An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views *

Hezi Azatchi † Yossi Levanoni ‡ Harel Paz § Erez Petrank ¶

ABSTRACT

With concurrent and garbage collected languages like Java and C# becoming popular, the need for a suitable non-intrusive, efficient, and concurrent multiprocessor garbage collector has become acute. We propose a novel mark and sweep on-the-fly algorithm based on the sliding views mechanism of Levanoni and Petrank. We have implemented our collector on the Jikes Java Virtual Machine running on a Netfinity multiprocessor and compared it to the concurrent algorithm and to the stop-the-world collector supplied with Jikes JVM. The maximum pause time that we measured with our benchmarks over all runs was 2ms. In all runs, the pause times were smaller than those of the stop-the-world collector by two orders of magnitude and they were also always shorter than the pauses of the Jikes concurrent collector. Throughput measurements of the new garbage collector show that it outperforms the Jikes concurrent collector by up to 60%. As expected, the stop-the-world does better than the on-the-fly collectors with results showing about 10% difference.

On top of being an effective mark and sweep on-the-fly collector standing on its own, our collector may also be used as a backup collector (collecting cyclic data structures) for the Levanoni-Petrank reference counting collector. These two algorithms perfectly fit sharing the same allocator, a similar data structure, and a similar JVM interface.

Keywords: Runtime systems, Memory management, Garbage collection, Concurrent garbage collection, On-the-fly garbage collection.

---

*Research was supported by generous funding from the Bar-Nir Bergreen Software Technology Center of Excellence - the Software Technology Laboratory (STL), and by the E. AND J. BISHOP RESEARCH FUND.

†IBM Haifa Research Labs. Work done while at the Technion. Email: hezia@cs.technion.ac.il.

‡Microsoft Corporation. Work done while at the Technion. Email: ylevanon@microsoft.com.

§Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: pharel@cs.technion.ac.il.

¶Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Email: erez@cs.technion.ac.il.

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

1. INTRODUCTION

Modern SMP servers with large heaps provide new challenges for the design of suitable garbage collectors. Garbage collectors designed for client machines may lead to inefficient running times on servers and non-incremental collectors may lead to unacceptable pauses. An *on-the-fly garbage collector* does not stop the program threads to perform the collection. Instead, the collector executes on a separate thread (or process) concurrently with the program threads (the mutators). On-the-fly collectors are useful for multithreaded applications running on multiprocessor servers, where it is important to fully utilize all processors and provide even response time, especially for systems in which stopping the threads is a costly operation.

In this paper, we present the design and implementation of a new efficient and non-intrusive garbage collector suitable for Java and C# running on modern SMP's and using large heaps. Our algorithm is a non-moving mark and sweep collector based on a "relaxed" snapshot of the heap (the sliding views). It is suitable for modern SMP's running concurrent programs. It is fully concurrent (on-the-fly) allowing short pause times. Namely, each thread is stopped for a short while to cooperate with the collector, but the threads never need to be stopped at the same time. In particular, unlike the mostly concurrent collector, our collector will always terminate without stopping the threads. Finally, our collector may be used with conservative JVM's.

Our sliding views mark and sweep collector may also be used as a backup tracing algorithm for the sliding views reference counting collector of Levanoni and Petrank [22]. Any reference counting collector may need to use a tracing collector to reclaim cyclic garbage. It is advantageous for the reference counting collector to have a tracing collector that may use a similar allocator and a similar JVM interface. The algorithm presented in this paper is inter-operable with the reference counting sliding view algorithm of Levanoni-Petrank meaning that they share the same allocator, and their data structure may be united so that it is possible to decide on a cycle by cycle basis which algorithm should be invoked. A preliminary version of our mark and sweep algorithm was implemented for that purpose and was used to produce the results of the reference counting collector reported in [22]. However, the tracing sliding view algorithm has not been reported previously and its properties have not yet been investigated. In this paper, we present a mature version of this collector accompanied by an implementation and measurements.

In the sequel, we assume that the reader is familiar with memory management standard terminology and algorithms. For a more detailed introduction to garbage collection and memory management the reader is referred to [19].

1.1 The main algorithmic ideas

The basic *mark and sweep* algorithm operates by stopping all program threads, *marking* any object which is directly reachable (either from a local or a global reference) and then recursively marking any object which is pointed to by a marked object. Then, any object, which is not marked is *swept*, i.e., reclaimed. Finally, program threads are resumed.

To simplify the presentation of our new collector, we start with a simple concurrent mark and sweep that uses a snapshot. Concurrent mark and sweep collectors perform some, or all, of the above steps concurrently with the program threads (the mutators). *Snapshot at the beginning* [30, 14] mark and sweep collectors exploit the fact that a garbage object remains garbage until the collector recycles it, i.e., being garbage is a stable property. Thus, snapshot at the beginning operates by:

1. stopping the mutators,
2. taking a snapshot (replica) of the heap and roots,
3. resuming the mutators,
4. tracing the replica,
5. sweeping all objects in the original heap whose replicated counterparts are unmarked. These reclaimed objects must have been unreachable at the time the snapshot was taken and hence they remain unreachable until the collector eventually frees them.

The problem with this approach is that making a snapshot of the heap is not realistic. It requires too much space and time. However, a useful property of today's benchmarks is that even if they employ a large heap, only a small part of it is modified at a time.

Loosely speaking, our algorithm works as follows. Assume first, that the heap does not change at all (which is not correct) and traverse the heap concurrently with the program activity. However, the heap *is* modified by the program threads and we cannot ignore it. Our solution is to record objects states before they are first modified by a mutator. Later, we trace according to the recorded state. A mechanism that remembers the object state before it is first modified by a mutator has been developed in [22] for monitoring changes in reference counts. We employ that mechanism for our algorithm.

However, we get substantial savings from adapting this mechanism to a mark and sweep collector. The first saving is obtained by the fact that recording is necessary only when the collector is active. The second saving is due to the fact that we need to record an object A only if the object A is modified after the collector started and before A is traced. This happens seldom. In particular, new objects are created marked (via the color toggle mechanism). Thus, updates to new objects, which are most frequent, do not require recording the values of the new object. Finally, and similarly to the reference counting saving, we need to record objects only once: the first time they get modified after the collection starts. All these savings make the write barrier very efficient. Usually, it only employs a fast path running only a couple of if statements. The long path of actually recording the object's state is taken infrequently (see the measured statistics in Section 6 below). The modified write barrier that we propose maintains the good properties of the original write-barrier from [22]. In particular, it allows concurrent threads to collect the information with no extra synchronization. More details appear in Section 2.1.

Finally, the algorithm described so far needs to stop all threads at the same time in order to determine the snapshot time. This is a must with a multiprocessor since we need to determine one specific time at which no thread is in the middle of an update operation or in the middle of creating a new object and at which all threads "know" that a snapshot time has been set. Such wide mutator synchronization increases the pause time, as all mutator threads must come to a halt together. In order to eliminate this synchronization, we let the collector determine the snapshot time for each mutator asynchronously at its own pace. This reduces the pause time to the level reported in this paper but requires some care to assure correctness. In particular, we get a fuzzy snapshot, called a "sliding view" of the heap. This view is not an accurate snapshot, but we can use it for collection with an additional aiding mechanism called "snooping". During the (short) time that the sliding view is determined, the write barrier records all pointer assignments. When marking the roots is over, snooping is stopped and all pointer slots recorded by the snooping mechanism are traced as if they were roots. This may lead to a small amount of floating garbage but it is required for correctness. Details appear in Section 2.2.

In our implementation, we employ the state-of-the-art engineering tricks such as color toggle [21, 16, 5, 18, 12], allowing a simple coloring of allocated objects, and saving some of the sweep work, we use a block oriented allocator [6, 13], bitmaps, etc. We do not elaborate on these engineering issues here.

1.2 Comparison to the Levanoni-Petrank collector

This work is based on the *sliding views* concept from [22]. The collector presented here is a different collector by nature since it is a mark and sweep collector. For today's benchmarks, the tracing collector runs faster. Our contribution here is in presenting the tracing collector, implementing it on the Jikes JVM and measuring its performance against two collectors supplied with Jikes: the on-the-fly reference counting collector and the stop-the-world tracing collector.

The algorithmic contribution in this paper is in the composition of several algorithmic ideas into one optimized collector. We start with the snapshot mark and sweep collector employing ideas from [14, 31]. We then modify this collector to make it suitable for stock hardware: instead of using the operating system copy-on-write feature, we let the mutators record modified objects via a write barrier. But now, we may borrow the mechanism of [22] for keeping track of modified objects and obtain a fast and non-intrusively collector. Once we make this connection, we note that optimization may be used on the combined modified collector. The write barrier must record a modified object only if the collector is tracing (rather than always as in [22]). Furthermore, recording is only required if the object being modified has not yet been traced. These two restrictions allow frequent use of a fast path for the write barrier and only infrequent actual recording of an object state.

1.3 Memory consistency

We start by describing our collector for a sequentially consistent memory. In Section 4 below, we provide modifications that allow the algorithm to run on platforms, which are not sequentially consistent. From our experience with the Netfinity (running the Pentium III Xeon processor), the modifications were not required for all the benchmarks that we used.

1.4 Implementation and results

The sliding views mark and sweep collector is implemented in Jikes [1], a JVM system written entirely in Java (with some primitives for manipulating raw memory). The system was run on a 4-way IBM Netfinity server. We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suites. These benchmarks are described in detail in SPEC's Web site[29].

It turns out that our algorithm is non-intrusive. The maximum pause time measured for all the run benchmarks was 2 ms, which is two orders of magnitude shorter than the pauses of the stop-the-world collector, but is even shorter than the concurrent Jikes collector. This pause time is the time it takes to scan the roots of a single thread. The rest of our handshakes are much faster. In the Jikes concurrent collector, the pause time is larger since (in addition to scanning the roots in each collection) it sometimes runs some allocator maintenance while threads wait. This is not required by our collector.

As for efficiency, our on-the-fly collector is slower than the stop-the-world collector by around 5-10%, which is "normal" for concurrent collectors. Comparing with the concurrent collector supplied with the Jikes JVM (see [2]), we obtained a throughput improvement of up to 60% (for the SPECjbb2000 benchmark).

1.5 Related work

The mark and sweep garbage collector was first presented by McCarthy [23]. Much research and engineering effort has been put into this algorithm since. Algorithms that perform garbage collection using a snapshot of the heap appear in [14, 31]. On-the-fly collectors are a special case of concurrent collectors. Concurrent collectors run on dedicated threads concurrently with the program threads, but allow short synchronization points in which all threads are suspended for some synchronization required by the collector. Many concurrent collectors have been proposed, see for example [25, 6, 26, 24].

The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [27, 28, 9] and continued in a series of papers [9, 15, 3, 4, 20, 21] culminating in the Doligez-Leroy-Gonthier (DLG) collector [11, 10]. A modern implementation of the DLG collector for Java appears in [12, 13]. On-the-fly collectors were mostly based on the mark and sweep algorithm, yet, an on-the-fly copying collector has appeared in [17] and on-the-fly reference counting collectors were proposed in [2, 22].

Although our collector comes from an advanced synergy of [31, 14] with [22], the outcome collector should be also compared to the collector of Doligez-Leroy-Gonthier [11, 10], which is the most advanced on-the-fly mark and sweep collector. The DLG collector also uses fine-grained synchronization, and it was used in a production JVM of IBM (see [12, 13]). Unfortunately, we are not aware of an implementation of that algorithm that is available for academic research (and in particular, it is not implemented on the Jikes platform). Therefore, it is not possible to show a direct comparison of throughput and latency. We expect the pause times to be similar as in both algorithms the longest pause emanates from marking the roots. In terms of efficiency, although the tracing algorithms are somewhat different, we do not see any theoretical comparison factors that may be stated without actually running the collectors. With respect to the write barrier more may be said. Ignoring the short interval in which the roots are marked and both collectors use an extended write barrier, the DLG collector marks gray the ex-target of any modified pointer. This means that the write barrier forces the mutator to touch a different object, whereas our write barrier touches only the modified object, copying the non-null pointers *at the first time the object is modified* and *only before it is traced*. Thus, our write barrier may take the short path more frequently and it may impose a better cache behavior. However, the actual answer must be done by a comparison of our collector with a serious and well-thought implementation of the DLG collector (which is not available for us). In any case, we believe that it is important to propose a (good) alternative to the state-of-the-art on-the-fly mark and sweep collector.

1.6 Organization

We start with an overview of the collector algorithm in Section 2 below. We provide the algorithmic details and pseudo-code in Section 3. In Section 4 we explain how to adapt the algorithm to platforms that do not provide sequentially consistent memory. We say a few words on the implementation for Java in Section 5 and in Section 6 we present performance results. We conclude in Section 7.

2. COLLECTOR OVERVIEW

In this Section we describe our new collector. For clarity of presentation, we start with an intermediate concurrent algorithm called *the snapshot algorithm*. In Section 2.2, we extend this intermediate algorithm making it on-the-fly.

2.1 Starting with a snapshot algorithm

We start with an intermediate algorithm called *the snapshot algorithm*. This is a concurrent collector that requires a synchronization point in which all mutators are halted together to determine a snapshot time in which no mutator is in the middle of an update operation or in the middle of creating a new object. Most of the ideas presented with this simpler collector apply to our on-the-fly collector. Note that the length of the pause for this algorithm is short, but it requires synchronizing all application threads, which might mean longer pauses, especially for operating systems that do not support an efficient suspension of all application threads.

The idea, as presented in Section 1.1, is to perform the marking phase after taking a snapshot of the heap. Once the heap is frozen in a snapshot, the marking phase may proceed on the snapshot view while the mutators go on modifying the real heap. At the end of the trace, unmarked objects may be safely reclaimed since dead objects can not be touched or modified by the mutators.

Since taking a real snapshot is too costly, our algorithm takes the following approach. In the beginning of the collection all mutators are stopped and implicitly agree on a snapshot time. At the same stop, their roots are being marked and all threads resume. From that moment on, the mutators use the following write barrier for each pointer modification. If the collector is still tracing, and if the modified object has not been traced yet, and if the modified object is not dirty, then the object becomes dirty and the values of its pointers are saved (copied) to a local buffer. The write barrier does the logging (and dirtying) only for non-dirty objects. Thus, actual logging of the object state is only required infrequently: when the collector has started, but has not yet traced the modified object, and when the object is modified for the first time. In that case, the saved values are the values of non-null pointers as existed during the snapshot time. Mostly, the write barrier runs the short path and finishes quickly. As an object is only saved once during a collection cycle, the number of objects that need to be saved is the number of objects that get modified during the collection. We ignore for a moment the possibility that mutators modify the same object concurrently. We will show later that the write barrier works well also in this case without requiring explicit synchronization.

Given the operations described above, the collector may trace the objects as if it has a heap snapshot. Non-dirty objects may be read from the heap, because they were not modified. The state of dirty objects at the time of the snapshot may be obtained from the local buffers. To finish the collection cycle, the mutators are notified that the write barrier is not required anymore, and sweep is run to reclaim unmarked objects. Finally, all the dirty marks on the

```
Procedure Update(o: Object, s: Slot, new: Object)
begin
    if TraceOn and o.color=white then
        local old := read(o)
        // was o written to since the snapshot time ?
        if ¬ Dirty(o) then
            // ... no; keep a record of the old values.
            Buffer[CurrPos] := ⟨o,old⟩
            CurrPos := CurrPos + length(o)
            Dirty(o) := true
    write(s, new)
end
```

Figure 1: Mutator Code: Update Operation

objects that appear in the buffers are cleared so that they become ready for the next collection.

We now return to the race issue raised above. What happens if two mutators modify the same object concurrently? Are the recorded values correct? Our write barrier is taken from the Levanoni-Petrank reference counting collector and is especially designed to handle such races without employing costly synchronization operations. A simplified version of the write barrier pseudo-code appears in Figure 1. (We study this simplified version since it clarifies all the relevant points. The actual write barrier is more efficient and it appears in Section 3 below.)

Two mutators that invoke the update barrier concurrently to modify the same location do not foil the collection. We remark that in normal benchmarks (and programs) mutators do not race over writing to the same location without synchronization. Such races rarely appear in programs (they do appear in programs that try to implement a lock, or programs that trust the various threads to write the same value, etc.). Such races usually appear when the program contains a bug. Either way (and even if the program contains a bug) we would like our collector to handle the situation properly and not fail during program execution. Our first analysis of this write barrier is based on sequential consistency. However, simple modifications may settle this issue and make the collector run correctly on weakly consistent platforms at a negligible throughput penalty. This issue is discussed in Section 4 below.

Looking at the write barrier pseudo-code we split the analysis into two cases. First, suppose one of the updating threads sets the dirty flag of an object before any other thread reads the dirty flag. In this case, only one thread records this object and the records properly reflect the pointer values at the snapshot time. The other case is that more than one thread finds the dirty bit clear. We will show that in this (rare) case, more than one mutator may log the value of an object, but it is guaranteed that all logs will reflect the same (correct) value corresponding to the object's state during the snapshot time.

Looking at the code, each thread starts by recording the old value of the object, and only then it checks the dirty bit. On the other hand, the actual update of o occurs after the dirty bit is set. Thus, if a thread detects a clear dirty bit, then it is guaranteed, since sequential consistency is assumed, that the value it records is the value of o before any of the threads has modified it. So while several threads may record the object o in their buffers, all of them must record the same (correct) information. To summarize, in case a race occurs, it is possible that several threads record the object o in their local buffers. However, all of them record the same correct value of o at the snapshot time. When using the information for the tracing, each of these records may be used. We conclude that even when races occur, the content of any heap pointer during the snapshot time can be obtained. The value of this pointer has either not been modified since the snapshot or it appears in the records taken by the mutators.

2.2 Using sliding views

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 local roots are marked. 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 local marking phase, scalability calls for eliminating complete halts from the algorithm. This is indeed the case with our sliding views algorithm, which avoids grinding halts completely.

A handshake [11, 10] 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 suspended one at a time, and only for a short interval, the duration of which depends on the size of the mutator's local state.

In the snapshot algorithm we had a fixed point of time at which we perform the trace. It was the time when all mutators were stopped. Namely, the snapshot algorithm is guaranteed to trace the same objects as if it had done the trace while keeping the mutators suspended. 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 has been taken from the reference counting collector of Levanoni and Petrank [22] and is similar to the way snapshots are taken in a distributed setting. Each mutator at a time will provide its view of (the modifications in) the heap, and special care will be taken by the system to make sure that while the information is gathered, concurrent modifications of the heap do not foil the collection.

Instead of stopping all mutators together for initiating a collection and marking their local roots, we stop one mutator at a time. The problem with such a relaxation is that the various threads start using the write barrier at different times. Furthermore, the scanning of the stacks is not done simultaneously and thus, a reference may be missed because it is moved from one location to another during the time we mark the threads' local roots.

Therefore we take a rather extreme, yet required, measure. Before we start marking the roots, we raise a snoop flag for each mutator. The local snoop flag is cleared when the local roots of the mutator are marked. Thus, throughout the time we mark local roots, the threads use a *snooping* mechanism via their write barrier. During this interval of time, all pointer updates are monitored. For each pointer update $p = O$ we add the object O to a local snooping buffer. All objects recorded in this manner will later be traced during the mark phase as if they were roots.

The snooping mechanism may lead to some floating garbage as we conservatively do not collect objects which have been recorded by the snooping mechanism (have been snooped), although such objects may become garbage before the cycle ends. However, if a snooped object becomes unreachable, it is guaranteed to be collected in the next cycle.

```
Procedure Update(o: Object, offset: int , new: Object)
begin
1.    if TraceOn and o.color=white then
2.        if o.LogPointer=NULL then // object not dirty
3.            TempPos := CurrPos
4.            foreach field ptr of o which is not NULL
5.                Buffer[++TempPos] := ptr
6.            // is it still not dirty?
7.            if o.LogPointer=NULL then
8.                // add pointer to object
9.                Buffer[++TempPos] := address of o
10.               //committing values in buffer
11.               CurrPos := TempPos
12.               // set dirty
13.               o.LogPointer =
14.                       address of Buffer[CurrPos]
15.   write( o, offset ,new)
16.   if Snoop and new != NULL then
17.       Snooped := Snooped ∪ { new }
end
```

Figure 2: Mutator code: Update Operation

3. THE GARBAGE COLLECTOR DETAILS

3.1 The log-pointer

One important choice that we made in our implementation affects the algorithmic details concerning the dirty bit. Each object must have a dirty bit signifying whether a pointer in the object has been modified since the sliding view started. Instead of using a single dirty bit per object we chose to dedicate a full word for the task. Indeed, this consumes space, but it allows keeping information about the dirty object. In particular, we use this word to keep a pointer to the location in the thread's local buffer where the object's pointers have been logged. A zero value (a null pointer) signifies that the object is not dirty (and not logged). We call this word the `LogPointer`.

Paying the extra price of allotting a whole word for the flag and transforming it into a pointer that identifies the logged contents of an object, rather than using a boolean bit-sized flag, enables an efficient tracing mechanism. Our tracing procedure does not need to "search" all local buffers to find out the recorded information about the object's state as recorded in the local buffers. Instead, it follows the pointer in the object header. Thus, the tracing procedure can always proceeds immediately after accessing the object's `LogPointer` field, either as dictated by the current objects' contents or according to the previous state of the object, as recorded in the log entry (pointed by the `LogPointer` field). Which of the two routes is taken is determined by the value of `LogPointer`.

3.2 Mutator cooperation

The mutators need to execute garbage-collection related code on three occasions: when updating an object, when allocating a new object and during handshakes. This is accomplished by the Update (Figure 2) procedure, the New (Figure 3) procedure and the handshake mechanism, respectively. The Update and New operations never interleave with a handshake. Namely, cooperation with a handshake waits until a currently executed Update or New operation finish.

In what follows we sometimes use the standard notation of denoting a marked object *black* and an unmarked object *white*.

3.2.1 Write barrier

Procedure Update (Figure 2) is activated at pointer assignment and its main task is to record the object whose pointer is modified.

```
Procedure New(size: Integer, o:Object)
begin
      Obtain an object o of size size from the allocator.
      o.color := AllocColor
      return o
end
```

Figure 3: Mutator code: Allocation Operation

We stress that the write barrier (the Update protocol) is only used with heap pointer modification. Modifications of local pointers in the registers or stack are not monitored. The logging should be done for a limited period: from the time local roots are marked till the tracing is done. The variable *TraceOn* is local to the mutator but is controlled by the collector. It tells the mutator whether the logging should be done. Thus, the first check is whether executing the write barrier is at all required. Next, we check whether the object is colored black. If it is the case, then the object is either new (i.e., this object was created during the current collection), or has already been traced during the current collection. In both cases, there is no need to log its old values (since this object won't be traced). Going through the pseudo-code, we see that each object's *LogPointer* is optimistically probed twice (lines 2 and 7) so that if the object is dirty (which is often the case), then the write barrier is extremely fast. If the object was not logged (i.e., the LogPointer of an object is NULL) then after the first probe, the objects values are recorded into the local *Buffer* (lines 3-5). The second probe at line 7 ensures that the object has not yet been logged (by another thread). If *LogPointer* is still NULL (in the second probe), then the recorded values are committed (line 9) and the buffer pointer is modified (line 11). In order to be able to distinguish later between objects and logged values, in line 9 we actually log the object's address with the least significant bit set on (while values are logged with least significant bit turned off). Then, the object's *LogPointer* field is set to point to these values (lines 13-14). After logging has occurred, the actual pointer modification happens. Finally, while marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local snooped buffer. This happens in lines 16-17. The variables *Buffer*, *CurrPos*, *Snoop* and *Snooped* are local to the thread.

Handling large objects. In our prototype we did not treat large objects in a special manner. However, buffering objects of substantial size, that contain a large amount of pointers, may exceed the 2ms pause time reported. To make sure this does not happen, one may associate dirty bits with areas smaller than object sizes. For example, the heap may be partitioned into cards of fixed size and each of them may be associated with a dirty bit (or a log pointer). Another possibility is to modify the write barrier and collector treatment of only large objects, so that they, only, are split into cards.

3.2.2 Creating a new object

Procedure New (Figure 3) is used when allocating an object. After the object is allocated, it is given a color, according to the allocation color. The allocation color is set by the collector during the various collection steps.

3.2.3 The handshake mechanism

Our handshake mechanism is the same as the one employed by the Doligez-Leroy-Gonthier collector [11, 10]. The mutator threads are never stopped together for cooperating with the collector. Instead, threads are suspended one at a time for the handshake. The stopping of the thread is not allowed while it is executing the write

```
Procedure Tracing-Collection-Cycle
begin
    Initiate-Collection-Cycle  // 1st and 2nd handshake
    Get-Roots    // 3rd handshake
    Trace-Heap
    Sweep        // 4th handshake
    Prepare-Next-Collection
end
```

Figure 4: Collector code: Tracing alg.

```
Procedure Initiate-Collection-Cycle
begin
    // first handshake
    for each thread T do
        suspend thread T
        Snoop := true
        resume T
    // second handshake
    for each thread T do
        suspend thread T
        TraceOn := true
        resume T
end
```

Figure 5: Collector code: Initiate-Collection-Cycle

barrier or while it is creating a new object. While a thread is suspended, the collector executes the relevant actions for the handshake and then the thread is resumed. The collector repeats this process until all threads have cooperated. At that time, the handshake is completed.

3.3 Phases of the collection

The collector algorithm runs in phases as follows.

- **First handshake:** during this handshake each mutator is stopped and the `Snoop` local flag, which activates the snooping mechanism, is set.
- **Second handshake:** during this handshake each mutator is stopped and the `TraceOn` local flag, which activates the logging mechanism, is set.
- **Third handshake:** during this handshake each mutator is stopped and the local roots of each mutator are marked. Also, the *Snoop* local flag is cleared.
- **Tracing:** after the third handshake is done, the collector traces the heap from the marked objects and from all snooped objects.
- **Fourth handshake:** during this handshake each mutator is stopped and the local flag `TraceOn` is cleared, so that the mutators stop recording updates in the buffers.
- **Sweep:** the collector sweeps the heap and reclaims allocated unmarked objects.
- **Clear dirty marks:** The collector clears the dirty marks of all objects previously recorded in the buffers.

3.4 Collector code

Collector's code for cycle k is presented in **Procedure Tracing-Collection-Cycle** (Figure 4). Let us briefly describe each of the collector's procedures.

```
Procedure Get-Roots
begin
    black := 1-black
    white := 1-white
    // third handshake
    for each thread T do
        suspend thread T
        AllocColor := black
        Snoop := false
        Roots := Roots ∪ State // copy thread local state.
        resume thread T
    for each thread T do
        // copy and clear snooped objects set
        Roots := Roots ∪ Snooped
        Snooped := ∅
end
```

Figure 6: Collector code: Get-Roots

```
Procedure Trace-Heap
begin
    for each object o ∈ Roots do
        push o to MarkStack
    while MarkStack is not empty
        obj = pop(MarkStack)
        Trace(obj)
end
```

Figure 7: Collector code: Trace-Heap

Procedure Initiate-Collection-Cycle (Figure 5) runs the first two handshakes. During the first handshake the *Snoop* flag is raised, signaling to the mutators that they should start snooping all stores into heap slots. During the second handshake the *TraceOn* flag is raised, signaling to the mutators that they should start logging old pointer values of objects modified for the first time. For correctness, it is important to separate the two handshakes. When any mutator starts logging values in its local buffer, all mutators should be already snooping. The modifications are done via handshake to make sure that on a multiprocessor each mutator sees its value properly.

Procedure Get-Roots (Figure 6) carries out the third handshake during which the *Snoop* flag is turned off and the thread local roots are accumulated into the *Roots* (global) buffer. Next, the *Snooped* buffer of each thread (containing snooped objects), is accumulated into *Roots*, and then cleared (for the next collection). In this procedure a color toggle is executed switching the values of black and white. The color toggle mechanism avoids races between the Sweep and the New procedures, and it avoids some redundant work of the Sweep procedure (see [21, 16, 5, 18, 12]). Note that it is correct to mark new objects black during the collection, since they are alive and have no children at the time of creation. The *AllocColor* variable of each thread is then set so that new objects are created black.

Procedure Trace-Heap (Figure 7) implements marking the roots and tracing the heap. (The threads are not stopped for this stage.)

Procedure Trace (Figure 8) traces a single object. It gets an object in the input. This object is traced only if its color is *white*. If it is white, the collector tries to determine the object content (in particular, its children) as reflected in the sliding view of the cycle. If the object has changed since the sliding view was taken (line 9), then its sliding view value is obtained from the relevant *Buffer* by checking the location pointed by *LogPointer* (line 10). Otherwise, the object has not changed since the sliding view was taken. In this case, we make a copy of the object and trace the copy so that tracing

```
Procedure Trace(o: Object)
begin
    if o.color = white then
        if o.LogPointer = NULL then // if not dirty
            temp := o // getting a replica
            // is still not dirty?
            if o.LogPointer = NULL then
                for each slot s of temp do
                    v := read(s)
                    push v onto MarkStack
        else // object is dirty
            BufferPtr := getOldObject(o.LogPointer)
            for each slot s of BufferPtr do
                v := read(s)
                push v onto MarkStack
        o.color := black
end
```

Figure 8: Collector code: Trace

```
Procedure Sweep
begin
    // fourth handshake
    for each thread T do
        suspend thread T
        TraceOn := false
        resume T
    Let swept point to the first object in the heap
    while swept does not point pass the heap do
        if swept.color = white then
            swept.color := blue
            return swept to the allocator
        advance swept to the next object
end
```

Figure 9: Collector code: Sweep

will not be affected by further concurrent execution of the program. Note, that the object is marked *black* only after determining the object's sliding view content (recall that the update procedure does not log *black* objects).

Procedure Sweep (Figure 9) starts with the fourth handshake, which turns off the *TraceOn* flag. As of this time, pointer values will not be recorded anymore. This is fine since tracing has completed. Next, all *white* objects are returned to the allocator and made *blue*, to signify that they have been reclaimed. Note that by the end of the sweep all objects are black or blue. The color toggle makes use of this fact. One may think of black as white and continue to use the same color for allocation. During the next mark, the meaning of black is switched with white and the next collection starts.

Procedure Prepare-Next-Collection (Figure 10) clears all dirty marks (i.e., all *o.LogPointer*s) that were set by mutators during this collection cycle. Clearing runs concurrently with program run. The global *Roots* buffer and the local *Buffer* of each thread are also cleaned.

```
Procedure Prepare-Next-Collection
begin
    Roots := ∅
    for each thread T do
        // clear all LogPointers
        foreach object o in Buffer
            o.LogPointer := NULL
        // clear objects buffer
        Buffer := ∅
end
```

Figure 10: Collector code: Prepare-Next-Collection

4. MEMORY WEAK CONSISTENCY

Modern SMP's do not always guarantee sequential consistency. Thus, it is important to check which modifications are required by our collector to make it work on a weakly consistent platform. In this section we provide the required modifications and discuss their cost. Due to lack of space we only present the main ideas and do not get into further savings possible for our collector.

Before going through the required modifications, we would like to stress that suspending a thread implies a synchronization barrier. Thus, a handshake serves implicitly as a synchronization barrier among all threads, guaranteeing, for example, that the setting of the snoop flag is visible to all processors before the second handshake.

Dependency 1: in the write barrier, the reads and writes of the *log-pointer* (serving as the dirty flag) and the pointer slot must be executed in the order stated in the algorithm, so that several mutators do not race and write inconsistent data into the local buffers. To solve this dependency, we note that the write barrier begins with a check whether the collector is on and whether the object is not dirty. We need to add a synchronization barrier after setting the *LogPointer* and before modifying the pointer. This is done only if both checks are validated, i.e., the collector is on and the object is not dirty.

Cost: The measures in 6 show that the write barrier rarely needs to actually log an object. Thus, the vast majority of the pointer updates require no cost for handling the first dependency with weakly consistent platform.

Dependency 2: Another interaction that relies on the order of operations is the interaction between the mutators running the write-barrier and the tracing collector. There are two problems here.

The first problem occurs when the collector discovers that the object is dirty and it then reads the buffer entry associated with the object. However, if sequential consistency is not guaranteed, the buffers may not yet contain the updated values (even though the *LogPointer* has already been set). The second problem occurs when the collector copies the object contents and then reads the *LogPointer* to find it null. The collector assumes that it has an unmodified copy of the object, as it was when the sliding view was taken. However, when sequential consistency is not guaranteed, it is possible that the collector read the contents of the object after it was modified, but because of memory access reordering the setting of the *LogPointer* flag has not yet become visible to the collector.

The idea for solving the first problem is to run the tracing in phases. First trace all objects that have not been modified and keep a list of all those objects that have been modified and still need to be traced. After this phase is done, the collector runs a handshake with the mutators to obtain their local buffers and provide them with new buffers. Now, a new phase begins in which we may trace through objects whose contents are recorded in the obtained buffers and through all objects that have not yet been modified. We run such phases again and again until the tracing is done. Checking the conditions that trigger the run of a new phase, one may check that one or two phases normally suffice for a typical benchmark. In particular, an object cannot be traced after the first handshake if it is *not* modified before the handshake, it is *not* traced before the handshake, and it *is* modified just after the first handshake (before it is traced). Such an event is rare in practice.

To solve the second problem, we use a "buffering" solution. Recall that because of the first dependency the mutators are running a synchronization barrier after setting the *LogPointer* and before modifying the pointer. Depending on some parameter m, the collector starts by making copies of *m* objects that appear to be not dirty. Next, it performs a synchronization barrier. Then, the *LogPointer* of each of the *m* copied objects is probed. If it is still null, then the copy of the object may be traced. Otherwise, the object is dirty and its content should be obtained from local buffers. The parameter *m* determines the frequency of running the synchronization barrier, and in this sense the larger *m* the better. However, a large *m* implies a large buffer for copying objects, and also a somewhat increased probability that the copied object has been modified during the (longer) time interval between the time it was copied and the time its *LogPointer* was checked.

Cost: Running a couple of additional handshakes for each collection cycle is of negligible cost compared to the overall running time of the collection cycle (and to the running time of the program). Running a synchronization barrier once for every *m* collector operations is negligible for *m* large enough.

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

5. AN IMPLEMENTATION FOR JAVA

We have implemented our algorithm in Jikes [1], a Java virtual machine (upon Linux Red-Hat 7.2). The entire system, including the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory). Jikes uses *safe-points*: rather than interrupting threads with asynchronous signals, each thread periodically checks a bit in a condition register that indicates that the runtime system wishes to gain control. This design significantly simplifies implementing the handshakes of the garbage collection. In addition, rather than implementing Java threads as operating system threads, Jikes multiplexes Java threads on *virtual-processors*, implemented as operating-system threads. Jikes establishes one virtual processor for each physical processor.

5.1 Memory allocator

Our implementation employs the non-copying allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [6]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists build from 16KB pages divided into fixed-size blocks. Large objects are allocated out of 4KB blocks with first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

6. MEASUREMENTS

Platform and benchmarks. We have taken measurements on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processor and 2GB of physical memory. The benchmarks we used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark. These benchmarks are described in detail in SPEC's Web site[29]. We feel that the multithreaded SPECjbb2000 benchmark is more interesting, as the SPECjvm98 are more appropriate for clients and our algorithm is targeted at servers. We also feel that there is a dire need in academic research for more multithreaded benchmarks. In this work, as well as in other recent work (see for example [2, 13]) SPECjbb2000 is the only representative of multithreaded applications.

Testing procedure. We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Our standard automated run runs each benchmark five times for each of the JVM's involved (each implementing a different collector). To get additional multithreaded benchmarks, we have also modified the _227_mtrt benchmark from the SPECjvm98 suite to run on a varying number of threads. We measured its run with 2, 4, 6, 8 and 10 threads. Finally, to understand better the behavior of our collector under tight and relaxed conditions, we tested it on varying heap sizes. For the SPECjvm98 suite, we started with a 24MB heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we used larger heaps, starting from 256MB heap size and extending by 64MB increments until a final large size of 704MB.

The compared collectors. We tested our concurrent collector against 2 collectors: the Jikes concurrent collector and the Jikes parallel load-balancing non-copying mark-and-sweep collector. Both collectors are distributed with the Jikes Research Java Virtual Machine package.

The concurrent collector is a modern on-the-fly pure reference counting collector developed at IBM and reported in Bacon et al. [2]. It has similar characteristics to our collector, namely, the mutators are only very loosely synchronized with the collector, allowing very low pause times. This collector is denoted hereafter *the Jikes concurrent collector*. We chose this collector, as it is the only on-the-fly collector that is available for comparison.

The stop-the-world collector associates a collector thread for each processor. This is a modern stop-the-world mark-and-sweep parallel collector initiated when an allocation fails. We refer to this collector later as *the Jikes STW (stop-the-world) collector*. We chose this collector as a representative efficient stop-the-world collector.

6.1 Pause times

The maximum pause times for the runs of the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in table 1. The SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 (with 1,2,3 warehouses) were run with a 256MB heap size. In these measurements, the number of program threads is smaller than the number of CPU's. Note that if the number of threads exceeds the number of processors, then large pause times appear because threads lose the CPU to other mutators or the collector. The length of such pauses depends on the operating system scheduler and is not relevant to the collector. Hence we report only settings in which the collector runs on a separate spare processor.

Our maximum pause time measured for all the run benchmarks was 2.04 ms. Our pause times are smaller than those of the Jikes concurrent collector for all tested benchmarks. One may wonder why these pause times are shorter than the ones reported for the Jikes concurrent collector. Usually, the longest pause time for an on-the-fly collector is the time required for scanning the roots of a single thread, which is the same for both collectors. We discovered that the longest pauses in the Jikes concurrent collector are due to freeing blocks for the allocator that is sometimes executed in addition to scanning the roots. For our collector the operation of scanning the roots is the longest pause. Other pauses are an order of magnitude shorter than the root-scanning handshake. Thus, our collector obtains shorter pauses than the Jikes concurrent collector.

As expected the maximum pause times measured for our collector were much smaller than those of the Jikes STW collector. In fact, the measurements show that the maximum pause times of the Jikes STW collector are larger by a factor of at least 200!

Note that pause measurement for the _222_mpegaudio benchmark is not included for the STW collector, since it has low allocation activity and no collection is executed during its run (using the STW collector).

Benchmarks	Maximum pause time (milliseconds)		
	Sliding Views	Jikes concurrent	Jikes STW
jess	1.3	2.77	261
db	0.66	1.84	193
javac	2.04	2.81	645
mpegaudio	0.54	0.8	-
jack	0.91	1.66	226
mtrt	0.91	1.80	376
jbb-1	0.6	1.79	324
jbb-2	0.73	2.6	422
jbb-3	0.93	3.15	517

Table 1: Maximum pause time in milliseconds

6.2 Server performance

6.2.1 Comparison against the Jikes concurrent collector

Our major benchmark is the SPECjbb2000 benchmark. SPECjbb2000 requires multi-phased run with increasing number of warehouses. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. The benchmark provides a measure of the throughput and we report the throughput ratio improvement. Note that a larger number is better, and we report the ratio between our collector and the compared collector. Thus, the higher the ratio, the better our collector behaves, and any ratio larger than 1 implies that our collector outperforms the compared collector.

The design point for the Jikes concurrent collector was for one collector CPU to be able to handle 3 mutator CPU's, so that for four-processor chip multiprocessors one CPU would be dedicated to collection. Thus, when comparing to the Jikes concurrent collector in this subsection, we also let the collector run on a separate spare processor and the results show mainly the ability of the concurrent collector to run concurrently without interfering with mutators work.

The measurements are reported for a varying number of warehouses and varying heap sizes in Figures 11 and 12. We can see that with small number of warehouses, both collectors act similarly with our collector doing a little better. When the number of warehouses is three and up, all 3 mutators' CPUs are in use, and the efficiency of the collector becomes more important. We can see that in this case, our collector outperforms the Jikes concurrent collector and obtains a performance improvement of up to 60%.

The SPECjvm98 benchmarks (and so also the modified _227_mtrt benchmark) provide a measure of the elapsed running time, which we report. Here, the smaller the better. In Figure 13 we report the running time ratio of our collector and the compared collector. For clarity of presentation, we report the inverse ratio, so that higher ratios still show better performance of our collector, and ratios larger than 1 imply our collector outperforming the compared collector.

As before, when running the SPECjvm98 benchmarks on a multiprocessor, we allow a designated processor to run the collector thread. Results are reported in Figure 13. Here again the collector runs concurrently with the program thread and good concurrency is the main factor in the comparison. Mostly, the collectors perform similarly with our collector usually slightly winning. The picture changes for _213_javac and _202_jess with which our collector does much better. Indeed the compared collector is known to perform badly on these benchmarks (see [2]).

Note that the cases in which the Jikes concurrent collector wins

Figure 11: SPECjbb2000 on a multiprocessor: throughput ratio for 1-4 warehouses compared to the Jikes concurrent collector.

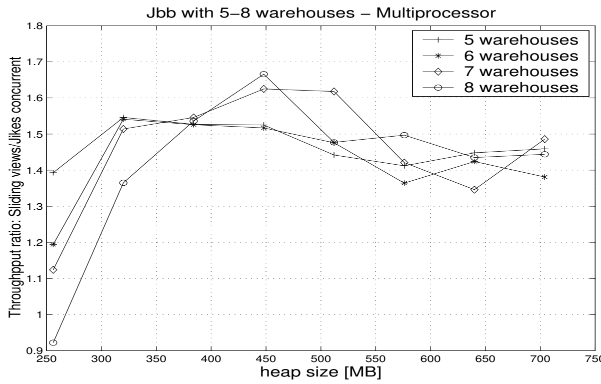

Figure 12: SPECjbb2000 on a multiprocessor: throughput ratio for 5-8 warehouses compared to the Jikes concurrent collector.

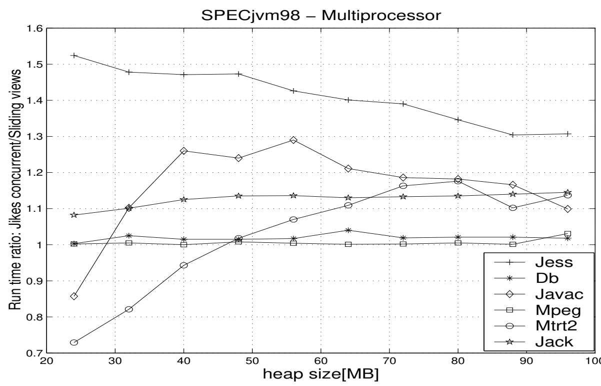

Figure 13: SPECjvm98 on a multiprocessor: run-time ratio compared to the Jikes concurrent collector.

with SPECjvm98 as well as with the modified _227_mtrt measurements presented below, is when the heap is tight. The reason for worse results on small heaps is that during those runs, we get short in memory (on both collectors), and so mutators are sometimes halted waiting for a collection cycle to terminate (and supply free space). These measurements demonstrate the superiority of reference counting (employed by the Jikes concurrent collector) for such settings. When frequent collections are performed, the tracing collector still has to trace the whole heap and sweep it, whereas the reference counting collector only needs to run over the latest modifications (in order to update the reference counts) and free the unreachable space. Note however, that this phenomena occurs only in highly stressful conditions. Normally, mutators are halted only in order to perform handshakes.

We do not include results for the _201_compress benchmark since its allocation activity is not significant.

Next, we report the measurements for the modified _227_mt-rt benchmark. We modified it to work with a varying number of threads (4, 6, 8, 10 threads) and the resulting throughput measures are reported in Figure 14. Note that a run with two threads appear with the SPECjvm98 measurements (reported as mtrt2 in Figure 13). Once more we allow a designated processor to run the collector thread, however since all 3 mutator CPU's are in use, the collector's efficiency plays the major factor in these measurements. Here, again, we can see that with small heaps the compared collector wins. As before, this happens because of the superiority of reference counting in a setting where frequent collections are required.

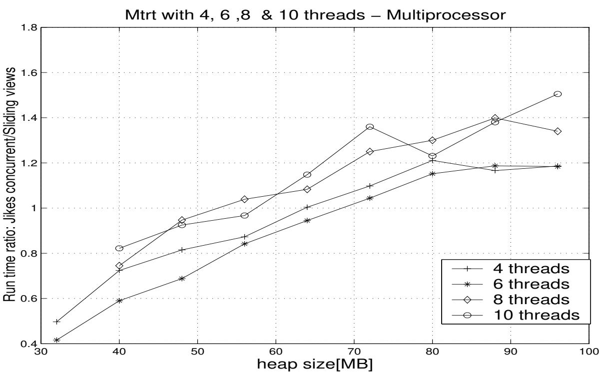

Figure 14: modified _227_mtrt results on a multiprocessor compared to the Jikes concurrent collector.

6.2.2 *Comparison against Jikes STW collector*

We have compared our collector performance over the SPECjbb2000 benchmark and SPECjvm98 benchmarks also against the Jikes STW collector. However, when comparing against the Jikes STW collector with four and up mutators (on our 4-way machine), our collector did not run on a spare processor but rather shared a processor with the program threads. Note, nevertheless, that we gave the collector (in this case) the highest priority, so that when a collection is triggered the collector would always get enough CPU.

The measurements of the SPECjbb2000 benchmark are reported for a varying number of warehouses and varying heap sizes in Figures 15 and 16. We can see that with a small (1-3) number of warehouses (when our collector runs on a dedicated processor), both collectors have similar throughput, except for 3 warehouses for small heap sizes, where the Jikes STW collector is slightly better.

Figure 16 shows that when running 4-8 warehouses over small heap, the Jikes STW collector outperforms our collector. This is the expected cost of running concurrently with program threads and using a write barrier. However, on large enough heap sizes, the compared collector is only slightly (3%-10%) better than our collector. The reason for the bad results over small heap sizes is that on those sizes our collector sometimes get short in memory, and so mutators are sometimes halted waiting for a collection cycle to terminate (and supply free space). In those cases the superiority of a parallel collector (over a concurrent collector) is expressed: the parallel collector always exploits all 4 CPUs, while our on-the-fly collector uses only one until free space is supplied.

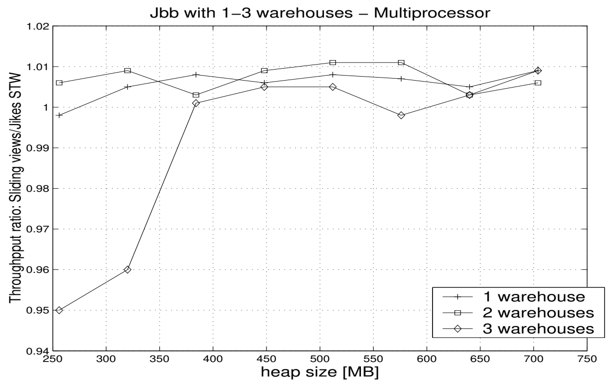

Figure 15: SPECjbb2000 on a multiprocessor: throughput ratio for 1-3 warehouses compared to the Jikes STW collector.

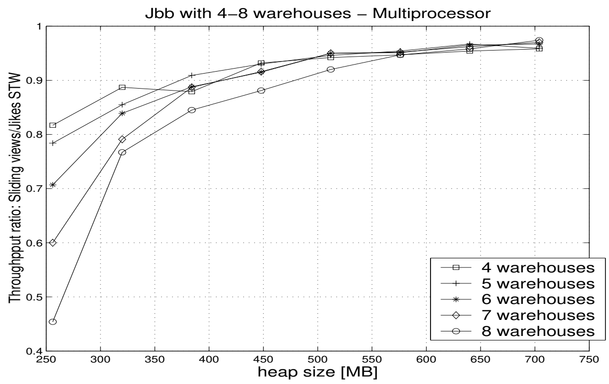

Figure 16: SPECjbb2000 on a multiprocessor: throughput ratio for 4-8 warehouses compared to the Jikes STW collector.

The measurements of the SPECjvm98 benchmark are reported in Figure 17. Here, our collector thread runs on a designated processor (i.e., the number of virtual processors is one more than the number of threads used by the benchmarks). The Jikes STW collector runs on the same number of CPU's (gaining efficiency from running the collector in parallel on them all). It can be seen that usually the collectors perform similarly. When running _213_javac and _227_mtrt with smaller heap sizes, our collector performs worst, for the same reasons described above: utilizing only one of two CPUs (three in case of _227_mtrt) when mutators are stucked due to lack of free space.

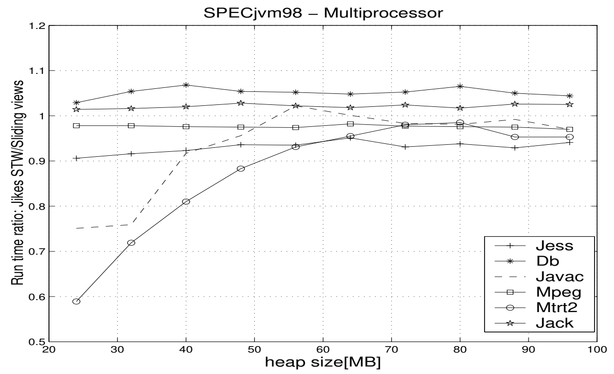

Figure 17: SPECjvm98 on a multiprocessor: run-time ratio compared to the Jikes STW collector.

Benchmarks	percent trace is on	percent not traced	percent not dirty	fraction of logging
compress	2.9	86.4	4.5	1/894
jess	5.9	3.3	3.8	1/13210
db	1.92	0.56	4.24	1/219354
javac	17.1	11.0	33.3	1/160
mpegaudio	0.04	86.0	4.6	1/64099
jack	4.2	10.6	1.4	1/16572
mtrt2	13.2	3.4	5.4	1/4116
jbb-1	2	7	8.6	1/8336
jbb-2	6.1	17.8	8.8	1/1033
jbb-3	23.3	17	8.5	1/299

Table 2: write-barrier: fraction of write-barrier executions that take the long path (on average)

6.3 Collector characteristics

6.3.1 Write-barrier measurements

The write-barrier (Figure 2) minimizes the number of object logging by using 3 filters. Table 2 shows the effect of each of those filters. Only write-barrier executions that pass those 3 filters would actually log non-null pointers of the modified object. The measurements were taken while the collector ran on a separate spare processor. The SPECjvm98 benchmarks were run with a 64MB heap size and SPECjbb2000 (with 1,2,3 warehouses) was run with a 256MB heap size.

Recall that logging should be done only from the time local roots are marked till the tracing is done. The second column shows the percentage of write-barrier executions that occur during this time. Those executions would pass the first filter (TraceOn flag was on). One can see that usually, as the number of mutators increases, the percentage of write-barrier executions that occur during this time increases, since memory is consumed faster making the collector run on a larger fraction of the overall time.

As objects that were already traced during the collection should not be logged, the third column shows the percentage of write-barrier executions in which the object (to be modified) was not yet traced, thus, this percentage of write barrier executions pass the second filter (given that it passed the first filter). Normally, a large fraction of pointer updated are initializations of newly allocated objects. As can be seen, for most benchmarks a vast majority of the objects (on which the write-barrier is executed) were already traced. This can be explained by the fact that new objects are created black.

Benchmarks	Heap size	update buffers	mark stack	snoop buffers	overall overhead
jess	64	0.26	0.05	0.12	0.43
db	64	0.28	0.15	0.07	0.5
javac	64	0.73	0.22	0.11	1.06
jack	64	0.13	0.55	0.07	0.75
mtrt	64	0.15	0.55	0.14	0.84
jbb-1	256	0.07	0.02	0.02	0.11
jbb-2	256	0.17	0.02	0.05	0.24
jbb-3	256	0.34	0.02	0.12	0.48
jbb-4-8	256	0.36	0.02	0.13	0.51

Table 3: Space overhead as a percentage of heap size

Since any object is logged at most once per collection, the fourth column shows the percentage of write-barrier executions in which the object was actually logged, given that it was not traced yet and the collector is currently tracing. This is the fraction of objects that pass the third filter (out of those which passed the first and second filter). The low percentage indicates that objects are usually modified many times. The write barrier makes sure that only one of those modifications take the long path of the write barrier.

The fifth column shows the fraction of write barriers that run the long path out of the number of all write barriers executed during the run. The measurements show that each one of the 3 filters is essential for making the long path write-barriers executions rare.

6.3.2 Write-barrier buffers' size

The space overhead consumed by the thread local buffers depends on the behavior of the application. In this section we provide some measurements providing some insight on this overhead for the benchmarks we ran. In table 3 we present the space consumed by these size-varying structures for each of the benchmarks. The numbers reported are the maximum sizes required throughout the execution. The second column presents the maximum overhead of the write-barrier buffers, the third column presents the maximum overhead of the *markStack* used for the traversal of the heap and the forth columns presents the maximum overhead of the *snoop* buffer. The last column summaries the total buffers' overhead.

The size of the buffers depends on application behavior. Specifically, the write-barrier buffers' size depends on the time consumed by the tracing phase (since the write-barrier is active only during the tracing phase), and on the number of processors used to run mutators during the tracing phase (if more processors are used to run mutators, then more objects are logged). For multithreaded benchmarks we report the overall space used for all buffers by all mutators. The measurements show that the space overhead is negligible compared to the heap size. Note that with SPECjbb2000, when the number of warehouses (mutators) go up, the volume of activity goes up and so does the space overhead of the buffers. Since we use a 4-way machine, only 3 mutators may run concurrently with the tracing operation, thus, above 3 warehouses, this overhead remains steady.

6.3.3 Profiling measurements

Our collector comprises of 4 phases: getting roots, tracing, sweeping and preparing for the next collection. Table 4 shows the percentage of time that the collector spends at each one of those phases. As can be seen, at least 97% of the collector work is spent on tracing and sweeping, while the other 2 phases are minor. The distri-

Benchmarks	percent get roots	percent trace	percent sweep	percent prepare next
jess	0.97	39.7	57.39	1.91
db	0.53	40.48	56.73	1.94
javac	0.77	57.71	39.13	2.36
mpegaudio	2.19	84.38	12.76	0.66
jack	0.9	34.85	63.02	1.22
mtrt2	0.7	54.28	43.89	1.1
jbb-1	0.29	25.03	74.13	0.55
jbb-2	0.26	28.45	70.21	1.08
jbb-3	0.37	49.55	47.62	2.45

Table 4: Percent time spent on each collection phase

bution between the tracing and the sweeping phases differs among the different benchmarks. It depends on the size of the live objects' graph and the amount of objects' freeing.

The measurements were taken while the collector ran on a separate spare processor. The SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 benchmark (with 1,2,3 warehouses) was run with a 256MB heap size.

6.4 Client performance

Although our collector is targeted at servers running on SMP platforms, as a sanity check, we also measured its performance against the Jikes concurrent collector and the Jikes STW collector on a uniprocessor. The behavior of the collector on a uniprocessor may demonstrate its efficiency. We measured our collector on a uniprocessor with the SPECjvm98 benchmark suite and the results appear in Figures 18 and 19. It turns out that our algorithm is better than the Jikes concurrent collector in almost all tests, and that its throughput does not fall below 80% of the Jikes STW collector's on most of the tests. These measurements do not serve much more than a sanity check since the compared collectors are also not targeted at running on a client machine.

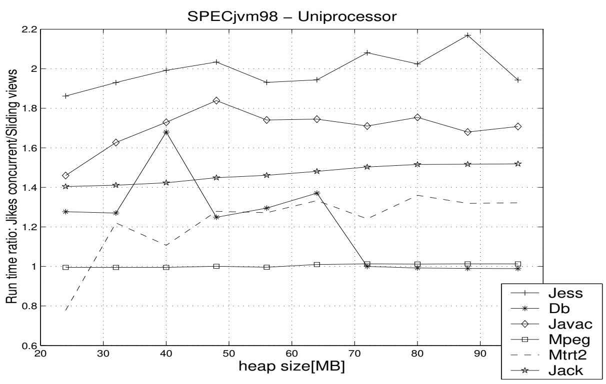

Figure 18: SPECjvm98 results on a uniprocessor compared to the Jikes concurrent collector.

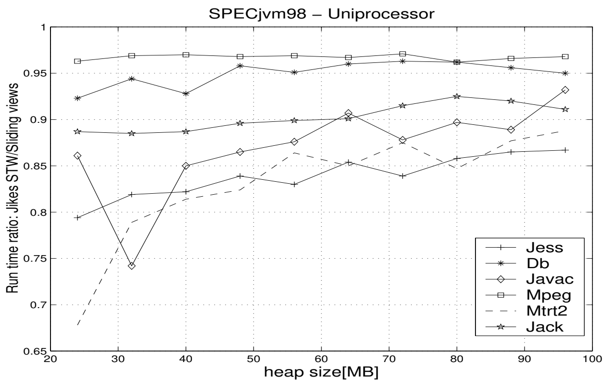

Figure 19: SPECjvm98 results on a uniprocessor compared to the Jikes STW collector.

7. CONCLUSIONS

We presented a novel on-the-fly mark and sweep garbage collector with low latency and high throughput. We have implemented our collector on the Jikes Research JVM running on a 4-way IBM Netfinity server and compared the behavior of our collector with the Jikes stop-the-world collector and the Jikes concurrent collector (both supplied with the Jikes JVM package). Comparisons to the Jikes stop-the-world collector show that the pauses have been reduced by a factor of at least 200. The longest pause measured between all runs of our collector was 2ms. When comparing the throughput with the stop-the-world collector, we see an anticipated reduction of throughput of around 10%. Comparing to the Jikes concurrent collector, we see that the pauses became shorter and the throughput has improved in almost all cases.

8. ACKNOWLEDGEMENTS

We thank David Detlefs and Eliot Moss for useful discussions.

9. REFERENCES

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeno in Java. *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

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

[3] 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 (Aarhus, Denmark), pages 14-22, New York, July 12-16 1982. Springer-Verlag.

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

[5] J. DeTreville. Experience with Concurrent Garbage Collector for Mudula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, November 1990.

[6] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, June 1991.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 274–284, 2000.

[13] 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.

[14] 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.

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

[16] Paul Hudak and Robert M. Keller. Garbage Collection and Task Deletion in Distributed Systems. In *ACM Symposium on Lisp and Functional Programming*, pp. 168-178, Pittsburgh, PA, August 1982.

[17] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC Without Stopping The World , Joint *ACM Java Grande — ISCOPE* 2001 Conference.

[18] L. Huelsbergen and P. Winterbottom. Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 50-54, 1998.

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

[20] 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.

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

[22] 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, pages 367-380, 2001.

[23] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[24] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 129-140, 2002.

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

[26] Tony Printezis and David Detlefs. A Generational Mostly-Concurrent Garbage Collector. In *International Symposium on Memory Management (ISMM '00)*, volume 36(1) of ACM SIGPLAN Notices, January 2001.

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

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

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

[30] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

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