Appears in the Proceedings of the 2000 International Symposium on Memory Management (ISMM), 15-16 October 2000, Minneapolis, Minnesota.

Concurrent Garbage Collection Using Program Slices on Multithreaded Processors

Manoj Plakal and Charles N. Fischer
Computer Sciences Department,
University of Wisconsin-Madison,
Madison, WI 53706, USA
{plakal,fischer}@cs.wisc.edu

Abstract

We investigate reference counting in the context of a multithreaded architecture by exploiting two observations: (1) reference-counting can be performed by a transformed program slice of the mutator that isolates heap references, and (2) hardware trends indicate that microprocessors in the near future will be able to execute multiple concurrent threads on a single chip. We generate a reference-counting collector as a transformed program slice of an application and then execute this slice in parallel with the application as a "run-behind" thread. Preliminary measurements of collector overheads are quite encouraging, showing a 25% to 53% space overhead to transfer garbage collection to a separate thread.

1 Introduction

Automatic memory management, or garbage collection, has become a "must have" component of modern programming languages; it makes programming both easier and more reliable. The benefits of garbage collection come at a non-trivial cost. Hence much research has been done in improving collectors [13, 27], especially tracing collectors (mark-and-sweep and copying). Collectors have been made, in particular refinements, generational, incremental, concurrent and parallel.

A less widely used and studied form of garbage collection is reference counting. Reference counting is a naturally incremental and locality-friendly approach since the collector's operations are distributed evenly over the application's computation and the working set of the collector closely matches that of the main application. Reference counting provides for instant recycling of garbage (which also improves memory access locality) and thus instant finalization for expensive objects. It is insensitive to heap residency and can exploit program structure (e.g., dead variable information). Reference counting is attractive in real-time and interactive environments due to the bounded overheads and pause times it can guarantee, in memory-constrained embedded environments due to immediacy of garbage detection, and in distributed environments due to its excellent locality. It is also simple to implement and has found wide use where its overheads are acceptable (e.g., the Perl and Python interpreters) or where it can be easily encapsulated in abstract data types (e.g., "smart pointer" libraries for C++).

The problems inherent in reference counting are well known. There is a space overhead to store reference counts, a time overhead to maintain them, and the difficulty in finding and reclaiming cyclic garbage. Space overheads can be minimized since a few bits should usually suffice for most reference counts and these can be stored in unused parts of object headers or compacted in a table. Time overheads can be minimized by Deutsch and Bobrow's deferred reference counting [5] which ignores updates to local variables and periodically scans the stack to determine true reference counts. Time overheads can also be tackled by using concurrency as in the SRC Modula-2+ collector [4], and as we propose in this work. Cyclic structures can be collected by using a tracing collector as a backup [12]. This paper concentrates primarily on using concurrency to reduce the time overhead of a reference counting collector. Existing solutions for the other problems (space and cycles) can be composed with our solution.

We propose a new way of implementing concurrent reference counting by exploiting two observations:

- The computations that maintain reference counts for heap-allocated objects can be obtained by suitably transforming a *program slice* that comprises all the mutator's instructions that manipulate heap references. This transformed slice can then be executed in parallel with the mutator.
- Hardware trends indicate that next-generation microprocessors will be able to execute multiple concurrent threads on a *single* chip through a variety of techniques: *Simultaneous Multithreading* (SMT, where multiple threads of execution are fetched and executed simultaneously in the same execution pipeline and which share functional units and all levels of caches e.g., Compaq 21464 [10]), *Chip Multiprocessing* (CMP, which is essentially a small-scale shared-memory multiprocessor on a single chip e.g., IBM Power4 [7,14], Sun MAJC [25], Compaq Piranha [1], NEC MP98 [11]) or *Coarse-grain Multithreading* (where multiple threads of execution share a processor but only one is being executed in a given cycle, this differs from traditional multiprogramming in that there is hardware support for extremely fast context-switch time e.g., IBM RS64-II/III [23,24] and Sun MAJC). All of these techniques give software the opportunity to implement closely-coupled threads cheaply by exploiting fast on-chip communication, as opposed to a typical shared multi-

processor architecture where communication between processors is comparatively expensive.

Existing concurrent reference counting techniques have the mutator log all updates to in-heap references (similar to the write barriers of concurrent tracing schemes). The collector is a generic routine that examines the change log to maintain reference counts and reclaim garbage. However, as we shall develop, this logging approach is but one point in a continuum of implementations of the same abstract model. The concurrent-slice implementation we propose reduces the amount of information logged by the mutator by having the collector repeat a subset of the mutator's actions. In the best case, the repeated actions correspond to a precise program slice of references to heap-allocated objects. Thus computations necessary to perform reference counting are removed from the mutator and isolated in an independent assist thread which runs behind the mutator. This approach is inspired by Patil and Fischer's *shadow processing* [19, 20, 18] which tried to speed up memory access checking by moving the instrumentation code conventionally inserted in the main application to a second "shadow" process which executes concurrently on an independent processor.

2 Related Work

Reference counting in practice: One of the most well known efficient implementations of sequential reference counting is Deutsch & Bobrow's deferred reference counting scheme [5]. They only maintain reference counts for updates to references in heap objects, ignoring local variables which can constitute a large fraction of pointer updates. Objects with counts equal to zero are only *potential* garbage (since local variables might point to them); they are put on a zero-count list which is periodically reconciled by having the collector pause the mutator and scan its stack and registers for references in local variables to obtain true counts for the objects in the list. The Lucent Inferno environment [28] includes the Dis Virtual Machine which provides a (non-concurrent) reference counting garbage collector combined with a tracing garbage collector [12].

Concurrent reference counting: The most widely known concurrent reference counting implementation is DeTreville's Modula-2+ collector [4]. He compared several concurrent collectors: pure reference counting, mark-and-sweep, copying and a combination of reference counting and mark-and-sweep. He considered the last combination scheme to be best in spite of the overhead of reference counting since the concurrent tracing schemes had poor memory locality on their target platform (the DEC SRC Firefly). Levanoni and Petrank [16] have proposed a concurrent reference counting algorithm which is designed to use fine-grain synchronization and be scalable on a multiprocessor system, but it has not yet been implemented. Kakuta et al [15] have proposed a concurrent reference counting algorithm for a LISP environment.

Customized garbage collectors: Colnet et al [3] have proposed automatically generating a mark-and-sweep collector for a specific mutator application. Their customizations include type-specific allocation and marking routines to speed up the collection process, exploiting the type system of Eiffel.

Concurrent program slices: Patil and Fischer [19, 20, 18] sliced out memory-access computation and executed it as a shadow "run-behind" process on a dual-CPU multiprocessor in order to implement a low-overhead memory access checker (similar to the Purify tool). They were able to demonstrate relatively low overheads for this shadow-processing scheme which inspired the research in this paper. A related notion is the idea of using a subset of a program's dynamic execution path to compute useful information, rather than using static analysis (and instrumentation) or dynamic profiling to guess the same information. This has recently been exploited in several "run-ahead" pre-fetching schemes [8, 2] as well as in aggressively speculative microarchitectures [21].

3 Reference-Counting with a Concurrent Program Slice

We first present an abstract model of a concurrent reference counting garbage collector for a typesafe language (such as Java) in which all references are to heap objects, and in which reference arithmetic is forbidden. We discuss how this abstract model has been concretely implemented in previous work and note that these implementations are but one point in a continuum of possible systems. We then describe how we will study another point in the continuum by extracting a program slice of the mutator and executing it concurrently on a multithreaded processor.

3.1 An Abstract Model of Concurrent Reference Counting

One can imagine a producer-consumer relation between the mutator and the collector (operating as multiple threads in a shared address space) with the mutator generating requests into a FIFO queue as it executes. The collector processes these requests and maintains reference counts for the objects named in each operation as specified by Table 1.

Mutator Operation	Request in FIFO Queue	Collector Operation
`p=new()`	`CreateRef(p)`	`p->rc = 1`
`p=q`	`AssignRef(p,q)`	`q->rc++` `p->rc--`
`p dead`	`KillRef(p)`	`p->rc--`

TABLE 1. Communication between mutator and collector. 'p' and 'q' are references to objects. p->rc is the reference count of p's referent.

Objects whose reference counts go to zero are considered garbage and returned to the memory allocator. Recursive freeing of objects pointed to by references embedded within this object (if their reference counts go to zero) can be done eagerly at the garbage reclamation point, or lazily by deferring it to the next allocation when this object is recycled by the allocator.

This (conceptual) arrangement requires a shared FIFO queue between the mutator (producer) and the collector (consumer), and some synchronization in the memory allocator. Reference counts and associated book-keeping information are assumed to be stored in object headers and can be accessed without having to synchronize with the mutator. Overheads could be reduced by using Deutsch-Bobrow deferred reference counting and only enqueuing updates of references in the heap, and periodically pausing the

mutator to scan its stack and registers for local references (or by having the mutator voluntarily enqueue information on local references at regular intervals).

3.2 A Continuum of Concrete Implementations of the Abstract Model

Even though our abstract model postulates a logical FIFO queue between the mutator and collector, the two do not have to physically communicate all the information inserted into the queue. We can imagine a *continuum* of concrete implementations of the abstract model, by varying the amount of information actually exchanged between the mutator and the collector. For the rest of this paper, we assume that the Deutsch-Bobrow deferred reference counting framework is used for all implementations.

At one extreme, the mutator logs all references that it manipulates and the collector is a standard log-processor as outlined in Section 3.1. This approach is adopted in the SRC Modula-2+ collector and it seems to be the only actively pursued form of concurrent reference counting. However, there are alternatives.

At the other extreme the mutator communicates *nothing* to the collector except for certain irreproducible values (e.g., return values of memory allocations, system calls and other interactions with the external environment). The collector is a complete copy of the mutator that runs behind it by a safe distance (we describe ways of doing this below). Also, *the mutator need not be paused* for local reference scanning since the collector is a copy of the mutator and could scan itself for this information. This may, of course, be unacceptably expensive as we are running two copies of the same program. The heap space overhead, as seen by the mutator, is comparable to that of a naive semi-space copying collector which essentially cuts the heap in half and copies live data from one semispace to the other when it gets full (though, the working set of this extreme is larger than the copying collector since it potentially uses the entire heap).

There are interesting points in the continuum between these two extremes. A more practical implementation requires that the collector ignore instructions that don't contribute to garbage collection. In particular, the collector could be a program slice [26] of the mutator that recomputes precisely enough information to maintain reference counts of heap-allocated objects. This could be done by using the instructions that read and write heap references as the criteria for a static interprocedural backward slice. This slice is suitably transformed to produce an executable thread that generates the same stream of FIFO requests as the original mutator would have and hence can be used as the collector to maintain reference counts (again without requiring the mutator to be paused).

Each implementation in the continuum has a cost which can be divided into mutator time overhead, and collector time and space overheads. The log-everything approach has low collector overheads but it requires the mutator to log two pointers on every store of a reference to the heap as well as a periodic pause for the collector to scan its stack and registers for local references. The recompute-everything approach has high collector overheads but it has potentially low mutator overheads in that the logging is reduced to a bare minimum and mutators need not be paused for stack scanning (since the collector can scan its own stack as it maintains copies of all mutator reference variables). The slicing approach strikes a compromise with lower collector space overheads while still retaining low mutator time overheads and again, not requiring mutator pauses. Note that there are other points in this continuum as well which trade off collector overhead with mutator overhead. In the next subsection, we describe our particular implementation which approximates the slicing approach.

3.3 Our Implementation of the Model

Our implementation infrastructure is a Java compiler which *statically compiles Java into a native executable* for execution on a uniprocessor as well as simulators of multithreaded processors. We expect the basic technique to be applicable to other configurations.

3.3.1 Generating the Collector

System Model We assume that all collectors that we generate will operate in the following manner. Each class definition in the original mutator application is used by the collector generation algorithm to generate a corresponding "shadow" class definition which contains a transformed subset of the components (data members and methods) of the original class definition. No direct reference is made to names of mutator classes in the collector, only shadow classes are used. Mutator objects created dynamically will have corresponding shadow objects in the collector. Shadow objects are what are manipulated by the collector when it executes. They contain (at least logically) a pointer to the original mutator object which this shadow represents, a reference count and any additional book-keeping information required by the reference-counting algorithm. The collector generation algorithm needs to ensure that such a mapping between mutator and shadow objects can be established (e.g., by logging the address of each new object created) and maintained as the mutator executes (by having the collector be aware of all mutator operations on references).

The Ideal World: A Precise Slice One way of generating the collector is to literally construct a static interprocedural slice of a Java program using all reference-manipulation instructions in the program as the criteria for a backward slice. Program slicing algorithms have been extended to handle object-oriented programs with classes and inheritance hierarchies [17]. However, we are not aware of actual implementations of full-fledged program slicers for Java, apart from the Bandera project which slices Java programs [9] for the specific purpose of generating specifications for model checkers. To avoid implementing a program slicer for Java, we propose a scheme which approximates a precise static slice.

The Real World: Our Approximation to a Precise Slice We want the collector to be able to reproduce the operations on references that are performed by the mutator. To do this, we exploit the type-safety and lack of reference arithmetic of Java. A reference variable can be assigned only one of the following three values: NULL, a return value from a system call (including the memory allocator) or the value of another reference variable. Non-reference variables do not contribute via data-flow to the *values* used to assign references, unlike weakly-typed languages such as C or C++ which allow pointer arithmetic as well as arbitrary casting between pointer and non-pointer types. However, non-reference variables *do* contribute via control dependences (control flow determined by non-references which affects reference-manipula-

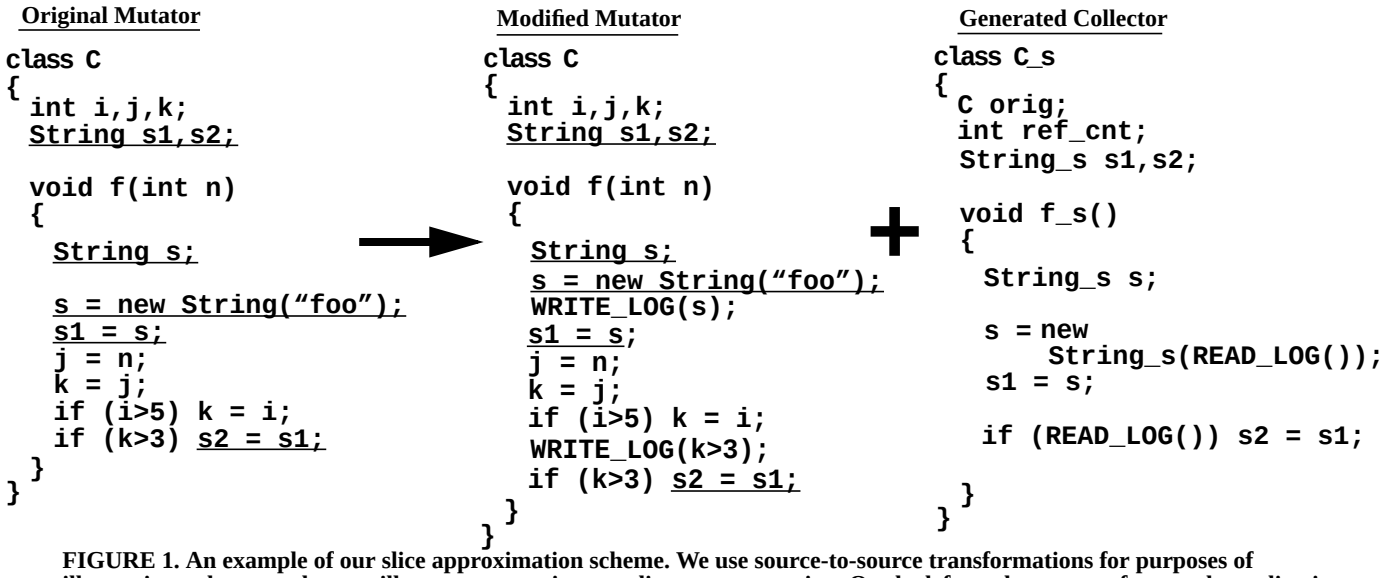

FIGURE 1. An example of our slice approximation scheme. We use source-to-source transformations for purposes of illustration only; our scheme will operate on an intermediate representation. On the left, we have part of a sample application program consisting of a single class C, where underlined constructs directly involve references. In the middle, we have the modified mutator generated by our scheme which logs the return value of the memory allocation as well as the condition that controls the reference assignment 's2 = s1'. On the right, we have the generated collector where the class C has a corresponding shadow class C_s which only includes the reference members of C, as well as a pointer to the object which it shadows and its reference count. Inside the shadow member function f_s(), only mutator statements which manipulate references are preserved, with occasional reads from the log. For clarity, we do not show the subsequent transformation of this "slice" to actually maintain the reference counts of the objects whose references are being created and assigned in the function. If we used a precise static slice instead of our approximation, then the collector would have had to include the variables i,j,k and n as well as the statements that operate on them while on the other hand, the mutator would not have to log the condition k>3 since the collector would have enough information to be able to compute that on its own.

tions) and reference-address generation (a non-reference value used to index into an array of references). Our slice approximation scheme makes a copy of all instructions (statements in class methods) and data (object and class members, local variables, method parameters and return values) in the mutator that directly involve object references, and makes the mutator log all control dependences and address generation that depend on non-reference values and affect reference values. In addition, all system calls that return references as results have their return values logged.

This scheme is biased towards producing a small collector slice. It copies the absolute minimum required mutator code (the reference-manipulations instructions which are the slice criteria) along with some reads from the log as opposed to including the code and data that would re-compute the logged information in the collector. Due to lack of space, we do not present a formal algorithm to generate the collector according to what we have just described. To illustrate our scheme in practice, we present an example of a generated collector for a simple mutator class in Figure 1.

More on the Shadow Heap A shadow object in the collector contains only the reference data fields of the corresponding mutator object[1], though of course they point to other shadow objects rather than mutator objects. Thus, one can imagine the shadow objects in the collector literally representing the connectivity graph of mutator objects, since each shadow object only contains references to other shadow objects. The collector's working set is then the total size of all the live shadow objects which translates into the fraction of the live mutator heap locations that contain references.

We are not restricted to shadowing every reference in a mutator object. We could choose to shadow some (and not have to log their updates) while we could leave some unshadowed (thus having to log their updates)[2]. This gives us a means of bounding shadow heap overhead by deciding how much gets shadowed. Furthermore, sophisticated encoding and compression schemes could be used to trade-off collector execution time with space overhead. The specific scheme we used for our evaluation in Section 4.2 creates shadow objects only for mutator objects which contain references, where each shadow object contains, in addition to the shadowed references, a pointer to its corresponding mutator object (into whose unused bits we squeeze a reference count).

Mutator Logging We have the mutator log relevant information to avoid having to compute a precise mutator slice as well as to keep the collector small in size. This information includes all system calls that return references, certain control flow conditions which control reference manipulation, and indices into arrays of references. We only need to log control flow if the control flow affects manipulation of reference variables, and the control flow condition is determined by contents of non-reference variables. Other control flow can either be omitted, copied into the collector (e.g., all non-virtual method calls) or be reproduced by the collector (e.g., virtual method calls can use the same virtual method table index in both mutator and collector). The control-flow information we do log

1. Note that an obvious optimization for classes with no reference data members is not to have a shadow object but directly manipulate book-keeping information in the mutator object header.

2. Note that arrays of references need to be shadowed in their entirety.

consists of 1 bit for two-way branches and a few bits (jump table index) for multi-way branches (switch statements). Similarly, we expect the index into an array of references to be much less than 16 bits worth of data, on average.

This information could be accumulated in spare processor registers in the mutator and then logged before allowing the collector to advance (described in the next subsection). Apart from the logging operation itself, this does not involve extra memory operations and we expect these computations not to be on the mutator's critical path (assuming no resource constraints). These operations *are* on the collector's critical path and that is alright since we are willing to let the collector lag behind while we want the mutator to continue executing without ever having to pause. This logging could be reduced further by using a more precise (though larger) slice of the mutator that included enough data fields for the collector to reproduce the information in the log.

At first blush, our scheme might seem like it involves too much logging. For example a loop that walks over a linked list and which has reference manipulations in its body needs to log the loop exit condition for each iteration of the loop. However, one must keep in mind that a traditional concurrent reference-counting scheme would involve logging more information since it needs to remember two pointer values for each update of a reference in a heap-allocated object, coupled with periodic mutator pauses for stack scanning. Hence in the case of the loop, it would record two pointers per iteration *per reference-update in the loop body* while our scheme would only log a boolean loop exit condition per iteration and not require mutator pauses. This is because our collector can reproduce the values of the references being updated thanks to its shadow heap.

To be fair, it is not clearly evident at this stage whether our logging will be cheap and whether the collector size (code and data) will be manageable: these are the issues which our final implementation and evaluation will resolve. Preliminary tests indicate that the size of the shadow heap is reasonable (see Section 4.2).

3.3.2 Keeping the Collector behind the Mutator

The collector must run at a safe distance behind the mutator. That is, the reference count maintenance corresponding to a mutator operation may be performed only after that operation has been executed (see Table 1). This is non-trivial since not all mutator operations are logged in a FIFO queue and the collector now reproduces some mutator computation on its own.

Our solution is to divide the execution of the mutator into disjoint sections which we call *epochs*. We ensure that the collector performs reference count manipulations of an epoch *only after the mutator has finished executing that epoch*. One simple way of doing this is to use a counter stored in memory shared between the mutator and collector and have the mutator increment the counter frequently as it executes. Counter increments demarcate epochs. We will ensure that epochs are of bounded length to avoid the case of the collector waiting for a long (or possibly infinite) time before it can proceed to the next epoch. One easy way to ensure this is to insert an increment on each loop backedge or function call. Values that have to be logged in an epoch can be accumulated in registers or scratch memory and then flushed to the log in a single operation just before indicating the end of that epoch (by incrementing the counter). There is a corresponding division into epochs of the collector's instruction stream where it checks the counter against a private copy of the counter to make sure that the mutator has completed execution of an epoch before it proceeds to read the log and update reference counts for that epoch. We *do not* use explicit synchronization to access the counter. There is a data race for the shared counter but it is harmless since the collector does not write to the counter and proceeds forward only when it reads an incremented value i.e., stale values only block the collector from proceeding to the next epoch and hence are a performance problem rather than a correctness problem. The placement of these counter updates is similar to the placement of thread-yield points in non-preemptive multithreaded systems.

4 Implementation Status and Evaluation

A full implementation of a concurrent collector is currently under development. Initially, we implemented a *sequential* reference-counting collector designed to be extended to our concurrent collector scheme. In this section, we use the sequential collector to estimate some of the overheads of a concurrent collector.

4.1 Infrastructure and Benchmarks

Our implementation infrastructure consists of the Strata [22] Java compiler and multithreaded processor simulators developed locally at Wisconsin. Strata (written in Java) statically compiles Java bytecodes into SPARC or MIPS executables and performs a number of standard local and global optimizations, including null-pointer and array-bounds check elimination. Currently, the compilation is entirely intraprocedural and multithreaded applications are not supported. The sequential collector we implemented uses the standard Deutsch-Bobrow scheme with 2-bit reference counts squeezed into object headers. The run-time system was instrumented to simulate the shadow scheme outlined in Section 3.3.1.

We used 8 benchmarks in all: the Strata compiler itself (more than 40K lines of Java), five programs from SPECjvm98 (the remaining three, mtrt and jess and javac, cannot currently be compiled with Strata) and Java versions of two well-known object-oriented benchmarks, Richards and Deltablue[1]. All benchmarks were compiled with Strata for the SPARC architecture with maximum optimizations. Strata was executed with one of its larger source files (1358 lines) as input, the SPECjvm98 benchmarks were run with their largest inputs (speed 100) while Richards and Deltablue were run for 100 iterations.

4.2 Preliminary Results

Table 2 shows the number and average sizes of all objects allocated. Our measurements are in general agreement with those reported by Dieckmann and Hölzle [6], with discrepancies mainly due to the difference in execution environments and object layout. Instance objects, on average, have 1 or 2 references, which implies small shadow objects. Moreover, the small size of instance objects suggests that overhead could be reduced by co-locating the shadow object with its associated mutator object. Note too that arrays of references typically contain a large fraction of null elements when they are reclaimed. This suggests that a sophisticated shadowing

1. Obtained from http://www.sun.com/research/people/mario/java_benchmarking

Benchmark	All Objects		Instance Objects			Array-of-reference objects		
	Number Allocated	Avg size (bytes)	Number Allocated	Avg size (bytes)	Avg Ref fields	Number Allocated	Avg size (bytes)	Avg % non-null
Strata	1.6M	40	1.2M	22	2.0	20K	347	16
Jack	6.9M	22	3.9M	17	1.3	160K	93	1.5
Raytrace	309K	20	251K	15	1.2	32K	29	58
Db	3.2M	20	3.1M	12	1.0	16K	1376	81
Compress	8K	15K	4.4K	19	2.1	19	2356	56
Mpegaudio	10K	366	5.3K	18	2.2	154	350	56
Richards	5.9K	37	3.4K	22	2.0	11	1689	52
Deltablue	418K	28	237K	18	1.5	89K	51	10

TABLE 2. Distributions of object sizes and densities relevant to our collector generation scheme. Instance objects refer to objects that are not arrays (i.e., instances of some class). Note that the average percentage of non-null elements of arrays of references (shown in the last column) is calculated over all reclaimed (garbage) arrays of references only.

Benchmark	Mutator Heap		Shadow Heap		Efficacy of reference counting			
	Total memory requested	Allocator High-water mark	Total memory requested	Allocator High-water mark	Total memory reclaimed	Total live memory at end	Total unclaimed garbage at end	Total unclaimed garbage with ref count stuck
Strata	67MB	34MB	22MB	12MB	34MB	2MB	31MB	6MB
Jack	174MB	18MB	56MB	9.7MB	158MB	1.2MB	15MB	3.6MB
Raytrace	6.4MB	4MB	1.8MB	1.2MB	2.7MB	613K	3MB	0.5MB
Db	77MB	9.4MB	46.5MB	2.4MB	69MB	1.4MB	6.8MB	130K
Compress	119MB	119MB	101K	83K	45K	9.7MB	110MB	94MB
Mpegaudio	3.7MB	3.8MB	109K	92K	47K	3.6MB	0	0
Richards	224K	357K	61K	57K	20K	204K	0	0
Deltablue	12MB	5.1MB	6.8MB	2MB	7.2MB	149K	4.7MB	1.4MB

TABLE 3. The memory overheads of our collector's shadow heap as well as reference counting in general. The statistics for unclaimed garbage were calculated by performing a mark-and-sweep of the heap just prior to the end of execution.

scheme that tracks only non-null references could further reduce shadow heap overhead.

Table 3 shows memory overheads of the shadow heap. It also details the effectiveness of using reference counting to collect the mutator's heap. We believe that the measurements for Strata, Jack and Raytrace (and to some extent, Db) are representative of large, real applications written in an object-oriented style.

The shadow heap overhead (measured by the ratio of the high-water marks) usually varies between 25% to 53%, with the exception of Compress, Mpegaudio and Richards which do not seem to be very interesting from a garbage collection point of view. There is more variance in reference counting's overall efficacy. To simplify our initial implementation, we use 2-bit reference counts, so any object with a reference count of 3 is "stuck" and cannot be collected. In general, reference counting (due to saturated counts and cyclic structures) does not leak excessively with the notable exceptions of Compress which seems to need more reference count bits and Strata which seems to be using a lot of cyclic data structures. We expect that for many applications reference counting, with at most an occasional full collection, will suffice.

5 Conclusions and Ongoing Work

We have proposed a new concurrent reference counting algorithm that maintains reference counts with a collector slice that executes in parallel with the mutator on a multithreaded processor. Preliminary experiments show that space overheads for the shadow heap are reasonable (25%-53% of the mutator heap). With more effort of the collector's part, shadow heap sizes can be further reduced.

Work currently in progress aims to complete the implementation of the collector generation algorithm and to compare our scheme's performance to that of conventional mark-and-sweep and generational collectors. We expect to produce effective multithreaded collectors that impose little mutator overhead.

6 Acknowledgements

This research was supported by the National Science Foundation under grant CCR-9974613. We would like to thank Timothy Heil and Subramanya Sastry for critical support with Strata, as well as Harish Patil, Rastistav Bodik, Anne Mulhern, Denis Gopan, Harit Modi, Ravi Rajwar and the anonymous reviewers for their many helpful comments and suggestions.

7 Bibliography

[1] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 12–14 2000.

[2] Fay Chang and Garth Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd Annual Symposium on Operating System Design and Implementation*, New Orleans, Louisiana, February 1999.

[3] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. In *Proceedings of the ACM SIGPLAN 1998 International Symposium on Memory Management*, Vancouver, Canada, October 17–19, 1998.

[4] John DeTreville. Experience with Concurrent Garbage Collectors for Modula-2+. Technical Report 64, Compaq Systems Research Center, November 1990.

[5] L. Peter Deutsch and Daniel G. Bobrow. An Efficient Incremental Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–576, September 1976.

[6] Sylvia Dieckmann and Urs Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, June 14–18, 1999.

[7] Keith Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), October 1999.

[8] James Dundas and Trevor Mudge. Improving Data Cache Performance By Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, Netherlands, June 11–15 1997.

[9] Matthew B. Dwyer, James C. Corbett, John Hatcliff, Stefan Sokolowski, and Hongjun Zheng. Slicing Multi-threaded Java Programs. Technical Report KSU CIS TR 99-7, Department of Computing and Information Sciences, Kansas State University, 1999.

[10] Joel S. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. In *Proceedings of the 1999 International Microprocessor Forum*, San Jose, California, October 4–8 1999. MicroDesign Resources.

[11] N. Nishi et al. A 1GIPS 1W Single-Chip Tightly-Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control Flow Execution. In *Proceedings of the 2000 IEEE International Solid-State Circuits Conference*, San Francisco, California, February 7–9 2000. IEEE Solid-State Circuits Society.

[12] Lorenz Huelbergen and Phil Winterbottom. Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization. In *Proceedings of the ACM SIGPLAN 1998 International Symposium on Memory Management*, Vancouver, Canada, October 17–19, 1998.

[13] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[14] Jim Kahle. Power4: A Dual-CPU Processor Chip. In *Proceedings of the 1999 International Microprocessor Forum*, San Jose, California, October 4–8 1999. MicroDesign Resources.

[15] K. Kakuta, H. Nakamura, and S. Iida. A Parallel Reference Counting Algorithm. *Information Processsing Letters*, 23:33–37, July 1986.

[16] Yossi Levanoni and Erez Petrank. A Scalable Reference Counting Garbage Collector. Technical Report CS-0967, Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel, November 1999.

[17] Donglin Lian and Mary Jean Harrold. Slicing Objects Using System Dependence Graph. In *Proceedings of the IEEE International Conference on Software Maintenance*, Washington, D.C., November 1998.

[18] Harish G. Patil. *Efficient Program Monitoring Techniques*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[19] Harish G. Patil and Charles N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, St Malo, France, May 1995.

[20] Harish G. Patil and Charles N. Fischer. Low-cost, Concurrent Checking of Pointer and Array Accesses in C programs. *Software: Practice and Experience*, 27(1):87 – 110, December 1997.

[21] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-1414, Computer Sciences Department, University of Wisconsin-Madison, April 2000.

[22] James E. Smith, Subramanya S. Sastry, Timothy Heil, and Todd M. Bezenek. Achieving High Performance via Co-designed Virtual Machines. In *International Workshop on Innovative Architecture*, 1999. See Strata Homepage at URL http://www.cae.wisc.edu/~ecearch/strata.

[23] S. Storino, A. Aipperspach, J. Borkenhagen, and S. Levenstein. A Commercial Multi-Threaded RISC Processor. In *Proceedings of the 1998 IEEE International Solid-State Circuits Conference*, San Francisco, California, February 5–7 1998. IEEE Solid-State Circuits Society.

[24] S. Storino and J. Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor Design. In *Hot Chips 1999: Proceedings of the 11th Annual International Symposium on High-Performance Chips*, Stanford University, California, August 15–17 1999.

[25] Marc Tremblay. An Architecture for the New Millenium. In *Hot Chips 1999: Proceedings of the 11th Annual International Symposium on High-Performance Chips*, Stanford University, California, August 15–17 1999.

[26] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[27] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jaques Cohen, editors, *Proceedings of the 1992 International Workshop on Memory Management*, pages 1–42, St Malo, France, September 17–19 1992.

[28] Phil Winterbottom and Rob Pike. The Design of the Inferno Virtual Machine. In *Hot Chips 1999: Proceedings of the 11th Annual International Symposium on High-Performance Chips*, Stanford University, California, August 15–17 1999.