A Pure Reference Counting Garbage Collector

DAVID F. BACON, CLEMENT R. ATTANASIO, V.T. RAJAN, STEPHEN E. SMITH
IBM T.J. Watson Research Center
and
HAN B. LEE
University of Colorado

While garbage collection via reference counting has many appealing properties, it has always suffered from some major limitations. We present new algorithms for reference counting that address these limitations by (1) improving on techniques for avoiding reference counting of stack variables, and (2) reclaiming cyclic garbage.

These algorithms are designed to operate with concurrent mutators in multiprocessor run-time systems. We present both proofs of correctness and an experimental evaluation in the context of the Jalapeño Java virtual machine.

We present measurements comparing our collector against a non-concurrent but parallel load-balancing mark-and-sweep collector (that we also implemented in Jalapeño), and evaluate the classical tradeoff between response time and throughput.

When processor or memory resources are limited, the Recycler usually runs at about 90% of the speed of the mark-and-sweep collector. However, with an extra processor to run collection and with a moderate amount of memory headroom, the Recycler is able to operate without ever blocking the mutators and achieves a maximum measured mutator delay of only 2.6 milliseconds for our benchmarks. End-to-end execution time is usually within 5%.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Languages, Measurement, Performance

Additional Key Words and Phrases: Cycle detection, garbage collection, reference counting

1. INTRODUCTION

Forty years ago, two methods of automatic storage reclamation were introduced: reference counting [Collins 1960] and tracing [McCarthy 1960]. Since that time tracing collectors and their variants (mark-and-sweep, semispace copying, mark-and-compact) have been much more widely used due to perceived deficiencies in reference counting.

Changes in the relative costs of memory and processing power, and the adoption of garbage collected languages in mainstream programming (particularly Java) have changed the landscape. We believe it is time to take a fresh look at reference counting, particularly

Author's address: D. Bacon, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2001 ACM 0164-0925/99/0100-0111 $00.75

Submitted for publication

as processor clock speeds increase while RAM becomes plentiful but not significantly faster. In this environment the locality properties of reference counting are appealing, while the purported extra processing power required is likely to be less relevant.

At the same time, Java's incorporation of garbage collection has thrust the problem into the mainstream, and large, mission critical systems are being built in Java, stressing the flexibility and scalability of the underlying garbage collection implementations. As a result, the supposed advantages of tracing collectors — simplicity and low overhead — are being eroded as they are being made ever more complex in an attempt to address the real-world requirements of large and varied programs.

Furthermore, the fundamental assumption behind tracing collectors, namely that it is acceptable to periodically trace all of the live objects in the heap, will not necessarily scale to the very large main memories that are becoming increasingly common.

There are two primary problems with reference counting, namely:

(1) run-time overhead of incrementing and decrementing the reference count each time a pointer is copied, particularly on the stack; and
(2) inability to detect cycles and consequent necessity of including a second garbage collection technique to deal with cyclic garbage.

In this paper we present new algorithms that address these problems and describe a new multiprocessor garbage collector based on these techniques that achieves maximum measured pause times of 2.6 milliseconds over a set of eleven benchmark programs that perform significant amounts of memory allocation.

Our collector, the *Recycler*, is novel in a number of respects:

—In normal operation, the mutators are only very loosely synchronized with the collector, allowing very low pause times;
—It is a pure concurrent reference counting collector; no *global* tracing is performed to collect cyclic garbage; and
—Cyclic garbage is collected using a new concurrent cycle detection algorithm that traces cycles *locally*.

In addition, we have implemented a non-concurrent ("stop-the-world") parallel load-balancing mark-and-sweep collector, and in this paper we provide comparative measurements of two very different approaches to multiprocessor garbage collection, for the first time quantitatively illustrating the possible tradeoffs.

The Recycler uses a new concurrent reference counting algorithm which is similar to that of Deutsch and Bobrow [1976] but has simpler invariants that make it easier to describe and to implement.

The concurrent cycle collector is the first fully concurrent algorithm for the detection and collection of cyclic garbage in a reference counted system. It is based on a new synchronous algorithm derived from the cyclic reference counting algorithm of Lins [1992a]. Our algorithm reduces asymptotic complexity from $O(n^2)$ to $O(n)$, and also significantly reduces the constant factors. The concurrent algorithm is a variant of the synchronous algorithm with additional tests to maintain safety properties that could be undermined by concurrent mutation of the data structures.

When the system runs too low on memory, or when mutators exhaust their trace buffer space, the Recycler forces the mutators to wait until it has freed memory to satisfy their allocation requests or processed some trace buffers.

The Recycler is implemented in Jalapeño [Alpern et al. 1999], a new Java virtual machine and compiler being developed at the IBM T.J. Watson Research Center. The entire system, including the collector itself, is written in Java (extended with unsafe primitives for manipulating raw memory).

We provide detailed pseudo-code for the algorithms and proofs of correctness based on an abstract graph induced by the stream of increment and decrement operations.

The rest of this paper is organized as follows: Section 2 describes our synchronous reference counting algorithm; Section 3 extends the algorithm to handl concurrent mutation. Section 4 contains a proof of correctness for our reference counting algorithm. Section 5 presents the synchronous algorithm for collecting cyclic garbage; Section 6 extends this algorithm to handle concurrent mutators. Section 7 contains proofs of correctness for the concurrent cycle collection algorithms. Section 8 describes the implementation, and Section 9 describes the parallel mark-and-sweep collector against which our collector is compared. Section 10 presents measurements of the running system and a comparison between the two garbage collectors. Section 11 describes related work and is followed by our conclusions.

This paper combines previously published material on the Recycler [Bacon et al. 2001; Bacon and Rajan 2001] and has been extended to contain complete algorithmic details and proofs that were previously omitted due to space constraints. Subsections 2.2, 3.1, 5.3, and 6.4 contain detailed pseudocode of the algorithms and can be skipped on a first reading of the paper.

2. THE SYNCHRONOUS REFERENCE COUNTING COLLECTOR

We will begin by describing a reference counting collector that does not handle cyclic structures. It can support multiple mutator threads running on multiple processors, but the collection itself is synchronous (also known as "stop-the-world"). This collector will then be extended, first to handle concurrent collection, and then to collect cyclic garbage.

The principles of the synchronous collector are simple. Every object has a reference count field, which for object `S` is denoted `RC(S)`. During normal operation, the collector only keeps track of writes to the heap, while ignoring writes to the stack. `RC(S)` may be out of date, but it is guaranteed that it will not reach zero until the object is garbage.

This guarantee is maintained as follows: increments and decrements to the heap are deferred by writing them into buffers, called `Inc` and `Dec`, respectively. When the program runs out of memory, a collection is triggered, which performs the following steps:

(1) scan the stack of each thread and increment the reference count for each pointer variable in the stack;
(2) perform the increments deferred in the `Inc` buffer;
(3) perform the decrements deferred in the `Dec` buffer, recursively freeing any objects whose `RC` drops to 0;
(4) clear the `Inc` and `Dec` buffers;
(5) scan the stack of each thread again and place each pointer variable into the `Dec` buffer.

The effect of these steps is that any pointers which are on the stack are accounted for during the current collection as increments, and during the next collection as decrements. By accounting for them during the current collection as increments, we ensure that objects pointed to by the stack are not collected. By accounting for them during the following

```
Increment(S)
  RC(S) = RC(S) + 1

Decrement(S)
  RC(S) = RC(S) - 1
  if (RC(S) == 0)
    Release(S)

Release(S)
  for T in children(S)
    Decrement(T)
  SystemFree(S)

Update(R, S)
  T = Fetch&Store(*R, S)
  if (S != null)
    EnqueueIncrement(S)
  if (T != null)
    EnqueueDecrement(T)

Allocate(n)
  P = SystemAllocate(n)
  if (P == null)
    Collect()
    P = SystemAllocate(n)
    if (P == null)
      throw OutOfMemoryError
  RC(P) = 1
  EnqueueDecrement(P)
  return P

EnqueueIncrement(S)
  if (full(Inc))
    Collect()
  append S to Inc

EnqueueDecrement(S)
  if (full(Dec))
    Collect()
  append S to Dec

Collect()
  for T in Threads
    Suspend(T)
  IncrementStack()
  ProcessIncrements()
  ProcessDecrements()
  DecrementStack()
  CollectCycles()
  for T in Threads
    Resume(T)

IncrementStack()
  for T in Threads
    for N in Pointers(Stack(T))
      Increment(N)

ProcessIncrements()
  for M in Inc
    Increment(M)
  clear Inc

ProcessDecrements()
  for M n Dec
    Decrement(M)
  clear Dec

DecrementStack()
  for T in Threads
    for N in Pointers(Stack(T))
      EnqueueDecrement(N)
```

Fig. 1. Algorithm 1 — a synchronous ("stop-the-world") multiprocessor reference-counting collector that does not collect cyclic garbage. The `Collect()` algorithm can easily be parallelized using a worklist approach and a barrier synchronization between increment and decrement processing.

collection as decrements we ensure that if field `RC(P)` was incremented because `P` was on the stack, and `P` is subsequently popped off the stack, that the reference count field will be appropriately decremented.

An object that remains on the stack for a long time will have its reference count repeatedly incremented and decremented at each collection. This is an obvious source for potential optimizations and will be discussed later.

Objects are allocated with a reference count of 1, and a decrement is immediately enqueued. Thus temporary objects that are never stored into the heap and whose stack pointers are popped quickly are freed at the next collection.

2.1 Environmental Assumptions

There are a number of basic assumptions that underly all of the collector variants described in this paper. First of all, we assume that there is an underlying storage manager which provides the routines `SystemAllocate()` and `SystemFree()`. This storage manager has considerable flexibility in its implementation, but a fundamental restriction is that when running in conjunction with the concurrent collection algorithms, it may not relocate objects.

While some researchers have argued that non-relocating collectors are not suitable for large-scale application because of their susceptibility to fragmentation, in our experience relocating collectors are equally susceptible to resource failure because they must allocate additional memory for semi-spaces. These problems become even more pronounced in a concurrent setting where more memory headroom is required and the collector may need to replicate mutable fields of objects. However, for a contrasting approach see, for example, the work of Cheng and Blelloch [2001].

All algorithms are presented for a language run-time system that supports multiple mutator threads and runs on multiple processors. The algorithms are designed to work properly on weakly ordered multiprocessors; synchronization is explicit. For total ordering of operations on a single memory word we assume the existence of operations `Test&Set()` and `Fetch&Store()`.

We also assume that thread switching is only performed at safe points, rather than at any arbitrary point in the mutator execution. Thus we do not handle potential race conditions such as might occur if the `Update()` operation were interrupted to perform garbage collection. This should not affect the applicability of the algorithms to other implementation environments, but some thought may be required to adapt them to more interrupt-prone run-time systems.

2.2 Pseudo-code and Explanation

Pseudo-code for the single-threaded reference counting algorithm is shown in Figure 1.

`Update(R,S)` The location pointed to by `R` is replaced by the pointer `S`. This is called by the language run-time system when a heap update is performed. If non-null, the new value is enqueued as an increment and the old (over-written) value is enqueued as a decrement.

The pointer replacement must be performed atomically on a multiprocessor due to race conditions with other `Update()` operations to the same location: if the pointers are not exchanged atomically, there could be lost updates leading to an incorrect reference count.

`Increment(S)` The reference count of `S` is increased by 1.

`Decrement(S)` The reference count of `S` is decreased by 1. If it reaches 0, `Release` is invoked to free object `S`.

`Release(S)` The reference counts of any objects pointed to by `S` are decremented, and then `S` is freed using the underlying storage manager.

`EnqueueIncrement(S)` The pointer `S` is appended to the `Inc` buffer for later processing. If the buffer is full, a collection is triggered first.

`EnqueueDecrement(S)` The pointer `S` is appended to the `Dec` buffer for later processing. If the buffer is full, a collection is triggered first.

`Allocate(n)` An object of n bytes is allocated and returned, using the underlying storage manager. If allocation fails, a collection is performed and the allocation is retried. If allocation fails again, an "out of memory" exception is thrown.

If allocation succeeds, the object's reference count is set to 1, a decrement is enqueued, and the object is returned.

`Collect()` Collect garbage and free buffers. This is performed in four complementary phases: `IncrementStack`, `ProcessIncrements`, `ProcessDecrements`, and `DecrementStack`.

`IncrementStack()` For each pointer `N` in the stack of each thread `T`, increments its reference count.

`ProcessIncrements()` For each entry `M` in the `Inc` buffer, invokes `Increment(M)`, which increments the reference count of `M`.

`ProcessDecrements()` For each entry `M` in the `Dec` buffer, invokes `Decrement(M)`, which decrements the reference count and if it reaches zero frees the object.

`DecrementStack()` For each pointer `N` in the stack of each thread `T`, enqueues it in the `Dec` buffer for processing at the next collection.

Note: it is assumed that the decrement buffer is always big enough to hold the pointers on the stack.

An implementation tradeoff that we have made is to enqueue the increments; an alternative is to perform them immediately using a fetch-and-add operation. The advantage of the algorithm as presented is that no atomic operations are required for reference counts. The best choice will depend on the target machine characteristics.

When implemented on a uniprocessor with thread switching occurring only at safe points, it is not necessary to implement the `Update()` operation with `Fetch&Store()`, and increments can also be processed immediately without requiring an atomic operation.

However, on a multiprocessor, an atomic exchange operation must be used. Consider if `Update()` had been implemented without it, using the statement

```
T = *R; *R = S;
```

Then it would be possible for the possible interleaving to occur if two processors were simultaneously updating the same location `R`:

Processor 1	*Processor 2*
`T = *R`	
	`T = *R`
`*R = S1`	
	`*R = S2`

As a result, processor 1 would emit a decrement of `R` and an increment of `S1`, and processor 2 would emit a decrement of `R` and an increment of `S2`. But this means that `R` is being decremented twice, while `S1` is not being decremented at all even though it is overwritten by `S2`. Therefore, an atomic exchange operation must be used to ensure that there is no interleaving within the pointer update.

3. THE CONCURRENT REFERENCE COUNTING COLLECTOR

In this section we describe the reference-counting garbage collection algorithm, for the time being ignoring the disposition of cyclic garbage which will not be detected. Our col-

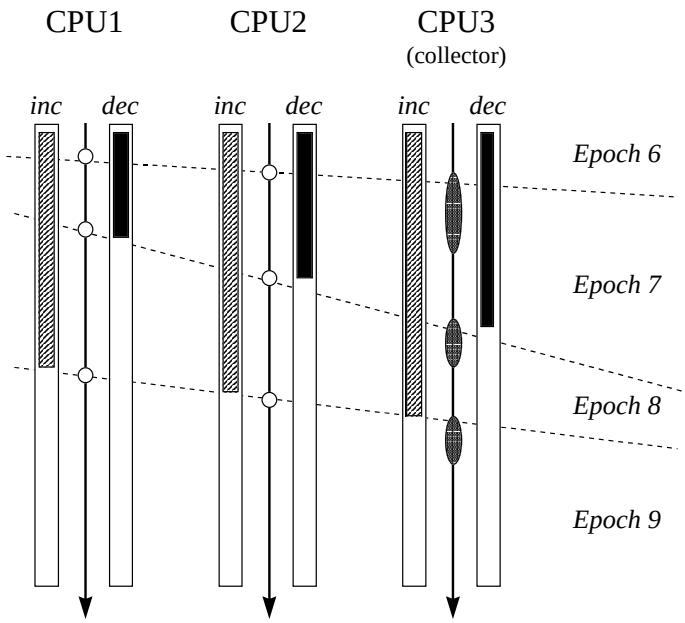

Fig. 2. The Concurrent Reference Counting Collector. Arrows represent the execution of the CPUs; bubbles are interruptions by the collector. Increment and decrement operations are accumulated by each CPU into a buffer. At the end of epoch 8, the collector, running on CPU 3, will process all increments through epoch 8 and all decrements through epoch 7.

lector shares some characteristics with the Deutsch-Bobrow algorithm and its descendants [Deutsch and Bobrow 1976; Rovner 1985; DeTreville 1990], as discussed in Section 11.

The Recycler is a producer-consumer system: the mutators produce operations on reference counts, which are placed into buffers and periodically turned over to the collector, which runs on its own processor. The collector is single-threaded, and is the only thread in the system which is allowed to modify the reference count fields of objects.

The operation of the collector is shown in Figure 2. During mutator operation, updates to the stacks are not reference-counted. Only heap updates are reference-counted, and those operations are deferred with by storing the addresses of objects whose counts must be adjusted into *mutation buffers*, which contain increments or decrements. Objects are allocated with a reference count of 1, and a corresponding decrement operation is immediately written into the mutation buffer; in this manner, temporary objects never stored into the heap are collected quickly.

Time is divided into *epochs*, which are separated by *collections* which comprise each processor briefly running its collector thread. Epoch boundaries are staggered; the only restriction being that all processors must participate in one collection before the next collection can begin.

Periodically, some event will *trigger* a collection cycle: either because a certain amount

of memory has been allocated, or because a mutation buffer is full, or because a timer has expired. In normal operation, none of these triggers will cause the mutator to block; however, they will schedule the collector thread to run on the first processor.

On the first processor when the collector thread wakes up it scans the stacks of its local threads, and places the addresses of objects in the stack into a *stack buffer*. It then increments its local epoch number, allocates a new mutation buffer, and schedules the collector thread on the next processor to run. Finally, it dispatches to the thread that was interrupted by collection.

The collector thread performs these same operations for each processor until it reaches the last processor. The last processor actually performs the work of collection.

The last processor scans the stacks of its local threads into a stack buffer. Then it processes increments: the reference count of each object addressed in the stack buffer for the current epoch computed by each processor is incremented. Then the mutation buffer for each processor for the current epoch is scanned, and the increment operations it contains are performed.

To avoid race conditions that might cause the collector to process a decrement before the corresponding increment has been processed, not only must we process the increment operations first, but we must process the decrement operations one epoch behind. So the last processor scans the stack buffers of the previous epoch, and decrements the reference counts of objects that they address, and then processes the mutation buffers of the previous epoch, performing the decrement operations.

During the decrement phase, any object whose reference count drops to 0 is immediately freed, and the reference counts of objects it points to are recursively decremented.

Finally, the stack and mutation buffers of the previous epoch are returned to the buffer pool, and the epoch number is incremented. The collection has finished and all processors have joined the new epoch, and now any processor can trigger the next collection phase.

3.1 Pseudo-code and Explanation

The pseudo-code of the algorithm is shown in Figure 3. The implementation is assumed to be running on `NCPU` processors, numbered from 1 to `NCPU`. Processors 1 to $\texttt{NCPU}-1$ run the mutators; processor `NCPU` runs the collector.

When a collection is triggered, epoch processing sweeps across the processors in order as a wave front starting with processor 1. When the wave-front reaches processor `NCPU`, the collection is performed.

Each processor contains a local data structure, called `CPU`, with a number of fields: `Id` is the processor number; `Epoch` is the current epoch in which the processor is participating; `Threads` is the set of threads local to the processor; `Ops` is the number of increment/decrement operations since the last collection was triggered; and `Alloc` is the number of bytes allocated since the last collection was triggered.

Each processor accumulates buffers of increments, decrements, and stack pointers. Conceptually, there is an infinite two-dimensional arrays of these buffers, indexed by processor number and epoch number. In practice, the arrays are of rank $(\texttt{NCPU}-1)\times 2$, since we only need the information from the last two epochs. The epoch number is taken modulo 2 to index these arrays, which are named `Inc`, `Dec`, and `Stk`. They correspond to the increment, decrement, and stack buffers, respectively.

The `Increment()`, `Decrement()`, `Release()`, and `Update()` operations with those for the synchronous collector in Figure 1. The other functions are:

```
EnqueueIncrement(S)
  MutationTrigger()
  E = CPU.Epoch mod 2
  while (full(Inc[CPU.Id,E]))
    Wait(CPU.Epoch)
  append S to Inc[CPU.Id,E]

EnqueueDecrement(S)
  MutationTrigger()
  E = CPU.Epoch mod 2
  while (full(Dec[CPU.Id,E]))
    Wait(CPU.Epoch)
  append S to Dec[CPU.Id,E]

MutationTrigger()
  if (++CPU.Ops > MaxOps)
    if (Trigger())
      CPU.Ops = 0

Allocate(n)
  CPU.Alloc += n
  if (CPU.Alloc > MaxAlloc)
    if (Trigger())
      CPU.Alloc = 0
  P = SystemAllocate(n)
  while (P == null)
    Wait(CPU.Epoch)
      P = SystemAllocate(n)
  RC(P) = 1
  EnqueueDecrement(P)
  return P

Trigger()
  if (! Test&Set(GCLock))
    Schedule(NextEpoch, 1)
    return true
  else
    return false

NextEpoch()
  if (CPU.Id < NCPU)
    ProcessEpoch()
    Schedule(NextEpoch,CPU.Id+1)
  else
    Collect()

ProcessEpoch()
  ScanStacks()
  SynchronizeMemory()
  CPU.Epoch = CPU.Epoch + 1

ScanStacks()
  E = CPU.Epoch mod 2
  for T in CPU.Threads
    for P in Pointers(Stack(T))
      append P to Stk[CPU.Id,E]

Collect()
  ProcessIncrements()
  ProcessDecrements()
  CollectCycles()
  Notify(CPU.Epoch)
  Fetch&Store(GCLock, 0)
  CPU.Epoch = CPU.Epoch + 1

ProcessIncrements()
  E = CPU.Epoch mod 2
  for P in 1..NCPU-1
    IncrementStack(P,E)
    for M in Inc[P,E]
      Increment(M)
    clear Inc[P,E]

IncrementStack(P,E)
  for N in Stk[P,E]
    Increment(N)

ProcessDecrements()
  E = (CPU.Epoch - 1) mod 2
  for P in 1..NCPU-1
    DecrementStack(P,E)
    for M in Dec[P,E]
      Decrement(M)
    clear Dec[P,E]

DecrementStack(P,E)
  for N in Stk[P,E]
    Decrement(N)
  clear Stk[P,E]
```

Fig. 3. Algorithm 2 — a concurrent reference-counting collector for a multi-threaded language without cyclic structures. Threads may not migrate and optimization of stack scanning is not performed.

`EnqueueIncrement(S)` Called by `Update()` when a new pointer is added to the heap. First, the `MutationTrigger()` is called to determine if it is time to initiate a collection. Then the epoch index is calculated and if the current increment buffer is full, the processor must wait until the epoch processing is complete. Normally, the buffer is not full and the increment is simply appended to the buffer.

`EnqueueDecrement(S)` Like `EnqueueIncrement(S)`, but called when a pointer is removed from the heap.

`MutationTrigger()` Checks whether the number of heap mutations by this processor, `CPU.Ops`, exceeds the constant `MaxOps`. If so, it calls `Trigger()` to attempt to trigger a garbage collection. If triggering is successful, `CPU.Ops` is reset to 0.

`Allocate(n)` Allocates `n` bytes of storage. First `n` is added to `CPU.Alloc`, the number of bytes allocated by this processor since the last collection was triggered. If as a result `CPU.Alloc` exceeds the constant `MaxAlloc`, the `Trigger()` function is called to initiate a garbage collection. If a collection was triggered, `CPU.Alloc` is reset to 0.

Next, the system allocator is called to obtain an `n`-byte block. If the resulting pointer is null, the allocator is out of space and a loop is entered which attempts the allocation again after the next epoch. Note that since collection is concurrent, there is no straightforward way to determine that the `Allocate()` function should give up and throw an error.

In the normal case, however, `P` is not null and is initialized with a reference count of 1. A decrement is enqueued, and the pointer to storage is returned.

`Trigger()` Attempts to initiate a garbage collection by acquiring the `GCLock` using a test-and-set operation. If successful, the `NextEpoch()` function is scheduled on processor 1 and `true` is returned; otherwise, a collection is already in progress and `false` is returned.

`NextEpoch()` Performs the epoch boundary processing for a processor. If the processor is one of the mutator processors, `ProcessEpoch()` is invoked and the `NextEpoch()` function is scheduled on the next processor; otherwise, the current processor is the last processor, responsible for collection, and `Collect()` is called.

`ProcessEpoch()` Performs local epoch boundary processing on the mutator processors. First, the `ScanStacks()` function accumulates objects pointers residing in the stacks of local threads into the local stack buffer. Then the system routine `SynchronizeMemory()` is called to ensure that all buffer updates are visible to the collection processor. Finally, the local epoch number is advanced.

`ScanStacks()` The epoch index is calculated, and then for each thread local to the processor, each pointer `P` in the stack of the thread is appended to the `Stk` buffer.

`Collect()` When the epoch wave-front reaches the final processor, this function is called to perform the actual work of garbage collection. First, all increments are processed, and then all decrements. Then `Notify` wakes up any threads that were forced to wait due to resource limitations. Finally, the `GCLock` is cleared, and the local epoch number is incremented.

`ProcessIncrements()` Applies all increments for a collection. First, the epoch index `E` is calculated. For each mutator processor, `IncrementStack()` is invoked to apply the stack increments, and then for each increment in each increment buffer, the increments are applied and then the increment buffer is cleared.

```
ScanStacks()
  E = CPU.Epoch mod 2
  for T in CPU.Threads
    if (Active(T,E))
      for P in Pointers(Stack(T))
        append P to Stk[T.Id,E]

IncrementStack(P,E)
  for T in P.Threads
    for N in Stk[T.Id,E]
      Increment(N)
```

```
DecrementStack(P,E)
  for T in P.Threads
    DecrementThread(T,E)

DecrementThread(T,E)
  F = (E+1) mod 2
  if (Active(T,F))
    for N in Stk[T.Id,E]
      Decrement(N)
    FreeBuffer(Stk[T.Id,E])
  else
    Stk[T.Id,F] = Stk[T.Id,E]
```

Fig. 4. Algorithm 3. Algorithm 2 extended to avoid repeated scanning of stacks belonging to idle threads.

`IncrementStack(P,E)` Applies the stack pointers for processor `P` and epoch `E` as increments.

`ProcessDecrements()` Applies all the decrements for a collection. First, the epoch index `E` of the *previous* epoch is calculated (because decrements are applied one epoch behind increments). Then for each processor, `DecrementStack()` is invoked to apply the stack decrements, and then for each decrement in each decrement buffer, the decrements are applied and the buffer is cleared.

`DecrementStack(P,E)` Applies the stack pointers for processor `P` and epoch `E` as decrements, and then clears the buffer.

3.2 Optimization of Stack Scanning

A problem with the algorithm as we have described it is that the stack of each thread is scanned for each epoch, even if the thread has been idle. As a result, the pause times will increase with the number of total threads in the system, and the collector will uselessly perform complementary increment and decrement operations in every collection on the objects referenced from the stacks of idle threads.

We now describe a refinement of the algorithm which eliminates this inefficiency; the refined algorithm is the one we have actually implemented.

Instead of a per-processor stack buffer, there are stack buffers for each thread, as well as a flag to keep track of whether the thread has been active in the current epoch, denoted `Active(T,E)`. When an epoch boundary occurs on a processor, instead of scanning the stacks of all threads, only the stacks of active threads are scanned.

When collection takes place on the last processor, the per-thread increment buffers are processed in the same manner as the per-processor buffers of the simpler algorithm. Any threads that were not active will have empty increment buffers.

In decrement processing, if the thread was active in the current epoch then the decrements from the previous epoch are applied as usual. However, if the thread was not active, then the decrements are not applied and the decrement buffer is *promoted* to be the decrement buffer of the current epoch, to be applied at the end of the next epoch.

Note that heap increments and decrements are unaffected by this optimization.

These refinements are shown in Figure 4. The procedures `ScanStacks` and `IncrementStack` are straightforward adaptations of the analogous routines from Figure 3,

modified to keep per-thread rather than per-processor stack buffers. `DecrementStack` invokes `DecrementThread` for each thread on the processor it is handling. `DecrementThread` then checks the `Active` flag for the current epoch (`F`) and either applies the decrements or promotes them.

A natural refinement is to apply this optimization to unchanged portions of the thread stack, so that the entire stack is not rescanned each time for deeply recursive programs. This is equivalent to the generational stack collection technique of Cheng et al. [1998]; so far we have not implemented this optimization since our benchmarks are not deeply recursive.

Also note that additional synchronization is required in order to allow thread migration to occur; otherwise a thread stack might be scanned twice or not at all. These implementation details have been omitted since they are not fundamental to the operation of the garbage collection algorithm.

3.3 Parallelization

Our collector is concurrent (it operates simultaneously with the mutators) but not parallel (the actual work of collection is only performed on the distinguished last CPU). The scalability of the collector is therefore limited by how well the collector processor can keep up with the mutator processors. Our design point was for one collector CPU to be able to handle about 3 mutator CPU's, so that for four-processor chip multiprocessors (CMPs) one CPU would be dedicated to collection.

It is possible to parallelize our algorithm, particularly the reference counting described in this section. Most straightforwardly, work could be partitioned by address, with different processors handling reference count updates for different address ranges. If these were the same address ranges out of which those processors allocated memory, locality would tend to be good except when there was a lot of thread migration due to load imbalance.

A scheme which is in many ways simpler and would have better load balance, would be to straightforwardly parallelize the reference count updates and use fetch-and-add operations to ensure atomicity on the reference count word. The problem is that now all operations on the reference count field will incur a synchronization overhead.

These solutions only address the problem of reference counting; cycle collection, which is discussed in Sections 5 and 6 is harder to parallelize, although it would be possible to use the techniques in this paper for a local "cluster" of processors and then use techniques borrowed from the distributed computing community to collect inter-cluster cycles [Rodrigues and Jones 1998].

4. PROOF OF CORRECTNESS: REFERENCE COUNTING ALGORITHM

4.1 The Abstract Graph

For the purpose of reasoning about the collection algorithms it is useful to define an abstract graph G_i for the i^{th} epoch of the garbage collector. At the end of each epoch the collector gets a set of increments and decrements from each of the mutator threads. If the increment refers to a new node, it implies the creation of that node. In addition, each increment implies addition of a directed edge between two nodes and each decrement implies deletion of an edge. The increments and decrements do not provide the source of the edges, so in practice we cannot build this graph, nor do we need to build it for the purpose of the algorithm. But for the purposes of the proof it is useful to conceptualize this graph. The

graph G_i denotes the graph that is generated by adding nodes for each reference to a new node, and then first inserting and then deleting edges to G_{i-1} corresponding to the increments of the i^{th} epoch and the decrements of the $i-1^{st}$ epoch. In addition, when a node is determined to be garbage and is freed, it is deleted from G_i. The system begins by executing epoch 1 with the abstract graph $G_0 = \emptyset$.

We can similarly define an abstract set of roots R_i for each epoch i. The roots are either in the mutator stacks or in global (class static) variables. The roots in the mutator stacks are named by the increments collected from the stack snapshots of each mutator for the epoch. The roots from the global variables are the sources in edges implied by increment and decrement operations whose source is a global variable instead of a heap variable. R_i is simply the union of these two types of roots.

Note that due to concurrency and the fact that decrements are processed an epoch behind increments, it is possible that both G_i and R_i can contain multiple edges corresponding to a single field of an object or static variable.

Given G_i and R_i, we can then define the garbage subgraph of G_i, which we denote Γ_i, as

$$\Gamma_i = G_i - R_i^\star$$

that is, the set difference G_i minus the transitive closure of the roots R_i.

4.2 Proof Overview

To prove that the concurrent reference counting algorithm is correct, we must prove two things: first, that an object is not freed while it is live (safety); and second, that all objects eventually stabilize at the correct reference count and objects with zero references counts are freed (liveness).

In our collector we treat the references to an object from heap and stack differently. To simplify the proof, initially we will assume that we treat pushing and popping on the stack the same way as we treat creation and deletion of references from a heap object. Namely, we will assume that we note a increment and decrement on the reference count for every push and pop respectively. Later, we will show that the different treatment of the stack does not change the correctness of the algorithm.

4.3 Safety

To prove safety we must prove that an object's reference count is not prematurely decremented to zero, even in the presence of concurrent updates to the object graph by multiple processors.

PROPOSITION 4.3.1. *There is no causal dependency from any program point in epoch i to any other program point in epoch $i-e$, where $e > 1$.*

The proposition is enforced by the implementation. One collection is not allowed to be initiated until the previous one has completed (by use of `GCLock` in Figure 3). Therefore, at any one time, there will be at most two epochs, i and $i+1$, executing concurrently across multiple processors. It is therefore possible for causal dependencies to exist from epoch $i+1$ to epoch i. But any longer reverse dependence is prevented by the use of `GCLock`.

Intuitively, one can think of two epoch boundaries together as forming the equivalent of a barrier synchronization, even though the implementation only requires lightweight pairwise synchronization. This is illustrated in Figure 5: the causal dependence (x, y) may

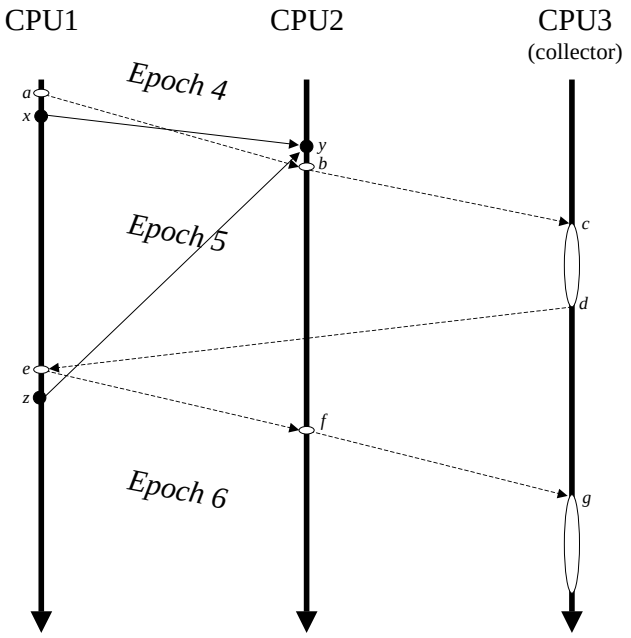

Fig. 5. A causal dependence from one epoch to the previous one is possible, as in (x, y), but a wider backward epoch dependence, like (z, y) is not. The pair-wise synchronizations (a, b) and (b, c) are the boundary between epoch 4 and 5. Similarly (e, f) and (f, g) are the boundary between epoch 5 and 6. The synchronization (d, e) enforced by `GCLock` prevents one epoch from starting before the previous one has completed.

occur since it is unaffected by the pair-wise synchronization (a, b), so there is a causal dependence from x in epoch 5 to y in epoch 4. However, the causal dependence (z, y) is not possible, because y precedes b (they are sequential events on processor 2), b precedes c (they are pair-wise synchronized), c precedes d (they are sequential events on processor 3), d precedes e (they are pair-wise synchronized), and e precedes z (they are sequential events on processor 1). Thus the chain of sequential dependence (y, b, c, d, e, z) must hold and z can not precede y.

THEOREM 4.3.1. *If there is a causal dependence from a write of a pointer to a particular object x to a deletion of a pointer to the same object x, then the increment of the reference count of x will be processed before the decrement to the reference count of x.*

PROOF. Assume that a reference to an object in the heap is created in epoch i and the reference is deleted in epoch j. Since the creation causally precedes the deletion, by proposition 4.3.1 $j \geq i - 1$. The increment will be processed at the end of epoch i. The decrement will be processed at the end of epoch $j + 1$. Since $j \geq i - 1$, the decrement will be processed at the end of some epoch $k \geq i$. If $k > i$ then the increment is processed first because epochs are processed in order. If $k = i$ then the increment is processed

first because within an epoch, all increments are processed before any decrements are applied. □

Note that the theorem only enforces the causal order from increments to decrements, and not any other order (decrement to increment, increment to increment, or decrement to decrement). Therefore, the reference count may be more or less than the "actual" number of references to the object in the memory (although note that this concept in itself assumes a total ordering a memory operations, which we do not assume). The only invariant which is enforced by this theorem is that the reference count does not prematurely reach zero.

4.4 Liveness

To prove the liveness of the collector we must also prove that if an object becomes garbage, its reference count in the abstract graph will reach actual number of references to it.

THEOREM 4.4.1. *If an object becomes garbage during epoch i (i.e. it is inaccessible from any of the roots) then its reference count will equal the number of references to it by the end of processing for epoch $i+1$ and hence it would be a part of Γ_{i+1}.*

PROOF. By assumption we have buffered an increment for each reference to this object that was created and a decrement for each reference to the object that was deleted. Therefore if we apply all of these increments and decrements the reference count of this object will be exactly the number of references to this object.

Once an object becomes garbage there can be no further mutations involving that object. Therefore there will be no increments or decrements in epoch $i+1$ or later. By the end of processing for epoch i we will have applied all increments involving this object and by the end of processing for epoch $i+1$ we will have applied all decrements involving this object.

Since our abstract graph G_{i+1} will reflect this, the object will be a part of Γ_{i+1}. □

Notice that the reference count will be zero if the object is not part of cyclic garbage.

If the object is a part of cyclic garbage, it will continue to have references from other garbage objects and therefore will have a non-zero reference count. We will dicuss how to deal with this in Sections 5, 6, and 7.

4.5 Equivalence of Deferred Stack Processing

THEOREM 4.5.1. *The use of stack buffers and object allocation as in Figure 3 is equivalent to the simplified model in which each push and pop corresponds to an increment and a decrement, respectively.*

PROOF. When an object is allocated its reference count is set to 1, and a decrement is enqueued. Initializing the count to 1 corresponds to enqueueing an increment for the stack push that corresponds to returning the allocated object in the simplified model. If the reference is no longer on the stack at the end of the epoch, then the decrement corresponds to the explicit pop in the simplified model. If the reference is still on the stack, then the decrement cancels out the increment which is applied since the pointer is found by scanning the stack.

When references to objects are pushed onto and popped from the stack we do not enqueue increments or decrements. Instead we scan the stack at the end of each epoch and buffer the references we see on the stack of each thread. Then at the end of epoch i we increment the reference count of an object for each reference to the object from the stack

buffer created for epoch i. We had incremented the reference count of the object at the end of epoch $i-1$ for each reference to the object from the stack buffer created for epoch $i-1$. To correct for this, at the end of epoch i we decrement the reference count for each reference to the object from the stack buffer created at the end of epoch $i-1$. Thus after the processing at the end of the epoch i, the reference count of an object includes the number of references to the object from the stacks of the threads on different processors. □

We do not include a proof of correctness for the optimization of Figure 4, which simply optimizes away the processing of unchanged thread stacks, which clearly will lead to complementary increments and decrements.

5. SYNCHRONOUS CYCLE COLLECTION

5.1 Previous Work on Cycle Collection

Previous work on solving the cycle collection problem in reference counted collectors has fallen into three categories:

—special programming idioms, like *groups* [Bobrow 1980], or certain functional programming styles;

—use of an infrequently invoked tracing collector to collect cyclic garbage [DeTreville 1990]; or

—searching for garbage cycles by removing internal reference counts [Christopher 1984; Martínez et al. 1990].

An excellent summary of the techniques and algorithms is in chapter 3 ("Reference Counting") of the book by Jones and Lins [1996]. The first algorithm for cycle collection in a reference counted system was devised by Christopher [1984]. Our synchronous cycle collection algorithm is based on the work of Martínez et al. [1990] as extended by Lins [1992a], which is very clearly explained in the chapter of the book just mentioned.

There are two observations that are fundamental to these algorithms. The first observation is that garbage cycles can only be created when a reference count is decremented to a non-zero value — if the reference count is incremented, no garbage is being created, and if it is decremented to zero, the garbage has already been found. Furthermore, since reference counts of one tend to predominate, decrements to zero should be common.

The second observation is that in a garbage cycle, all the reference counts are internal; therefore, if those internal counts can be subtracted, the garbage cycle will be discovered.

As a result, when a reference count is decremented and does not reach zero, it is considered as a candidate root of a garbage cycle, and a local search is performed. This is a depth-first search which subtracts out counts due to internal pointers. If the result is a collection of objects with zero reference counts, then a garbage cycle has been found and is collected; if not, then another depth-first-search is performed and the counts are restored.

Lins [1992a] extended the original algorithm to perform the search lazily by buffering candidate roots instead of exploring them immediately. This has two advantages. Firstly, after a time, the reference count of a candidate root may reach zero due to other edge deletions, in which case the node can simply be collected, or the reference count may be re-incremented due to edge additions, in which case it may be ignored as a candidate root. Secondly, it will often prevent re-traversal of the same node.

Unfortunately, in the worst case Lins' algorithm is quadratic in the size of the graph, as for example in the cycle shown in Figure 6. His algorithm considers the roots one at a

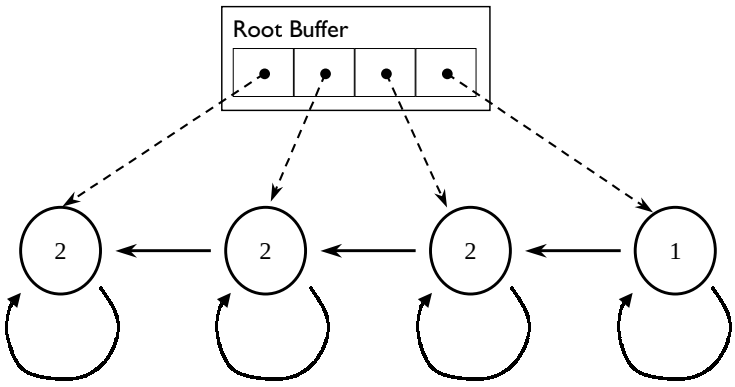

Fig. 6. Example of compound cycle that causes Lins' algorithm to exhibit quadratic complexity.

time, performing the reference count subtraction and restoration passes for that root before moving on.

Therefore, Lins' algorithm will perform a complete scan from each of the candidate roots until it arrives at the final root, at which point the entire compound cycle will be collected.

5.2 The Synchronous Cycle Collection Algorithm

In this section we describe our synchronous cycle collection algorithm, which applies the same principles as those of Martínez et al. and Lins, but which only requires $O(N + E)$ worst-case time for collection (where N is the number of nodes and E is the number of edges in the object graph), and is therefore competitive with tracing garbage collectors.

We also improve the practicality of the algorithm by allowing resizing of collected objects, and show how significant constant-time improvements can be achieved by ruling out inherently acyclic data structures.

Our synchronous algorithm is similar to Lins' algorithm: when reference counts are decremented, we place potential roots of cyclic garbage into a buffer called `Roots`. Periodically, we process this buffer and look for cycles by subtracting internal reference counts.

There are two major changes that make the algorithm linear time: first of all, we add a *buffered* flag to every object, which is used to prevent the same object being added to the root buffer more than once per cycle collection. This in turn places a linear bound on the size of the buffer.

Secondly, we analyze the entire transitive closure of `Roots` as a single graph, rather than as a set of graphs. This means that the complexity of the algorithm is limited by the size of that transitive closure, which in turn is limited by $N + E$ (since `Roots` is bounded by N by the use of the buffered flag). Of course, in practice we hope that the transitive

Color	Meaning
Black	In use or free
Gray	Possible member of cycle
White	Member of garbage cycle
Purple	Possible root of cycle
Green	Acyclic
Red	Candidate cycle undergoing Σ-computation
Orange	Candidate cycle awaiting epoch boundary

Table I. Object Colorings for Cycle Collection. Orange and red are only used by the concurrent cycle collector and are described in Section 6.

closure will be significantly smaller.

In practice we found that the first change (the use of the buffered flag) made almost no difference in the running time of the algorithm; however, the second change (analyzing the entire graph at once) made an enormous difference in run-time. When we applied Lins' algorithm unmodified to large programs, garbage collection delays extended into minutes.

5.3 Pseudo-code and Explanation

We now present detailed pseudo-code and an explanation of the operation of each procedure in the synchronous cycle collection algorithm.

In addition to the buffered flag, each object contains a color and a reference count. For an object `T` these fields are denoted `buffered(T)`, `color(T)`, and `RC(T)`. In the implementation, these quantities together occupy a single word in each object.

All objects start out black. A summary of the colors used by the collector is shown in Table I. The use of green (acyclic) objects will be discussed below.

The algorithm is shown in Figure 7. The procedures are explained in detail below. `Increment` and `Decrement` are invoked externally as pointers are added, removed, or overwritten. `CollectCycles` is invoked either when the root buffer overflows, storage is exhausted, or when the collector decides for some other reason to free cyclic garbage. The rest of the procedures are internal to the cycle collector. Note that the procedures `MarkGray`, `Scan`, and `ScanBlack` are the same as for Lins' algorithm.

`Increment(S)` When a reference to a node `S` is created, the reference count of `T` is incremented and it is colored black, since any object whose reference count was just incremented can not be garbage.

`Decrement(S)` When a reference to a node `S` is deleted, the reference count is decremented. If the reference count reaches zero, the procedure `Release` is invoked to free the garbage node. If the reference count does not reach zero, the node is considered as a possible root of a cycle.

`Release(S)` When the reference count of a node reaches zero, the contained pointers are deleted, the object is colored black, and unless it has been buffered, it is freed. If it has been buffered, it is in the `Roots` buffer and will be freed later (in the procedure `MarkRoots`).

`PossibleRoot(S)` When the reference count of `S` is decremented but does not reach zero, it is considered as a possible root of a garbage cycle. If its color is already purple, then it is already a candidate root; if not, its color is set to purple. Then the *buffered*

```
Increment(S)
  RC(S) = RC(S) + 1
  color(S) = black

Decrement(S)
  RC(S) = RC(S) - 1
  if (RC(S) == 0)
    Release(S)
  else
    PossibleRoot(S)

Release(S)
  for T in children(S)
    Decrement(T)
  color(S) = black
  if (! buffered(S))
    SystemFree(S)

PossibleRoot(S)
  if (color(S) != purple)
    color(S) = purple
    if (! buffered(S))
      buffered(S) = true
      append S to Roots

CollectCycles()
  MarkRoots()
  ScanRoots()
  CollectRoots()

MarkRoots()
  for S in Roots
    if (color(S) == purple
        and RC(S) > 0)
      MarkGray(S)
    else
      buffered(S) = false
      remove S from Roots
      if (color(S) == black and
          RC(S) == 0)
        SystemFree(S)

ScanRoots()
  for S in Roots
    Scan(S)

CollectRoots()
  for S in Roots
    remove S from Roots
    buffered(S) = false
    CollectWhite(S)

MarkGray(S)
  if (color(S) != gray)
    color(S) = gray
    for T in children(S)
      RC(T) = RC(T) - 1
      MarkGray(T)

Scan(S)
  if (color(S) == gray)
    if (RC(S) > 0)
      ScanBlack(S)
    else
      color(S) = white
      for T in children(S)
        Scan(T)

ScanBlack(S)
  color(S) = black
  for T in children(S)
    RC(T) = RC(T) + 1
    if (color(T) != black)
      ScanBlack(T)

CollectWhite(S)
  if (color(S) == white
      and ! buffered(S))
    color(S) = black
    for T in children(S)
      CollectWhite(T)
    SystemFree(S)
```

Fig. 7. Synchronous Cycle Collection

flag is checked to see if it has been purple since we last performed a cycle collection. If it is not buffered, it is added to the buffer of possible roots.

`CollectCycles()` When the root buffer is full, or when some other condition, such as low memory occurs, the actual cycle collection operation is invoked. This operation has three phases: `MarkRoots`, which removes internal reference counts; `ScanRoots`, which restores reference counts when they are non-zero; and finally `CollectRoots`,

which actually collects the cyclic garbage.

`MarkRoots()` The marking phase looks at all the nodes `S` whose pointers have been stored in the `Roots` buffer since the last cycle collection. If the color of the node is purple (indicating a possible root of a garbage cycle) and the reference count has not become zero, then `MarkGray(S)` is invoked to perform a depth-first search in which the reached nodes are colored gray and internal reference counts are subtracted. Otherwise, the node is removed from the `Roots` buffer, the `buffered` flag is cleared, and if the reference count is zero the object is freed.

`ScanRoots()` For each node `S` that was considered by `MarkGray(S)`, this procedure invokes `Scan(S)` to either color the garbage subgraph white or re-color the live subgraph black.

`CollectRoots()` After the `ScanRoots` phase of the `CollectCycles` procedure, any remaining white nodes will be cyclic garbage and will be reachable from the `Roots` buffer. This procedure invokes `CollectWhite` for each node in the `Roots` buffer to collect the garbage; all nodes in the root buffer are removed and their `buffered` flag is cleared.

`MarkGray(S)` This procedure performs a simple depth-first traversal of the graph beginning at `S`, marking visited nodes gray and removing internal reference counts as it goes.

`Scan(S)` If this procedure finds a gray object whose reference count is greater than one, then that object and everything reachable from it are live data; it will therefore call `ScanBlack(S)` in order to re-color the reachable subgraph and restore the reference counts subtracted by `MarkGray`. However, if the color of an object is gray and its reference count is zero, then it is colored white, and `Scan` is invoked upon its children. Note that an object may be colored white and then re-colored black if it is reachable from some subsequently discovered live node.

`ScanBlack(S)` This procedure performs the inverse operation of `MarkGray`, visiting the nodes, changing the color of objects back to black, and restoring their reference counts.

`CollectWhite(S)` This procedure recursively frees all white objects, re-coloring them black as it goes. If a white object is buffered, it is not freed; it will be freed later when it is found in the `Roots` buffer.

5.4 Acyclic Data Types

A significant constant-factor improvement can be obtained for cycle collection by observing that some objects are inherently acyclic. We speculate that they will comprise the majority of objects in many applications. Therefore, if we can avoid cycle collection for inherently acyclic objects, we will significantly reduce the overhead of cycle collection as a whole.

In Java, dynamic class loading complicates the determination of inherently acyclic data structures. We have implemented a very simple scheme as part of the class loader. Acyclic classes may contain:

—scalars;

—references to classes that are both acyclic and `final`; and

—arrays of either of the above.

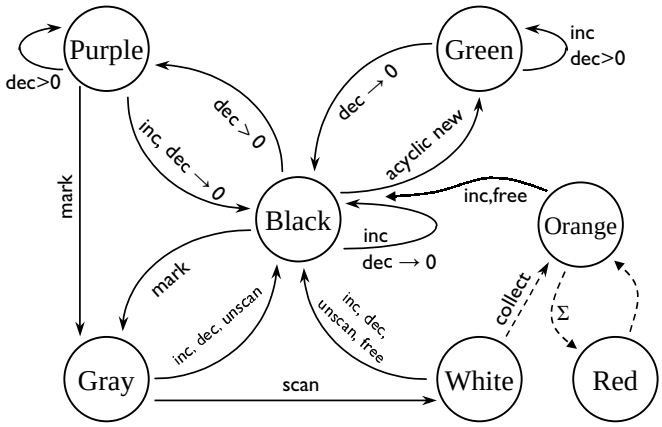

Fig. 8. State transition graph for cycle collection.

Our implementation marks objects whose class is acyclic with the special color green. Green objects are ignored by the cycle collection algorithm, except that when a dead cycle refers to green objects, they are collected along with the dead cycle. For simplicity of the presentation, we have not included consideration of green objects in the algorithms in this paper; the modifications are straightforward.

While our determination of acyclic classes is very simple, it is also very effective, usually reducing the objects considered as roots of cycles by an order of magnitude, as will be shown in Section 10. In a static compiler, a more sophisticated program analysis could be applied to increase the percentage of green objects.

6. CONCURRENT CYCLE COLLECTION

We now describe a concurrent cycle collection algorithm based on the principles of our synchronous algorithm of the previous section.

6.1 Two Phase Cycle Collection

Now that we have abstracted the concurrent system to a collection of mutators emitting streams of increment and decrement operations, and a reference counting collector which merges and applies these operations, we can describe in overview how the algorithm operates.

The concurrent cycle collection algorithm is more complex than the synchronous algorithm. As with other concurrent garbage collection algorithms, we must contend with the fact that the object graph may be modified simultaneously with the collector scanning it; but in addition the reference counts may be as much as a two epochs out of date (because decrements are deferred by an epoch).

Our algorithm relies on the same basic premise as the synchronous algorithm: namely, that given a subset of nodes, if deleting the internal edges between the nodes in this subset reduces the reference count of every node in the subset to zero, then the whole subset of

nodes is cyclic garbage. (The subset may represent more than one independent cycles, but they are all garbage cycles.)

However, since the graph may be modified, we run into three basic difficulties. Firstly, since we can not rely on being able to retrace the same graph, the repeated traversal of the graph does not define the same set of nodes. Secondly, the deletion of edges can disconnect portions of the graph, thus making the global test by graph traversal difficult. Thirdly, reference counts may be out of date.

Our algorithm proceeds in two phases. In the first phase, we use a variant of the synchronous algorithm as described in Section 5 to obtain a candidate set of garbage nodes. We then wait until an epoch boundary and then perform the second phase in which we test these to ensure that the candidates do indeed satisfy the criteria for garbage cycles.

The two phases can be viewed as enforcing a "liveness" and a "safety" property. The first phase enforces liveness by ensuring that potential garbage cycles are considered for collection. The second phase ensures safety by preventing the collection of false cycles induced by concurrent mutator activity.

6.2 Liveness: Finding Cycles to Collect

Essentially, we use the synchronous algorithm to find candidate cycles. However, due to concurrent mutator activity, the graph may be changing and the algorithm may produce incorrect results.

To perform the concurrent cycle collection, we need a second reference count for each object, denoted `CRC(S)`. This is a hypothetical reference count which may become incorrect due to concurrent mutator activity. In the implementation, we are able to fit both reference counts, the color, and the buffered flag into a single header word by using a hash table to hold count overflows, which occur very rarely.

The liveness phase of the concurrent algorithm proceeds in a similar manner to the synchronous cycle collection algorithm, except that when an object is marked gray its cyclic reference count (`CRC`) is initialized to its true reference count — the "true" reference count (`RC`) is not changed. Henceforward, the mark, scan, and collect phases operate upon the cyclic reference count instead of the true reference count. In the `CollectWhite` procedure, instead of collecting the white nodes as garbage, we color them orange and add them to a set of *possible* garbage.

By using the cyclic reference count we ensure that in the event of concurrent mutator activity, the information about the true reference count of the objects is never lost.

In absence of mutator activity, the liveness phase will yield the set of garbage nodes, and the safety phase will certify that this indeed is a set of garbage of nodes and we can collect them.

However, the presence of concurrent mutator activity can cause live nodes to enter the list in three different ways. Firstly, the mutator can add an edge, thus causing the `MarkGray` procedure to incorrectly infer that there are no external edges to a live object. Secondly, the mutator can delete an edge, thus causing scan procedure to incorrectly infer a live object to be garbage. Thirdly, the deletion of edges concurrent to running of the `MarkGray` and scan procedure can create gray and white nodes with various values of cyclic reference counts. While eventually the reporting of the mutator activity will cause these nodes to be detected and re-colored, if these nodes are encountered before they are re-colored they can mislead the runs of the above procedures into inferring that they are garbage.

The output of phase one is a set of nodes believed to be garbage in the `CycleBuffer` data structure. The `CycleBuffer` is divided into discrete connected components, each of which forms a potential garbage cycle. Due to mutator activity, the contents of the `CycleBuffer` can be a superset of the actual set of garbage nodes and can contain some nodes that fail tests in the safety phase (this is discussed in detail in Section 7).

6.3 Safety: Collecting Cycles Concurrently

The second ("safety") phase of the algorithm takes as input a set of nodes and determines whether they form a garbage cycle. These nodes are marked with a special color, orange, which is used to identify a candidate set in the concurrent cycle collector.

The safety phase of the algorithm consists of two tests we call the Σ-test and the Δ-test. If a subset of nodes of the object graph passes both the Σ-test and the Δ-test, then we can be assured that the nodes in the subset are all garbage. Thus, correctness of the safety phase of our algorithm is not determined by any property of the output of the liveness phase which selects the subgraphs. This property of the safety phase of the algorithm considerably simplifies the proof of correctness as well as modularizing the code.

In theory, it would be possible to build a cycle collector which simply passed random sets of nodes to the safety phase, which would then either accept them as garbage or reject them as live. However, such a collector would not be practical: if we indeed pick a random subset of nodes from the object graph, the chances that they form a complete garbage subgraph is very small. The job of the liveness phase can be seen as finding likely sets of candidates for garbage cycles. If the mutator activity is small in a given epoch, this would indeed be very likely to be true.

The Σ-test consists of two parts: a *preparation* and an actual *test*. In the preparation part, which is performed immediately after the candidate cycles have been found, we iterate over the subset and initialize the cyclic reference count of every node in the subset to the reference count of the node. Then we iterate over every node in the subset again and decrement the cyclic reference count of any children of the node that are also in the subset. At the end of the preparation computation, the cyclic reference count of each node in the subset represents the number of references to the node from nodes external to the subset. In the actual test, which is performed after the next epoch boundary, we iterate over every node in the subset and test if its cyclic reference count is zero.

If it is zero for every member of the set, then we know that there exists no reference to this subset from any other node. Therefore, any candidate set that passes the Σ-test is garbage, unless the reference count used during the running of the preparation procedure is outdated due to an increment to one of the nodes in the subset.

This is ascertained by the Δ-test. We wait until the next epoch boundary, at which point increment processing re-colors all non-black nodes and their reachable subgraphs black. Then we scan the nodes in the candidate set and test whether their color is still orange. If they are all orange, we know that there has been no increment to the reference count during the running of the preparation procedure and we say that the candidate set passed the Δ-test.

Any subset of garbage nodes that does not have any external pointers to it will pass both the tests. Note that we do not have to worry about concurrent decrements to the members of the subset, since it is not possible for the reference count of any node to drop below zero.

However, it is possible for a set of garbage to have pointers to it from other garbage

cycles. For example in Figure 6, only the candidate set consisting of the last node forms isolated garbage cycle. The other cycles have pointers to them from the cycle to their right.

We know that the garbage cycles in the cycle buffer cannot have any forward pointers to other garbage cycles (if they did, we would have followed them and included them in a previous garbage cycle). Hence, we process the candidate cycles in the cycle buffer in the reverse of the order in which we found them. This reasoning is described more formally in Lemma 7.2.1 in Section 7.

When a candidate set passes both tests, and hence is determined to be garbage, then we free the nodes in the cycle, which causes the reference counts of other nodes outside of the cycle to be decremented. By the stability property of garbage, we can decrement such reference counts without concern for concurrent mutation.

When we decrement a reference count to an orange node, we also decrement its cyclic reference count (CRC). Therefore, when the next candidate cycle is considered (the previous cycle in the buffer), if it is garbage the Σ-test will succeed because we have augmented the computation performed by the preparation procedure.

Hence when we reach a candidate set, the cyclic reference count does not include the count of any pointers from a known garbage node. This ensures that all the nodes in Figure 6 would be collected.

A formalism for understanding the structure of the graph in the presence of concurrent mutation, and a proof of correctness of the algorithm is presented in Section 7.

6.4 Pseudo-code and Explanation

We now present the pseudo-code with explanations for each procedure in the concurrent cycle collection algorithm. The pseudo-code is shown in Figures 9 and 10. The operation of `FindCycles` and its subsidiary procedures is very similar to the operation of the synchronous algorithm of Figure 7, so for those procedures we will only focus on the differences in the concurrent versions of the procedures.

`Increment(S)` The true reference count is incremented. Since the reference count is being incremented, the node must be live, so any non-black objects reachable from it are colored black by invoking `ScanBlack`. This has the effect of re-blackening live nodes that were left gray or white when concurrent mutation interrupted a previous cycle collection.

`Decrement(S)` At the high level, decrementing looks the same as with the synchronous algorithm: if the count becomes zero, the object is released, otherwise it is considered as a possible root.

`PossibleRoot(S)` For a possible root, we first perform `ScanBlack`. As with `Increment`, this has the effect of re-blackening leftover gray or white nodes; it may also change the color of some purple nodes reachable from `S` to black, but this is not a problem since they will be considered when the cycle collector considers `S`. The rest of `PossibleRoot` is the same as for the synchronous algorithm.

`CollectCycles()` Invoked once per epoch after increment and decrement processing due to the mutation buffers from the mutator threads has been completed. First, `FreeCycles` attempts to free candidate cycles discovered during the previous epoch. Then `FindCycles` collects new candidate cycles and `SigmaPreparation` prepares for the Σ-test to be run in the next epoch.

```
Increment(S)
  RC(S) = RC(S) + 1
  ScanBlack(S)

Decrement(S)
  RC(S) = RC(S) - 1
  if (RC(S) == 0)
    Release(S)
  else
    PossibleRoot(S)

Release(S)
  for T in children(S)
    Decrement(T)
  color(S) = black
  if (! buffered(S))
    SystemFree(S)

PossibleRoot(S)
  ScanBlack(S)
  color(S) = purple
  if (! buffered(S))
    buffered(S) = true
    append S to Roots

CollectCycles()
  FreeCycles()
  FindCycles()
  SigmaPreparation()

FindCycles()
  MarkRoots()
  ScanRoots()
  CollectRoots()

MarkRoots()
  for S in Roots
    if (color(S) == purple
        and RC(S) > 0)
      MarkGray(S)
    else
      remove S from Roots
      buffered(S) = false
      if (RC(S) == 0)
        SystemFree(S)

ScanRoots()
  for S in Roots
    Scan(S)

CollectRoots()
  for S in Roots
    if (color(S) == white)
      CurrentCycle = empty
      CollectWhite(S)
      append CurrentCycle
        to CycleBuffer
    else
      buffered(S) = false
    remove S from Roots

MarkGray(S)
  if (color(S) != gray)
    color(S) = gray
    CRC(S) = RC(S)
    for T in children(S)
      MarkGray(T)
  else if (CRC(S) > 0)
    CRC(S) = CRC(S) - 1

Scan(S)
  if (color(S) == gray
      and CRC(S) == 0)
    color(S) = white
    for T in children(S)
      Scan(T)
  else
    ScanBlack(S)

ScanBlack(S)
  if (color(S) != black)
    color(S) = black
    for T in children(S)
      ScanBlack(T)

CollectWhite(S)
  if (color(S) == white)
    color(S) = orange
    buffered(S) = true
    append S to CurrentCycle
    for T in children(S)
      CollectWhite(T)
```

Fig. 9. Concurrent Cycle Collection Algorithm (Part 1)

```
SigmaPreparation()
  for C in CycleBuffer
    for N in C
      color(N) = red
      CRC(N) = RC(N)
    for N in C
      for M in children(N)
        if (color(M) == red
            and CRC(M) > 0)
          CRC(M) = CRC(M)-1
    for N in C
      color(N) = orange

FreeCycles()
  last = |CycleBuffer|-1
  for i = last to 0 by -1
    C = CycleBuffer[i]
    if (DeltaTest(C)
        and SigmaTest(C))
      FreeCycle(C)
    else
      Refurbish(C)
  clear CycleBuffer

DeltaTest(C)
  for N in C
    if (color(N) != orange)
      return false
  return true

SigmaTest(C)
  externRC = 0
  for N in C
    externRC = externRC+CRC(N)
  return (externRC == 0)

Refurbish(C)
  first = true
  for N in C
    if ((first and
        color(N)==orange) or
        color(N)==purple)
      color(N) = purple
      append N to Roots
    else
      color(N) = black
      buffered(N) = false
    first = false

FreeCycle(C)
  for N in C
    color(N) = red
  for N in C
    for M in children(N)
       CyclicDecrement(M)
  for N in C
    SystemFree(N)

CyclicDecrement(M)
  if (color(M) != red)
    if (color(M) == orange)
      RC(M) = RC(M) - 1
      CRC(M) = CRC(M) - 1
    else
      Decrement(M)
```

Fig. 10. Concurrent Cycle Collection Algorithm (Part 2)

`FindCycles()` As in the synchronous algorithm, three phases are invoked on the candidate roots: marking, scanning, and collection.

`MarkRoots()` This procedure is the same as in the synchronous algorithm.

`ScanRoots()` This procedure is the same as in the synchronous algorithm.

`CollectRoots()` For each remaining root, if it is white a candidate cycle has been discovered starting at that root. The `CurrentCycle` is initialized to be empty, and the `CollectWhite` procedure is invoked to gather the members of the cycle into the `CurrentCycle` and color them orange. The collected cycle is then appended to the `CycleBuffer`. If the root is not white, a candidate cycle was not found from this root or it was already included in some previously collected candidate, and the buffered flag is set to

false. In either case, the root is removed from the `Roots` buffer, so that at the end of this procedure the `Roots` buffer is empty.

`MarkGray(S)` This is similar to the synchronous version of the procedure, with adaptations to use the cyclic reference count (CRC) instead of the true reference count (RC). If the color is not gray, it is set to gray and the CRC is copied from the RC, and then `MarkGray` is invoked recursively on the children. If the color is already gray, and if the CRC is not already zero, the CRC is decremented (the check for non-zero is necessary because concurrent mutation could otherwise cause the CRC to underflow).

`Scan(S)` As with `MarkGray`, simply an adaptation of the synchronous procedure that uses the CRC. Nodes with zero CRC are colored white; non-black nodes with CRC greater than zero are recursively re-colored black.

`ScanBlack(S)` Like the synchronous version of the procedure, but it does not need to re-increment the true reference count because all reference count computations were carried out on the CRC.

`CollectWhite(S)` This procedure recursively gathers white nodes identified as members of a candidate garbage cycle into the `CurrentCycle` and colors them orange as it goes. The `buffered` flag is also set true since a reference to the node will be stored in the `CycleBuffer` when `CurrentCycle` is appended to it.

`SigmaPreparation()` After the candidate cycles have been collected into the `CycleBuffer`, this procedure prepares for the execution of the Σ-test in the next epoch. It operates individually on each candidate cycle `C`. First, each node `S` in `C` has its CRC initialized to its RC and its color set to red. After this only the nodes of `C` are red. Then for any pointer from one node in `C` to another node in `C`, the CRC of the target node is decremented. Finally, the nodes in `C` are re-colored orange. At the end of `SigmaPreparation`, the CRC field of each node `S` contains a count of the number of references to `S` from outside of `C`.

`FreeCycles()` This procedure iterates over the candidate cycles in the reverse order in which they were collected. It applies the safety tests (the Σ-test and the Δ-test) to each cycle and if it passes both tests then the cycle is freed; otherwise it is refurbished, meaning that it may be reconsidered for collection in the next epoch.

`DeltaTest(C)` This procedure returns true if the color of all nodes in the cycle are orange, which indicates that their have been no increments to any of the nodes in the cycle.

`SigmaTest(C)` This procedure calculates the total number of external references to nodes in the cycle, using the CRC fields computed by the `SigmaPreparation` procedure. It returns true if the number of external references is zero, false otherwise.

`Refurbish(C)` If the candidate cycle has not been collected due to failing a safety test, this procedure re-colors the nodes. If the first node in the candidate cycle (which was the purple node from which the candidate was found) is still orange, or if any node has become purple, then those nodes are colored purple and placed in the `Roots` buffer. All other nodes are colored black and their `buffered` flags are cleared.

`FreeCycle(C)` This procedure actually frees the members of a candidate cycle that has passed the safety tests. First, the members of `C` are colored red; after this, only the nodes in `C` are red. Then for each node `S` in `C`, `CyclicDecrement` decrements reference counts in non-red nodes pointed to by `S`.

`CyclicDecrement(M)` If a node is not red, then it either belongs to some other candidate cycle or not. If it belongs to some other candidate cycle, then it is orange, in which case both the RC and the CRC fields are decremented (the CRC field is decremented to update the computation performed previously by the `SigmaPreparation` procedure to take the deletion of the cycle pointing to `M` into account). If it does not belong to some other candidate cycle, it will not be orange and a normal `Decrement` operation is performed.

For ease of presentation, we have presented the pseudo-code in a way that maximizes readability. However, this means that as presented the code makes more passes over the nodes than is strictly necessary. For instance, the first pass by `SigmaPreparation` can be merged with `CollectWhite`, and the passes performed by `DeltaTest` and `SigmaTest` can be combined. In the implementation, the passes are combined to minimize constant-factor overheads.

7. PROOF OF CORRECTNESS: CYCLE COLLECTION ALGORITHM

In this section we prove the correctness of the concurrent cycle collection algorithm presented in Section 6. We use the concepts of the abstract graph G_i, the roots R_i, and the garbage subgraph Γ_i as defined in Section 4.1.

7.1 Safety: Proof of Correctness

A garbage collector is *safe* if every object collected is indeed garbage. In this section we prove the safety of our algorithm.

At the end of epoch $i+1$, the procedure `ProcessCycles` invokes `FreeCycles` to collect the cycles identified as potential garbage in the collection at the end of epoch i.

Let the set B_i denote the contents of the `CycleBuffer` generated during the cycle collection at the end of epoch i. This is the collection of orange nodes generated by the concurrent variant of the synchronous cycle detection algorithm, which used the set of purple nodes (denoted P_i) as roots to search for cyclic garbage.

B_i is partitioned into the disjoint sets $B_{i1} \ldots B_{in}$. Each B_{ik} is a candidate garbage cycle computed by the cycle collection algorithm from a particular purple node in P_i.

Due to concurrent mutation, B_i may contain nodes from $G_i \cup G_{i+1}$.

THEOREM 7.1.1. *Any set B_{ik} containing nodes that do not exist in G_i (that is, $B_{ik} - G_i \neq \emptyset$) will fail the Δ-test.*

PROOF. The only way for new nodes to be added to G_i in G_{i+1} is by increment operations. However, all concurrent increment operations will have been processed before we apply the Δ-test. Processing an increment operation invokes `ScanBlack`, so that if the node in question is in B_{ik}, then that node (at least) will be re-colored from orange to black. The presence of that black node in B_{ik} will cause the Δ-test to fail. □

Therefore, for a given epoch i with G_i, R_i, and Γ_i, let

Ω_i denote the set containing the sets $B_{ik} \in B_i$ that passed the Δ-test. The sets in Ω_i are denoted Ω_{ij}. Since all Ω_{ij} have passed the Δ-test, all $\Omega_{ij} \subseteq G_i$.

C denote one of the sets Ω_{ij}, namely a set of nodes believed to be a garbage cycle that has passed the Δ-test.

S denote a specific node in that collection ($S \in C$).

$RC(S)$ denote the reference count at a node S in the i^{th} epoch. By definition RC(S) is the reference count of node S in graph G_i.

$RC(S, C)$ denote the number of references to S from nodes within C.

$RC^{\Sigma}(S, C)$ denote the number of references to S from nodes within C as determined by the Σ-test.

$CRC(S)$ denote the hypothetical reference count for $S \in C$ as computed by the Σ-test.

$\overline{C}$ denote $G_i - C$, the complement of C in G_i.

THEOREM 7.1.2. *If C passes the Σ-test, that is if we have computed the values of $CRC(S)$ for each $S \in C$ as described in the procedure* `SigmaPreparation` *in Section 6 and*

$$\sum_{S \in C} CRC(S) = 0$$

then C is a set of garbage nodes ($C \subseteq \Gamma_i$).

PROOF. From the above definitions, for every $S \in C$,

$$RC(S) = RC(S, C) + RC(S, \overline{C})$$

Since we delay the processing of the decrements by one epoch, this ensures that the following properties are true:

$RC(S)$ is non-negative and if it is zero, then S is a garbage node.

$RC(S, \overline{C})$ is non-negative and if it is zero for every node S in C then C is a collection of garbage nodes.

During the Σ-test, we determine the number of references to node S from nodes within C. Therefore by definition,

$$CRC(S) = RC(S) - RC^{\Sigma}(S, C).$$

The $RC^{\Sigma}(S, C)$ may differ from the $RC(S, C)$ because there may be new references to S from nodes within C that were added to G_i, thereby increasing $RC^{\Sigma}(S, C)$; or because there may be references to S from nodes in C that were deleted from G_i, thereby decreasing $RC^{\Sigma}(S, C)$. If the collection passes the Δ-test, then no references were added to S at any time during the last epoch.

Therefore,

$$RC(S, C) \geq RC^{\Sigma}(S, C)$$

and

$$\begin{aligned} CRC(S) &= RC(S) - RC^{\Sigma}(S, C) \\ &\geq RC(S) - RC(S, C) = RC(S, \overline{C}) \end{aligned}$$

If $CRC(S)$ is zero from the Σ-test, since $RC(S, \overline{C})$ is non-negative, it follows that $RC(S, \overline{C})$ has to be zero too. Further, if $RC(S, \overline{C}) = 0$ for every node in the collection C, then the whole collection C is garbage. □

Theorem 7.1.1 and Theorem 7.1.2 show that the Δ-test and the Σ-test are sufficient to ensure that any set C of nodes that passes both tests is garbage. The following theorem shows that both the tests are necessary to ensure this as well.

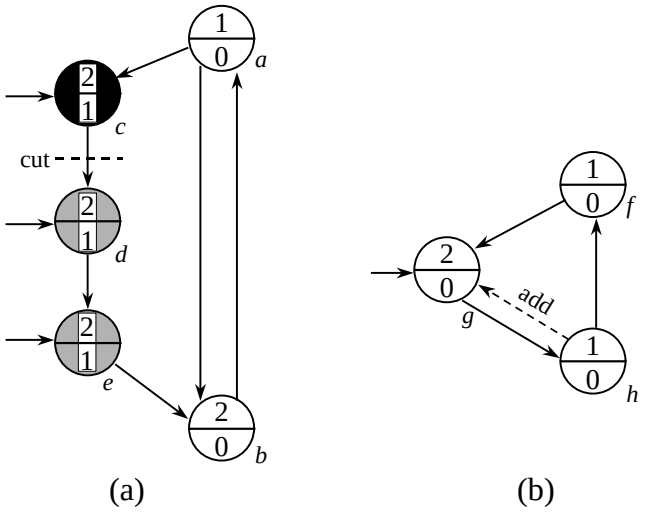

Fig. 11. Race conditions uniquely detected (a) by the Σ-test, and (b) by the Δ-test. The purple nodes from which cycle collection started were a and f. Inside of each node is shown the reference count, or RC (top) and the cyclic reference count, or CRC (bottom).

THEOREM 7.1.3. *Both Δ-test and Σ-test are necessary to ensure that a candidate set B_{ij} contains only garbage nodes.*

PROOF. We prove the theorem by example. Consider the graph of nodes shown in Figure 11 (a). At the end of epoch i, a cycle is detected from the purple node a, which is the starting point from which cycle collection is run. If the edge between nodes c and d is cut by a concurrent mutator between the execution of the `MarkGray` and the `Scan` routines, then the nodes a and b will be collected by the `CollectWhite` routine and form a set B_{ij}. These nodes are not garbage. However, since there have been no increments to the reference counts of either of these nodes, this set will pass Δ-test.

The removal of the edge $c \rightarrow d$ will be part of epoch $i + 1$, but since decrements are processed an epoch later, it will not be applied until the end of epoch $i + 2$. Therefore the decrement to node d will not have an effect on the nodes a and b in the `FreeCycles` operation performed in epoch $i + 1$, which would therefore collect the live nodes a and b.

Waiting for an additional epoch does not ensure that nodes a and b will be detected by Δ-test, since during epoch $i + 1$ the edge from d to e could be cut. Indeed, by making the chain of nodes $\{c, d, e\}$ arbitrarily long and having a malicious mutator cut edges at just the right moment, we can have the non-garbage set of nodes B_{ij} pass the Δ-test for arbitrarily many epochs. Hence the Δ-test alone cannot detect all live nodes in B_i.

Now consider the graph of nodes shown in Figure 11 (b). The cycle is detected by the collection at the end of epoch i by starting at the purple node f. If a new edge is added from node h to node g before the `MarkGray` routine is run (shown as the dashed line in the figure), the reference count of the node g will be out of date. If the cycle collector observes the newly added edge, the sum of the reference counts in $\{f, g, h\}$ will equal the sum of the

edges. Hence the set of nodes $\{f, g, h\}$ will be collected by the `CollectWhite` routine and form the set B_{ik}.

The addition of the edge will take place in epoch $i+1$, and so it will be processed at the end of epoch $i+1$. Since the Σ-test is applied to the reference counts in effect at the end of epoch i, the increment to node g will not be taken into account and B_{ik} will pass the Σ-test. Hence Σ-test alone cannot detect all live nodes in B_i. □

Notice that we are not claiming that the two race conditions shown in Figure 11 are an exhaustive list of all possible race conditions that our algorithm will face. But these two are sufficient to show the necessity of both the tests. Thus the two tests are both necessary and sufficient to ensure the safety of the algorithm.

Finally we prove here the following lemma that will be used in the next section, since the proof uses the notation from the present section. We define a *complete* set of nodes as one which is closed under transitive closure in the transpose graph; that is, a complete set of nodes includes all of its parents.

LEMMA 7.1.1. *If $C \subseteq \Gamma_i$ is a complete set of nodes, then C will pass both the Σ-test and the Δ-test.*

PROOF. By the stability property of garbage, there can be no changes to the reference counts of the nodes $S \in C$, since $S \in \Gamma_i$. Therefore, C passes the Δ-test.

By the same reasoning,

$$RC(S, C) = RC^{\Sigma}(S, C).$$

Since C is a complete set,

$$RC(S) = RC(S, C).$$

Therefore,

$$\begin{aligned} CRC(S) &= RC(S) - RC^{\Sigma}(S, C) \\ &= RC(S) - RC(S, C) \\ &= 0 \end{aligned}$$

Hence, C will pass the Σ-test. □

7.2 Liveness: Proof of Correctness

A garbage collector is *live* if it eventually collects all unreachable objects. Our concurrent algorithm is subject to some extremely rare race conditions, which may prevent the collection of some garbage. Therefore, we prove a weak liveness condition which holds provided that the race condition does not occur in every epoch.

We can only demonstrate weak liveness because it is possible that a candidate cycle $C \in \Omega_i$ contains a subset which is a complete garbage cycle, and some nodes in that subset point to other nodes in C which are live (see Figure 12). This is a result of our running a variant of the synchronous cycle collection algorithm while mutation continues, thus allowing race conditions such as the ones shown in Figure 11 to cause `CollectCycles` to occasionally place live nodes in a candidate set B_{ik}. The resulting candidate set B_{ik} will fail either the Σ-test or the Δ-test for that epoch. If this occurs, the cycle will be reconsidered in the following epoch. Therefore, unless the race condition occurs indefinitely, the garbage will eventually be collected.

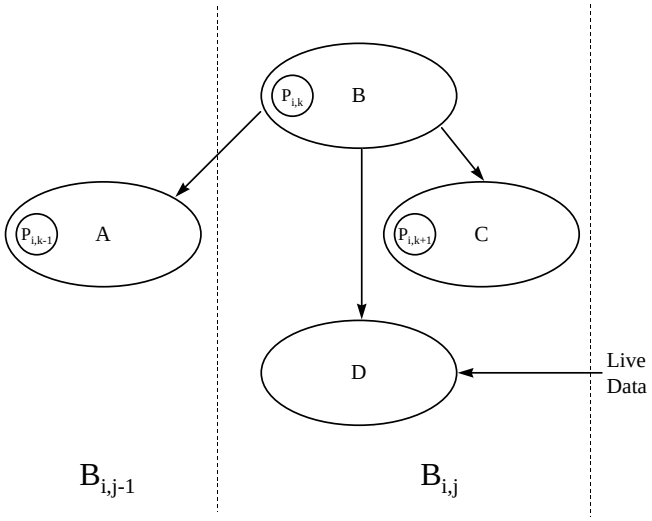

Fig. 12. Concurrent mutation of the nodes in D can cause candidate sets $B_{i,j-1}$ and $B_{i,j}$ to become undiscovered garbage.

Any garbage nodes that are not collected in epoch i are called the *undiscovered garbage* of epoch i.

In practice, we have been unable to induce the race condition that leads to undiscovered garbage, even with adversary programs. However, this remains a theoretical limitation of our approach.

We have solutions to this problem (for instance, breaking up a set that fails either of the two tests into strongly connected components), but have not included them because they complicate the algorithm and are not required in practice. We are also investigating another alternative, in which the entire cycle collection is based on performing a strongly-connected component algorithm [Bacon et al. 2001]; this alternative is also promising in that the number of passes over the object graph is substantially reduced.

In this section we will prove that if a set of garbage nodes is free from the race condition leading to undiscovered garbage, it will be collected in the epoch i; otherwise it will be considered again in the epoch $i + 1$.

We previously defined P_i as the set of purple nodes in epoch i, that is the set of nodes from which the cycle detection algorithm begins searching for cyclic garbage in `Collect-Cycles`.

THEOREM 7.2.1. *The purple set is maintained correctly by the concurrent cycle collection algorithm: every garbage node is reachable from some purple node. That is,*

$$\Gamma_i \subseteq P_i^\star.$$

PROOF. The `Decrement` procedure ensures that the only garbage in the set G_i is cyclic garbage. In addition, `Decrement` adds all nodes having decrement to non-zero to the purple set. Thus we know that any cyclic garbage generated during the processing of increments and decrements in epoch i is reachable from the purple set. What remains to be proved is that the purple set contains roots to any uncollected cyclic garbage from the previous epochs.

We know this to be trivially true for epoch 1. We assume that it is true for epoch i and prove that, after running the `CollectCycles` routine, it is true for epoch $i+1$. The result then follows by induction.

Let P be a member of the purple set in epoch i that is the root of a garbage cycle. The `CollectRoots` routine ensures that the root of each cycle generated by it is stored in the first position in the buffer and takes all white nodes that are reachable from it and unless it has been collected before (therefore reachable from another root node) puts it in the current cycle. Since `CollectCycles` is a version of the synchronous garbage collection algorithm, and there can be no concurrent mutation to a subgraph of garbage nodes, all such garbage nodes will be in the current cycle. In addition, any other uncollected purple node reachable from P and the cycle associated with will be added to the current cycle. If this latter purple node is garbage, then it will continue to be reachable from P and hence proper handling of P will ensure proper handling of this node and its children.

The `Refurbish` routine will put this first node back into the purple root set unless: a) the current cycle is determined to be garbage, in which case the entire cycle is freed or b) the first node is determined to be live, in which case it is not the root of a garbage cycle.

Hence the purple set P_{i+1} will contain the roots of all the garbage cycles that survive the cycle collect in epoch i. □

COROLLARY 7.2.1. *The cycle buffer generated in the i^{th} epoch contains all the garbage nodes present in G_i. That is,*

$$\Gamma_i \subseteq B_i \subseteq P_i^{\star}.$$

PROOF. By Theorem 7.2.1, the root of every garbage cycle is contained in P_i. The procedure `CollectCycles` is a version of the synchronous garbage collection algorithm. There can be no concurrent mutation to a subgraph of garbage nodes. Therefore, all garbage nodes will be in put into the cycle buffer B_i. □

Unfortunately, due to concurrent mutation B_i may contain some live nodes too. Let $B_{ij} \in B_i$ denote a set of nodes that were collected starting from a root node $P_{ik} \in P_i$. If all the nodes in B_{ij} are live, then it will fail one of the two tests (Δ-test or Σ-test) and its root will be identified as a live node and discarded.

It is however possible that B_{ij} contains a set of garbage nodes as well as a set of live nodes as shown in Figure 12. There are three purple nodes $P_{i,k-1}, P_{i,k}, P_{i,k+1}$. `CollectWhite` processes $P_{i,k-1}$ first and creates the candidate set $B_{i,j-1}$ which contains the nodes in A. Then `CollectWhite` processes $P_{i,k}$ and creates the candidate set $B_{i,j}$ which contains the nodes in B, C, and D. This includes both the garbage nodes in C reachable from another purple node not yet considered ($P_{i,k+1}$) and live nodes in D.

In this case B_{ij} will fail one of the two safety tests and the algorithm will fail to detect that it contained some garbage nodes. Furthermore, the algorithm will fail to detect any other garbage nodes that are pointed to by this garbage, such as $B_{i,j-1}$ in the figure. The roots $P_{i,k-1}$ and P_{ik} will be put into P_{i+1}, so the garbage cycles will be considered again in $(i+1)^{st}$ epoch, and unless a malicious mutator is able to fool `CollectCycles` again, it will be collected in that epoch. But the fact remains that it will not be collected during the current epoch.

Let U_i be the set of nodes undiscovered garbage in epoch i. That is, every member of U_i either has a live node in its cycle set or its cycle set is pointed to by a member of U_i. We will show that all the other garbage nodes (i.e. the set $\Gamma_i - U_i$) will be collected in epoch i.

LEMMA 7.2.1. *There are no edges from garbage nodes in B_{ik} to B_{il} where $k < l$.*

PROOF. The `CollectRoots` routine takes all white nodes that are reachable from the root of the current cycle, colors them orange, and places them in a set B_{il}. Any nodes that it reaches that were previously colored orange are not included in the set because they have already been included in some previous set B_{ik}. Thus all nodes that are reachable from the current root exist in the current cycle or in a cycle collected previously. Since the nodes in B_{ik} were collected before the nodes in B_{il} there can be no forward pointers from the first to the second set, unless some edges were added after the running of the `CollectRoots` routine. However, since there can be no mutation involving garbage nodes, this is not possible. □

Let K_i be the set of all nodes collected by the procedure `FreeCycles`.

THEOREM 7.2.2. *All the garbage that is not undiscovered due to race conditions will be collected in the i^{th} epoch. That is,*

$$K_i = \Gamma_i - U_i.$$

PROOF. From Corollary 7.2.1 above, we know that every node in Γ_i is contained in B_i.

If a cycle B_{ij} fails the Δ-test, then we know that it has a live node. In that case, a node in this cycle is either live, in which case it does not belong to Γ_i, or it is a garbage node that is undiscovered garbage, hence it belongs to U_i. Thus none of these nodes belong to $\Gamma_i - U_i$.

If a cycle B_{ij} fails the Σ-test, it means that there is some undeleted edge from outside the set of nodes in B_{ij}. By Lemma 7.2.1, it cannot be from a garbage node that comes earlier in the cycle buffer. If this is from a live node or from a garbage node that is undiscovered garbage, then garbage nodes in this cycle, if any, belong to the set U_i. If it is not from an undiscovered garbage node, then that garbage node belongs to a discovered garbage set later in the cycle buffer.

But in the `FreeCycles` routine we process the cycles in the reverse of the order in which they were collected. As we free garbage cycles, we delete all edges from the nodes in it. By Lemma 7.2.1, the last discovered garbage set cannot have any external pointers to it. Therefore it will pass the Σ-test also. In addition, when we delete all edges from this set, the next discovered garbage set will pass the Σ-test. Hence, every discovered garbage cycle set B_{ij} will pass both the tests. □

COROLLARY 7.2.2. *In the absence of the race condition leading to undiscovered garbage, namely a mixture of live and garbage nodes in some set B_{ik}, all garbage will be collected. That is,*

$$K_i = \Gamma_i.$$

PROOF. In this case, there are no live nodes in any B_{ik} that contains garbage and hence U_i is a null set. The result follows from Theorem 7.2.2. □

8. IMPLEMENTATION

The Recycler is implemented in Jalapeño [Alpern et al. 1999], a Java virtual machine written in Java and extended with unsafe primitives that are available only to the virtual machine. Jalapeño uses *safe points* – rather than interrupting threads with asynchronous signals, each thread periodically checks a bit in a condition register that indicates that

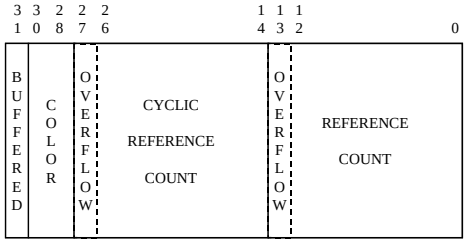

Fig. 13. Layout of the reference count word. When the overflow bit is set, excess reference count is stored in a hash table.

the runtime system wishes to gain control. This design significantly simplifies garbage collection.

Implementing the garbage collector in Java creates a number of its own problems: the memory allocator must bootstrap itself; the collector must avoid any allocation and must make sure it does not prematurely collect its own internal data structures.

All information required by the reference counting collector is stored in one extra word in the object header. We are implementing other object model optimizations that in most cases will eliminate this per-object overhead.

The Recycler is an exact collector, and makes use of the object and stack reference maps generated for use with the mark-and-sweep collector.

8.1 Memory Allocator

Since long allocation times must be treated as mutator pauses, the design of the memory allocator is crucial. The design of the allocator also strongly affects the amount of work that can be shifted to the collection processor; the more concurrent access to the allocation structures is possible, the better.

We currently use an allocator which is less than ideal for the Recycler; it was adapted from the non-copying parallel mark-and-sweep collector described in the next section. Using the terminology of Wilson et al. [1995], small objects are allocated from per-processor segregated free lists built from 16 KB pages divided into fixed-size blocks. Large objects are allocated out of 4 KB blocks with a first-fit strategy.

9. THE PARALLEL MARK-AND-SWEEP COLLECTOR

In this section we briefly describe the parallel non-copying mark-and-sweep collector with which the Recycler will be compared; more detail is available in the work of Attanasio et al. [2001].

Each processor has an associated collector thread. Collection is initiated by scheduling each collector thread to be the next dispatched thread on its processor, and commences when all processors are executing their respective collector threads (implying that all mutator threads are stopped).

The parallel collector threads start by zeroing the mark arrays for their assigned pages, and then marking all objects reachable from roots (references in global static variables and

Program	Description	Applic. Size	Threads
201.compress	Compression	18 KB	1
202.jess	Java expert system shell	11 KB	1
205.raytrace	Ray tracer	57 KB	1
209.db	Database	10 KB	1
213.javac	Java bytecode compiler	688 KB	1
222.mpegaudio	MPEG coder/decoder	120 KB	1
227.mtrt	Multithreaded ray tracer	571 KB	2
228.jack	Parser generator	131 KB	1
specjbb 1.0	TPC-C style workload	821 KB	3
jalapeño	Jalapeño compiler	1378 KB	1
ggauss	Cyclic torture test (synth.)	8 KB	1

Table II. The benchmarks used to evaluate the Recycler. The benchmarks include the complete SPEC suite and represent a wide range of application characteristics.

in mutator stacks). The Jalapeño scheduler ensures that all suspended threads are at safe points, and the Jalapeño compilers generate stack maps for the safe points identifying the location of references within stack frames. This allows the collector threads to quickly and exactly scan the stacks of the mutator threads and find the live object references.

While tracing reachable objects, multiple collector threads may attempt to concurrently mark the same object, so marking is performed with an atomic operation. A thread which succeeds in marking a reached object places a pointer to it in a local work buffer of objects to be scanned. After marking the roots, each collector thread scans the objects in its work buffer, possibly marking additional objects and generating additional work buffer entries.

In order to balance the load among the parallel collector threads, collector threads generating excessive work buffer entries put work buffers into a shared queue of work buffers. Collector threads exhausting their local work buffer request additional buffers from the shared queue of work buffers. Garbage collection is complete when all local buffers are empty and there are no buffers remaining in the shared pool.

At the end of collection the mark arrays have marked entries for blocks containing live objects, and unmarked entries for blocks available for allocation. If all blocks in a page are available, then the page is returned to the shared pool of free heap pages, and can be reassigned to another processor, possibly for a different block size.

Collector threads complete the collection process by yielding the processor, thus allowing the waiting mutator threads to be dispatched.

The design target for this collector is multiprocessor servers with large main memories. When compiled with the Jalapeño optimizing compiler, this collector was able to garbage collect a 1 GB heap with millions of live objects in under 200 milliseconds on a 12-processor PowerPC-based server. This statistic should give some indication that we are not comparing the Recycler against an easy target.

10. MEASUREMENTS

The Recycler is a fairly radical design for a garbage collector. We now present measurements showing how well various aspects of the design work.

The Recycler is optimized to minimize response time, while the mark-and-sweep collector is optimized to maximize throughput. We present measurements of both systems

Program	Obj Alloc	Obj Free	Byte Alloc	Obj Acyclic	Incs	Decs
201.compress	0.15 M	0.13 M	240 MB	76%	0.46 M	0.53 M
202.jess	17.4 M	17.2 M	686 MB	20%	55.1 M	71.6 M
205.raytrace	13.4 M	13.1 M	361 MB	90%	3.59 M	16.3 M
209.db	6.6 M	5.9 M	193 MB	10%	67.0 M	66.7 M
213.javac	16.1 M	14.1 M	195 MB	51%	41.6 M	51.8 M
222.mpegaudio	0.30 M	0.27 M	25 MB	76%	12.1 M	6.70 M
227.mtrt	14.0 M	13.5 M	381 MB	90%	4.5 M	17.3 M
228.jack	16.8 M	16.4 M	715 MB	81%	16.8 M	33.0 M
specjbb 1.0	33.3 M	33.0M	1034 MB	59%	52.4 M	84.5 M
jalapeño	19.6 M	18.4 M	676 MB	7%	62.6 M	65.6 M
ggauss	32.4 M	32.0 M	1163 MB	$< 1\%$	56.9 M	77.2 M

Table III. Benchmark allocation and mutation characteristics.

that illustrate this classical tradeoff in the context of multiprocessor garbage collection.

All tests were run on a 24 processor IBM RS/6000 Model S80 with 50 GB of RAM. Each processor is a 64-bit PowerPC RS64 III CPU running at 450 MHz with 128 KB split L1 caches and an 8 MB unified L2 cache. The machine runs the IBM AIX 4.3.2 Unix operating system.

10.1 Benchmarks

Table II summarizes the benchmarks we have used. Our benchmarks consist of the full suite of SPEC benchmarks (including SPECjbb); the Jalapeño optimizing compiler compiling itself; and `ggauss`, a synthetic benchmark designed as a "torture test" for the cycle collector: it does nothing but create cyclic garbage, using a Gaussian distribution of neighbors to create a smooth distribution of random graphs.

SPEC benchmarks were run with "size 100" for exactly two iterations, and the entire run, including time to JIT the application, was counted.

We performed two types of measurements: response time oriented and throughput oriented. Since our collector is targeting response time, most of the measurements presented are for the former category

For response time measurements, we ran the benchmarks with one more CPU than there are threads. For throughput measurements, we measured the benchmarks running on a single processor. The first scenario is typical for response time critical applications (multiprocessor workstations, soft real-time systems, etc.) The second scenario is typical of multiprogrammed multiprocessor servers.

Table II summarizes the benchmarks and shows the number of objects allocated and freed by each program; the difference is due to the fact that some objects are not collected before the virtual machine shuts down. It also shows the number of bytes requested over the course of each program's execution.

To get a broad overview of the demands each program will place on the Recycler, we show the number of increment and decrement operations performed, as well as the percentage of objects created that are acyclic according to our very simple test performed at class resolution time. These measurements confirm our basic hypotheses: the number of reference count operations per objects is usually small (between 2 and 6), so that reference counting will be efficient – the exceptions are `db` and `mpegaudio`, which perform about

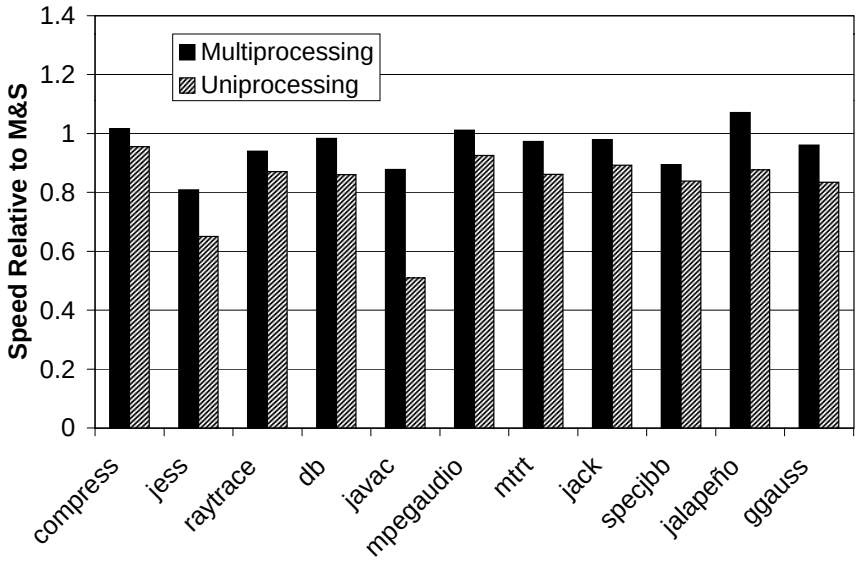

Fig. 14. Relative Speed of the Recycler compared to the Parallel Mark-and-Sweep Collector. In the multiprocessing environment the Recycler offers much lower pause times while remaining competitive with Mark-and-Sweep in end-to-end execution time.

20 and 60 mutations per object, respectively. The effect of these mutation rates will be seen in subsequent measurements.

The number of acyclic objects varies widely, indicating that the system may be vulnerable to poor performance for applications where it can not avoid checking for cycles on many candidate roots. In practice this turns out not to be a problem.

10.2 Application Performance

Our collector has a new and unusual design, and there are obvious questions about its overhead and applicability in practice. While these will be addressed below in more detail, we believe that the ultimate measure of a garbage collector is, How well does the application perform?

When we undertook this work, our goal was to develop a concurrent garbage collector that only suffered from rare pauses of under 10 milliseconds, while achieving performance within 5% of a tuned conventional garbage collector.

Figure 14 shows how well we have succeeded. It shows the speed of applications running with the Recycler relative to the speed of the same applications running with the parallel mark-and-sweep collector. The “multiprocessing” bar shows the response time oriented measurement, where an extra processor is allocated to run the collector; the “uniprocessing” bar shows the throughput oriented measurement, where the collector runs on the same

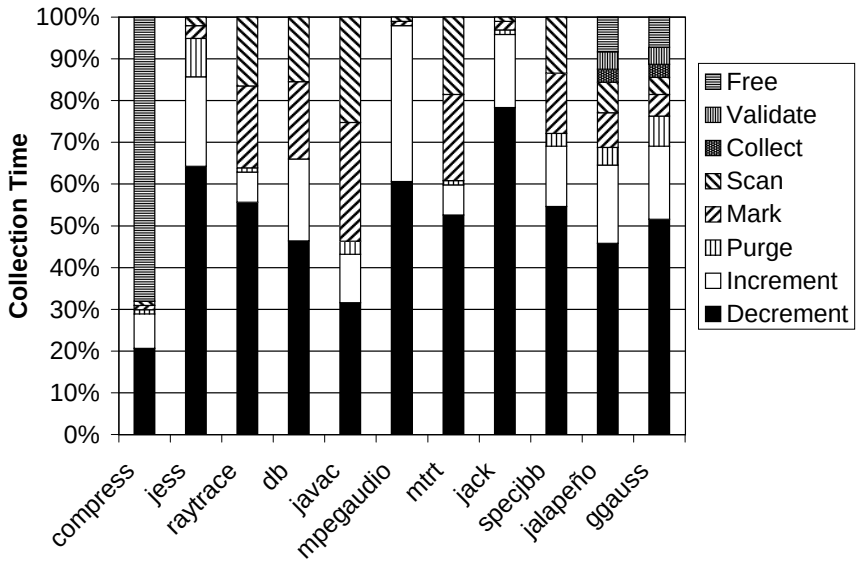

Fig. 15. Collection Time Breakdown. The time devoted to various phases varies widely depending on the characteristics of the program.

processor as the mutator(s).

For our design point, namely the multiprocessor environment, all but two of the benchmarks run within about 95% of the speed of the baseline (mark-and-sweep). The exceptions are `jess` and `javac`. For three out of eleven benchmarks, the Recycler even provides a moderate application speedup.

In a single processor environment, performance generally drops off by 5 to 10%, since the work of the collector is no longer being overlapped with the mutators. However, the performance of `jess` and `javac` is quite poor in this environment. Subsequent sections will investigate the characteristics that lead to this performance problem.

Depending on ones' point of view, the Recycler can be viewed as having successfully extracted parallelism from the application and distributed it to another processor; or as having introduced a significant overhead into the collection process.

However, we believe that there is a significant body of users who will appreciate the benefits provided by the Recycler, while being willing to pay the associated cost in extra resources. Technology trends such as chip multiprocessing (CMP) may favor this as well.

10.3 Collector Costs

Figure 15 shows the distribution of time spent on the collector CPU by the Recycler. This is work that is overlapped with the mutators in the multiprocessing case, and these measurements are from the multiprocessing runs.

Program	Concurrent Reference Counting (The Recycler)						Parallel Mark-and-Sweep			
	Ep.	Pause Time		Pause	Coll.	Elap.	GC	Max	Coll.	Elap.
		Max.	Avg.	Gap	Time	Time		Pause	Time	Time
compress	41	1.0 ms	0.5 ms	53 ms	1.3 s	238 s	7	186 ms	1.2 s	242 s
jess	93	2.2 ms	1.1 ms	120 ms	63.4 s	136 s	24	237 ms	5.2 s	110 s
raytrace	101	1.1 ms	0.7 ms	84 ms	25.2 s	99 s	9	374 ms	2.7 s	93 s
db	215	1.0 ms	0.5 ms	136 ms	73.5 s	183 s	4	414 ms	1.1 s	180 s
javac	182	2.3 ms	0.9 ms	285 ms	104.1s	147 s	12	531 ms	2.8 s	129 s
mpegaudio	21	0.7 ms	0.5 ms	36 ms	4.2 s	271 s	4	172 ms	0.7 s	274 s
mtrt	66	2.2 ms	0.6 ms	150 ms	22.9 s	74 s	10	530 ms	4.0 s	72 s
jack	153	1.3 ms	0.7 ms	122 ms	31.1 s	147 s	23	190 ms	4.1 s	144 s
specjbb	72	1.3 ms	0.5 ms	493 ms	136.7 s	(2103)	6	1127 ms	4.7 s	(2351)
jalapeño	330	2.6 ms	0.6 ms	192 ms	93.9 s	154 s	4	162 ms	0.6 s	287 s
ggauss	405	0.5 ms	0.2 ms	222 ms	99.8 s	282 s	24	171 ms	3.7 s	271 s

Table IV. Response Time. Maximum pause time is 2.7 milliseconds while the elapsed time is generally within 5% of Mark-and-Sweep. The smallest gap between pauses is 36 ms, and is usually much larger.

For most applications, the majority of time is spent processing decrements in the mutation buffers. Decrement processing includes not only adjustments to the reference count and color of the object itself, but the cost of freeing the object if its reference count drops to zero. Freeing may be recursive.

The memory allocator is largely code shared with the parallel mark-and-sweep collector, and is not necessarily optimized for the reference counting approach. Considerable speedups are probably possible in the decrement processing and freeing.

A smaller but still significant portion of the time is spent in applying mutation buffer increments. The `mpegaudio` application spend almost all of its collector time in increment and decrement processing. This is because it performs a very high rate of data structure mutation, while containing data that is almost all determined to be acyclic by the class loader.

The Purge phase removes and frees objects in the root buffer whose reference counts have dropped to zero. If the size of the root buffer is sufficiently reduced and enough memory is available, cycle collection may be deferred until another epoch. Purging is a relatively small cost, except for `jess` and `ggauss`.

The Mark and Scan phases perform approximately complementary operations and take roughly the same amount of time. The Mark phase colors objects gray starting from a candidate root, and subtracts internal reference counts. The Scan phase traverses the gray nodes and either recolors them black and restores their reference counts or else identifies them as candidate cycle elements by coloring them white.

The performance problems with `javac` are largely due to the fact that it has a large live data set which is frequently mutated, causing pointers into it to be considered as roots. These then cause the large live data set to be traversed, even though this leads to no garbage being collected: it spends over 50% of its time in Mark and Scan.

Only three benchmarks, namely `compress`, `jalapeño` and `ggauss`, actually spend a significant amount of time actually collecting cyclic garbage. The case of `compress` is particularly interesting: it uses many large buffers (roughly 1 MB in size), which are referenced by cyclic structures which eventually become garbage. While the amount of mutation and the number of objects is small, the Recycler performs all zeroing of large

Program	Buffer Space (KB)		Possible Roots (M)		
	Mutation	Root	Possible	Buff.	Roots
compress	128	131	0.4	0.03	0.01
jess	1920	1180	54.3	9.4	0.2
raytrace	416	393	3.4	42.1	0.3
db	896	131	60.8	3.8	3.8
javac	1792	524	38.5	9.1	4.5
mpegaudio	43616	131	6.4	0.07	0.01
mtrt	992	786	4.2	1.0	0.6
jack	448	131	16.6	0.9	0.2
specjbb	4832	660	51.3	6.9	2.8
jalapeño	1280	655	53.8	11.1	6.9
ggauss	1568	393	51.4	18.8	7.7

Table V. Effects of Buffering. The buffer requirements are small, and filtering significantly reduces the roots that must be considered for cycle collection (see also Figure 16).

objects (since this would otherwise be counted as a mutator pause), and this is counted as part of the Free phase. This accounts for `compress` running faster under the Recycler: we have parallelized block zeroing!

10.4 Response Time

While we have shown that the Recycler is quite competitive with the mark-and-sweep collector in end-to-end execution times, the Recycler must also meet stringent timing requirements.

Table IV provides details on both pause times and end-to-end execution times for the benchmarks running under both the Recycler and the parallel mark-and-sweep collector. The benchmarks are being run in our standard response time oriented framework: there is one more processor than there are mutator threads.

The longest measured delay was 2.6 ms for the `jalapeño` benchmark.

The longest type of delay occurs when an allocation on the first processor must fetch a new block and triggers a new epoch, which immediately causes the collector thread to run, scan the stack of the mutator threads, and switch the mutation buffers. On return from the collector, the allocator must still fetch a newly freed block of memory and format it. Therefore the maximum delay experienced by the application is usually when calling the allocator, and that delay is slightly more than the maximum epoch boundary pause.

Maximum pause time is only part of the story, however. It is also necessary that mutator pauses occur infrequently enough that the mutator can achieve useful work without constant interruptions.

Cheng and Blelloch [2001] have formalized this notion for their incremental collector as *maximum mutator utilization*, which is the fraction of time the mutator is guaranteed to be able to run within a given time quantum. This is a natural measure for a highly interleaved collector like theirs which interrupts the mutator at every allocation point, but is less relevant for our collector which normally only interrupts the mutator infrequently at epoch boundaries.

We provide a measurement of the smallest time between pauses (“Pause Gap”), which ranges from 36 ms for `mpegaudio` to almost half a second for `specjbb`. Thus for `mpegaudio`, the mutator may be interrupted for as much as 0.7 ms, but it will then run

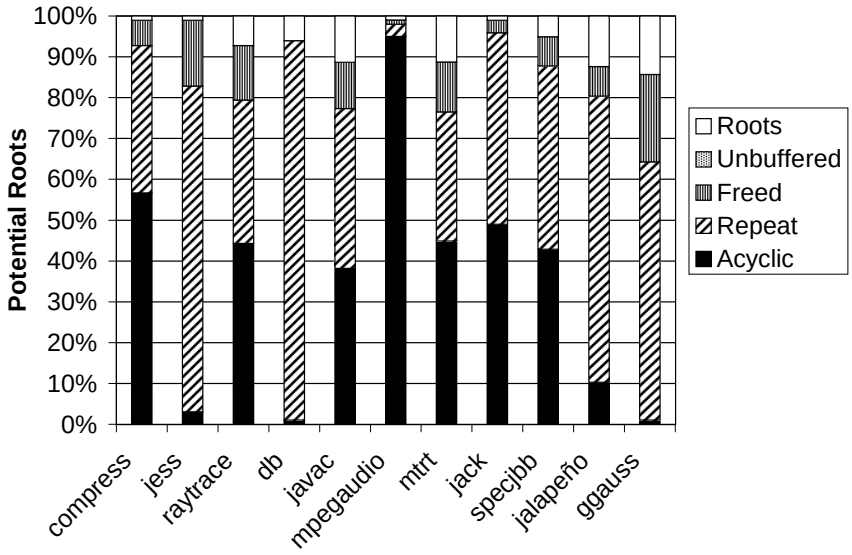

Fig. 16. Root Filtering

Program	Epochs	Roots Checked	Cycles Found		References Traced	Trace/ Alloc	M&S Traced
			Collected	Aborted			
compress	41	6,067	101	0	123,739	0.84	1,800,816
jess	93	226,707	0	0	14,870,730	0.85	8,558,011
raytrace	101	270,900	3	1	35,611,945	2.64	4,009,684
db	275	3,791,011	0	0	83,056,779	12.49	2,004,687
javac	182	4,520,382	3,874	3	168,570,902	10.45	4,550,773
mpegaudio	21	9,638	0	0	176,634	0.58	1,065,008
mtrt	66	273,109	13	0	114,054,072	0.78	4,217,820
jack	154	199,827	701	0	1,783,240	0.10	6,651,059
specjbb	72	2,786,822	0	0	96,338,266	2.98	4,081,266
jalapeño	330	6,938,814	388,945	7	50,389,369	2.57	1,463,823
ggauss	405	7,666,111	269,302	0	28,970,954	0.89	5,851,686

Table VI. Cycle Collection. Many applications check a large number of roots without finding much cyclic garbage, and race conditions leading to aborted cycles are rare. The number of references that have to be traced by the two collectors vary widely depending on the program.

Program	Heap Size	Reference Counting			Mark-and-Sweep		
		Epochs	Coll. Time	Elapsed Time	GCs	Coll. Time	Elapsed Time
compress	64 MB	46	1.3 s	247 s	7	1.1 s	236 s
jess	64 MB	116	44.1 s	166 s	19	4.2 s	108 s
raytrace	64 MB	195	15.3 s	108 s	9	2.5 s	94 s
db	64 MB	276	33.7 s	207 s	4	1.2 s	178 s
javac	64 MB	234	111.8 s	249 s	12	2.9 s	127 s
mpegaudio	64 MB	31	4.8 s	296 s	3	0.5 s	274 s
mtrt	64 MB	157	17.2 s	115 s	10	3.6 s	99 s
jack	64 MB	191	21.7 s	158 s	20	3.7 s	141 s
specjbb	72 MB	85	25.2 s	(705)	5	1.9 s	(841)
jalapeño	256 MB	289	48.3 s	186 s	4	1.1 s	163
ggauss	40 MB	516	65.1 s	327 s	40	6.2 s	273 s

Table VII. Throughput. Unlike the previous tables, the programs are run on a single processor. Even on a single processor the throughput of the Recycler is reasonable for most applications.

for *at least* 36 ms.

Interestingly, the programs with shorter pause gaps also seem to have shorter maximum pause times. As a result, the mutator utilization remains good across a spectrum of applications.

Although the Recycler spends far more time performing collection than the mark-and-sweep collector, this collection time is almost completely overlapped with the mutators. We are investigating ways to reduce this overhead, including both algorithmic [Bacon et al. 2001] and implementation improvements.

The `specjbb` benchmark performs a variable amount of work for a given time period, so its throughput scores are shown in parentheses.

Our maximum pause time of 2.6 ms is two orders of magnitude shorter than that reported by Doligez and Leroy [1993] and by Nettles and O'Toole [1993], both for a concurrent dialect of ML. While processor speeds have increased significantly in the last seven years, memory systems have progressed far less rapidly. We believe our system represents a substantial increase in real performance; however, only "head-to-head" implementations will tell. Our work is a beginning in this direction.

10.5 Buffering

The Recycler makes use of five kinds of buffers of object references: mutation buffers, stack buffers, root buffers, cycle buffers, and mark stacks. The four buffer types have been described in the algorithm section; mark stacks are used to express the implicit recursion of the marking procedures explicitly, thereby avoiding procedure calls and extra space overhead.

All five types of buffers consumes memory, and it is clearly undesirable for the garbage collector to consume memory. In practice, only the mutation and root buffers turn out to be of significant size. The thread stacks never have more than a few hundred object references, so the stack buffers are of negligible size (although they could become a factor on a system with large numbers of threads, or for applications which are deeply recursive).

Table V shows the instantaneous maximum buffer space utilization ("high water mark") for both mutation buffers and root buffers. Mutation buffer consumption is reasonable,

with the exception of `mpegaudio`, which uses 43 MB (!) of mutation buffer space. This is a direct result of the very high per-object mutation rate reflected in the measurements in Table II, showing that `mpegaudio` performs about 60 mutations per allocated object.

We are implementing some preprocessing strategies which should reduce the buffer consumption by about a factor of 2. We also have not yet tuned the feedback algorithm between the mutators and the collector, which should further reduce buffer consumption. In particular, we hope to take advantage of Jalapeño's dynamic profiling, feedback, and optimization system [Arnold et al. 2000] to improve space consumption for programs like `mpegaudio`.

Table V also shows the effectiveness of our strategies for reducing the number of objects that must be traced by the cycle collector. Every decrement that does not actually free an object potentially leaves behind cyclic garbage, and must therefore be traced. The number of such decrements is shown ("Possible"), as well as the number that are actually placed in the buffer ("Buffered"), and the number that are left in the buffer after purging ("Roots"). Purging checks for objects that have been modified while the collector waits to process the buffer; objects whose reference count has been incremented are live and can be removed from consideration as roots, and objects whose reference count has been decremented to zero are garbage and can be freed.

While the number of possible candidate roots is high (as many as 60 million for `db`), the combination of the various filtering strategies is highly effective, reducing the number of possible roots by at least a factor of seven. Only `ggauss`, our synthetic cycle generator, requires a large fraction of roots to be buffered.

While filtering is highly successful, Figure 16 shows that no one technique is responsible for its success. On average, about 40% of possible roots are excluded from consideration because the are acyclic, while another 30% are eliminated because they are already in the root buffer ("Repeat"). The balance between these two factors varies considerably between applications, but on balance the two filtering techniques remove about 70% of all candidate roots before they are ever put in the root buffer.

Another 10% or so are freed during root buffer purging, because a concurrent mutator has decremented the reference count of the object to zero while it was in the buffer. Surprisingly, the number of objects in the buffer whose reference count is incremented, allowing them to be removed ("Unbuffered") is very small and often zero.

Finally, between 1 and 15% of the possible roots are left for the cycle collection algorithm to traverse, looking for garbage cycles. Thus the filtering techniques are a key component of making the cycle collection algorithm viable in practice.

10.6 Cycle Collection

Table VI summarizes the operation of the concurrent cycle collection algorithm. There were a number of surprising results. First of all, despite the large number of roots considered, the number of garbage cycles found was usually quite low. Cyclic garbage was significant in `jalapeño` and our torture test, `ggauss`. It was also significant in `compress`, although the numbers do not show it: multi-megabyte buffers hang from cyclic data structures in `compress`, so the application runs out of memory if those 101 cycles are not collected in a timely manner.

Note that `javac`, which spends over 50% of its garbage collection time searching for cyclic garbage to collect, actually collects less than 4,000 cycles. This explains `javac`'s poor performance in the single processor environment.

The number of cycles aborted due to concurrent mutation was smaller than we expected, but these invalidations only come into play when race conditions fool the cycle detection algorithm.

Finally, Table VI compares the number of references that must be followed by the concurrent reference counting ("Refs. Traced") and the parallel mark-and-sweep ("M&S Traced") collectors. The reference counting collector has an advantage in that it only traces locally from potential roots, but has a disadvantage in that the algorithm requires three passes over the subgraph. Furthermore, if the root of a large data structure is entered into the root buffer frequently and high mutation rates force frequent epoch boundaries, the same live data structure might be traversed multiple times.

In this category, there is no clear winner. Each type of garbage collection sometimes performs one to two orders of magnitude more tracing than the other. To calibrate the amount of tracing performed, "Trace/Alloc" shows the number of references traced per allocated object for the reference counting collector.

10.7 Throughput

In the previous section we measured our collectors in an environment suited to response time; we now measure them in an environment suited to throughput. Table VII shows the results of running our benchmarks on a single processor. The mark-and-sweep collector suffers somewhat since it is no longer performing collection in parallel.

However, in this environment, the lower overhead of the mark-and-sweep collector dominates the equation, and it outperforms the the Recycler, sometimes by a significant margin.

Of course, the Recycler is not designed to run in a single-threaded environment; nevertheless, it provides a basis for comparing the inherent overhead of the two approaches in terms of overall work performed.

11. RELATED WORK

While numerous concurrent, multiprocessor garbage collectors for general-purpose programming languages have been described in the literature [Dijkstra et al. 1976; Huelsbergen and Winterbottom 1999; Kung and Song 1977; Lamport 1976; Lins 1992b; Rovner 1985; Steele 1975], the number that have been implemented is quite small and there is limited only some implementations have actually been run on multiprocessors [Appel et al. 1988; Cheng and Blelloch 2001; DeTreville 1990; Huelsbergen and Larus 1993; Doligez and Leroy 1993; Domani et al. 2000; Nettles and O'Toole 1993].

The work of DeTreville [1990] on garbage collectors for Modula-2+ on the DEC Firefly workstation is the only comparative evaluation of multiprocessor garbage collection techniques. His algorithm is based on the reference counting collector of Rovner [1985] backed by a concurrent tracing collector for cyclic garbage. Unfortunately, despite having implemented a great variety of collectors, he only provides a qualitative comparison. Nevertheless, our findings agree with DeTreville's in that he found reference counting to be highly effective for a general-purpose programming language on a multiprocessor.

The Recycler differs in its use of cycle collection instead of a backup mark-and-sweep collector. The Recycler also uses atomic exchange operations when updating heap pointers to avoid race conditions leading to lost reference count updates; DeTreville's implementation required the user to avoid race conditions and was therefore unsafe.

Huelsbergen and Winterbottom [1999] describe a concurrent algorithm (VCGC) that is used in the Inferno system to back up a reference counting collector. They report that ref-

erence counting collects 98% of data; our measurements for Java show that the proportion of cyclic garbage is often small but varies greatly. The only measurements provided for VCGC were on a uniprocessor for SML/NJ, so it is difficult to make meaningful comparisons.

The only other concurrent, multiprocessor collector for Java that we know of is the work of Domani et al. [2000]. This is a generational collector based on the work of Doligez et al. [Doligez and Leroy 1993; Doligez and Gonthier 1994], for which generations were shown to sometimes provide significant improvements in throughput. In a subsequent paper [Domani et al. 2000], they provided limited response time data in the form of user-level measurements for a single benchmark, `pBOB`, which is the predecessor of `SPECjbb`. They reported pause times of 30–40 milliseconds for `pBOB` run with three mutator threads. However, these numbers are not directly comparable with ours due to both differences in underlying hardware and because their measurements were not performed directly in the virtual machine.

The other implemented concurrent multiprocessor collectors [Appel et al. 1988; Cheng and Blelloch 2001; Huelsbergen and Larus 1993; Doligez and Leroy 1993; Nettles and O'Toole 1993] are all tracing-based algorithms for concurrent variants of ML, and with the exception of Cheng and Blelloch's collector have significantly longer maximum pause times than our collector. Furthermore, ML produces large amounts of immutable data, which simplifies the collection process.

The collector of Huelsbergen and Larus [1993] for ML achieved maximum pause times of 20 ms in 1993, but only for two small benchmarks (Quicksort and Knuth-Bendix). Their collector requires a read barrier for mutable objects that relies on processor consistency to avoid locking objects while they are being forwarded. Read barriers, even without synchronization instructions, are generally considered impractical for imperative languages [Jones and Lins 1996], and on weakly ordered multiprocessors their barrier would require synchronization on every access to a mutable object, so it is not clear that the algorithm is practical either for imperative languages or for the current generation of multiprocessor machines.

Cheng and Blelloch [2001] describe a parallel, concurrent, and incremental collector for SML. They take a much different approach, essentially trying to solve the problem of making a compacting garbage collector meet stringent time bounds. Their approach requires such overheads as duplicating mutable fields, which we did not consider acceptable in our collector. On the other hand, their collector is scalable while ours is not.

11.1 Reference Counting

The Recycler shares with Deutsch and Bobrow's Deferred Reference Counting algorithm [Deutsch and Bobrow 1976] the observation that reference counting stack assignments is prohibitive, and that periodic scanning of the stack can be used to avoid direct counting of stack references. The principal difference is the manner in which the stack references are handled. Deferred Reference Counting breaks the invariant that zero-count objects are garbage, and requires the maintenance of a Zero Count Table (ZCT) which is reconciled against the scanned stack references. The ZCT adds overhead to the collection, because it must be scanned to find garbage.

The Recycler defers counting by processing all decrements one epoch behind increments, and by its use of stack buffers. The result is a simpler algorithm without the additional storage or scanning required by the ZCT, albeit at the expense of additional buffer

space.

In recent work, Levanoni and Petrank [2001] have explored how a reference counting system can limit the number of increments and decrements that must be applied for each memory location in each epoch. They make use of the insight that the intermediate values of a pointer cell do not really matter; for the purposes of garbage collection the only relevant values are what the cell points to at the beginning of the epoch and what it points to at the end of the epoch.

Plakal and Fischer [2000] have explored how program slicing can be used to automatically move reference counting operations into a concurrent thread on a simultaneous multithreading (SMT) architecture.

11.2 Cycle Collection

As described in Section 5, our cycle collection algorithm is derived from the synchronous algorithm devised by Martínez et al. [1990] and extended by Lins to lazily scan for cyclic garbage [Lins 1992a; Jones and Lins 1996]. Our synchronous variant differs in a number of important respects: its complexity is linear rather than quadratic; it avoids placing a root in the root buffer more than once per epoch; and it greatly reduces overhead by not considering inherently acyclic structures.

Cannarozzi et al. [2000] have presented a very different approach to garbage collection which handles the cycle collection problem by dynamically maintaining sets of objects that can point to each other using the union-find algorithm. A set is collected when the allocating stack frame of its oldest member is popped. While this technique suffers from certain drawbacks, notably an inability to handle multi-threaded programs and potential storage leaks into the base stack frame of the program, it is nevertheless very intriguing because it eliminates the need for tracing entirely.

Lins [1992b] has presented a concurrent cycle collection algorithm based on his synchronous algorithm. Unlike the Recycler, Lins does not use a separate reference count for the cycle collector; instead he relies on processor-supported asymmetric locking primitives to prevent concurrent mutation to the graph. His scheme has, to our knowledge, never been implemented. It does not appear to be practical on stock multiprocessor hardware because of the fine-grained locking required between the mutators and the collector. Our algorithm avoids such fine-grained locking by using a second reference count field when searching for cycles, and performing safety tests (the Σ-test and the Δ-test) to validate the cycles found.

Jones and Lins [1993] present an algorithm for garbage collection in distributed systems that uses a variant of the lazy mark-scan algorithm for handling cycles. However, they rely on much more heavy-weight synchronization (associated with message sends and receives and global termination detection) than our algorithm. The algorithm has never been implemented.

In terms of cycle collection systems that have been implemented, the closest to our work is that of Rodrigues and Jones [1998], who have implemented an algorithm for cycle collection in distributed systems. They use a tracing collector for local cycles and assume that inter-processor cycles are rare, and they use considerably more heavy-weight mechanisms (such as lists of back-pointers) than we do; on the other hand they also solve some problems that we do not address, like fault tolerance.

The Recycler's concurrent cycle collector could in the worst case require space proportional to the number of objects (if it finds a cycle consisting of all allocated objects). This

is not directly comparable to concurrent tracing collectors, which push modified pointers onto a stack that must be processed before the algorithm completes. Since the same pointer can be pushed multiple times, the worst case complexity appears as bad or worse than the Recycler's. In practice, each algorithm requires a moderate amount of buffer memory.

12. CONCLUSIONS

We have presented novel algorithms for concurrent reference counting and cycle detection in a garbage collected system, provided proofs of correctness, and have demonstrated the efficacy of these techniques in the Recycler, a garbage collector implemented as part of the Jalapeño Java virtual machine.

Our work is novel in two important respects: it represents the first practical use of cycle collection in a reference counting garbage collector for a mainstream programming language; and it requires no explicit synchronization between the mutator threads or between the mutators and the collector.

Another contribution of our work is our proof methodology, which allows us to reason about an abstract graph that never exists in the machine, but is implied by the stream of increment and decrement operations processed by the collector. In effect we are able to reason about a consistent snapshot without ever having to take such a snapshot in the implementation.

Over a set of eleven benchmark programs including the full SPEC benchmark suite, the Recycler achieves maximum measured application pause times of 2.6 milliseconds, about two orders of magnitude shorter than the best previously published results.

We have measured the Recycler against an highly tuned non-concurrent but parallel mark-and-sweep garbage collector. When resources are scarce, the throughput-oriented design of the mark-and-sweep collector yields superior execution times. But with an extra processor and some extra memory headroom, the Recycler runs without ever blocking the mutators, and achieves maximum pauses that are about 100 times shorter without sacrificing end-to-end execution time.

The Recycler uses a novel concurrent algorithm for detecting cyclic garbage, and is the first demonstration of a purely reference counted garbage collector for a mainstream programming language. It is competitive with the best concurrent tracing-based collectors.

We believe these quantitative reductions will create a qualitative change in the way garbage collected languages are perceived, programmed, and employed.

ACKNOWLEDGMENTS

We thank Jong Choi, Anthony Cocchi, Alex Dupuy, David Grove, Wilson Hsieh, Chet Murthy, Rob Strom, and Mark Wegman for providing helpful feedback at various stages of this project. Mike Burke and Mark Mergen provided moral support. David Grove also provided invaluable help with the optimizer.

The anonymous referees from PLDI and ECOOP provided very detailed and thoughtful feedback which has greatly improved the quality of the paper.

REFERENCES

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPERD, J. C., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices, 34,* 10, 314–324.

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

ARNOLD, M., FINK, S., GROVE, D., M.HIND, AND SWEENEY, P. 2000. Adaptive optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications. SIGPLAN Notices, 35,* 10, 47–65.

ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. 2001. A comparative evaluation of parallel garbage collectors. In *Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science. Springer-Verlag, Cumberland Falls, Kentucky.

BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. See PLDI [2001].

BACON, D. F., KOLODNER, H., NATHANIEL, R., PETRANK, E., AND RAJAN, V. T. 2001. Strongly-connected component algorithms for concurrent cycle collection. Tech. rep., IBM T.J. Watson Research Center and IBM Haifa Scientific Center. Apr.

BACON, D. F. AND RAJAN, V. T. 2001. Concurrent cycle collection in reference counted systems. In *Proceedings of the* 15^{th} *European Conference on Object-Oriented Programming*, J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Budapest, Hungary.

BOBROW, D. G. 1980. Managing re-entrant structures using reference counts. *ACM Trans. Program. Lang. Syst. 2,* 3 (July), 269–273.

CANNAROZZI, D. J., PLEZBERT, M. P., AND CYTRON, R. K. 2000. Contaminated garbage collection. See PLDI [2000], 264–273.

CHENG, P. AND BLELLOCH, G. 2001. A parallel, real-time garbage collector. See PLDI [2001], 125–136.

CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proceedings of the Conference on Programming Language Design and Implementation. SIGPLAN Notices, 33,* 6, 162–173.

CHRISTOPHER, T. W. 1984. Reference count garbage collection. *Software – Practice and Experience 14,* 6 (June), 503–507.

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Commun. ACM 3,* 12 (Dec.), 655–657.

DETREVILLE, J. 1990. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, DEC Systems Research Center, Palo Alto, California. Aug.

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Commun. ACM 19,* 7 (July), 522–526.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1976. On-the-fly garbage collection: An exercise in cooperation. In *Hierarchies and Interfaces*, F. L. Bauer et al., Eds. Lecture Notes in Computer Science, vol. 46. Springer-Verlag, New York, 43–56.

DOLIGEZ, D. AND GONTHIER, G. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the* 21^{st} *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, New York, Portland, Oregon, 70–83.

DOLIGEZ, D. AND LEROY, X. 1993. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the* 20^{th} *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, New York, Charleston, South Carolina, 113–123.

DOMANI, T., KOLODNER, E. K., LEWIS, E., SALANT, E. E., BARABASH, K., LAHAN, I., LEVANONI, Y., PETRANK, E., AND YANOVER, I. 2000. Implementing an on-the-fly garbage collector for Java. See ISMM [2000], 155–166.

DOMANI, T., KOLODNER, E. K., AND PETRANK, E. 2000. A generational on-the-fly garbage collector for Java. See PLDI [2000], 274–284.

HUELSBERGEN, L. AND LARUS, J. R. 1993. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the* 4^{th} *ACM Symposium on Principles and Practice of Parallel Programming. SIGPLAN Notices, 28,* 7 (July), 73–82.

HUELSBERGEN, L. AND WINTERBOTTOM, P. 1999. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management. SIGPLAN Notices, 34,* 3, 166–174.

ISMM 2000. *Proceedings of the ACM SIGPLAN International Symposium on Memory Management. SIGPLAN Notices, 36,* 1 (Jan., 2001). Minneapolis, MN.

JONES, R. E. AND LINS, R. D. 1993. Cyclic weighted reference counting without delay. In *PARLE'93 Parallel Architectures and Languages Europe*, A. Bode, M. Reeve, and G. Wolf, Eds. Lecture Notes in Computer Science, vol. 694. Springer-Verlag, 712–715.

JONES, R. E. AND LINS, R. D. 1996. *Garbage Collection*. John Wiley and Sons.

KUNG, H. T. AND SONG, S. W. 1977. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*. IEEE Press, New York, New York, 120–131.

LAMPORT, L. 1976. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*. 50–54.

LEVANONI, Y. AND PETRANK, E. 2001. An on-the-fly reference counting garbage collector for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. *SIGPLAN Notices, 36*, 10.

LINS, R. D. 1992a. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett. 44*, 4 (Dec.), 215–220.

LINS, R. D. 1992b. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming 35*, 1–5 (Sept.), 563–568. *Proceedings of the* 18^{th} *EUROMICRO Conference* (Paris, France).

MARTÍNEZ, A. D., WACHENCHAUZER, R., AND LINS, R. D. 1990. Cyclic reference counting with local mark-scan. *Inf. Process. Lett. 34*, 1, 31–35.

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM 3*, 184–195.

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

PLAKAL, M. AND FISCHER, C. N. 2000. Concurrent garbage collection using program slices on multithreaded processors. See ISMM [2000], 94–100.

PLDI 2000. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. *SIGPLAN Notices, 35*, 6. Vancouver, British Columbia.

PLDI 2001. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. *SIGPLAN Notices, 36*, 5 (May). Snowbird, Utah.

RODRIGUES, H. C. C. D. AND JONES, R. E. 1998. Cyclic distributed garbage collection with group merger. In *Proceedings of the* 12^{th} *European Conference on Object-Oriented Programming*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, Brussels, 249–273.

ROVNER, P. 1985. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Tech. Rep. CSL–84–7, Xerox Palo Alto Research Center. July.

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

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, H. Baker, Ed. Lecture Notes in Computer Science, vol. 986. Springer Verlag, Kinross, Scotland.

Submitted August 2001