



THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-09-02

Efficient Concurrent Mark-Sweep Cycle Collection

**Daniel Frampton, Stephen M Blackburn,
Luke N Quinane, John N Zigman**

October 2009

ANU Computer Science Technical Report Series

This technical report series is published by the School of Computer Science, College of Engineering and Computer Science, The Australian National University. Prior to 2009 this series was published as *Joint Computer Science Technical Report Series* jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
School of Computer Science
College of Engineering and Computer Science
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical-DOT-Reports-AT-cs-DOT-anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

<http://cs.anu.edu.au/techreports/>

Recent reports in this series:

- TR-CS-09-01 Peter Christen, Ross Gayler, and David Hawking. *Similarity-Aware Indexing for Real-Time Entity Resolution*. August 2009.
- TR-CS-08-03 Jie Cai, Alistair P. Rendell, Peter E. Strazdins, and H'sien Jin Wong. *Predicting Performance of Intel Cluster OpenMP with Code Analysis Method*. November 2008.
- TR-CS-08-02 Paul Thomas. *Implementation of PIS*. June 2008.
- TR-CS-08-01 Stephen M. Blackburn, Sergey I. Salishev, Mikhail Danilov, Oleg A. Mokhovikov, Anton A. Nashatyrev, Peter A. Novodvorsky, Vadim I. Bogdanov, Xiao Feng Li, and Dennis Ushakov. *The Moxie JVM Experience*. April 2008.
- TR-CS-07-05 Peter Strazdins. *Research-Based Education in Computer Science at the ANU: Challenges and Opportunities*. August 2007.
- TR-CS-07-04 Stephen M. Blackburn and Kathryn S. McKinley. *Immix Garbage Collection: Fast Collection, Space Efficiency, and Mutator Locality*. August 2007.

Efficient Concurrent Mark-Sweep Cycle Collection

Daniel Frampton

Daniel.Frampton@anu.edu.au

Stephen M Blackburn

Steve.Blackburn@anu.edu.au

Luke N Quinane

luke@quinane.id.au

John N Zigman

John.Zigman@anu.edu.au

Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia

ABSTRACT

Since 1960, reference counting has been a popular means of garbage collection. Reference counters achieve low pause times by using local data to determine liveness, but the use of this local data leaves the collector unable to collect cyclic garbage. Recent advances such as the use of coalescing and generations have dramatically improved the throughput of reference counting collectors. However, the efficient collection of cyclic garbage remains a stumbling block. This paper responds to this shortcoming with MSCD, a concurrent tracing algorithm that takes advantage of information available within a reference counted environment. MSCD outperforms alternatives such as trial deletion and backup tracing by up to a factor of two. This advantage is the result of three insights: 1) objects subject to races during concurrent tracing are trivially identified using data already established by the reference counter, 2) the trace performed by the mark phase of the collector can safely omit objects statically identified as inherently acyclic, and 3) the sweep phase of the collector can be reduced to just those objects which are potentially cyclic garbage. We show that MSCD works with state of the art reference counting collectors — which allow large numbers of heap mutations to be ignored — without affecting the correctness or completeness of the algorithm. We provide detailed performance comparisons of concurrent and non-concurrent versions of our cycle collector, a simple mark-sweep cycle detector, and a high-performance implementation of trial deletion.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Algorithms

Keywords

Java, Mark-Sweep, Cycle Collection, Reference Counting

1. Introduction

Since it was first described by Collins in 1960 [10], reference counting has held a tantalizing appeal due to its conceptual simplicity, low pause times, and prompt reclamation. The appeal was tantalizing because reference counting suffered two major drawbacks: poor throughput, and an inability to collect cyclic data. The first problem has been addressed by deferred reference counting [11], coalescing [14], and most recently with the use of generations [8, 4] and ulterior reference counting [8]. This combination yields throughput competitive with the fastest tracing garbage collectors [8]. Backup collectors are used to address the second problem, but their overhead is such that unless additional CPU resources are deployed [5, 17], throughput may suffer noticeably in applications which generate cyclic garbage. A collector which exhibits low pause times and high throughput even in the face of cyclic garbage, has thus long been a goal.

We address this longstanding objective with MSCD, a concurrent tracing algorithm that outperforms alternatives including trial deletion [9, 16, 15, 5] by close to a factor of 2, and backup mark sweep by up to 30%.¹ MSCD achieves this with a novel tracing algorithm which exploits information available within a reference counted environment to a) operate concurrently with no additional synchronization overheads, b) avoid tracing inherently acyclic data, and c) limit sweeping to potentially cyclic garbage.

Reference counting garbage collectors work by keeping counts of incoming references to each object, incrementing the count as each new reference to the object is generated, and decrementing the count as each reference to the object is deleted or overwritten. When an object's count reaches zero, there are no references to it, so it may be reclaimed. The beauty of this algorithm lies in its reliance only on local information to *directly* identify garbage. By contrast, tracing collectors must trace through all live objects before *indirectly* inferring those that are dead as those objects that were not identified as live. Unfortunately, the reliance on local information means that reference counting is unable to identify self sustaining cycles of garbage (consider two objects which point only to each other). Consequently, reference counters are generally augmented with cycle detection algorithms.

There are two broad approaches to cycle detection, *backup tracing* [13] and *trial deletion* [9, 16, 15, 5, 13], both of which may be executed either synchronously or concurrently with the application. Backup tracing involves occasionally tracing the whole heap and identifying as garbage those objects left unmarked during the trace. Trial deletion works on the basis that if a member of an unreachable self-sustaining cycle were to be temporarily deleted, the cycle would collapse and the reference count of the temporar-

¹MSCD stands for Mark Sweep Cycle Detector. We hope to have a more imaginative name for the final version of this paper.

ily deleted object would fall to zero, confirming that it was in fact a dead cycle. Trial deletion thus attempts the deletion of selected candidates, and, if they are identified as cyclic garbage, they are properly deleted. Candidates are selected by observing that a dead cycle can only be created in the uncommon case where an object’s reference count is decremented to a non-zero value (empirically, most decrements go to zero).

Both approaches suffer shortcomings. Standard backup tracing is oblivious to its context in a reference counted heap, missing opportunities for lower cost concurrency control, higher throughput and better scalability, which we exploit. Although trial deletion exhibits improved best-case behavior, in the common and worst case, it is substantially more expensive. While in some circumstances this overhead can be absorbed by dedicating additional CPU resources to the task [5, 17], we are interested in a more general solution where such resources may not be available.

Our contribution is a new concurrent backup tracing algorithm that outperforms standard backup tracing and trial deletion. Our good performance is achieved in two ways. First, we identify a new low overhead variant of snapshot-at-the-beginning concurrent collection [22, 3] that does not require any additional snapshot operations, but instead uses the set of non-zero decrements (already maintained by the underlying reference counter) to identify all objects potentially exposed to a race condition. Second, we specifically target the backup trace to the task of cycle collection. We do this by pruning both the mark and sweep phases of the trace, a) avoiding marking certain inherently acyclic objects and b) only sweeping potentially cyclic garbage.

The remainder of the paper is structured as follows. Section 2 describes the background and related work, Section 3 describes the MSCD algorithm, Section 4 describes our evaluation methodology, Section 5 evaluates MSCD against trial deletion and conventional backup tracing, Section 6 discusses future work, and Section 7 concludes the paper.

2. Background and Related Work

In this section we discuss the key background work upon which MSCD builds. First we briefly review reference counting and discuss various approaches to cycle detection. We then discuss the lightweight snapshot write barrier used by MCS. Finally, we give an overview of concurrent garbage collection in the context of cycle detection.

2.1 Reference Counting and Cycle Detection

Collins [10] first described reference counting garbage collection in 1960. Naive implementations of reference counting required increment and decrement operations to be performed at every pointer mutation, which is extremely expensive. Sixteen years later, Deutsch and Bobrow [11] addressed this source of inefficiency by deferring (temporarily ignoring) mutations to frequently mutated pointers, specifically those in registers and on the stack. The deferred pointers were accounted for at collection time when they were fully enumerated. More recently, Levanoni and Petrank [14] observed that all but the first and last in any chain of mutations to a given pointer could be coalesced. Only the initial and final states of the pointer are necessary to generate correct increments and decrements since the intervening mutations generate increments and decrements which cancel each other out. Azatchi and Petrank [4] and Blackburn and McKinley [8] concurrently and independently added generations to reference counting. Blackburn and McKinley [8] did so in a general framework of ulterior reference counting, which generalizes Deutsch and Bobrow’s notion of deferral to include heap pointers such as those within a copying nursery. Blackburn and McKinley showed that with a copying nursery, ulterior reference count-

ing could achieve throughput matching the fastest tracing collectors while still attaining good pause times.

Jones and Lins [13] give a good description of the various approaches that deal with cyclic data structures. Most approaches are not general, being either language specific, or dependent on programmer intervention. However, there exist two general approaches, backup tracing [11] and trial deletion [9, 16, 15, 5].

Backup tracing simply performs a mark-sweep trace of the whole heap from time to time. It must trace all live objects in the heap and sweep the entire heap for dead objects. Unless the tracing is performed concurrently, reference counting’s advantages of prompt reclamation and low pause times are lost. Concurrent mark-sweep collection requires some form of synchronization to avoid races with the mutator [20, 12, 3]. MSCD exploits its reference counting context to a) require no synchronization above that which already exists in a state-of-the-art reference counting collector, b) avoid marking certain objects statically identified as acyclic, and c) only sweep potentially cyclic objects.

Trial deletion is also described as partial mark-sweep [13]. Intuitively, trial deletion determines which objects are being kept alive only by virtue of reachability from some candidate object. If the candidate is only alive by virtue of reachability from itself, then it is part of a self-sustaining garbage cycle and should be collected. A trial deletion collector works by ‘trailing’ the deletion of selected candidate objects in three phases. Each phase performs a transitive closure over the subgraph reachable from the candidates: 1) The subgraph is traversed, adjusting reference counts on all objects in the subgraph to reflect the hypothetical death of the candidates. At the end of this phase the reference counts reflect only the references from objects external to the sub-graph. Any object with a count of zero is only reachable from within the subgraph. 2) The second phase restores the counts of all externally reachable objects and their referents. 3) The third phase again traces the subgraph, sweeping any objects with a zero count.

The original implementation by Christopher [9] effectively applied this three-phase approach using *all* objects in the heap as the candidate set. The advantage of the approach over simple mark-sweep is that it does not require information about external roots, which may be unavailable in some environments. However, it is substantially less efficient. Martinez et al [16] noted that a garbage cycle could only be created when a pointer to a shared object is removed. They thus check for cyclic garbage whenever a reference count is decremented to a non-zero value. Lins [15] noted that this could be prohibitively expensive, and performed cycle detection lazily, periodically targeting the set of candidate objects whose counts experienced decrements to non-zero values. Bacon and Rajan [5] made three key improvements to Lin’s algorithm. They observed that by performing the three phases of trial deletion to each candidate sequentially could exhibit worst case quadratic complexity. They solved this by performing each phase over the candidates en masse. Second, they observed that many objects can be statically identified as inherently acyclic and thus be ignored in many phases of the algorithm. Finally, they extend the algorithm to allow it to run concurrently with the mutator. Our contribution is that like Bacon and Rajan, we exploit elements of the reference counting context in our algorithm, but we do so starting with a concurrent tracing algorithm which we found to be substantially more efficient than trial deletion.

2.2 A Lightweight Snapshot Write Barrier

We now describe a low-cost, low-synchronization snapshot-at-the-beginning write barrier which is used in both reference counting collectors and snapshot-at-the-beginning concurrent tracing collec-

```

1 public void writeBarrier(ObjectReference srcObj,
2                          Address srcSlot,
3                          ObjectReference tgtObj)
4   throws InlinePragma {
5   if (getLogState(srcObj) != LOGGED)
6     writeBarrierSlow(srcObj);
7   srcSlot.store(tgtObj);
8 }
9
10 private void writeBarrierSlow(ObjectReference srcObj)
11   throws NoInlinePragma {
12   if (attemptToLog(srcObj)) {
13     enumeratePointersToSnapshotBuffer(srcObj);
14     modifiedObjectBuffer.push(srcObj);
15     setLogState(srcObj, LOGGED);
16   }
17 }
18
19 private boolean attemptToLog(ObjectReference object)
20   throws InlinePragma {
21   int oldState;
22   do { /* perform conditional store */
23     oldState = object.prepare();
24     if (oldState == LOGGED) return false;
25   } while (oldState == BEING_LOGGED ||
26           !object.attempt(oldState, BEING_LOGGED));
27   return true;
28 }

```

Figure 1: A Low Overhead Coalescing RC Write Barrier.

tors. The semantics and performance of this barrier are key to MSCD. The barrier is a slightly more efficient variation on one first used by Levanoni and Petrank [14] in 2001 for on-the-fly reference counting with backup mark-sweep, and since then has been used for ulterior reference counting and simple reference counting with trial deletion [8, 6], as well as on-the-fly mark-sweep collection [3], among others [4, 17].

Naive reference counting barrier implementations perform explicit increment and decrement operations unconditionally at the time of each pointer mutation. Better locality may be achieved by buffering increment and decrement operations and applying them en masse periodically [5]. In either case, if mutator parallelism (ie multiple user threads) is to be supported, it is necessary that the barrier operations be synchronized with the pointer store operation. The cost of performing atomic operations on every pointer mutation is significant.

Figure 1 shows Java source code for the low synchronization barrier implemented in MMTk [6]. The barrier allows ‘before’ and ‘after’ images of the pointers within each mutated object to be established and ensures that each object is only ever remembered once. It achieves this by a) taking a snapshot of the state of all pointer fields of each object prior to its first mutation since a GC (line 13), and b) recording the mutated object so that the ‘after’ state of its fields may be enumerated at GC time (line 14).

In a reference counting GC, objects referenced in the ‘before’ state receive a decrement, and objects referenced in the ‘after’ state receive an increment. It is essential to the correctness of reference counting that a) the ‘before’ snapshot of an object is not exposed to interleaving with a concurrent mutation, and b) that each object is only enqueued once for enumeration of its ‘after’ value. The synchronized guard in line 12 is therefore key to the correctness of this barrier for reference counting.

In a snapshot-at-the-beginning tracing collector (described below), only the ‘before’ image (line 13) is essential, so it may omit line 14. Furthermore, the correctness of the barrier for the snapshot-at-the-beginning tracing collector only depends on some thread capturing a consistent before-image of the mutated object. Therefore,

the snapshot-at-the-beginning tracing collector may omit synchronization altogether, removing the guard in line 12.

In line 5 a check is made to see whether the object being mutated has already been logged. For both collectors, this requires only an *unsynchronized* test of a bit in the object header. If the object has not been logged since the last GC, an out of line call is made to the write barrier slow path (lines 10-28) which may contain synchronization code, as described in the preceding paragraphs. All subsequent mutations of the object fall straight through to line 7 and simply perform the (unsynchronized) pointer store. The common case thus only involves an unsynchronized test of a bit in the source object’s header in addition to performing the pointer store. Synchronization only occurs in lines 22-26, where a *prepare/attempt* idiom is used to perform a conditional store on the infrequently taken slow path.

2.3 Concurrent Garbage Collection

An important attribute of MSCD is its capacity to run concurrently without any synchronization overhead above that inherent in the snapshot write barrier described in the previous section, which is already employed by the underlying reference counter. There is a substantial literature on concurrent garbage collection [13], with seminal work dating back to the 70’s [20, 12]. Here we focus on work that pertains to concurrent cycle detection and MSCD.

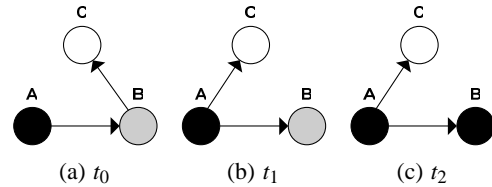


Figure 2: The Mutator-Collector Race.

Figure 2 illustrates the fundamental race that a concurrent tracing collector must address. We use the standard tri-color convention: *Black* represents a node which has been visited and need not be re-visited by the collector. *Grey* represents a node which the collector must visit. *White* represents unvisited nodes, which at the end of collection will be garbage [13]. At t_0 , the collector marks A’s sole referent, B, grey and enqueues it before marking A black (reachable). At t_1 , the mutator adds a pointer from A to C and deletes the pointer from B to C. At time t_2 the collector marks B as black (reachable) and since it has no referents, does not enqueue any objects for marking. C is thus left white (unreachable) at the end of the collection although it is live. The problem is due to the creation of a pointer from a black node to a white node (A to C).

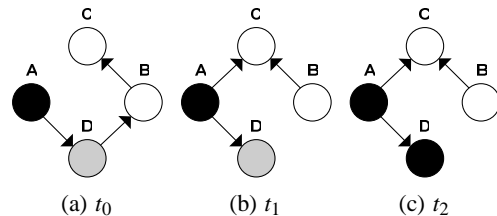


Figure 3: Adding an Extra Node to the Race.

The conditions for this race have been previously described as follows [13]:

- C1. A pointer from a black object to a white object is created.
- C2. The original reference to the white object is destroyed.

However, when, as in Figure 3, we introduce an intervening node, D, between A and B, it becomes clear that C2 needs to be

weakened to account for the white object becoming *indirectly* unreachable. We therefore postulate a weaker necessary condition:

C2.¹ The original *path* to the white object is destroyed.

Of the many solutions to this problem, we will focus on those that use a write barrier, namely those characterized by Wilson as *incremental update* and *snapshot-at-the-beginning* [21], which are predominant among contemporary concurrent non-copying tracing collectors. The goal of all such approaches is to identify the creation of any (potential) black to white pointers (C1 above). Once identified, the collector can visit the referent white objects, fixing the problem generated by the race.

There are two seminal incremental update algorithms, both dating from the mid 70's. The more conservative, due to Dijkstra et al [12], marks the *target* of a new reference grey if it was white, regardless of the color of the source. When new objects are created during collection, they are marked grey.² This ensures that there are never black to white references. Steele [20] takes a less conservative approach, marking the *source* grey if it was black and the target was white. Steele has a less conservative, more complex algorithm for determining the color of new objects than Dijkstra et al. Abstractly, Steele's algorithm retreats the grey wave-front while Dijkstra's algorithm advances it.

Intuitively, snapshot-at-the-beginning algorithms snapshot the state of all edges in the heap prior to a concurrent collection and ensure that any objects reachable in that snapshot and any newly allocated objects are kept alive by the collector. This level of conservatism means that objects which become garbage during the course of the collection cannot be collected. In practice, it would be too expensive to snapshot the whole heap. Yuasa's algorithm [22] approximates this conservatively by marking grey the *referent* of any overwritten pointer. This requires a check of the referent *every* time a pointer is overwritten. Azatchi et al [3] improve on this algorithm substantially by using a variation of the lightweight snapshot barrier described in the previous section. The first time an object is mutated, its before image is logged and a pointer to the log is recorded in an additional word in the object's header. The existence of the log pointer serves to mark the object as logged. The mark phase ensures it always traces the logged copy of the object (the before image), logging the object first if it is not already logged. Azatchi et al's algorithm is also 'on-the-fly', meaning that in addition to the tracing algorithm being concurrent, they have a concurrent means of establishing the roots for the concurrent trace.

Our algorithm is most similar to Azatchi et al's with two major differences. First, we avoid the one word overhead for storing the log pointer. We ensure all before images are traced by adding the before images to the set of grey objects. We may then treat all logged objects as if they were marked, since their before image is guaranteed to be traced. Second, we exploit the fact that our underlying reference counting algorithm already uses the lightweight snapshot barrier. We therefore incur no synchronization overhead above that which is intrinsic to our underlying reference counting collector. This means that unlike Azatchi et al, we never need to perform a synchronized logging operation during our mark phase.

Furthermore, we substantially prune the set of objects which may have been the subject of a race. We do this through the observation that any object that was the subject of such a race (object C in Figure 2) must have incurred a reference count decrement to a non-zero value. This observation is key to our algorithm and is described and justified in the following section. Finally, we target our tracing algorithm at cycle detection, and in doing so limit the scope

²More precisely, as an artifact of the freelist structure used by Dijkstra et al, in some instances new objects are marked black.

of our mark phase by avoiding tracing objects which are statically known to be acyclic and limit the sweep phase to potentially cyclic objects.

Bacon and Rajan's [5] trial deletion algorithm operates concurrently with the mutator. Recently, Paz et al [17] implemented concurrent trial deletion using the snapshot barrier after identifying a race condition in the Bacon and Rajan collector.

In summary, MSCD adapts an efficient concurrent tracing algorithm to exploit its setting as a backup to a reference counting algorithm. This allows us to perform concurrent tracing at no additional synchronization overhead, and target the trace to discovering cyclic garbage.

3. The MSCD Algorithm

MSCD is a snapshot-at-the-beginning mark-sweep collector that will *only* collect cyclic garbage. We define cyclic garbage as any unreachable data kept alive by a cycle. Only collecting cyclic garbage is sufficient since the reference counting collector will promptly reclaim acyclic garbage and only collecting cycles allows for greater efficiency by limiting the collection scope. MSCD has three optimizations over a conventional snapshot-at-the-beginning. 1) By using information gathered during reference counting, we significantly reduce the set of objects that may have been subject to a mutator race. 2) We limit the mark phase to avoid inherently acyclic objects. 3) We limit the sweep phase to only sweep potentially cyclic garbage.

The remainder of this section is structured as follows. First we give an overview of the unoptimized base algorithm upon which MSCD builds. Then we describe and argue for the correctness of MSCD's optimizations to concurrency, marking, and sweeping. Finally we explain how an MSCD invocation may span multiple phases of the underlying reference counter and the heuristics that can be used to trigger MSCD.

3.1 The Base Algorithm

The foundation on which the MSCD algorithm is built, closely follows a conventional snapshot-at-the-beginning algorithm. At the highest level the base algorithm – similar to that of Levanoni and Petrank [14] – can be described in terms of three phases and a single data structure, the *mark queue*. The mark queue is the algorithm's work queue, containing all *grey* (unvisited) objects. There are two ways an object can be added to the mark queue: 1) when an unmarked object is encountered during the trace, and 2) when a potential mutator race is detected (via the write barrier). The phases of the base snapshot-at-the-beginning algorithm are as follows:

1. **Roots.** All objects referenced by roots outside the collected space are added to the initially empty mark queue.
2. **Mark.** The mark queue is exhaustively processed.
 - 2.1 **Process** Each object is popped off the queue until the queue is empty. If the object is not already marked, its mark bit is set and each of its referents are added to the mark queue.
 - 2.2 **Check.** Any objects potentially subject to a collector-mutator race not already been added to the mark queue must be added. If this set is non-zero, return to step 2.1
3. **Sweep.** Any objects in the collected space that have not been marked are swept into the free list.

Each of these phases is refined in the sections that follow, as we add optimizations for concurrency (§3.2), marking (§3.3), and sweeping (§3.4). In Step 1 of the base algorithm, roots are established by coinciding the start of the algorithm with a reference

counting collection and using the same root set (deferred and ulterior reference counting collectors must enumerate all external references into the collected heap). In the base algorithm, all references recorded by the snapshot barrier during the course of Steps 1 and 2 are added to the mark queue in Step 2.2. This is done by performing Step 2.2 during a reference counting collection, when the before image established by the snapshot barrier is enumerated for decrements. In the base algorithm Step 3 is performed by walking the heap and placing all unmarked objects on the free list.

Throughout the rest of this paper we will refer to this algorithm as the *base snapshot-at-the-beginning algorithm* or simply the *base algorithm*. This is the cycle detection algorithm used by Levanoni and Petrank [14] (they use it with a reference counter that establishes its root set on-the-fly). We now describe each of the optimizations we apply to the base algorithm.

3.2 Concurrency

Our first optimization is to exploit information at hand in our reference counting context to substantially reduce the conservatism of the base algorithm. We begin by defining the *fixup set* as the set of all objects added to the grey queue in the ‘check’ phase (Step 2.2). Our optimization reduces the size of the fixup set from the set of all objects which suffered overwritten pointers, to the subset of those which also experienced a decrement to a non-zero reference count.

Recall the necessary conditions C1 and C2’ for a race, described in Section 2.3: ‘C1. A pointer from a black object to a white object is created.’, and ‘C2’. The original path to the white object is destroyed’. We now make the following claims:

1. For C2’ to occur, *either the white object or some object in the original path still connected to the white object will be subject to a reference count decrement to a non-zero value.*
2. When C2’ arises, *it is correct and sufficient to add to the fixup queue either the white object or any object in the original path still connected to the white object.*

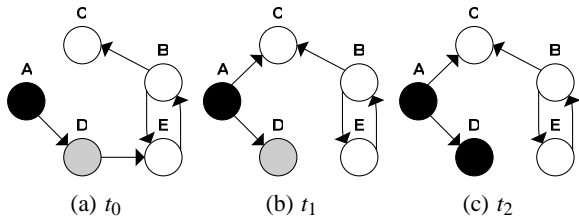


Figure 4: Adding a Cycle to the Mutator-Collector Race.

We justify our first claim as follows. For the path from A to C (in Figures 2, 3, and 4) to have been broken, one of three cases must have occurred:

- a) A pointer to C was destroyed,
- b) A pointer to some object X was destroyed, where X formed part of an *acyclic* path from A to C (B or D in Figure 3, D in Figure 4), or
- c) A pointer to some object X was destroyed, where X formed part of a *cyclic* path from A to C (E in Figure 4).

To understand case a), consider time t_1 in Figure 2. The pointer from A to C must be established before deleting the pointer from B to C. Therefore C experiences a decrement to a non-zero reference count. For case b), consider Figure 3. The deletion of the reference to the acyclic path will cause the object(s) in the path to be transitively reclaimed by the reference counter (ie B will be reclaimed, deleting the pointer to C), reducing case b) to case a). For case c), consider Figure 4. The deletion of the reference will cause some

object in the cyclic path to C to have its reference count reduced (ie object E), and since it is also part of a cycle, it must experience a decrement to a non-zero reference count (so object E will be added to the fixup queue). Thus whenever condition C2’ arises, either the white object or some object in the path to the white object will experience a decrement to a non-zero reference count and thus be added to the fixup set, satisfying our first claim.

Our second claim is trivial. Since any object in the fixup set will be traced, it is sufficient to add any object that reaches the white object to the fixup queue. Furthermore, once an object which forms the original path to the white object is made unreachable, the path cannot be changed by the mutator. We therefore claim that it is sufficient and correct to use the set of objects which experienced a decrement to a non-zero reference count as the fixup set.

Now Martinez et al and all trial deletion algorithms that followed noted that a decrement to a non-zero reference count is a necessary condition for the generation of cyclic garbage (Section 2.1). Three interesting, previously established properties follow: 1) Decrements to non-zero reference counts are empirically known to be uncommon, which is why they are used to reduce the set of candidates for trial deletion. 2) The condition is trivially identified by the reference counter during batch-processing of decrements. 3) The condition is robust to coalescing of reference counts (exploited by Blackburn and McKinley [8] and discussed by Paz et al [17]).

MSCD therefore reduces its set of fixup candidates to just those that experience decrements to non-zero reference counts, which Bacon and Rajan referred to as *purple* objects [5]. It is important to note that the correctness of this optimization to MSCD only depends on purple object identification occurring during heap tracing, while it is required at all times for trial deletion. In the results section we will show that the overhead of continually maintaining the purple sets is measurable, so performing the operation only on demand may be sensible. However, the MSCD sweeping optimization described below does depend on the purple set being continually maintained between each invocation of the cycle detector.

3.3 Marking

Our second optimization is to reduce the scope of the mark phase by avoiding objects which were statically identified as being inherently acyclic. We use a simple method of determining acyclic classes in Java proposed by Bacon and Rajan [5]. A class is said to be acyclic if it contains no pointer fields, or if it can point only to acyclic classes. Bacon and Rajan set a *green* bit in an object’s header at allocation time if it is acyclic, and curtail the scope of each trial deletion trace to avoid tracing green objects. We trivially modify Step 2.2 above to consider an object marked if the object is either marked *or* green. Since by definition a green object may not point to a non-green object, only green objects will fail to be correctly marked by our optimized trace. As long as the sweep phase considers objects marked whether they are marked *or* green, no objects can be incorrectly collected as a result of this optimization. The effectiveness of the optimization depends on the proportion of green objects in the heap. Table 1 shows that for many benchmarks a large proportion of all objects are allocated green.

3.4 Sweeping

Our third optimization is to limit the scope of sweeping to sweep only potentially cyclic objects and their children. We do this by using the same definition of potentially cyclic as used in Section 3.2: objects subject to decrement to non zero reference counts, which we refer to as *purple*. Rather than sweep the entire heap for unmarked objects, we note that the collector need only collect cyclic garbage, and therefore target our sweep at the potentially cyclic garbage identified by the purple set. This optimization is complete

since all acyclic garbage will be collected by the reference counting collector.

3.5 Interaction With The Reference Counter

To this point we have not discussed in detail the relationship between the operation of MSCD and the phases of the underlying reference counter. In short, there are just three requirements. First, the root set needs to be established atomically with respect to the mutator – achieved by piggy backing on an invocation of the deferred reference counter which must also establish roots. This may be done in either a stop-the-world or on-the-fly manner [14]. Secondly, the fixup set must be added to the grey queue (Step 2.2 above) at a point where the set is known to be complete. This is trivial when the reference counter operates in stop-the-world phases. If all mutators are suspended, then it is sufficient to first process all increments and then all decrements before determining the fixup set. The third requirement is that the reference counter not free any objects known to MSCD (which would make MSCD’s reference a dangling pointer). Since MSCD only maintains a mark queue (grey objects) and a fixup queue (purple objects), it is sufficient to address these. The reference counter therefore does not free any grey or purple objects. Instead it adds them to a ‘free buffer’ for freeing at the completion of the MSCD invocation.

3.6 Invocation Heuristics

A detailed analysis of heuristics for invoking cycle detectors is outside the scope of this paper. The most simple policy is to invoke the cycle detector whenever the underlying reference counter is unable to free as much memory as the user requires. More generally, a heap fullness threshold could be used as a trigger for cycle detection. Trial deletion and MSCD cycle detectors both depend on the size of the purple set, so they may use the size of the purple set as a cycle detection trigger. Significantly, since both trial deletion and MSCD place exactly the same requirement upon the underlying reference counter (the establishment of the purple set), the collectors can be interchanged dynamically at runtime. Although backup tracing does not require the establishment of the purple set, it is always possible to switch to it from MSCD or trial deletion. It is however only possible to switch from backup tracing immediately after a complete cycle detection.

Finally, our implementation of MSCD will be publicly available as a patch against Jikes RVM,³ and we hope will soon be properly integrated into the standard MMTk/Jikes RVM release.

4. Methodology

This section first briefly describes Jikes RVM and MMTk which are publicly available and provide a common implementation framework for the collectors evaluated in this paper. We then present the characteristics of the machines on which we do all experiments, and some features of our benchmarks.

4.1 Jikes RVM and MMTk

We use MMTk in Jikes RVM version 2.3.4+CVS, with patches to support *pseudo-adaptive* compilation. MMTk is a flexible high performance memory management toolkit used by Jikes RVM [6]. Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [1, 2]. We use configurations that precompile as much as possible, including key libraries and the optimizing compiler and turn off assertion checking (the *Fast* build-time configuration). The adaptive compiler uses sampling to select

³See the Jikes RVM *Research Archive* tracker at the Jikes RVM web page, <http://www.jikesrvm.org>.

methods to optimize, leading to high performance but a lack of determinism. Since our goal is to focus on application and garbage collection interactions, our *pseudo adaptive* approach deterministically mimics adaptive compilation.⁴

By using the MMTk framework we are able to perform an apples-to-apples comparison of the collectors, as all base mechanisms are shared by the different collectors. MMTk accounts for all space consumed by metadata as part of the overall memory consumption (including work queues, free list metadata, snapshot buffers, and decrement buffers).

Benchmark	Green	Cycle	Alloc	Min Heap
antlr	85%	13%	301MB	13MB
bloat	43%	12%	684MB	22MB
fop	69%	28%	66MB	24MB
hsqldb	65%	14%	592MB	21MB
jython	0.6%	4%	462MB	13MB
pmd	17%	24%	322MB	20MB
ps	46%	2%	572MB	9MB
xalan	89%	58%	77MB	99MB
_201_compress	91%	93%	116MB	14MB
_209_db	11%	1%	90MB	16MB
_228_jack	72%	2%	271MB	8MB
_213_javac	47%	23%	241MB	20MB
_202_jess	8%	2%	300MB	9MB
_222_mpegaudio	6%	45%	5MB	8MB
_227_mtrt	21%	6%	173MB	16MB
_205_raytrace	20%	4%	163MB	12MB
pseudojbb	47%	14%	315MB	36MB

Table 1: Benchmark Characteristics.

4.2 Experimental Platform

We use an 2.2GHz AMD 64 3500+ for our experiments. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, exclusive 512KB 16-way set associative L2 cache. The Athlon has 2GB of dual channel 400 DDR RAM configured as 2 × 1GB DIMMs with an nForce3 Ultra MSI K8N Neo2 motherboard and 800MHz front side bus. It runs Linux 2.6.10.

We run each benchmark at a particular parameter setting five times and use the second fastest of these. The variation between runs is low, and we believe this number is the least likely disturbed by other system factors and the natural variability of the adaptive compiler.

4.3 Benchmarks

Table 1 shows key characteristics of each of the 17 benchmarks we use. The DaCapo suite [7] is a recently developed suite of non-trivial real-world open source Java applications. We use version beta050224. We also use the SPECjvm98 suite and *pseudojbb*, a variant of SPEC JBB2000 [18, 19] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. The first column shows the fraction of allocated objects by volume which are statically determined to be green (acyclic) using the definition given in Section 3.3. The second column shows the fraction of objects, by volume, which become cyclic garbage. The third column shows the total volume of allocation. The final column shows the minimum heap in which each benchmark will run using MMTk’s default GenMS collector.

5. Results

This section compares the performance of three cycle detectors: MSCD, backup mark-sweep and trial deletion. We explore performance three ways: 1) First, we take a limit study of raw cycle

⁴Xianglong Huang and Narendran Sachindran jointly implemented the pseudo adaptive compilation mechanism.

Benchmark	Backup Tracing			MSCD			Trial Deletion		
	CD Time msec	Visits millions	Visit Cost nsec	CD Time /BT	Visits /BT	Visit Cost /BT	CD Time /BT	Visits /BT	Visit Cost /BT
_202_jess	89.98	11.62	7.74	0.81	0.88	0.93	1.76	1.17	1.51
_205_raytrace	91.25	11.95	7.64	0.78	0.87	0.90	1.64	1.28	1.28
_209_db	92.38	12.12	7.62	0.73	0.85	0.86	1.52	1.15	1.32
_213_javac	118.33	14.59	8.11	0.94	0.92	1.02	1.94	1.99	0.98
_222_mpegaudio	66.74	8.94	7.46	0.77	0.90	0.86	1.65	1.23	1.34
_227_mtrt	99.75	12.90	7.73	0.80	0.87	0.92	1.65	1.32	1.25
antlr	95.78	11.84	8.09	0.91	0.90	1.01	1.86	1.40	1.33
bloat	116.56	14.12	8.25	0.81	0.91	0.89	1.65	1.55	1.07
fop	108.97	12.94	8.42	1.02	0.92	1.10	1.88	1.93	0.97
hsqldb	104.29	13.01	8.02	0.83	0.81	1.02	1.70	1.44	1.18
jython	105.62	13.24	7.98	0.81	0.90	0.90	1.70	1.30	1.30
xalan	108.82	13.23	8.23	0.86	0.90	0.95	1.73	1.49	1.16
pseudobb	126.29	13.83	9.13	0.71	0.84	0.84	1.41	1.55	0.91
geomean				0.83	0.88	0.93	1.69	1.43	1.19
mean	101.9	12.64	8.03	0.83	0.88	0.94	1.70	1.45	1.20

Table 2: Throughput Limit Study, 8MB Collection Period, Average Costs Per Cycle Detection.

detection throughput, where the cycle detector is forced to run at set intervals in a non-concurrent setting, 2) We then examine concurrency, measuring the efficiency of the MSCD concurrency optimization and 3) Finally, we compare overall performance in a more natural setting, where the collectors are invoked only when deemed necessary by a cycle detection heuristic.

5.1 Throughput Limit Study

We begin with a limit study where we analyze the fundamental efficiency of three cycle collectors: the base snapshot-at-the-beginning mark sweep, trial deletion, and MSCD. All three cycle detectors are correct and complete, and are able to collect any cyclic garbage present in a given heap. Abstractly, our approach is to expose each collector to a large number of cycle detection opportunities and measure the efficiency with which they process them. Concretely, we achieve this by forcing the cycle detectors to collect a reference-counted heap after a fixed volume of allocation. In each case the cycle detector is invoked in a stop the world manner, factoring out issues relating to concurrency. It should be clear that this is simply an analytical tool, not a practical way to collect cycles. We will examine the overall cost of the collectors in a natural setting with more realistic invocation heuristics in the sections that follow.

Since all collectors execute in a stop the world setting, this analysis is limited to examining the effectiveness of the MSCD mark and sweep optimizations (Sections 3.3 and 3.4). The effectiveness of the MSCD concurrency optimization is addressed in Section 5.2.

Figure 5 shows the overall throughput of MSCD and trial deletion algorithms normalized against the snapshot-at-the-beginning base algorithm for cycle detection periodicities ranging from 128KB to 128MB. Here we show the average time for each cycle detection,

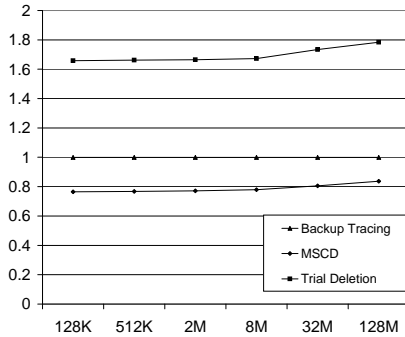


Figure 5: Throughput Limit Study, Varying Collection Period.

taking the geometric mean of this value for all 17 benchmarks. We see that MSCD tends to outperform the base algorithm by around 20% and that trial deletion tends to perform around 70% worse than the base algorithm. Figure 5 suggests that for this set of benchmarks, the overall throughput of the various collectors is fairly stable across a wide range of invocation frequencies. While we do not discount the possibility of different trends beyond the scope of our experiment, we note that our upper limit (128MB) is in the order of the total volume of allocation for many benchmarks (see Table 1).

In Table 2 we see a breakdown of results for each of the 17 benchmarks at an 8MB invocation period (the middle point in our 128K to 128MB range). The first three columns show *absolute* numbers for cycle detection time (msec), the number of nodes visited (millions), and the average node visit cost (msec/million visits \equiv nsec/visit) for the base algorithm. The remaining six columns show the corresponding data for MSCD and trial deletion, *normalized* against the results for the base algorithm. We see that for all benchmarks, trial deletion is substantially and consistently slower than the base algorithm. The best result we see is a 41% degradation for pseudobb. By contrast, MSCD outperforms the base algorithm on 16 of the 17 benchmarks, with a degradation of just 2% on fop. On all other benchmarks we see consistent improvements of around 10 to 30%.

The fifth column indicates that on average MSCD visits about 12% fewer nodes than the base algorithm, indicating the efficacy of the MSCD mark optimization (Section 3.3). This reduction is not as great as might be expected from the often very high proportion of green objects shown in Table 1. There are two explanations for this. First, we must still visit (but not scan) all green objects on the fringe of a green subgraph in order to determine that they are green. These non-scanning visits are included in our visit count. Secondly, the numbers in Table 1 reflect the total proportion of *allocated* objects which are green, whereas the results here depend on the proportion of green objects present in the heap at any time. Thus if green objects were disproportionately short lived, Table 1 would tend to over-represent their significance in the heap.

The sixth column indicates that the average cost of a node visit is also lower in MSCD than the base algorithm. This is due to two factors. First, as mentioned above, the mark optimization means that in MSCD, visits to green objects are cheap, as they do not involve a scan. Second, since this figure is calculated by dividing the total CD time by the number of nodes visited, the MSCD sweep optimization (Section 3.4) reduces the overall CD time, and thus reduces the average visit cost.

5.2 Concurrency

All of the experiments described in the previous section were performed in a stop-the-world setting, and therefore precluded any analysis of our concurrency optimization. Recall that the concurrency optimization (described in Section 3.2) allows us to prune the fixup set, which is used by the snapshot-at-the-beginning collector to account for any races with the mutator. Since the only effect of the optimization is to prune the fixup set and thus reduce the total number of nodes to be processed, we expected it to be a straightforwardly effective optimization.

It was our intention to measure the efficacy of our concurrency optimization in the context of a symmetric multithreaded (SMT) processor (specifically an Intel Pentium 4 with hyperthreading). Our rationale is that this provides a reasonable opportunity for true concurrency and an opportunity for the cycle detector to consume potentially wasted instruction level parallelism without extravagantly dedicating an entire CPU to the task of cycle detection. Unfortunately, at the time of submission we are unable to resolve a concurrency bug affecting both the base algorithm and MSCD, that is only exposed in SMT and SMT settings.

Both algorithms performed correctly in a time-slicing context, so we performed a preliminary analysis in that environment. To our surprise (and disappointment!), the optimization had no measurable impact. We observed a small slowdown and noticed that it was due to the overhead of continuously maintaining the set of purple objects (those that experience a decrement to a non-zero value). Since the concurrency optimization only requires the purple set to be maintained while the collection is in progress, we re-ran the experiments with this further optimization. We were disappointed to find that although we avoided the purple set slowdown, any net advantage over the base algorithm was negligible.

After further analysis, we noted that since the mutator and collector were executing in time slices, not truly concurrently, opportunities for data races were extremely small, so the fixup work was generally a no-op. Therefore the utility of the optimization was small. We hope to have more encouraging results for a final version of this paper.

5.3 Overall Performance

Finally, we evaluate the performance of the collectors triggered only when guided by straightforward heuristics. For all of these experiments we used a simple heuristic which triggered a cycle detection only when the reference counter was unable to reclaim enough space in a fixed sized heap. Because we do not have a concurrent implementation of trial deletion, each of the cycle detectors were invoked in a stop-the-world manner.

Figure 6 shows detailed performance of three representative benchmarks, giving total time, overall GC time (inclusive of cycle detection), cycle detection time, and mutator time as a function of heap size. We measure the performance of the base algorithm, MSCD with mark and sweep optimizations, MSCD with just the mark optimization, and trial deletion. All of the graphs plot time in seconds on a logarithmic scale. Recall that we use a naive heuristic which triggers cycle detection on the basis of heap fullness. For `_213.jvac`, cycle detection costs become noticeable at heap sizes less than $\times 2.5$, while for `_202.jess` and `bloat` cycle detection costs are noticeable for heap sizes less than $\times 4$ (Figures 6(g)–6(i)).

The most surprising result in Figure 6 is that the MSCD with mark and sweep optimizations performs worse than MSCD with just the mark optimization except in the tightest heaps. The sweep optimization uses the purple set (potentially cyclic garbage) to target the sweep at just those objects, avoiding sweeping the whole heap. Closer analysis reveals that any advantage in a more targeted sweep is lost to purple set maintenance. The correctness of the

sweep optimization requires that the purple set contain *all* purple objects identified since the last cycle detection. This is true of trial deletion also.

Purple set maintenance costs both space and time. The space overhead leads to heap pressure and consequently increased GC load, evident in Figures 6(g) and 6(i). Here we see trial deletion and MSCD with mark and sweep optimizations continue to perform measurable cycle detection work in large heaps while the others perform none. The time overhead is due to the need to filter the purple set periodically to remove objects which have been collected by the reference counter. As the cycle detections become less frequent, the size of the purple set accumulates over a longer time, becomes larger, and requires more filtering, explaining why at the tightest heaps the sweep optimization is not harmful. This result suggests a hybrid approach. Since the sweep optimization only makes sense when the purple set is small, a cap could be placed on the purple set size. Once the cap was exceeded, the purple set could be discarded and not maintained until the next cycle detection phase – where the cap would be reinstated and the process started again. The hybrid would thus dynamically choose whether to use the sweep optimization. We have not yet evaluated this hybrid. The overhead of dealing with the purple object buffer could also be addressed by using a purple object bitmap. This would come at a constant space overhead, but would avoid the need for filtering. A combination of modest buffers and a bitmap updated by a single thread would avoid the need for atomic bitmap updates. We have not yet evaluated any of these alternatives.

There are two other notable results to be drawn from Figure 6. The first is that in benchmarks such as `_202.jess` which allocate large numbers of short lived objects, the overhead of setting the grey bit on newly allocated objects is measurable. This is clear in Figure 6(d), where trial deletion holds a clear mutator time advantage over the others. Our initial experience was that setting the green bit in newly allocated acyclic objects was also expensive, but we modified Jikes RVM’s optimizing compiler to ensure that the state of the green bit was statically determined. The final result is perhaps the most striking of all, and that is the need for good heuristics for triggering of cycle detection. Our heuristic is obviously naive, as `_202.jess`, which has very little cyclic garbage, spends as much as half of its total running time performing cycle detection in tight heaps. As has been noted previously [8, 6, 17], generational reference counting can substantially reduce the cycle detection load. It does so three ways. 1) Only mature objects are exposed to time and space overheads associated with the reference counted object header, such as the cost of setting the grey and green bits. 2) Short lived cyclic garbage is efficiently collected in the nursery. 3) Since nursery objects, which are heavily mutated, are not subject to increments and decrements, the purple object maintenance load is significantly reduced.

6. Future Work

While the demands of scalability and responsiveness continue to grow, we see reference counting, particularly when combined with generational collection, as an increasingly important approach to garbage collection. There are many clear avenues for future work, some of which we wish to address in the final version of this paper. First, we wish to apply our detailed approach and analysis of the performance of cycle detection algorithms in a truly concurrent setting, as we had intended in Section 5.2. The widespread availability of SMT processors is, to us, an obvious opportunity for concurrent garbage collection. Second, heuristics for triggering cycle detection are of utmost importance and yet, to our knowledge, under-evaluated in the literature. Third, our analysis of the vari-

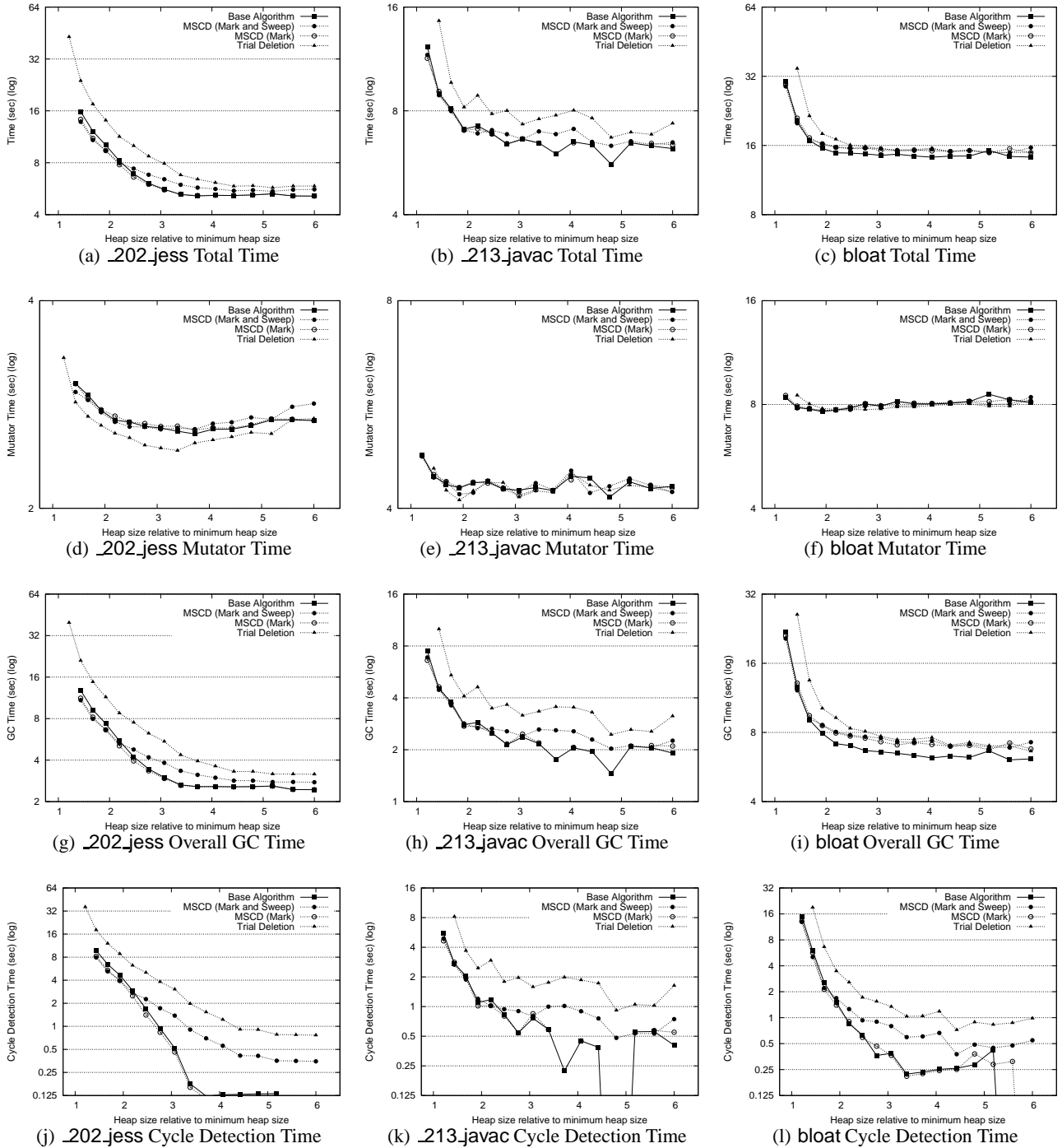


Figure 6: Total, Mutator, GC and Cycle Detection Performance

ous approaches to concurrent cycle detection opens the door to a number of hybrid approaches, both in the application of our optimizations, and to the choice between mark-sweep and trial deletion. Our infrastructure and the demands of the respective algorithms allow a choice to be made at the end of each cycle detection as to which collector will be used next. Fourth, we plan to add ulterior reference counting to our analysis in the final version of this paper. Finally, we would like to explore parallelizing the concurrent

mark-sweep for use in large-scale SMP settings.

7. Conclusion

Recent major improvements to the performance of reference counting [14, 8, 4] have renewed interest in reference counting and placed renewed pressure on the need for efficient cycle detection. In this paper we present a detailed analysis of concurrent cycle detection, offer a base algorithm with minor improvements over the state of the art snapshot-at-the-beginning collector [3], and present a new

algorithm, MSCD, which optimizes the base algorithm three ways. We show that in the limit, MSCD is more efficient at detecting cycles than the base algorithm and trial deletion. We also analyzed the algorithms when triggered according to a naive heuristic and noted that our sweep optimization degraded performance – unless cycle detections occurred very frequently. Finally we proposed dynamic application of the optimizations and identified avenues for future work.

8. REFERENCES

- [1] B. Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, Nov. 1999.
- [2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.
- [3] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 269–281. ACM Press, 2003.
- [4] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003.
- [5] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proc. of the 15th ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.
- [6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.
- [7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.
- [8] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.
- [9] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.
- [10] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.
- [11] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
- [12] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science*, No. 46. Springer-Verlag, New York, 1976.
- [13] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.
- [14] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, Oct. 2001.
- [15] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.
- [16] A. D. Martinez, R. Wachenchauser, and R. D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.
- [17] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan. Efficient on-the-fly cycle collection. In *Fourteenth International Conference on Compiler Construction*, Edinburgh, Scotland, Apr. 2005.
- [18] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
- [19] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.
- [20] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, Sept. 1975.
- [21] P. R. Wilson. Uniprocessor garbage collection techniques. In *IWMM'92, Proceedings of International Workshop on Memory Management, St Malo, France, Sep 16–18*, volume 637 of *Lecture Notes in Computer Science*, 1992.
- [22] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.