Proc. of the European Conference on Object-Oriented Programming, Glasgow, UK, July 2005

Derivation and Evaluation of Concurrent Collectors

Martin T. Vechev[1], David F. Bacon[2], Perry Cheng[2], and David Grove[2]

[1] Computer Laboratory, Cambridge University
Cambridge CB3 0FD, U.K.

[2] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Abstract. There are many algorithms for concurrent garbage collection, but they are complex to describe, verify, and implement. This has resulted in a poor understanding of the relationships between the algorithms, and has precluded systematic study and comparative evaluation. We present a single high-level, abstract concurrent garbage collection algorithm, and show how existing snapshot and incremental update collectors, can be derived from the abstract algorithm by reducing precision. We also derive a new hybrid algorithm that reduces floating garbage while terminating quickly. We have implemented a concurrent collector framework and the resulting algorithms in IBM's J9 Java virtual machine product and compared their performance in terms of space, time, and incrementality. The results show that incremental update algorithms sometimes reduce memory requirements (on 3 of 5 benchmarks) but they also sometimes take longer due to recomputation in the termination phase (on 4 of 5 benchmarks). Our new hybrid algorithm has memory requirements similar to the incremental update collectors while avoiding recomputation in the termination phase.

1 Introduction

The wide acceptance of the Java programming language has brought garbage collected languages into the mainstream. However, the use of traditional synchronous ("stop the world") garbage collection is limiting the domains into which Java and similar languages can expand. The need for concurrent garbage collection is primarily being driven by two trends: the first is increased heap sizes, which make the pauses longer and less tolerable; the second is the increase in the use of, and complexity of, real-time systems, for which even short pauses are often unacceptable. Therefore there is need for rapid improvement in various kinds of incremental and concurrent collector technology.

Unfortunately, concurrent garbage collectors are one of the more difficult concurrent programs to construct correctly. The study of concurrent collectors began with Steele [28], Dijkstra [14], and Lamport [21].

Concurrent collectors were considered paradigmatic examples of the difficulty of constructing correct concurrent algorithms. Steele's algorithm contained an error which he subsequently corrected [29], and Dijkstra's algorithm contained an error discovered and corrected by Stenning and Woodger [14]. Furthermore, some correct algorithms [9] had informal proofs that were found to contain errors [26].

These problems also manifest themselves in practice because concurrent bugs generally have a non-deterministic effect on the system and are non-repeatable, so that connecting the cause of the error to the observed effect is particularly difficult.

Many incremental and concurrent algorithms have been introduced in the last 30 years [1, 3, 4, 6, 7, 10, 11, 12, 13, 16, 17, 18, 20, 22, 24, 25], but there has been very little comparative evaluation of the properties of the different algorithms due to the complexity of implementing even one algorithm correctly. As noted in [2], because of these constraints, current state-of-the-art concurrent systems are generally not quantitatively compared against each other and the exact relationships among the different concurrent schemes are largely unknown.

For example, early collectors were all examples of *incremental update* collectors which "chase down" modifications to the object graph that are made by the program during collection. Yuasa [30] introduced snapshot collectors, which do not attempt to collect garbage allocated after collection begins, but do not require any rescanning of the object graph. Thus, snapshot collectors trade off reliable termination for a potential increase in floating garbage. However, costs and benefits relative to incremental update techniques have not been systematically studied.

This paper presents a high-level algorithm for concurrent collection that subsumes and generalizes several previous concurrent collector techniques. This algorithm is significantly more precise than previous algorithms (at the expense of constant-factor increases in both time and space), and more importantly yields a number of insights into the operation of concurrent collection. For instance, the operation of concurrent write barriers can be viewed as a form of degenerate reference counting; in our algorithm, we do true reference counting and are thereby able to find live data more precisely.

Existing algorithms can then be viewed as instantiations of the generalized algorithm that sacrifice precision for compactness of object representation and speed of the collector operations (especially the write barriers).

Additionally, we argue that all of the existing concurrent algorithms fundamentally share a deeper structure. And there is a whole continuum of existing algorithms, which we have not yet explored, but could be uncovered if we start from such a structure. Moreover, by having a common abstract algorithm, much of the construction of the practical collector will be simplified.

The contributions of this paper are:

– A generalized, extendable, abstract concurrent collection algorithm, which is more precise than previous algorithms;
– A demonstration of how the abstract algorithm can be instantiated to yield existing snapshot and incremental update algorithms;
– A new snapshot algorithm (derived from the abstract algorithm) that allocates objects unmarked ("white") and reduces floating garbage without re-scanning of the heap required by incremental update algorithms;
– An implementation of four concurrent collectors in a production-quality virtual machine (IBM's J9 JVM product): Snapshot (after Yuasa), two incremental- update (after both Dijkstra and Steele), and our hybrid snapshot algorithm; and
– A quantitative experimental evaluation comparing the performance of the different algorithms.

2 An Abstract Collector

This section presents the abstract collector algorithm. The algorithm is designed for maximum precision and flexibility, and keeps much more information per object than would be practical in a realistic implementation. However, the space overhead is only a constant factor, and thus, does not affect the asymptotic complexity of the algorithm, while the additional information allows a potential reduction in complexity.

Similarly, a number of operations employed by the abstract algorithm also have constant time overheads that would be undesirable in a realistic collector. In particular, there is no special treatment of stack variables: they are assumed to be part of the heap and therefore every stack operation may incur a constant-time overhead for the collector to execute an associated barrier operation. There are a number of collectors for functional languages (such as ML and Haskell) that treat the stack in exactly this way.

Our generalized concurrent collection algorithm makes use of the framework of Bacon et al. [5]: they showed that for synchronous ("stop the world") garbage collection, tracing and reference counting can be considered as dual approaches to computing the reference count of an object. Tracing computes a least fixpoint, and reference counting computes a greatest fixpoint. The difference between the greatest and least fixpoints is the cyclic garbage. In most practical tracing collectors, the reference count is collapsed into a single bit.

Furthermore, they showed that all collectors could be considered as a combination of tracing and reference counting, and that any incrementality is due to the use of a reference counting approach with its write barriers.

This insight is now extended to concurrent tracing collectors: we show that they are also a tracing/reference counting hybrid. The collector traces the original object graph as it existed at the time when collection started, but does reference counting for pointers to live objects that could be lost due to concurrent mutation.

The abstract algorithm makes use of the variables depicted in Table 1. In the discussion that follows, we elaborate more on the semantics of each shared variable.

2.1 Restrictions and Assumptions

The algorithms we discuss are non-moving and concurrent, but not parallel. That is, the collector is single-threaded. The ideas derived from this discussion, however, are easily extendable to algorithms using multiple spaces, such as generational ones.

Furthermore, the algorithm performs synchronization with atomic sections rather than isolated atomic (compare-and-swap) operations. Atomic sections are relatively expensive on a multiprocessor, so that although the algorithm can be executed on a multiprocessor it is better suited to a uniprocessor system based on safe points, in which low-level atomicity is a by-product of the implementation style of the run-time system.

Additionally, we assume that the concurrency between the mutators and the collector is bounded by a single cycle. This is a common underlying assumption in most practical algorithms. Essentially, this means that all mutator operations started in collector cycle N finish in that cycle. They do not carry over to cycle N + 1, for example. No pipelining between the collector phases is assumed: sweeping is followed by marking.

Shared Variable	Description	Computed By	Value Domain
Global Variables			
Phase	Current Collector phase	Collector	[Idle, Tracing, Sweeping]
Hue	Scanned Part of the Heap	Collector	[0, $\lvert H\rvert$]
Per-Object Variables			
Marked	Mark flag	Collector	Boolean
SRC	Scanned reference count	Mutator	[0, $\lvert P\rvert$]
Shade	Scanning progress within object	Collector	[0, $\lvert N\rvert$]
Recorded	Recorded in buffer by barrier	Mutator	Boolean
DontSweep	Allocated after Hue	Mutator	Boolean

Table 1. Shared variables, $|N|$ is object size, $|P|$ is maximum number of pointers in the heap and $|H|$ is the number of objects in the heap.

For the sake of presentation, we also make a number of simplifying assumptions about the heap. We assume that all heap objects are the same size S and consist only of collector meta-data and object data fields which are all pointers. The fields of an object X are denoted $X[1]$ through $X[S]$.

2.2 Tracing

The abstract algorithm is shown in Fig 1 and 2. We begin by describing the outer collection loop and the tracing phase of collection cycle.

The `Collect()` procedure is invoked to perform a (concurrent) garbage collection. When it starts, the `Phase` of the collector is `Idle`, and the first thing it does is to atomically mark the root object and set the collector phase to `Tracing`. Atomicity is required because mutators can perform operations dependent on the collection phase.

Because all variables live in the heap, there is only a single root that must be marked atomically. In a realistic collector that avoided write barriers on stack writes, this single operation would be replaced by atomic marking of all of the roots – which could be on stacks or on global variables.

The core of the algorithm is the invocation of `Trace()`, which is performed repeatedly until the concurrently executing mutators have not modified the object graph in a way that could result in unmarked live objects.

Tracing in our algorithm is very similar to the tracing in a synchronous collector: it repeatedly gets an object from the mark stack and scans it.

Shades of Grey In the `Scan()` procedure the first major difference appears. Like a standard tracing collector, we iterate over the fields of the object and mark them. However, as each field is read, the *Shade* of the object is incremented.

The use of shades is one of the generalizations of our algorithm. Most concurrent collectors use the well-known *tri-color* abstraction: an object is white if it has not been seen by the collector, grey if it has been seen, but all of its fields have not been seen, and black if both it and its fields have been seen.

```
Collect()
    atomic
        Mark(root);
        Phase = Tracing;

    do
        Trace();
    while (ProcessBarriers());

    atomic
        ProcessBarriers());
        Trace();
        Phase = Sweeping;

    Sweep();
    Phase = Idle;

Trace()
    while(! markStack.empty())
        Obj = markStack.pop();
        Scan(Obj);

Scan(Obj)
    for (field = 1; field <= Obj.Size; field++)
        atomic
            Ptr = Obj[field];
            Obj.Shade = field;
        Mark(Ptr);

Mark(Obj)
    if (! Obj.Marked)
        markStack.push(Obj);
        Obj.Marked = true;

ProcessBarriers()
    retrace = false;
    atomic
        while (true)
            if (barrierBuffer.empty())  return retrace;
            Obj = barrierBuffer.remove();
            Obj.Recorded = false;
            if ((INSTALLATION_COLLECTOR && Obj.SRC == 0) ||
                (DELETION_COLLECTOR && Obj.SRC == 0 && isLeaf(Obj)))
                continue;
            if (! Obj.Marked)
               Mark(Obj);
               retrace = true;
```

Fig. 1. Abstract Collector Code

```
Sweep()
    for (i = 1; i <= Heap.Size; i++)
        Hue = i;
        Obj = Heap[i];
        if (! Obj.Marked && ! Obj.DontSweep)
            FREE(Obj)
        Reset(Obj)

    Hue = 0;

Reset(Obj)
    Obj.Shade = Obj.SRC = 0;
    Obj.Marked = Obj.Recorded = Obj.DontSweep = false;
```

```
atomic WriteBarrier(Obj, field, New, isAllocated)
    if (Phase == Tracing)
        Old = Obj[field];

        if (field < Obj.Shade) // Already scanned by collector
            if (! New.Marked)
                if (DELETION_COLLECTOR)
                    if (isAllocated)
                        Remember(New);
                else
                    if (! New.Recorded)
                        Remember(New);
                New.SRC++;
            if (! Old.Marked)
                Old.SRC--;
        else if (DELETION_COLLECTOR && ! Old.Marked
                 && ! Old.Recorded &&
                 (!isLeaf(Obj) || (isLeaf(Obj) && Old.SRC > 0)))
            Remember(Old);
    Obj[field] = New;

atomic AllocateBarrier(Obj, field, New)
    Reset(New);
    if (Phase == Sweeping)
        if (Heap.free >= Heap.Hue)
            New.DontSweep = true;
    else
        WriteBarrier(Obj, field, New, true);

Remember(Obj)
    BarrierBuffer.append(Obj);
    Obj.Recorded = true;
```

Fig. 2. Abstract Mutator Code

The color of an object represents the progress of the tracing wavefront as it sweeps over the graph. However, the tri-color abstraction loses information because it does not track the progress of sweeping within the object. Fundamentally, the synchronization between the collector and the mutator depends on whether an object being mutated has been seen yet by the collector. Therefore, by losing information about the marking progress, the precision of the algorithm is compromised.

The *Shade* of an object is simply a generalization of the tri-color abstraction: objects are still white, grey, or black, but there are many shades of grey. The shade represents the exact progress of marking within the object. When *Shade* is 0, the object is white. When it is the same as the number of fields in the object, the object is black. We will describe how the shade information is used when we present the write barrier executed by the mutator.

Once the `Scan()` procedure has updated the shade, it marks the target object. The `Mark()` procedure pushes the object onto the mark stack if it was not already marked.

2.3 Mutator Interaction

We now turn to the interaction between the mutator and collector by considering the actions of the mutator when it changes the object graph. The connectivity graph can be modified by both pointer modification and object allocation.

Write Barrier The write barrier is depicted by the procedure `WriteBarrier()` in Fig 2. In our presentation of the algorithm, the entire write barrier is atomic. Finer-grained concurrency is possible, but is not discussed in this paper.

The write barrier takes a pointer to the object being modified, the field in the object that is being modified, the new pointer that is being stored into the object, and a flag indicating whether the new pointer refers to an object that was just allocated.

If the collector is not in its tracing phase, it simply performs the write: because it is the tracing phase that determines reachability of objects, only object graph mutations during tracing can affect reachability (object graph additions – via allocation – require some additional synchronization, which is described below).

An object can be protected either (1) when a pointer to it is stored or (2) when a pointer to it is overwritten. We call saving the pointer at 1 an *installation barrier* and saving the pointer at 2 a *deletion barrier*. The Dijkstra-style barrier is an instance of an installation barrier; the Yuasa-style barrier is an instance of a deletion barrier.

Earlier, we described our collector as a combination of tracing and reference counting. The reference counting is done in the write barrier. In particular, we keep a count of the number of references to an unmarked object from scanned portions of the heap. This is called the Scanned Reference Count or *SRC*. The *SRC* is one of the most important aspects of our abstract algorithm and allows for a number of interesting insights.

The SRC allows us to defer reachability decisions from the time of a write barrier to the time when collector tracing is finished. For example, if a pointer to an object is installed into the scanned portion of the heap, and subsequently removed from the scanned portion of the heap, then it can not possibly affect the liveness of the object.

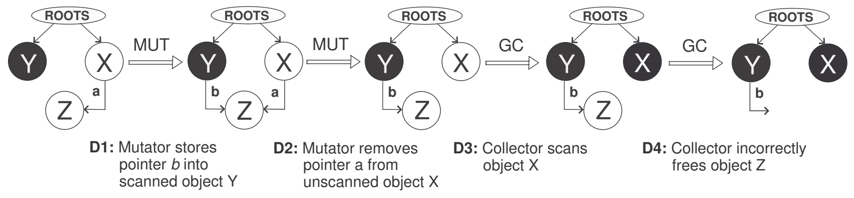
ROOTS
MUT
GC
Y
X
Z
a
b
D1: Mutator stores pointer *b* into scanned object Y
D2: Mutator removes pointer a from unscanned object X
D3: Collector scans object X
D4: Collector incorrectly frees object Z

Fig. 3. Erroneous collection of live object Z via deletion of direct pointer a from object X.

Object Allocation Besides pointer assignments, the mutator can also add objects to the connectivity graph. Similarly to pointer assignments, the allocation interacts with the tracing phase. In addition, allocation also interacts with the sweeping phase of the collector. This is performed in the procedure `AllocateBarrier()` in Fig 2.

In terms of reachability, if the collector is in its tracing phase, object allocation can be seen as just another pointer modification event. The main difference between allocation and pointer writes is that upon allocation we know that the new pointer is unique. We also know that the new object does not contain any outgoing pointers.

During the sweeping phase, the collector iterates over the heap, reclaims all unreachable objects and resets the state of the live objects. We assume that we can designate which parts of the heap the collector has passed indicated by the variable *Heap.Hue*. The variable is similar to *Shade*, except *Shade* is applied per object while *Hue* is applied per heap. That is, we have one *Hue* variable. Similarly to *Shade*, the variable is monotonic within the same collector cycle.

If the mutator allocates during the collector's sweeping phase, we require a mechanism to protect the object from being collected erroneously. The field *DontSweep* indicates if the object has been allocated in a part of the heap that the collector has yet to reach in its sweeping action.

2.4 Lost Object Problem

In a concurrent interleaving between the application and the collector, the program can accidentally hide pointers during collector heap marking. A mutator can store a pointer into a portion of the heap the collector has already scanned, and subsequently destroy all paths from an unscanned reachable portion of the heap to that object. The problem can be broken down into hiding directly and transitively reachable objects. For illustration purposes an object with a black color is one that the collector has marked reachable and has scanned all of its children. A white-colored object is one that the collector has not yet reached.

The sequence for *directly* hidden objects is depicted in Fig. 3. Each state of the graph is shown in time steps. In the initial state, three are objects: scanned object Y, unscanned but reachable object X and object Z which is not yet marked, but is reachable only from X via pointer *a*. In step D1, a mutator copies pointer *a* and stores it into the scanned object Y resulting in pointer *b*. In step D2, the mutator removes the only pointer to Z

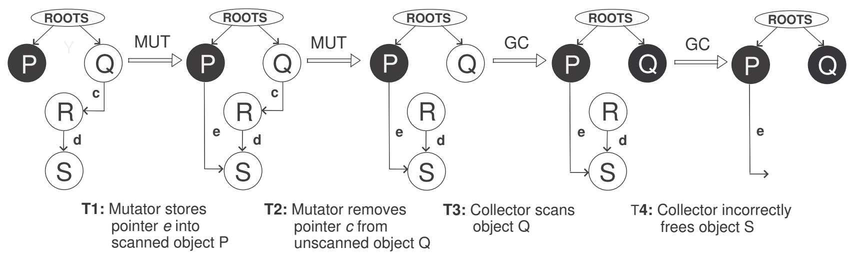
ROOTS
P
Q
R
S
c
d
e
MUT
MUT
GC
GC
T1: Mutator stores pointer e into scanned object P
T2: Mutator removes pointer c from unscanned object Q
T3: Collector scans object Q
T4: Collector incorrectly frees object S

Fig. 4. Erroneous collection of live object S via deletion of pointer c from object Q which transitively reaches S through R.

from an unscanned but reachable object X. The mutator is then immediately preempted by the collector and in step D3, the collector processes object X, turns it black (scanned) and assumes that its marking phase is completed. Next, in step D4, the collector starts its sweeping phase and erroneously frees object Z, although Z is reachable from Y via pointer *b*. In this case we say that object Z is *directly* hidden from the collector.

Alternatively, an object can be hidden *transitively*. This case is illustrated in Fig. 4. In the initial state, object P is scanned and Q, R, and S are reachable but not yet seen. Starting from this state, in step T1, the mutator introduces pointer *e* from a scanned and visited object P to object S. In step T2, the mutator destroys the unscanned pointer *c* from Q to R, essentially, destroying the only path starting from Q to object S. Next, in step T3, the collector preempts the mutator and scans object Q as shown and assumes to have finished the tracing phase. In step T4, the collector incorrectly frees object S. In this case we say that object S was *transitively* hidden from the collector.

The lost object problem consists of two main events in time: storing a pointer to the particular object to be lost and in a subsequent step destroying all other paths to that object. The two well-known solutions to this problem operate at either of these two steps. They either operate at state D1/T1 or at state D2/T2. Dijsktra's and Steele's solutions operate at states D1/T1 and aim to prevent the un-acknowledged introduction of pointers from scanned portions of the heap to reachable but unmarked objects. They essentially speculate that a pointer destruction will occur sometime in the future, and this will lead to hiding of the object. Alternatively, solutions can operate at steps D2/T2. When a pointer is destroyed as in steps D2 and T2, we reason that a pointer to the object must have been introduced earlier and make the target of the overwritten object reachable. This is the solution chosen by Yuasa. For example, Yuasa would make Z live when pointer *a* is removed in step D2 or pointer *c* is removed in step T2. In the transitive case, even though object R might have become unreachable when the pointer is destroyed in step T2, Yuasa's solution requires that object R is kept live as a potential only path left to the hidden object S.

2.5 Design Alternatives

The abstract algorithm maintains rich object and heap-level information. This section attempts to provide an intuitive understanding of the abstract algorithm.

The essence of the abstract algorithm is that it allows for deferring reachability decisions from the mutator to the collector. That is, in the write barrier the mutator detects a potential problem and nominates a candidate pointer for the collector. Subsequently, before the termination of its tracing phase, the collector examines the nominated pointers and optionally discards unnecessary candidates. The specific choices of which pointers are selected by the mutator and which pointers are processed by the collector are discussed in the following sections.

Mutator Selection When a mutator hits the write barrier, it can protect an object using either the *installation* choice or the *deletion* choice. Intuitively, to protect an object, the mutator speculates about reachability, since it has no knowledge of how the graph changes before the collector has finished tracing. In the abstract algorithm, the mutator detects a potential problem, but does not make explicit decisions whether the object is reachable at the end of tracing.

If the *installation* choice is utilized, the object is nominated by the mutator as soon as the *SRC* becomes > 0, thereby, protecting the object *directly* rather than transitively. The installation choice speculates that right after the *SRC* becomes > 0, the only path to the object from an unscanned, but reachable object will be destroyed. Immediately after nominating the pointer, the SRC could be decremented back to zero effectively undoing the previous operation.

For the *deletion* approach, if a pointer in an unscanned object is overwritten, another object can become hidden either transitively or directly. If the *SRC(X)* is > 0 and a pointer to object X is overwritten from an unscanned portion of the heap, we need to protect object X directly. Therefore, the mutator must nominate this pointer. Alternatively, if the *SRC(X)* is 0, we might need to protect some transitively reachable object from X. The key is to recognize that if X does not contain any outgoing pointers, then no object can be hidden transitively. In such cases, we do not need to nominate X.

Determining whether object X is a leaf can be done by using the type of the object. Examples of acyclic types are scalar arrays as well as newly allocated objects before pointers are stored into them. Objects of acyclic types are leaves for their entire lifetime while newly allocated objects can be leaves only temporarily.

Moreover, even if object X is not a leaf, a write barrier could possibly perform nested checks and determine that at, for example, two-levels deep all objects pointed from X are leaves and their SRC is 0. In this case, we can again refrain from nominating the overwritten X pointer.

In some way, it would be logical to make a conclusion that the deletion choice should be more precise, since it always reasons about an event which has already occurred: the *SRC* of some object has become > 0. The *installation* choice speculates about the future, that may be at some point an unscanned pointer will be destroyed. Although a deletion collectors reasons about past event and should have more information, it has no practical way of determining those transitive objects whose *SRC* > 0. In contrast, the *installation* choice always has an immediate access to the critical object.

Besides pointer events, the mutator can modify the connectivity graph via object allocation. Allocation can be seen as an instance of a write barrier with special knowledge that the target pointer is unique. For installation choice collectors, allocation events are treated exactly as all pointer events. For deletion choice collectors, if the resulting pointer from an allocation request is stored into a scanned portion of the heap, it is possible that the object will be lost. We can then think of allocation as a normal pointer store, except that immediately after the pointer store into a scanned region of the heap, an unscanned virtual pointer to the object is overwritten. Since the virtual event cannot be captured by the barrier, we *simulate* it in the barrier. The flag *isAllocated* is passed specifically for this reason from the `AllocateBarrier()` to the `WriteBarrier()` procedure.

Characterizing graphs that allow different barrier choices per object is an interesting though primarily theoretical question. It is generally not possible to make that decision arbitrary without some local knowledge of the graph.

Finally, if all barriers occur on leaf objects, the deletion choice will always require us to nominate fewer pointers. Of course, in both barrier choices precisely the same number of objects will be marked live. This can be logically explained by the fact that in both cases, the immediate object is available during the pointer store therefore we can reason locally about reachability. In that case, for leaf objects, the decision of which barrier to use can be made per-object rather than per-collector-cycle. We do not deal with this topic further.

Collector Choice Once the collector has finished the initial tracing of the heap, there could be a number of unmarked candidates nominated by the mutator. It is possible that in between the time when the mutator has nominated a candidate and the collector sees it, the candidate is no longer necessary.

Similarly to the mutator's pointer selection mechanism, the collector also uses a mechanism to filter out unnecessary candidates. This selection mechanism for the collector is the same as that for the mutator. This can be seen in the write barrier processing phase, the procedure `ProcessBarriers()` in Fig. 1.

Although the collector uses the same mechanism as the mutator, it is possible that candidates nominated by the mutator are ignored by the collector. For example, if the *installation* choice is used and if the object's *SRC* is > 0, when the collector sees such pointers, the corresponding object must be retraced. If the object's *SRC* is 0 however, then the object was recorded by the mutator, but before tracing finished, its *SRC* dropped to 0. Such objects are skipped by the collector in this phase. They have either become garbage or are live but hidden. In the latter case, the object is reachable transitively from a chain of reachable objects starting at an object whose *SRC* is > 0. We therefore only need to re-trace objects whose *SRC* is > 0. Similar reasoning although with a different selection criteria is applied to the deletion choice.

Maintaining an accurate *SRC* has several advantages. First, the SRC prevents us from inducing *floating garbage*. That is because at the time a pointer store occurs, the mutator *nominates* objects that could be *potentially* hidden from the collector. It need not make an explicit decision whether they will actually be reachable once the tracing is complete. The reachability is left to the collector when the barrier tracing phase occurs.

It is because of the *SRC* that the mutator does not need to make such explicit decisions about reachability. Secondly, the collector must start re-scanning only from specific objects. For example, for the installation choice it does not need to consider objects whose *SRC* is 0.

3 Transformations: Trading Precision for Efficiency

The abstract algorithm of the previous section provides a much higher degree of precision than previously published and implemented algorithms, but it is also impractical. In this section we describe how practical collectors can be derived via orthogonal transformations of the abstract collector. Since the transformations are orthogonal, and since the reduction in precision can be modulated, this framework allows the derivation of a much broader set of algorithms than have previously been described, as we will show in the following section.

The transformations presented are (1) reduction in write barrier overhead by treating multiple pointers as roots; (2) reduction in root processing by eliminating re-scanning of the root set; (3) reduction in object space overhead and barrier time overhead by reducing the size of the scanned reference count (SRC); (4) reducing object space overhead by reducing the precision of the per-object shade; (5) conflation of shade and SRC to further reduce object space overhead and speed up the write barrier.

These transformations are not strictly semantics-preserving, since the set of collected objects is changed. However, they are invariant-preserving in that live data is never collected (the collector safety property).

3.1 Root Sets: Eliminating Write Barriers

In the abstract algorithm, all memory is reached from a single root. Thus stacks and global variables are treated as objects like any other. Such an approach is actually used in some implementations of functional programming languages [12]. However, in systems with a significant level of optimization, the cost of such an approach is prohibitive because the mutation rate of the stack is generally extremely high and every stack mutation must include a write barrier.

Therefore, we can transform an abstract algorithm with a uniform treatment of memory into an algorithm which partitions memory into two regions: the roots and the *heap*. The roots generally include the stack and may also contain the static variables and other distinguished pointer data.

In common parlance the static variables are generally considered to be roots, but if they are barriered then they are in effect treated as fields of the “global variable object”, and only the pointer to that “object” is a true root. From the point of view of the root transformation, the only issue is that the memory is partitioned into two sets, the roots and the heap, such that there are no pointers from the heap into the roots.

In the abstract collector, there is a single root pointer. Therefore, examining the root is an inherently atomic operation. With the addition of multiple roots, they must either be processed atomically or a further transformation must be applied to incrementalize root processing [15]. In this work we restrict ourselves to algorithms with atomic root processing.

In particular, at the beginning of every collection, we stop the mutators and mark all heap objects directly reachable from roots, placing them on the work queue (mark stack). Subsequently, when the roots are mutated, no write barrier is executed.

Since the roots are processed atomically at the beginning of collection, they are in effect a *scanned object*. However, since we no longer perform a barrier on mutation, the SRC field of objects referenced by mutated roots is no longer guaranteed to be correct and they will not have been placed in the barrier buffer. Therefore, the algorithm must be adjusted to correct or accomodate this imprecision.

The imprecision can be corrected by atomically *re-scanning* the roots before barrier processing. Consider a sequence of stores into a particular root pointer. These stores must be treated like stores into a scanned portion of the heap, so that the SRC of the installed and overwritten pointers must be incremented and decremented, respectively. If we rescan the roots, then any pointers which were scanned previously will have already been marked, and the SRC will be unaffected.

When a pointer to an unscanned object is stored into a root for the first time, the pointer that is being overwritten must point to a marked object, since all direct referents of roots are marked atomically at the start of collection. Thus the SRC of the overwritten pointer would not have changed if the write had been barriered. However, the SRC of the newly installed pointer would have been incremented, but if the roots are rescanned this pointer will be discovered and since it points to an unmarked object it is known to be a new pointer, and the SRC is incremented. Thus in the case of a single store to a root, the SRC is correct.

Inductively, if there are multiple stores to a root, then each subsequent store will cause the SRC of the overwritten pointer to be decremented and the SRC of the installed pointer to be incremented. The decrement will cancel the increment that was performed on the same pointer when it was previously installed. Therefore, a sequence of stores to a particular root pointer will result in the SRC of all objects except the last one to remain unchanged. [1]

Since that object is found by rescanning, rescanning will compute an accurate SRC, and the transformation that separates the memory into roots and heap leaves the precision of the algorithm unchanged.

3.2 Root Rescan Elimination

As we have just shown, the special treatment of roots does not affect the precision of the collector if root re-scanning is used to correct the SRC. However, re-scanning is undesirable because it increases the running time of the algorithm.

If root re-scanning is eliminated, then the SRC values may be under-approximations (because the increment of the final pointer stored in a root will have been missed). Since increments may keep objects live that would otherwise have been collected, this means

[1] This is the same reasoning that was applied by Barth to eliminate redundant reference count updates at compile time [8], and by Levanoni and Petrank to remove redundant reference count updates between epochs in a concurrent reference counting collector [23].

that any reclamation of an object based on its SRC being 0 is unsafe. Therefore, the algorithm must be conservative in such cases and precision will be sacrificed.

Furthermore, when an installation barrier is used the installation of pointers into the scanned portion of the heap is what causes them to be remembered in the barrier buffer for further tracing during barrier processing. This means that regardless of the imprecision of the SRC, objects that would have had a non-zero SRC must be seen during barrier processing. In effect, this means re-scanning can not be eliminated for algorithms that use the installation barrier.

For algorithms that use the deletion barrier, the only pointers to new objects that are remembered in the write barrier are the newly allocated objects. Therefore, as long as those objects are placed in the barrier buffer by the allocator, and the SRC-based computation in the barrier processing is eliminated, then the root re-scanning can be safely eliminated.

Since no collector decisions are based on the value of the SRC, it is redundant and can be eliminated. The result is an algorithm with more floating garbage (in particular, all newly allocated objects are considered live), of which Yuasa's algorithm is an example.

3.3 Shade Compression

The shade of an object represents the progress of the collector as it processes the individual pointers in the object. The precision of the shade can always safely be reduced as long as the processing of the pointers in the object in the write barrier treats the imprecise shade conservatively.

In particular, since many objects have a small number of pointers N, it is efficient to treat the shade which originally had the range $[0, N]$ as the set $\{0, [1 \ldots N-1], N\}$. These three values represent an object for which collector processing has not yet begun, is in progress, or has been completed. This is the standard tri-color abstraction introduced by Dijkstra, where the three values are called white, grey, and black, respectively.

When N is small, the chance is low that the mutator will store a pointer into the object currently being processed by the collector, so the reduction in precision is likely to be low. However, with large objects (such as pointer arrays) the reduction in precision can be more noticable. Some collectors therefore treat sections of the array independently, in effect mapping equal-sized subsections of the array into different shades.

3.4 Scanned Reference Count Compression

The scanned reference count (SRC) can range from 0 to the number of pointers in the system. However, the number of references to an object is usually small, and the SRC will be even lower (since it only counts references from the scanned portion of the heap to unmarked objects). Therefore, the SRC can be compressed and the loss of precision is likely to be low.

However, the compression must be conservative to ensure that live objects are not collected. This is accomplished by making the SRC into a "sticky" count [27]: once it reaches its maximum value, it is never decremented. As a result, the SRC is an over-approximation, which is always safe since it will only cause additional objects to be treated as live.

An important special case for collectors that use an installation barrier is a one-bit SRC, since in this case the SRC becomes equivalent to the Recorded flag, allowing those two fields to be collapsed.

3.5 Conflation of Shade and Scanned Reference Count

In a collector using an installation barrier with a one-bit sticky SRC and tri-color shade, an object with a stuck SRC must be scanned by the collector. Similarly, a grey object must be scanned by the collector. Thus the meaning of these two states can be collapsed and the grey color can be used to indicate a non-zero (stuck) SRC, which also represents the Recorded flag.

This is in fact the representation used by most collectors that have been implemented. In effect, they have collapsed numerous independent invariants into a small number of states. This helps to understand why such algorithms are bug-prone: collapsing the states corresponding to algorithmic invariants relies on subtle transformations and simultaneously reduces redundancy in the representation.

4 Using Transformations to Derive Practical Collectors

In this section, we derive various practical algorithms by applying the previously discussed transformations to the abstract collector algorithm. Some of the schemes are well-known concurrent algorithms such as Dijkstra and Yuasa, while others are new derivations.

4.1 Derivation of a Dijkstra Algorithm

The Dijkstra algorithm is an instance of an abstract installation collector and to derive it we apply the following transformations:

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *SRC compression* to a single sticky bit
4. *Conflation of Shade* and *SRC*

Although at the end of the transformation steps, we arrive at a practical Dijkstra algorithm, the intermediate steps also represent valid algorithms with different precision.

The compressions of *SRC* and *Shade* can lead to floating garbage. However, unlike *Shade* and *SRC* compressions, the *Conflation* transformation does not lead to increased floating garbage. On the other side, it reduces, both, space consumption in the header of the object, and, complexity of the write barrier.

The *Root Sets* transformation also preserves the treatment of allocated objects. When a new object is allocated and stored into the roots, the mutator will not nominate the pointer because the store will occur into a scanned partition (roots) and the

```
MarkRoots()
    while (! roots.end())
      Obj = roots.get();
      Mark(Obj);

MarkRootsDirect()
    while (! roots.end())
      Obj = roots.get();
      if (Obj.isAllocatedInThisCycle)
          Obj.Color = black;

Collect()
    atomic
        MarkRoots();
        Phase = Tracing;

    do
        Trace();
    while (ProcessBarriers());

    atomic
        MarkRootsDirect();
        ProcessBarriers());
        Trace();
        Phase = Sweeping;

    Sweep();

    Phase = Idle;

atomic WriteBarrier(Obj, field, New)
    if (Phase == Tracing)
        Old = Obj[field];

        if (New.Color == white && New.isAllocatedInThisCycle)
            New.Color = black;

        if (   Obj.Color != black && Old.Color == white
            && !Old.Recorded )
            Remember(Old);
    Obj[field] = New;
```

Fig. 5. Pseudo Code For The Hybrid Collector

write barrier is not active on the roots. If the pointer to the allocated object gets stored in the heap, then it will be processed in the mutator write barrier, similarly to all other objects existing at collection startup. If the allocated object dies before the roots are rescanned, the collector will not mark that object as live.

A Steele-like collector is similar to a Dijkstra collector except that its transformation covers a wider range of rescanning. A Steele algorithm is not limited to rescanning only the roots, but can also rescan heap partitions. However, the barrier processing phase and the selection criteria are exactly the same as in the Dijkstra collector.

4.2 Derivation of a Yuasa Algorithm

Our second derived collector is a Yuasa snapshot algorithm. The Yuasa algorithm is an instance of a deletion collector. The algorithm can be derived by applying the following transformations to the abstract collector:

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *Root Rescan Elimination*

During barrier processing, deletion collectors can skip objects whose *SRC* is 0 and are leafs. However, since *Root Rescan elimination* prevents the roots rescanning process, an accurate *SRC* cannot be computed, and subsequently the collector selection criteria cannot be applied. Therefore, in order to preserve the safety property of the abstract collector, the collector must mark all overwritten pointers and rescan from them. The *SRC* is removed since it cannot serve its primary purpose: a guide for the collector selection criteria.

The Yuasa collector is the most conservative approach to floating garbage. It does not allow any destruction in the connectivity graph once the collector has started and it effectively allocates only reachable object (black).

One fundamental difference between Yuasa and Dijkstra algorithms is that in the presence of a *Root Sets* transformation, installation collectors must never use the *Root Rescan elimination*, while deletion collectors have no requirement to apply it. The rescanning in deletion collectors is done mostly to eliminate floating garbage, albeit, at the expense of triggering work to rescan the roots.

Also, although at the end of our derivation, we arrived at a Yuasa algorithm, the result of every intermediate step is a valid deletion collector.

4.3 Derivation Of a Hybrid Algorithm

The third derived practical algorithm is the Hybrid collector. The Hybrid algorithm is an instance of an abstract deletion collector. The Hybrid algorithm can be derived by applying the following transformations:

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *SRC compression* to a single sticky bit

4. *Conflation of Shade and SRC*
5. *Root Rescan elimination for existing objects*
6. *Over approximate Shade*

The first two transformation steps are the same as for the Yuasa algorithm. However, in the Hybrid algorithm, we utilize the rescanning of roots only for newly allocated objects. The roots rescan transformation is parameterized to be active only for existing objects. The idea is to obtain a deletion Yuasa algorithm for the existing heap graph while maintaining a less restricted policy for newly allocated objects, similarly to the Dijkstra collector. By eliminating rescanning for existing objects, we can remove the *SRC* for those objects. Whenever the collector encounters an existing object during its barrier processing phase it will always mark the object, without applying any selection criteria.

After step 5 we still have a working deletion algorithm, but we would like to obtain more of the properties of Yuasa, namely, bounded re-tracing of newly allocated objects triggered by roots rescanning. To do that, we perform an additional transformation where if a newly allocated pointer is stored into the heap, the object is marked as reachable for this collection cycle. This simply means that if a newly allocated pointer is stored into the heap, we always increase the *SRC*, ignoring what the color of the destination object is. This is clearly a trivial over-approximation transformation on Shade of the destination object as indicated in step 6. With this, the collector now only needs to trace from existing objects and not from newly allocated objects. Newly allocated objects are essentially allocated white and colored black either during roots rescanning or during a pointer store in the heap.

The *Hybrid* collector is particularly suited for hard real-time applications, where it is desirable to achieve a bound on the roots rescanning work while reducing the floating garbage.

The skeleton code for the algorithm is illustrated in Fig. 5. $MarkRootsDirect$ is the procedure that performs the one-level deep rescanning procedure for the roots partition while the $isAllocatedInThisCycle$ bit is used to differentiate between newly allocated and existing objects.

5 Experimental Evaluation

We have implemented a concurrent collector framework in IBM's J9 virtual machine. The collector supports both standard work-based collection (for every a units of allocation the collector performs ka units of collection work) as well as time-based collection (the collector runs for c out of q time units). This collector has been built as a second-generation Metronome real-time collector [4].

However, in this paper we will concentrate on work-based collection because its use is more common in more widely used soft real-time systems, and is likely to provide a better basis for comparison with other work. Isolated experiments have shown that the trends we report for work-based collection generally hold for time-based collection as well.

Our collector is implemented in a J2ME-based system that places a premium on space in the virtual machine. Therefore, we use the microJIT rather than the much

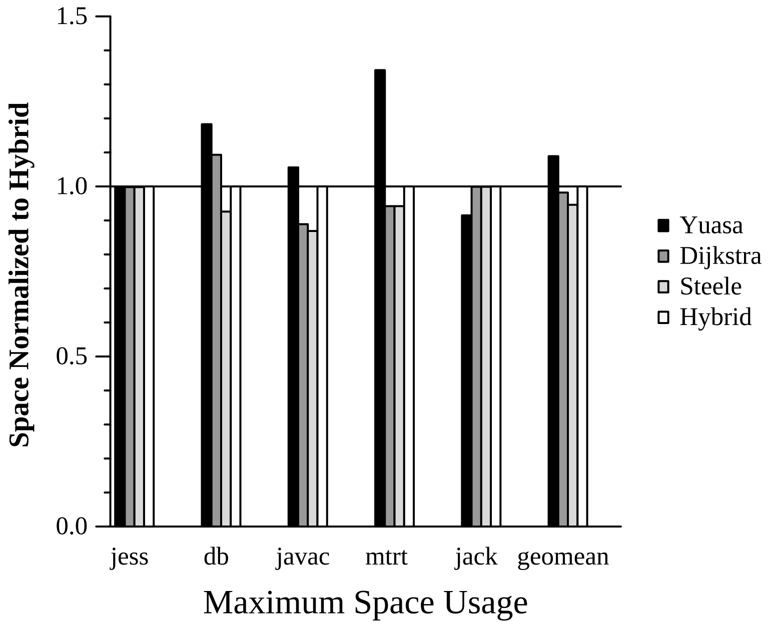

Fig. 6. Summary of the maximal space usage of the four collector algorithms. Data is normalized to the Hybrid algorithm. Shorter bars represent lower space usage.

more resource-intensive optimizing compiler. The microJIT is a high-quality single-pass compiler, producing code roughly a factor of 2 slower than the optimizing JIT.

The system runs on Linux/x86, Windows/x86, and Linux/ARM. The measurements presented here were performed on a Windows/x86 machine with a Pentium 4 3GHz CPU and 500MB of RAM.

The measurements presented all use a collector to mutator work ratio of 1.5, that is, for every 6K allocated by the mutator, the collector processes 9K. Collection is triggered when heap usage reaches 10MB.

We have measured the SPECjvm98 benchmarks, which exhibit a fairly wide range of allocation behavior (with the exception of compress, which performs very little allocation).

Figure 7 summarizes the performance of the four collector algorithms. The left graph shows the maximum heap size, the right graph total execution time. Both graphs are normalized to the Hybrid algorithm, and shorter bars represent better performance (less heap usage or shorter execution times). A geometric mean is also shown. These graphs summarize more detailed performance data which can be found in the Appendix.

5.1 Space Consumption

As expected, the incremental update collectors (Dijkstra and Steele) often require less memory than the snapshot collector (Yuasa). This is because the incremental update collectors allocate white (unmarked) and only consider live those objects which are

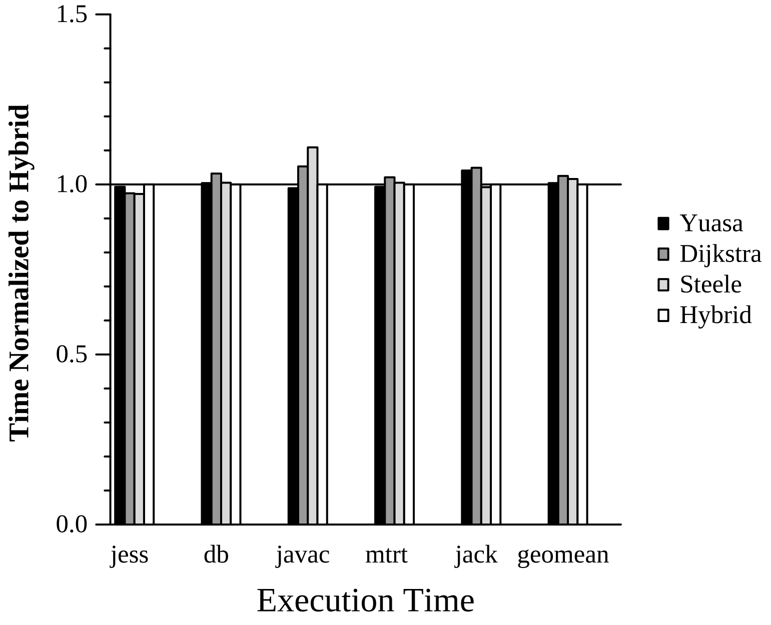

Fig. 7. Summary of overall execution time of the four collector algorithms. Data is normalized to the Hybrid algorithm. Shorter bars represent faster execution time.

added to the graph. However, there is no appreciable difference on 2 of the five benchmarks (jess and jack), which confirms that the space savings from incremental update collectors are quite program-dependent.

The use of Steele's write barrier instead of Dijkstra's theoretically produces less floating garbage at the expense of more re-scanning, since it marks the source rather than the target object of a pointer update. This means that if there are multiple updates to the same object, only the most recently installed pointer will be re-scanned.

However, the Steele barrier only leads to significant improvement in one of the benchmarks (db). This is because db spends much of its time performing sort operations. These operations permute the pointers in an array, and each update triggers a write barrier. With a Steele barrier, the array is tagged for re-scanning. But with a Dijkstra barrier, each object pointed to by the array is tagged for re-scanning. As a result, there is a great deal more floating garbage because the contents of the array are being changed over time.

Finally, the hybrid collector which we introduced, a snapshot collector that allocates white (unmarked), significantly reduces the space overhead of snapshot collection: the space overhead over the best collector is at worst 13% (for javac), which is quite reasonable.

5.2 Execution Time

While the incremental update collectors are generally assumed to have an advantage in space, their potential time cost is not well understood. Incremental update collectors may have to repeatedly re-scan portions of the heap that changed during tracing. Termination could be difficult if the heap is being mutated very quickly.

Our measurements show that incremental update collectors do indeed suffer time penalties for their tighter space bounds. The Dijkstra barrier causes significant slowdown in db, javac, mtrt, and jack. The Steele barrier is less prone to slowdown – only suffering on javac – but it does suffer the worst slowdown, about 12%. These measurements are total application run-time, so the slow-down of the collector is very large – this represents about a factor of 2 slowdown in collection time.

Once again, our hybrid collector performs very well – it usually takes time very close to the fastest algorithm. Thus the hybrid collector appears to be a very good compromise between snapshot and incremental update collectors.

Because its only rescanning is of the stack, it suffers no reduction in incrementality from a standard Yuasa-style collector, which must already scan the stack atomically. Its advantage over a standard snapshot collector is that it significantly reduces floating garbage by giving newly allocated objects time to die. But because it never rescans the heap, it avoids the termination problems of incremental update collectors and is still suitable for real-time applications.

As shown by the more detailed graphs in the appendix, the primary reason why the Yuasa and Hybrid algorithms are quicker is that the Dijkstra and Steele collectors both scan significantly more data during barrier buffer processing.

The benchmark with the most unusual behavior is jack, for which the Yuasa snapshot collector uses the *least* memory, while the Steele algorithm uses the least time. We are still in the process of investigating this behavior.

6 Conclusions

We have presented an abstract concurrent garbage collection algorithm and showed how incremental update collectors in the style of Dijkstra, and snapshot collectors in the style of Yuasa, can be derived from this abstract algorithm by reducing precision through various transformations.

We have also used the insights from this formulation to derive a new type of Hybrid snapshot collector which allocates its objects unmarked, and therefore induces less floating garbage.

We have implemented all four collectors in a production virtual machine and compared their time and space requirements. Incremental update collectors do indeed suffer less floating garbage, while the pure snapshot collector sometimes uses significantly more memory. The Hybrid collector greatly reduces the space cost of snapshot collection.

Incremental update collectors can significantly slow down garbage collection, leading to noticeable slow-downs in application execution speed. Our new Hybrid snapshot collector is generally about as fast as the fastest algorithm. For most applications, this

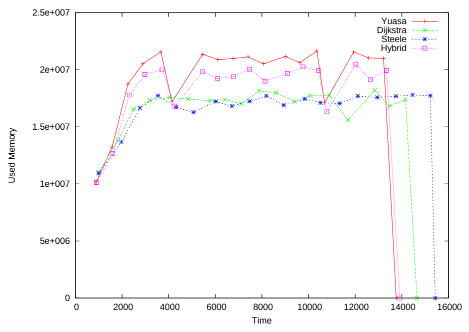

Fig. 8. Space vs. Time: javac

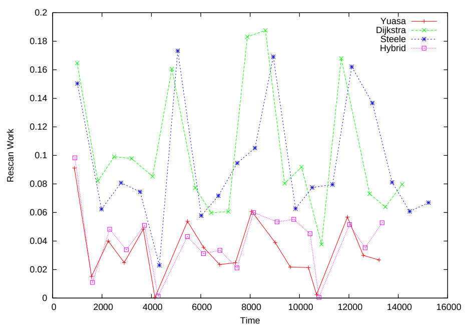

Fig. 9. Collector Rescanning Work: javac

collector will represent a good compromise between time and space efficiency, and has the notable advantage of snapshot collectors in terms of predictable termination.

We hope this work will spur further systematic study of algorithms for concurrent collection and further quantitative evaluation of those algorithms.

Appendix: Detailed Performance Data

This section includes graphs that illustrate for each benchmark, the behavior of the four collectors with respect to space utilization and barrier-induced work.

Figure 8 shows space usage over time by javac. Each data point represents the amount of data in use when the tracing and barrier processing terminated, but before sweeping. This represents the point of maximum memory use. The Yuasa-style collector consistently uses more memory than the others, but it also terminates the quickest (at termination, memory consumption is 0).

The reason why the Yuasa and Hybrid algorithms are quicker can easily be seen in Figure 9: the Dijkstra and Steele collectors both scan significantly more data during barrier buffer processing. Note that barrier-induced scanning is still significant even for the pure snapshot (Yuasa) collector. This is because pointers to some objects that are part of the snapshot may have been overwritten and not discovered during marking. Therefore, the snapshot it “completed” during barrier buffer processing. However, the total work will be based on the live data in the object graph at the time collection began, whereas in the incremental update algorithms it varies.

The rescanning overhead that we observed above for the db benchmark with Dijkstra’s barrier can be seen clearly in Figure 11: rescanning typically causes about 20% of the heap to be re-visited, while rescanning for the other three collectors is negligible.

Details for the remaining benchmarks are found in Figures 12 through 17.

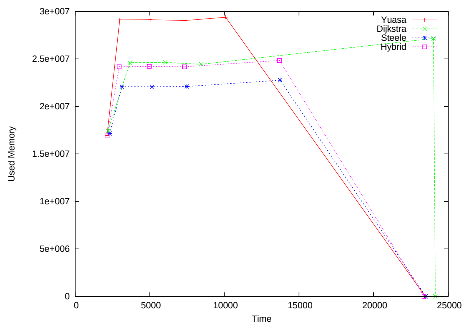

Fig. 10. Space vs. Time: db

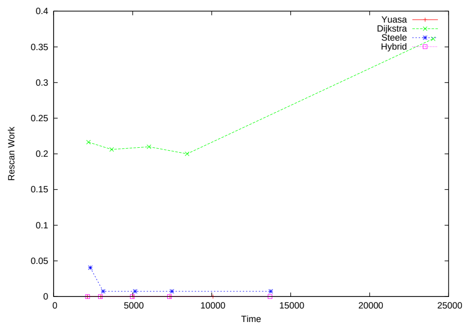

Fig. 11. Collector Rescanning Work: db

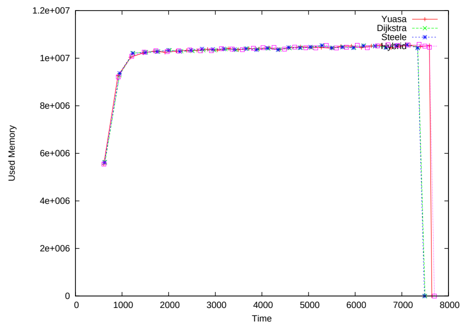

Fig. 12. Space vs. Time: jess

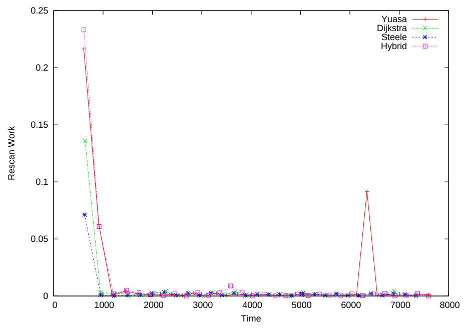

Fig. 13. Collector Rescanning Work: jess

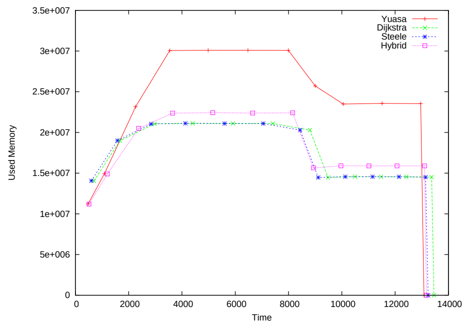

Fig. 14. Space vs. Time: mtrt

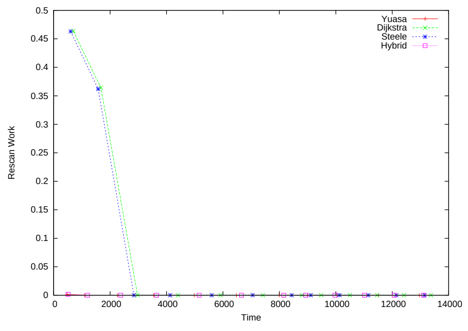

Fig. 15. Collector Rescanning Work: mtrt

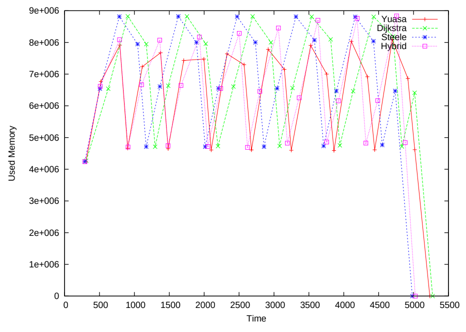

Fig. 16. Space vs. Time: jack

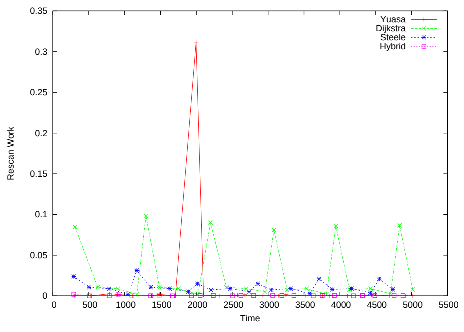

Fig. 17. Collector Rescanning Work: jack

References

[1] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices*, *23*, 7 (July), 11–20.

[2] AZATCHI, H., LEVANONI, Y., PAZ, H., AND PETRANK, E. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (Oct 2003), ACM Press, pp. 269–281.

[3] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 92–103.

[4] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[5] BACON, D. F., CHENG, P., AND RAJAN, V. T. A unified theory of garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 2004), pp. 50–68.

[6] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[7] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.

[8] BARTH, J. M. Shifting garbage collection overhead to compile time. *Commun. ACM 20*, 7 (July 1977), 513–518.

[9] BEN-ARI, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (1984), 333–344.

[10] BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), ACM Press, pp. 157–164.

[11] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[12] CHEADLE, A. M., FIELD, A. J., MARLOW, S., PEYTON JONES, S. L., AND WHILE, R. L. Non-stop Haskell. In *Proc. of the Fifth International Conference on Functional Programming* (Montreal, Quebec, Sept. 2000). *SIGPLAN Notices*, *35*, 9, 257–267.

[13] CHENG, P., AND BLELLOCH, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Jun 2001), ACM Press, pp. 125–136.

[14] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (1978), 966–975.

[15] DOMANI, T., KOLODNER, E. K., LEWIS, E., SALANT, E. E., BARABASH, K., LAHAN, I., LEVANONI, Y., PETRANK, E., AND YANORER, I. Implementing an on-the-fly garbage collector for java. In *Proceedings of the second international symposium on Memory management* (Oct 2000), ACM Press, pp. 155–166.

[16] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[17] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[18] HUDSON, R. L., AND MOSS, E. B. Incremental garbage collection for mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*.

[19] *Proc. of the ACM SIGPLAN International Symposium on Memory Management* (Vancouver, B.C., Oct. 1998).

[20] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[21] LAMPORT, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

[22] LAROSE, M., AND FEELEY, M. A compacting incremental collector and its performance in a production quality compiler. In ISMM [19], 1–9.

[23] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (Oct 2001), ACM Press, pp. 367–380.

[24] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[25] NORTH, S. C., AND REPPY, J. H. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture* (Portland, Oregon, Sept. 1987), G. Kahn, Ed., vol. 274 of *Lecture Notes in Computer Science*, pp. 113–133.

[26] PIXLEY, C. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing 3, 1 6*, 3 (Dec. 1988), 41–49.

[27] ROTH, D. J., AND WISE, D. S. One-bit counts between unique and sticky. In ISMM [19], pp. 49–56.

[28] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[29] STEELE, G. L. Corrigendum: Multiprocessing compactifying garbage collection. *Commun. ACM 19*, 6 (June 1976), 354.

[30] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.