run for each of the suite's members. All measurements are in seconds.

While we have targeted our collector for multi-processor environments we still wanted to verify that it is performant in a single-processor setting. To that end we have used the SPECjvm98 benchmark suite. We used the suite using the test harness, performing standard[1] automated runs of all the benchmarks in the suite. In a standard automated run, each benchmark is ran twice and all benchmarks are ran on the same JVM one after the other. Figure 19 shows the elapsed time of the entire automated run and the time for each double run of each benchmark. As can be seen from figure 19, the reference counting collector was only 3.6% percent slower than the original JVM. Given that we pay the overheads of concurrent collection while we're not benefiting from the availability of multiple processors these are very good results.

6.3 Collector characteristics

We also include some measurements of the collector characteristics. Due to lack of space, we only mention a couple of them briefly. First, we measured the number of objects that have reached a stuck count (i.e., $o.RC = 3$). Recall that we keep only two bits for the reference count and an object whose RC is increased to 3 is considered stuck. This reference count is resolved only in the following run of the mark and sweep collector. It turns out that for most benchmarks this happens to less than 1% of the objects. For compress, javac, and mpegaudio this number was higher: between 3.7% to 4.7% of the objects.

Stuck pointers and cycles prevent the reference counting collector from collecting all dead objects. We check the effectiveness of the collector in Figure 20. Except for javac, which uses many cyclic structures, and to a lesser degree the db benchmark, the benchmarks have demonstrated a low degree of sensitivity to reference counting. This supports the assumption that we may use reference counting for most garbage collection cycles and only occasionally resort to tracing.

7. CONCLUSIONS

We have presented a novel on-the-fly reference counting garbage collector with low latency and high throughput. The algorithm uses extremely low synchronization overhead: the barriers for modifying a reference and the barrier for creating a new object are very short and in particular, require

[1]The standard run requires running the harness through a Web server while we performed the tests directly off the disk. Aside from that, the executions were standard.

Benchmark	% Reclaimed by tracing	% Reclaimed by RC
jbb	97.5%	96.5%
compress	73.5%	72.1%
db	99.6%	90.5%
jack	99.6%	96.8%
javac	99.6%	66.1%
jess	99.8%	99.5%
mpegaudio	74.2%	69.6%

Figure 20: Percentage of objects reclaimed by (1) the original collector and (2) the reference counting collector when used without the backing mark-and-sweep collector.

no strong synchronized operations such as a compare-and-swap instruction. Furthermore, there is no particular point in which all threads must be suspended simultaneously. Instead, each thread cooperates with the collector by being shortly suspended four times during each collection cycle.

We have implemented our collector on SUN's Java Virtual Machine 1.8 and presented measurements showing excellent latency and a throughput that is comparable to the original mark-sweep-compact collector.

8. ACKNOWLEDGMENT

We thank Hillel Kolodner for helpful discussions.

9. REFERENCES

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988.

[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] D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. To appear in the *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 20-22 2001.

[4] D. Bacon and V. Rajan. Concurrent Cycle Collection in Reference Counted Systems. To appear in the *Fifteenth European Conference on Object-Oriented Programming* (ECOOP), University Eötvös Lorand, Budapest, Hungary, June 18-22 2001.

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

[6] Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.

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

[8] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. *ICLP*, pages 276–293, 1987.

[9] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In Proceedings of *the First International Symposium on Memory Management*, volume 34(3) of ACM SIGPLAN Notices, October 1998, pages 37-48.

[10] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[11] Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988.

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

[13] John DeTreville. Experience with garbage collection for modula-2+ in the topaz environment. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

[14] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[15] Sylvia Dieckmann and Urs Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), Lecture Notes on Computer Science, Springer Verlag, June 1999.

L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[16] 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.

[17] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL* 1994.

[18] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL* 1993.

[19] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanover. Implementing an On-the-fly Garbage Collector for Java. *The 2000 International Symposium on Memory Management*, October, 2000.

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

[21] Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. Parallel conservative garbage collection with fast allocation. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP* '91 Workshop on Garbage Collection in Object-Oriented Systems, 1991.

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

[23] Atsuhiro Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy reference counting: An incremental garbage collection method for parallel inference machines. *ICLP*, pages 1241–1256, 1988.

[24] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[25] Maurice Herlihy and J. Eliot B Moss. Non-blocking garbage collection for multiprocessors. Technical Report CRL 90/9, DEC Cambridge Research Laboratory, 1990.

[26] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996.

[27] Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. Technical Report 88.384, IBM Haifa Research Lab, November 1999. Available at http://www.cs.princeton.edu/~erez/publications.html.

[28] Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS0967, Technion, Israel Institute of Technology, November 1999. Available at http://www.cs.technion.ac.il/~erez/publications.html.

[29] James S. Miller and B. Epstein. Garbage collection in MultiScheme. In *US/Japan Workshop on Parallel Lisp, LNCS 441*, pages 138–160, June 1990.

[30] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[31] Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *IPL*, 55(4):229–234, August 1995.

[32] Manoj Plakal and Charles N. Fischer. Concurrent Garbage Collection Using Program Slices on Multithreaded Processors. *ISMM* 2000.

[33] David J. Roth and David S. Wise. One-bit counts between unique and sticky. *ACM SIGPLAN Notices*, pages 49–56, October 1998. ACM Press.

[34] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the Effectiveness of GC in Java. *The 2000 International Symposium on Memory Management (ISMM '00)* 2000.

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

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

[37] Standard Performance Evaluation Corporation, `http://www.spec.org/`

[38] Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In *LFP*, pages 159–166, August 1984.

[39] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 1991.

[40] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[41] David S. Wise. Stop and one-bit reference counting. *IPL*, 46(5):243–249, July 1993.

[42] David S. Wise. Stop and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.

[43] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.