Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June, 2001 (SIGPLAN Not. 36,5)

Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector

David F. Bacon   Clement R. Attanasio   Han B. Lee[*]   V.T. Rajan   Stephen Smith

IBM T.J. Watson Research Center

ABSTRACT

The deployment of Java as a concurrent programming language has created a critical need for high-performance, concurrent, and incremental multiprocessor garbage collection. We present the *Recycler*, a fully concurrent pure reference counting garbage collector that we have implemented in the Jalapeño Java virtual machine running on shared memory multiprocessors.

While a variety of multiprocessor collectors have been proposed and some have been implemented, experimental data is limited and there is little quantitative basis for comparison between different algorithms. We present measurements of the Recycler and compare it 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 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%.

1. INTRODUCTION

In this paper we present a new multiprocessor garbage collector 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*.

[*]Work done at IBM. Current address: Department of Computer Science, University of Colorado, Boulder, CO 80309.

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 [13] but maintains the invariant that objects whose reference count drops to zero can be collected, and therefore avoids the need for ancillary tables.

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 [23]. Our algorithm reduces asymptotic complexity from $O(n^2)$ to $O(n)$, and also significantly reduces the constant factors.

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 [1], 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).

In concurrently published work, we provide detailed pseudocode for the cycle collection algorithm and a proof of correctness based on an abstract graph induced by the stream of increment and decrement operations [5]. This paper concentrates on describing the system as a whole, and on the comparative measurements.

The rest of this paper is organized as follows: Section 2 presents our algorithm for concurrent reference counting. Section 3 presents the synchronous algorithm for collecting cyclic garbage; Section 4 extends this algorithm to handle concurrent mutators. Section 5 describes the implementation, and Section 6 describes the parallel mark-and-sweep collector. Section 7 presents measurements of the running system and a comparison between the two garbage collectors. Section 8 describes related work and is followed by our conclusions.

2. 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 collector shares some characteristics with the Deutsch-Bobrow algorithm and its descendants [13, 29, 12], as discussed in Section 8.

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

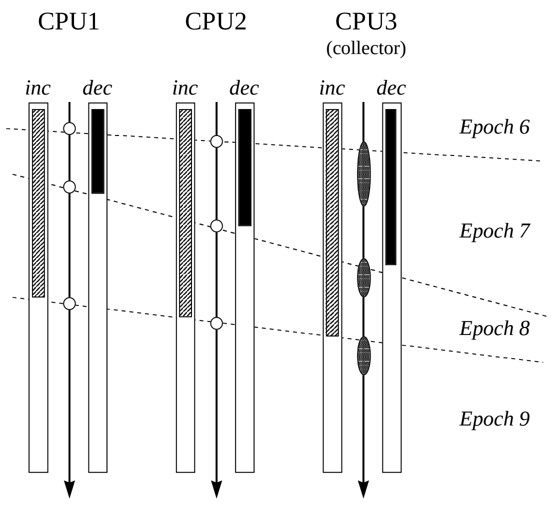

Figure 1: The Concurrent Reference Counting Collector. Arrows represent the execution of the CPUs; bubbles are interruptions by the collector. Increment and decrement operations are accumulated by each CPU into a buffer. At the end of epoch 8, the collector, running on CPU 3, will process all increments through epoch 8 and all decrements through epoch 7.

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 1. During mutator operation, updates to the stacks are not reference-counted. Only heap updates are reference-counted, and those operations are deferred with a write barrier 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.

2.1 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. When the collector runs on a processor, instead of scanning the stacks of all threads, it only scans the stacks of active threads.

When buffer processing occurs on the last processor, the collector iterates over all threads, and if the thread was active in the current epoch, it processes the stack buffer and increments each object it refers to. If the thread was inactive, it does not perform any increments; instead, the stack buffer of the previous epoch is *promoted* and becomes the stack buffer of the current epoch.

The collector then scans the mutation buffer of each processor for the current epoch, and performs the increment operations.

Then the collector iterates over all of the threads again, and if a thread has a stack buffer for the previous epoch, the objects referred to are decremented. Note that if the thread was idle, its stack buffer of the previous epoch will have been promoted in the increment phase, and no decrements will be performed for the idle thread.

Finally, the decrements of the mutation buffers of the previous epoch are performed, and the collection completes as before.

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 [9]; so far we have not implemented this optimization since our benchmarks are not deeply recursive.

2.2 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 col-

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 1: Object Colorings for Cycle Collection. Orange and Red are only used by the concurrent cycle collector.

lector 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 3 and 4 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 [28].

3. SYNCHRONOUS CYCLE COLLECTION

Since the early 1960's when both mark-and-sweep [26] and reference counting [11] were first proposed for automatic garbage collection, a deficiency of many collectors based on reference counting has been their inability to collect cyclic garbage. Solutions to this problem have ranged from placing the responsibility for breaking cycles on the programmer, to providing special programming abstractions [6] to using an infrequent mark-and-sweep collector as a backup to the reference counting collector [12].

In passing, it should be noted that cycles can be problematic for tracing collectors as well. Cyclic garbage greatly increases the false retention effects in conservative collectors, sometimes to unacceptable levels [7]. Cycles can also disturb generational collectors by causing large amounts of dead data to be moved into the "old" generation. Finally, cycles can cause poor performance for the train algorithm by requiring cars to be moved multiple times.

We now expand the algorithm of the previous section to handle cyclic garbage. Following our philosophy of using a pure reference-counting approach, rather than a hybrid of reference-counting and tracing, we find cyclic garbage by performing localized cycle detection.

In this section we describe a synchronous "stop the world" cycle collector so that the concerns raised by concurrent mutator activity can be factored out. In Section 4 we extend the algorithm to handle

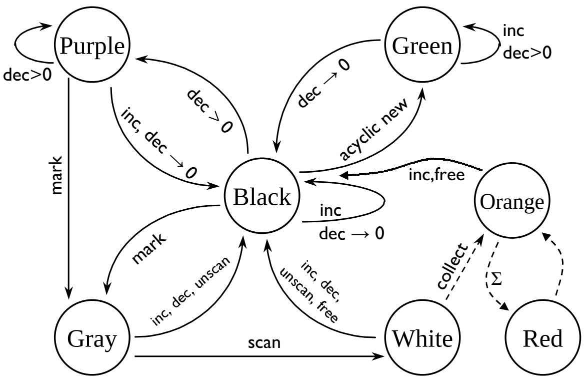

Figure 2: State transition graph for cycle collection.

concurrent mutation.

First of all, we observe that some objects are inherently acyclic, and 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.

Some classes can be statically determined to be acyclic: those that contain only scalars and references to final acyclic classes (that is, classes that are acyclic and may not be subclassed), and arrays of final acyclic classes. In Java, an important special case of the latter group are arrays of scalars.

In the Recycler, part of the reference counting field in each object is reserved for cycle collection, which uses a coloring scheme as detailed in Table 1. An object is colored green at object creation time if the class of the object is statically determined to be acyclic.

Note that for static compilation, the class graph could be analyzed to determine which classes are acyclic. However, in the presence of dynamic class loading our more restrictive formulation must be used, since an acyclic class could later be subclassed with a cyclic class.

For those objects that are potentially cyclic, we use a technique first described by Christopher [10] in which garbage cycles are identified by tentatively subtracting internal references, and observing whether the resulting structure has regions with zero reference count.

Our algorithm is based variant of the coloring algorithm proposed by Martínez et al [25] and extended by Lins [23]. An excellent description of the latter algorithm is contained in the book by Jones and Lins [20]; we will briefly describe our algorithm and highlight the differences with Lins' algorithm.

The basic approach is based on the fact that a live cycle must contain at least one object with a reference count of two or more. Therefore, whenever a reference count is decremented to a non-zero value, we record the pointer in a *root buffer* and color the object purple, meaning that it is a potential root of a garbage cycle. We also set a *buffered* flag in the object to ensure that we do not record the pointer in the root buffer more than once. The state transition graph for our algorithm is shown in Figure 2.

Optimistically, we hope that eventually the potential root will either become garbage (by its reference count dropping to zero) or will become linked to some other live structure, causing its reference count to increase again, at which point we re-color it black. In either of these cases, we know that the object is not part of a

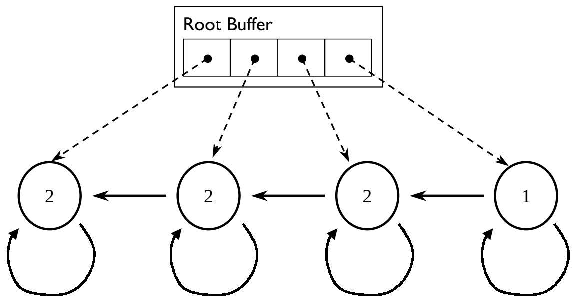

Figure 3: Example of compound cycle that causes Lins' algorithm to exhibit quadratic complexity.

garbage cycle.

Periodically, we process the root buffer. The data structures rooted by those objects that are still purple are traversed, and the reference counts due to internal pointers are subtracted, with the objects traversed marked gray indicating that they are candidates for garbage cycle removal (inherently acyclic – green – objects are not marked or traversed). This is the *marking* phase of the algorithm.

A garbage cycle will have only internal pointers, and therefore subtracting the counts due to internal pointers will cause the reference counts in the cycle to drop to zero. A second traversal starting from the once-purple roots finds such objects and colors them white. This is the *scanning* phase of the algorithm. Gray objects whose reference counts are non-zero are colored black, along with all objects reachable from them. In the process, the reference counts are restored, in a process called *unscanning* (or "scan black" in Jones and Lins' book).

Finally, the white objects are swept into the free list, the reference counts of green objects they refer to are decremented, and the root buffer is cleared. This is the *collection* phase of the algorithm.

Lins' algorithm performs the mark, scan, and collect phases together for each candidate root in turn. Unfortunately, this makes the algorithm O(n^2) in the worst case, as for example in the cycle shown in Figure 3. His algorithm will perform a complete scan from each of the candidate roots until it arrives at the final root, at which point the cycle will be collected. Lins' algorithm also does not use a buffered flag, and may therefore consider the same root multiple times.

Our algorithm performs each phase in its entirety for all of the roots, and is therefore linear in the size of the object graph – $O(N+E)$ – since the mark, scan, and collect phases each traverse at worst each object and each pointer. In the worst case our algorithm would traverse the entire heap 3 times. In practice, we observe far less memory traversal, since we eliminate consideration of green nodes, and many candidate roots become garbage before they are scanned.

4. CONCURRENT CYCLE COLLECTION

In this section we briefly describe the concurrent cycle collector. A fuller description of the details of the algorithm, including detailed pseudo-code and a formal proof of correctness, is presented in concurrently published work [5].

The concurrent cycle collection algorithm is somewhat 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 while it is being scanned by the collector. In addition the reference counts may be as much as two epochs out of date.

Our algorithm relies on the *stability property* of garbage: once an object becomes garbage, it can not cease being garbage. The basic approach is to use the sequential algorithm of the previous section to find what appear to be garbage cycles. We then perform two validation tests to ensure that the garbage cycles were not detected erroneously due to concurrent mutator activity.

An important characteristic of our algorithm is that the validation tests are relatively simple and are independent of the algorithm used to detect the candidate cycles. This greatly simplifies the proof of correctness.

Since we can not rely on being able to re-trace the same graph in order to restore reference counts that have been subtracted due to internal pointers, we instead maintain two reference counts for each object: one is the *true* reference count (usually just called the RC), and the other is the *cyclic reference count* (or CRC).

Both counts, the color, and the buffered flag are stored in a single 32-bit word in the object header. The RC and CRC are each 12 bits plus an overflow bit. When the overflow bit is set, the excess count is stored in a hash table. In practice this hash table never contains more than a few entries.

The algorithm proceeds in a similar manner to the synchronous algorithm, except that when an object is marked gray its cyclic reference count is initialized to its true reference count, and henceforward the algorithm operates only on the cyclic reference count, leaving the true reference count unchanged. However, in the *collect* phase instead of marking white objects black and freeing them, we instead mark them orange (as shown in the transition graph in Figure 2) and place them in a *cycle buffer*. Different cycles are delineated by nulls.

4.1 Safety Tests

So far we have simply used a synchronous cycle collection algorithm in a concurrent setting, without any synchronization between the mutators and the collector. Therefore, some of the cycles found may not really be garbage. To prevent collection of live data, we perform two validation steps: one to verify the number of external references into the cycle; and a second, to check for concurrent addition of references into the cycle which would make it live.

The Σ-test checks for external references to a cycle, and proceeds as follows: for each cycle, set the cyclic reference count (CRC) of each object to its true reference count (RC). Then follow the pointers from each of the objects in the cycle, subtracting one from the CRC of each reached object. Finally, take the sum of the CRC's of each object in the cycle. This sum is the total number of external references into the cycle. If it is zero, then the cycle is garbage provided that no additional edges were concurrently added into the cycle.

An important feature of the Σ-test is that it operates on a fixed set of nodes; it does not rely on following pointers within the objects to elaborate the set, since those pointers are subject to concurrent mutation. This is the key insight of the Σ-test.

The Δ-test runs after the next epoch and checks for concurrent modification to the cycle. It scans the objects in each cycle and checks whether they are still orange (if their reference count changed, they would have been recolored). If all objects in a cycle are still orange, and it has passed the external reference test, then the cycle is garbage and is collected.

Note that it is not necessary to wait to observe the effect of concurrent decrement operations, since they only reduce the external reference count of the cycle.

4.2 Liveness

So far we have described an algorithm that is safe – it does not collect reachable data, by virtue of the two validation tests. However, we must also demonstrate liveness – that all garbage is collected.

Acyclic garbage is handled by the basic reference counting technique of Section 2. Roots of cyclic garbage are entered into the root buffer by the process described in the previous section for the sequential algorithm. The concurrent variant of the sequential algorithm then searches for dead cycles from those roots.

If there is no concurrent mutation, the synchronous algorithm will find garbage due to the stability property of garbage.

If a cycle is identified as garbage but fails either of the validation tests, then its root (the first object in the buffer) and any members that have been colored purple due to decrements are entered into the root buffer and reconsidered during the next cycle collection. This ensures that any cycle abandoned due to concurrent mutation is correctly reconsidered.

The only other possible case is that the sequential algorithm does not find a garbage cycle due to concurrent mutation, and therefore never enters it into consideration by the validation tests. However, in that case, the concurrent modification will subsequently be visible to the collector as a decrement, which will introduce the object in question into the set of possible roots, which is then considered correctly.

4.3 Collection of Cycles

To collect the cycles, we process the cycle buffer in reverse order. To see why this is necessary, consider Figure 3. The concurrent algorithm will consider each of the objects as a separate cycle, and if we collected the cycles in the same order they appeared in the buffer, we would only collect the rightmost cycle on each successive epoch, which is clearly unacceptable.

By collecting the cycles in reverse order, and decrementing both the RC and CRC fields of any objects referenced by collected objects, by the time we reach the earlier, dependent cycle, its external reference count will have dropped to zero, and it can be collected.

We observe that if a cycle is collected, then the external reference count (ERC) of any dependent cycles can be updated by subtracting the number of edges from the collected cycle to the dependent cycle. Since the collected cycle is garbage, it is not possible that those edges were subject to concurrent mutation, and it is not necessary to re-compute the ERC. Therefore, if the ERC of the dependent cycle reaches zero, and it passes the Δ-test, then it is also a garbage cycle and can be collected.

There are also certain types of dependent graphs not detected in a single epoch by our algorithm that would be detected if a fully general SCC algorithm were run. However, such an algorithm may require constructing a supergraph as large as the original object graph, and once again we believe the likelihood of such data structures in practice is very low.

4.4 Isolated Markings

Since the algorithm for finding candidate cycles is coloring objects concurrently with the execution of the mutators, it is possible that the mutators can cut an edge that causes arbitrary gray or white subgraphs to be isolated from the collector. These subgraphs could later "fool" the algorithm into producing an incorrect result.

We handle this problem by always recoloring the reachable graph of a gray or white object to black when its reference count is incremented or decremented (in the case of a decrement, the root object is colored purple and considered as a root). This means that the colors will always be properly reset after at most two epochs.

In the meantime, false positives are handled by the validation tests. False negatives are not an issue since the objects are inherently live (because they have been concurrently mutated), and any garbage cycles involving concurrent decrements will be found when the object is recolored purple and added to the root buffer.

5. IMPLEMENTATION

The Recycler is implemented in Jalapeño [1], 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 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.

5.1 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 [31], 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.

6. THE PARALLEL COLLECTOR

In this section we briefly describe the parallel non-copying mark-and-sweep collector with which the Recycler will be compared.

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 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.

Program	Description	Applic. Size	Threads	Obj Alloc	Obj Free	Byte Alloc	Obj Acyclic	Incs	Decs
201.compress	Compression	18 KB	1	0.15 M	0.13 M	240 MB	76%	0.46 M	0.53 M
202.jess	Java expert system shell	11 KB	1	17.4 M	17.2 M	686 MB	20%	55.1 M	71.6 M
205.raytrace	Ray tracer	57 KB	1	13.4 M	13.1 M	361 MB	90%	3.59 M	16.3 M
209.db	Database	10 KB	1	6.6 M	5.9 M	193 MB	10%	67.0 M	66.7 M
213.javac	Java bytecode compiler	688 KB	1	16.1 M	14.1 M	195 MB	51%	41.6 M	51.8 M
222.mpegaudio	MPEG coder/decoder	120 KB	1	0.30 M	0.27 M	25 MB	76%	12.1 M	6.70 M
227.mtrt	Multithreaded ray tracer	571 KB	2	14.0 M	13.5 M	381 MB	90%	4.5 M	17.3 M
228.jack	Parser generator	131 KB	1	16.8 M	16.4 M	715 MB	81%	16.8 M	33.0 M
specjbb 1.0	TPC-C style workload	821 KB	3	33.3 M	33.0M	1034 MB	59%	52.4 M	84.5 M
jalapeño	Jalapeño compiler	1378 KB	1	19.6 M	18.4 M	676 MB	7%	62.6 M	65.6 M
ggauss	Cyclic torture test (synth.)	8 KB	1	32.4 M	32.0 M	1163 MB	$< 1\%$	56.9 M	77.2 M

Table 2: Benchmarks and their overall characteristics. The benchmarks include the complete SPEC suite and represent a wide range of application characteristics.

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.

7. 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 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.

7.1 Benchmarks

Table 2 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 2 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 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.

7.2 Application Performance

Our collector has a new and unusual design, and there are obvious questions about its overhead and applicability in practice.

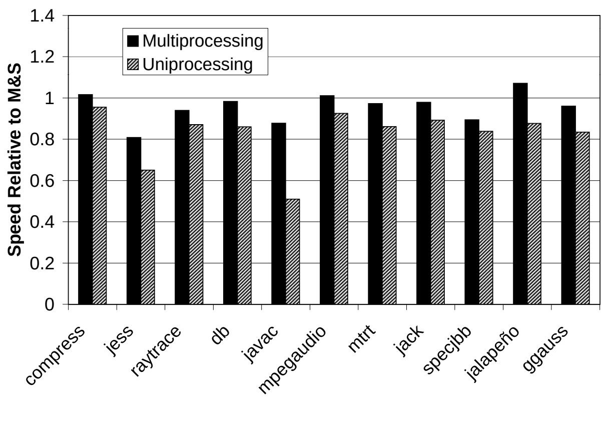

Figure 4: Relative Speed of the Recycler compared to the Parallel Mark-and-Sweep Collector. In the multiprocessing environment the Recycler offers much lower pause times while remaining competitive with Mark-and-Sweep in end-to-end execution time.

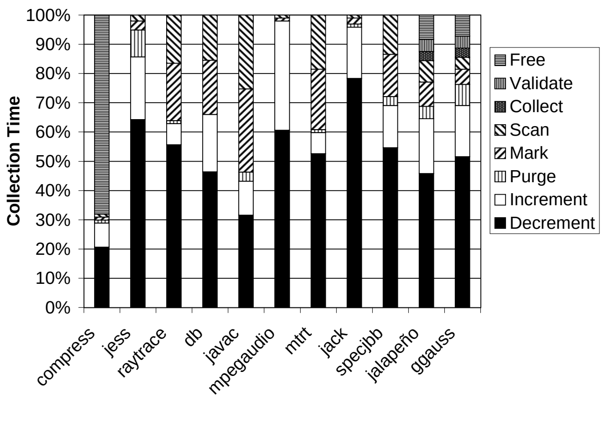

Figure 5: Collection Time Breakdown. The time devoted to various phases varies widely depending on the characteristics of the program.

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 4 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 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.

7.3 Collector Costs

Figure 5 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.

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 inter-

Program	Concurrent Reference Counting (The Recycler)						Parallel Mark-and-Sweep			
	Epochs	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 3: 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.

Program	Buffer Space (KB)		Possible Roots (M)		
	Mutation	Root	Possible	Buff.	Roots
compress	128	131	0.40	0.03	0.01
jess	1920	1180	54.3	9.36	0.23
raytrace	416	393	3.40	42.1	0.27
db	896	131	60.8	3.8	3.8
javac	1792	524	38.5	9.1	4.5
mpegaudio	43616	131	6.42	0.07	0.01
mtrt	992	786	4.2	0.96	0.56
jack	448	131	16.6	0.85	0.20
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 4: Effects of Buffering. The buffer requirements are small, and filtering significantly reduces the roots that must be considered for cycle collection (see also Figure 6).

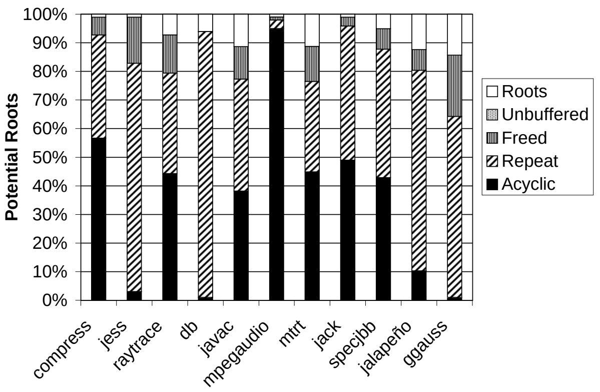

Figure 6: Root Filtering

esting: 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 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!

7.4 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 3 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 [8] 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 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

Program	Epochs	Roots Checked	Cycles Found		Refs. Traced	Trace/ Alloc	M&S Traced
			Coll.	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 5: 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.

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 [4] 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 [16] and by Nettles and O'Toole [27], 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.

7.5 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 4 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 2, 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 [3] to improve space consumption for programs like `mpegaudio`.

Table 4 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 6 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.

7.6 Cycle Collection

Table 5 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

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 6: 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.

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 5 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.

7.7 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 6 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.

8. RELATED WORK

While numerous concurrent, multiprocessor garbage collectors for general-purpose programming languages have been described in the literature [12, 14, 16, 18, 19, 21, 22, 24, 29, 30], the number that have been implemented is quite small and of these, only a few actually run on a multiprocessor [2, 12, 18, 16, 17, 27].

DeTreville's work on garbage collectors for Modula-2+ on the DEC Firefly workstation [12] is the only comparative evaluation of multiprocessor garbage collection techniques. His algorithm is based on Rovner's reference counting collector [29] 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 [19] describe a concurrent algorithm (VCGC) that is used in the Inferno system to back up a reference counting collector. They report that reference 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 [17]. This is a generational collector based on the work of Doligez et al [16, 15], for which generations were shown to sometimes provide significant improvements in throughput. No response time measurements were provided.

The other implemented concurrent multiprocessor collectors [2, 18, 16, 27] are all tracing-based algorithms for concurrent variants of ML, and generally have significantly longer maximum pause times than our collector. In addition, ML produces large amounts of immutable data, thereby simplifying the collection process.

The collector of Huelsbergen and Larus [18] 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 [20], 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.

In concurrently published work, Cheng and Blelloch [8] 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.

8.1 Reference Counting

The Recycler shares with Deutsch and Bobrow's Deferred Reference Counting algorithm [13] 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.

8.2 Cycle Collection

As described in Section 3, our cycle collection algorithm is derived from the synchronous algorithm devised by Martínez et al [25] and extended by Lins to lazily scan for cyclic garbage [23, 20]. 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.

Lins has presented a concurrent cycle collection algorithm [24] 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.

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.

9. CONCLUSIONS

We have presented the Recycler, a concurrent multiprocessor garbage collector for Java implemented in Java. The Recycler comprises novel algorithms for concurrent reference counting and cycle collection. 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 nonconcurrent 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.

Acknowledgements

We thank Anthony Cocchi, Jong Choi, Alex Dupuy, David Grove, Chet Murthy, 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 provided very detailed and thoughtful feedback which has greatly improved the quality of the paper, as has its shepherd, Wilson Hsieh.

Rob Strom suggested the method we employed for detecting dead cycles. Kathryn McKinley stimulated this work with her talk on a sophisticated generational collector, convincing us (despite her best efforts) of the viability of reference counting.

10. REFERENCES

[1] ALPERN, B., ET AL. Implementing Jalapeño in Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1999). *SIGPLAN Notices, 34*, 10, 314–324.

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

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

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

[5] BACON, D. F., AND RAJAN, V. T. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming* (Budapest, Hungary, June 2001), J. L. Knudsen, Ed., vol. 2072 of *Lecture Notes in Computer Science*, Springer-Verlag.

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

[7] BOEHM, H. Personal communication. Hewlett-Packard Laboratories, 2000.

[8] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May).

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

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

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

[12] DETREVILLE, J. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, DEC Systems Research Center, Aug. 1990.

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

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

[15] DOLIGEZ, D., AND GONTHIER, G. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conf. Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 70–83.

[16] DOLIGEZ, D., AND LEROY, X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conf. Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Jan. 1993), pp. 113–123.

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

[18] HUELSBERGEN, L., AND LARUS, J. R. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proc. of the Fourth ACM Symposium on Principles and Practice of Parallel Programming* (May 1993). *SIGPLAN Notices*, *28*, 7 (July), 73–82.

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

[20] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.

[21] KUNG, H. T., AND SONG, S. W. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science* (1977), pp. 120–131.

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

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

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

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

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

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

[28] RODRIGUES, H. C. C. D., AND JONES, R. E. Cyclic distributed garbage collection with group merger. In *Proc. of the Twelfth European Conference on Object-Oriented Programming* (Brussels, July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 249–273.

[29] ROVNER, P. 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 1985.

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

[31] WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management* (Sept. 1995).