A Parallel, Real-Time Garbage Collector[*]

Perry Cheng Guy E. Blelloch
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{pscheng,guyb}@cs.cmu.edu

ABSTRACT

We describe a parallel, real-time garbage collector and present experimental results that demonstrate good scalability and good real-time bounds. The collector is designed for shared-memory multiprocessors and is based on an earlier collector algorithm [2], which provided fixed bounds on the time any thread must pause for collection. However, since our earlier algorithm was designed for simple analysis, it had some impractical features. This paper presents the extensions necessary for a practical implementation: reducing excessive interleaving, handling stacks and global variables, reducing double allocation, and special treatment of large and small objects. An implementation based on the modified algorithm is evaluated on a set of 15 SML benchmarks on a Sun Enterprise 10000, a 64-way UltraSparc-II multiprocessor. To the best of our knowledge, this is the first implementation of a parallel, real-time garbage collector.

The average collector speedup is 7.5 at 8 processors and 17.7 at 32 processors. Maximum pause times range from 3 ms to 5 ms. In contrast, a non-incremental collector (whether generational or not) has maximum pause times from 10 ms to 650 ms. Compared to a non-parallel, stop-copy collector, parallelism has a 39% overhead, while real-time behavior adds an additional 12% overhead. Since the collector takes about 15% of total execution time, these features have an overall time costs of 6% and 2%.

1. INTRODUCTION

The first garbage collectors were non-incremental, non-parallel collectors appropriate for batch workloads on uniprocessors [14, 4, 21]. However, parallel or multi-threaded applications that run on multiprocessors need parallel garbage collectors since even a dedicated processor cannot keep up with the memory requirements of more than a few processors. Early efforts on designing parallel collectors include Halstead's collector for Multilisp [18]. The Multilisp collector, however, has problems scaling because it does not load-balance the collection work. More recently, Endo used load balancing to achieve a scalable parallel mark-sweep collector [12]. These ideas were extended by Flood et. al. for a copying collector [15]. Other researchers using incremental and concurrent collectors tackled the problem of eliminating the lengthy pauses applications experience during garbage collection. The elimination of such pauses is necessary to make languages with garbage collection useful in applications with real-time constraints. Incremental collectors break up each garbage collection into smaller pieces of work and interleave these increments of collection work within the execution of the program [1]. Concurrent collectors run a single collector thread concurrently with one or more application threads [23, 8]. These collectors, however, do not consider running multiple collector threads in parallel.

In an earlier paper, we presented a collector algorithm that is both scalably parallel and real-time [2]. By making all aspects of the collector incremental (*e.g.*, incrementally copying arrays) and allowing an arbitrary number of application and collector threads to run in parallel, we were able to achieve tight theoretical bounds on the pause time for any application thread as well as bound the total memory usage. Since the main purpose of that paper was to prove the time and space bounds, the interface was somewhat simplified and the algorithm had features that were asymptotically but not practically efficient. Furthermore the algorithm was a paper design with no implementation.

In this paper we present an implementation based on our previous collector algorithm along with an experimental analysis of the implementation. The collector is designed for shared-memory multiprocessors and implemented within the runtime system [6] for the TILT SML compiler. To make the algorithm efficient in practice and to make it work within the context of an existing language and compiler, we had to extend the algorithm in several ways. These extensions are described in this paper. The important additions related to performance include extending the algorithm to work with generations, increasing the granularity of the incremental steps, separately handling global variables, delaying the copy on write, reducing the synchronization costs of copying small objects, paralellizing the processing of large objects, and reducing double allocation during collection. The most important extension of the interface was to allow program stacks—the original algorithm assumed activations records were allocated on the heap.

To evaluate the algorithm and implementation we imple-

[*]This work was supported in part by the National Science Foundation under grants CCR-9706572 and CCR-0085982, in part by DARPA CSTO, issued by ESC/ENS under Contract No. F19628-95-C0050, and in part by an IBM Graduate Student Fellowship in Computer Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2000 ACM 0-89791-88-6/97/05..$5.00

mented several variants, including stop-copy and concurrent (real-time) variants, and semi-space and generational variants. The implementations were evaluated on a set of 15 benchmarks on a Sun Enterprise 10000 shared-memory multiprocessor. Our experiments show that compared to a baseline non-incremental, non-parallel collector, scalable parallelism on average has a 39% overhead while concurrency and real-time bounds add another 12%. Since the collector takes about 15% of the total execution time, these translate to an overall time cost of 6% and 2%, respectively. The average speedup of the collector is 7.5 at 8 processors and 17.7 at 32 processors. As for real-time bounds, our maximum pause time is about 3 ms to 5 ms. Furthermore we analyze the real-time performance in terms of a metric we call the *minimum mutator utilization* (MMU). The MMU allows one to see not just the maximum pause time, but the fraction of time that the processor is available to the *mutator* (application) for any time window of a specified size (worst case across the full execution). This is a much more useful measure of real-time capabilities than just the maximum pause time. Depending on the parameter setting, our real-time collector provides the program with 10% to 15% processor access in any 10 ms window.

The features of our collector can be summarized as follows.

- **Parallel** and **Concurrent**. Any number of collector and mutator (application) threads can run simultaneously.
- **Copying**. The collector is a replicating collector, copying reachable data from the from-space to the to-space while the program is executing.
- **Real-Time**. All phases of the collector are incremental, allowing pause times to be strictly bounded.
- **Handles Large Objects**. Large objects, like arrays of arbitrary size, are copied both incrementally and in parallel.
- **Uses Stacklets**. The program stacks of each thread are partitioned into stacklets to permit parallel processing and real-time bounds even for deep stacks.
- **Barriers**. Our collector never requires a read barrier (*i.e.* mutators read data directly with no overhead). A write barrier is imposed on modifications to heap objects but not to values in the stacks or registers.
- **Header Words**. For all small objects the per-object information used by the collector is kept in the tag word, requiring no extra space.

2. BACKGROUND AND DEFINITIONS

We describe a traditional *semispace* copying garbage collector [4], define when a collector is incremental, parallel, or concurrent, and introduce a new measure for analyzing the real-time behavior of a collector.

2.1 A Semispace Stop-Copy Collector

In a semispace collector, heap memory is divided into two equally-sized regions: the *from-space* and the *to-space*. During normal execution, the mutator allocates new objects from from-space. Eventually, continued allocation exhausts from-space causing the program to be suspended while the collector reclaims memory. The program's data can be viewed as a directed graph consisting of *internal nodes* corresponding to objects in the heap, *root nodes* corresponding to non-heap pointer values (i.e. in registers, global variables, and on the stack), and edges corresponding to pointers between objects, and from the root nodes into the heap. The objects that can be reached from the root nodes are called the *reachable* objects. Under this interpretation, the collector traverses the memory graph starting from the roots and copies the reachable objects from from-space into to-space. When all reachable objects are copied, the collector is *flipped* off by updating the root values and reversing the roles of from-space and to-space.

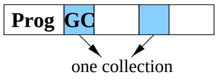
A useful terminology for describing garbage collection is the tri-color abstraction [8] which assigns one of three colors (white, gray, and black) to objects/roots. When a collection starts, all objects are white and the root nodes are gray. When a white object is copied, it becomes gray, signifying that it has been copied but that it may still refer to white objects. When all of a gray object's children are copied it becomes black. A collection is complete when all reachable objects are black, or equivalently when there are no gray objects or roots left. At any time during the collection, the set of gray objects or roots can be seen as the frontier of the graph traversal. To properly update all the pointers during collection so that they point to the new copy, the collector needs to know the new location of each object. This association is typically maintained with a *forwarding pointer* from the old copy to the new one.

2.2 Types of Garbage Collectors

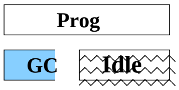
The first garbage collectors, including the one described in the previous section, were non-incremental (*stop-collect*) and programs (also called *mutators*) were periodically suspended while the collector reclaimed memory. Over the years, various techniques were introduced to improve collector efficiency, to reduce or eliminate pause times, and to adapt a collector suitable for use on multiprocessors. Many of these techniques or collector properties are often used in isolation to address a particular problem, creating a misperception that the techniques are mutually exclusive. In fact, these techniques sometimes complement each other. To avoid confusion, we define the following terms:

- Incremental[1]: A single collection is divided into multiple increments whose executions are interleaved with the application on a single processor. Typically, the rate of collection is related to (and greater than) the allocation rate so that the collection is guaranteed to terminate.
- Concurrent[23, 8]: At least one program thread and one collector thread are executing concurrently.
- Parallel[18, 12]: Multiple collector threads are collecting concurrently.

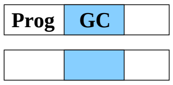
The reader should be aware that these terms are not used consistently across the literature. For example, some papers refer to concurrent collectors as being parallel while others exclude the notion of incrementality when they use the term parallel. The diagram below shows one collection cycle for the 3 types of collectors for the 2-processor case:



Incremental

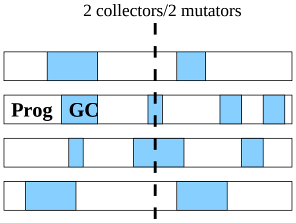


Concurrent



Parallel

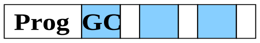
Both our original and modified collectors are simultaneously incremental, concurrent and parallel. Each processor runs incrementally, but since the processors are not synchronized, any combination of mutator and collector threads can run concurrently. This can be illustrated as follows:



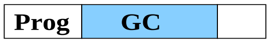
incremental, concurrent, and parallel

2.3 What is a Real-time Collector?

Empirical studies of real-time garbage collectors typically report pause times, with particular emphasis on the maximum pause time. What can a programmer deduce from knowing that the maximum pause time is 25ms? Pessimistically, he can deduce this is unsuitable for applications that require a response time of 10ms. However, he cannot safely assume the collector is suitable for mouse tracking, which requires a response time of about 50ms. The problem is that statistics about the pause time do not characterize the occurrences of the pauses. A burst of frequent short pauses is not much different from a single long pause.



Several short pauses



One long pause

To capture both the size and placement of pauses, we propose a new notion called *utilization*. In any time window, we define the utilization of that window to be the fraction of time that the mutator executes. For any window size, the minimum utilization over all windows of that size captures the access the mutator has to the processor at that granularity. For example, mouse tracking might require a response time of 50ms and the mouse-tracking code might take 1ms to execute. In that case, it is sufficient to have a minimum utilization of 2% at a granularity of 50ms. The minimum mutator utilization (MMU) is a function of window size and generalizes both maximum pause time and collector overhead. The maximum pause time is the window size at and below which the MMU is zero and the collector overhead is the complement of the MMU at the granularity of the total execution time.

3. THEORETICAL ALGORITHM

In this section, we review our previous collector algorithm [2] by a series of modifications to Cheney's simple copying collector [4]. Our treatment adds, in turn, parallelism, incremental collection, and concurrency. We end by describing the space and time bounds guaranteed by the collector. The reader is referred to our earlier paper for more details.

The collector is designed for a shared memory multiprocessor with sequentially consistent memory. We assume that all objects are stored in a shared global pool of memory. In addition to the typical uniprocessor instructions, the collector uses a `FetchAndAdd` instruction, which atomically reads a memory location and stores the incremented value back to the location, and a `CompareAndSwap`, which atomically reads a memory location and, if the read value equals some given value, updates the same memory location with a new value. The collector interfaces with the application via 3 memory-related operations: allocating space for a new object, initializing the fields of a new objects, and modifying the field of an existing object.

3.1 Scalable Parallelism

While running, the collector needs to keep track of all the gray objects so that it can perform its traversal. The manner in which the set of gray objects is maintained and the order in which they are visited is important. Cheney proposed a clever technique for maintaining the set implicitly with no additional space cost [4] by keeping them in contiguous locations in to-space. This technique, however, restricts the traversal order to breadth-first, and is also difficult to implement in a parallel setting. For these reasons, our collector represents the set of gray objects using an explicitly-managed local (per-processor) stack with pointers to the gray objects.

For the collector to scale to multiple processors, it is important that no processor is idle during the collection. This can happen if one processor runs out of gray objects while collectively there are more gray objects to be processed. Our collector uses work sharing to avoid idleness. In addition to the local per-processor stacks, we add a shared stack of gray objects accessed by all processors. Each processor periodically transfers gray objects between its local stack and the shared stack. Thus no processor idles until there are no more gray objects in the shared stack. Other parallel collectors have used work-stealing for sharing work [12, 15].

Our shared stack is designed to allow fast parallel access. It separates in time the pushes from the pops, but allows an arbitrary number of pushes (or pops) to run in parallel. The pushes can proceed in parallel by using a `FetchAndAdd` to reserve a target region in the shared stack prior to the data transfer. This scheme fails when pushes and pops are concurrent since the target regions are no longer non-overlapping and the subsequent data transfer cannot proceed independently. To preventing concurrent pushes and pops while permitting multiple pushes or multiple pops, we use a form of synchronization called room synchronization. This synchronization is described later in section 4.3.

The main correctness issue that arises with multiple collector threads is the possibility that a white object is erroneously copied twice, resulting in an incorrect graph in to-space. To prevent this from occurring, a per-object synchronization is used for object copying. Before an object is copied, a copier must gain exclusive access to the object by atomically swapping into the forwarding pointer field a flag designating that copying is in progress. Other processors, if any, that had desired to copy the object would have failed to perform the atomic swap and instead noticed that copying is in progress. These processors busy-wait until the forwarding pointer is installed. We refer to this mechanism as the *copy-copy synchronization*.

3.2 Incremental and Replicating Collection

Baker proposed the first incremental collector [1]. For every unit of space that is allocated while the collector is on, the collector would copy k units of data. This allowed him to bound the pause time by showing that each copy takes bounded time, and bound the memory usage by showing that the collector will make sufficient progress toward the collection. The complication in designing an incremental collector is that the mutator can execute while the collection is only partially complete. Baker's algorithm handles this by keeping a to-space invariant—the mutator can only see the *copied* version of objects, which are in to-space. However, preserving this invariant means that each read of a pointer by the mutator requires an additional check. The check, called a *read-barrier*, has been experimentally found to be quite impractical since reads are very frequent.

To avoid a read-barrier, our algorithm uses a variant of the replicating collector suggested by Nettles and O'Toole [22]. In a replicating collector the mutator can only see the *original* copy, which is in from-space. While the collector executes, the mutator continues to modify the objects in from-space, which collectively form the *primary* memory graph, while the collector generates a *replica* memory graph in to-space. To maintain the invariants necessary for correctness, replicating collectors require a *write barrier*. That is, whenever an object's field is modified, both the old and new object must be copied and colored gray. Since writes are significantly less frequent than reads in nearly all languages and programs, this is less of a problem than the read-barrier. This is particularly true in mostly functional languages such as SML. As with Baker's algorithm, while it is on our collector collects k units of data for every unit that is allocated. This is done on a per-processor basis—a processor that allocates a unit of space must copy (*i.e.*, gray) k units.

3.3 Concurrency

A collector is concurrent if the program and collector can execute simultaneously. Since the program only manipulates the primary memory graph, which is not disturbed by the collector (except for the forwarding pointer field), the collector does not interfere with the mutator. On the other hand, the collector generates the replica graph by traversing the primary graph, which is being modified by the mutator (hence the name). Because a primary object might be modified after it has been copied, a replica object will not necessarily be a faithful replica unless the corresponding modification is made to the replica. A race condition arises if a primary object is being modified around the same time that it is independently copied. Correctly handling this situation requires a *copy-write synchronization* which delays the mutator's update to the replica if the modified field of the object is being copied by the collector, which is indicated by a flag in the replica object. More details on the necessity and correctness of this synchronization can be found in our earlier paper [2].

The application model in our original algorithm prohibited different threads from concurrently modifying the same memory location. Such concurrent writes create a potential problem in the collector. Normally, the write barrier requires that when a thread modifies a (primary) object, it must perform the corresponding update to the replica object. If two threads modify the same primary object, they will then perform concurrent updates to the same replica object. A *write-write* synchronization is necessary to prevent a thread from updating the replica with an out-of-date primary value. More details on this synchronization can be found in Cheng's thesis [5].

3.4 Space and Time Bounds

The key property of our previous collector were the time and space bounds. These can be summarized as follows.

- Each memory operation will take no more than ck time for some constant c. Intuitively, c is the time it takes to collect one word and k is the number of words we collect per word allocated.
- If any application uses maximum reachable space R, maximum object count N, and maximum memory graph depth D (the depth of a path is the sum of the sizes of objects along that path), our collector on a P-way multiprocessor requires at most $2(R(1+1.5/k)+N+5PD)$ space for any $k > 1$. For most applications, the PD term is relatively small and objects are tagged so the space requirement reduces to $2R(1 + 1.5/k)$.

The proofs of these properties are contained in our previous paper [2]. It should be noted that since the emphasis of the algorithm was to prove the time and space bounds we kept the algorithm as simple as possible. This ment that the algorithm has some features that were asymptotically but not practically efficient. Furthermore it did not consider global-variables or program stacks—we assumed activation records and globals were stored in the heap.

4. EXTENDED ALGORITHM

In this section, we present the extensions to the original algorithm necessary for practical efficiency and applicability in an language with stacks and globals. We can classify the extensions into three categories. The first group includes features that are visible to the application and compiler such as the collector's ability to handle globals and programs stacks. The challenge here lies not in adding code to the collector to process globals and stacks but rather to do so in such a way that the collector's scalability and real-time bounds are not compromised. The second group of changes systematically eliminates the unnecessarily fine granularity of the collector which imposes a huge overhead. Finally, there are fundamental extensions to the algorithm which produce significant performance improvements.

4.1 Globals, Stacks, and Stacklets

Globals. Most compilers assign globally accessible variables to static locations in the data segment. Because global locations are fixed at compile time, a procedure can always access them. Like registers, globals are directly accessible and so must be considered a root value when the collection starts. Similarly, a global location must be replaced with its replica value when the collection ends. Unlike the register set though, there may be arbitrarily many globals. Thus the process of turning the collector off (which involves updating global values) may take an unbounded amount of time making any real-time bounds impossible.

This problem can be solved by replicating globals like other heap-allocated object. That is, a global is actually a pair of locations, one containing the primary global value which is accessed by the mutator. The other location is available to the collector for storing the replica global value. Because this alternate location can be modified without effecting the mutator, it can be updated long before the end of the collection. A flag is used to indicate to the mutator which of the two location holds the primary value and this flag is toggled at the end of a collection when the roles of the semi-spaces are reversed (only a single flag is needed for all the global variables). This arrangement doubles space consumption for global locations and possibly imposes a slight penalty when accessing globals. The penalty, if any, is dependent on how globals are normally compiled. In theory, since globals are already accessed indirectly through a base pointer, there should be no cost. In practice, however, taking advantage of this indirection prevents using the globals facilities already present in vendor-supplied assemblers and linkers.

Stacks and Stacklets. Another source of trouble concerning a timely flip comes from stacks of activation records that are used by many compilers. (Implementations that use heap-allocated *read-only* activation records do not have this problem.) As the stack can hold pointer values, these locations must be replaced with their replicas when the garbage collector is turned off. However, the stack can be very deep and disrupt the desired time bounds. In fact, the depth of the stack is a dynamic property and so this problem is arguably worse than that of globals.

Since we cannot wait until the end of a collection to flip all the stack slots, we must replicate the stack as the collection proceeds so that the stack work is distributed over the entire collection. Replicating the stack is, however, difficult since the mutator is continually modifying existing stack locations and pushing and popping stack frames. Instead we break up the stack into fixed-sized stacklets which are linked together with some glue code.

In this system, a deep stack becomes a string of fixed-sized stacklets, at most one of which is active at any time. While the mutator is executing in the active stacklet, the collector has a chance to replicate the other stacklets. By the time the collector is ready to turn off, only the most recent stacklet needs to be processed. Since this stacklet is bounded in size, we can also bound the processing time. During the collection, the collector processes older stacklets before younger ones since younger stacklets are more likely to die and require no replication.

Finally, traditional systems require that the size of a stack be fixed on creation of a thread. Since the precise stack requirements of a thread are generally unknown, programmers typically must estimate a safe upper bound and hope that the stack does not overflow. Safe overestimates waste space or may be impossible for applications that generate many threads or threads whose stack requirements vary greatly. Stacklets are useful in this context as well since they bound the wasted space per stack to the size of one stacklet. In addition, the programmer is not required to compute or guess a safe stack size [16].

4.2 Granularity

The original algorithm was designed for ease of proving correctness, time, and space properties. Unfortunately, asymptotic bounds fail to capture overheads a direct implementation may incur. In fact, the original algorithm invokes the collector whenever the collector allocates space for an object, initializes a field of a new object, or modifies a field of an existing object. For a language which typically inlines these operations such as SML, the increase in code size alone is prohibitive. For example, initializing a field, which normally takes one instruction, would become a function call. Even for languages that do not inline these operations, the cost of a function call overhead for every field initialization, space allocation, and field modification is too high. In fact, the overhead incurred is more than just from a function call, including also glue code for switching from the mutator to the collector as well as increasing the instruction cache miss rate. We next describe two changes which combine to batch up collector work, thus reducing the overhead associated with overly frequent switching.

Block Allocation and Free Initialization. Even though `FetchAndAdd` is a scalable construct in theory [17], using it for every memory allocation is inadvisable because of its expense relative to an ordinary load-add-store sequence. Furthermore, since objects are often of a size comparable to that of a cache line, allocating memory in such a fine grain fashion would split cache lines across different processors, causing false sharing.

Instead of allocating memory directly from the shared heap, each processor maintains a local pool of memory in from-space and, when the collector is on, a local pool in to-space. Objects are allocated from the pools if the request can be satisfied. Otherwise, the local pool is discarded and a new local pool is allocated from the shared heap using a `FetchAndAdd`. Field initialization does not require calling the collector. Such a two-level allocation scheme has several benefits. By greatly reducing the frequency of the more expensive `FetchAndAdd`, the cost of memory allocation is reduced to the usual copying collector allocation code which is short enough that it can be inlined. Cheap allocation is important for functional languages like ML. In addition, cache behavior is improved since related objects, which are those manipulated by one processor, have improved spatial locality.

Since the collector is no longer invoked per allocation or field initialization, context switching costs are much more reasonable. Instead, when the collector is on, an increment of collection work is performed whenever a local pool is discarded. Since the size of the work increment effects the real-time bounds, the size of the local pool must be chosen to balance efficiency and real-time demands.

Finally, local pools make it possible for the space for copying an object to be eagerly allocated before a processor is certain it will be the copier. If it should fail to be a copier, the memory is easily returned to its local pool without causing fragmentation. Without local pools, however, memory already allocated from a shared heap cannot be returned without destroying the contiguity of the unallocated space in the heap. The ability to eagerly allocate space allows the busy-wait in the copy-copy synchronization for small objects to be eliminated.

Write Barrier. The need for a write barrier stems from the collector's incrementality. If the i^{th} field of object x containing pointer value y is updated with pointer value z, the write barrier grays the object y and if x has been copied, performs the corresponding update to x's replica and, in so doing, grays object z. The code associated with these actions is substantial compared to the actual pointer update which is one instruction. Inlining this code is impractical.

Even a function call is likely unacceptable because of the disruption to the instruction cache and processor pipeline and code size blowup. Instead, we elect to only record the relevant information, the triple $< x, i, y >$, deferring the processing for later. Note that the value z is not necessary since it is obtainable from x and i. Further, if the updated location contains a non-pointer value, y may be omitted. This form of recording is similar to the write barrier often used in a generational collector, except that a generational collector needs to record only $x + i$. The memory traffic generated by the write barrier is very regular and well-behaved. In addition, on machines with multi-issue, the write barrier may have a low to zero cycle cost. No write barrier is necessary for the registers or the stacks.

During execution, each thread has a write log for storing the triples $< x, i, y >$ that correspond to the mutations. Whenever the write log is full, the collector is then invoked to process the write log. In fact, the collector may process the log whenever it is invoked, such as when its local pool is exhausted. Like block allocation, the write log serves to batch up write barrier work, eliminating frequent context switches.

Small and Large Objects. In our original algorithm, all objects, regardless of size, were treated in the same way. Because of the presence of large objects and the need to meet real-time bounds, all objects were copied incrementally, one field at a time.

This turned out to have a significant overhead since each field requires reinterpretation of the tag word as well as transferring the object from and to the local stack. Therefore, instead of copying and scanning the fields of a small object one at a time, the entire object is locked down and the object is copied all at once. In addition to reducing the synchronization operations, the small object is in the local stack once and its tag is decoded once. In practice, any object less than 512 bytes can be considered small so most objects are small, making this optimization very important.

Because the original algorithm scans fields one at a time in order, the collection can be incremental even when large objects are present. However, this scheme prevents the collection of any object, even large ones, from occurring in parallel. The processing of large objects can be parallelized by breaking them up into multiple fixed-sized segments and associating a tag with each segment. The tags are stored as extra header words of the object. The size of the segment can be chosen for an appropriate degree of parallelism (*e.g.*, 512 bytes). A smaller segment size allows increased parallelism but creates an increased space overhead due to more segment tags. Also, by using a per-segment tag, the copy-write synchronization is less likely to create contention for large objects.

4.3 Algorithmic Modifications

Reducing Double Allocation. While the collector is on, all objects that are allocated in from-space by the mutator must be copied into to-space. This allocation barrier policy is necessary to preserve the reachability invariant. As with the write barrier, the code for this double allocation, in comparison to the normal allocation, is substantial. Deferring the double allocation is simple and does not even have an additional recording cost like the write barrier. All that is required is to replicate all objects in the current local pool whenever a new local pool is about to be allocated.

In preliminary experiments, however, we found the cost of the double allocation to be substantial. This can be seen in the following analysis. Consider an application which in steady-state has L live data and is running with a fixed-sized heap of H. The liveness ratio is then $r = L/H$. At the start of the collection, there is L live data which needs to be copied by the end of the collection. During the collection, L/k additional data is allocated and hence copied. This increases the effective survival rate from r to $(r + r/k)/(1 + r/k)$. For not atypical values of $r = 0.2$ and $k = 2$, the survival rate is increased from 20% to 27% which increases the collection time by 35%.

To reduce double allocation, we divide collection into two phases, aggressive and conservative. In the first phase (aggressive), we perform collection without double-allocating new objects, so that at the end of this phase only data that was live at the beginning of the collection will have been copied. The second phase (conservative) begins by recomputing the root set and consists of a second much shorter collection during which all new objects are double-allocated. At the end of the second phase, the replica memory graph is complete and the collection can be terminated. The 2-phase scheme greatly reduces double allocation by confining it to a short second phase. In the first phase, L data is copied while L/k data is allocated. Of this, Lr/k is live and so we copy Lr/k data plus the additional Lr/k^2 since we are performing double allocation. The final effective survival rate is reduced to $(r + r^2/k + r^2/k^2)/(1 + r/k + r^2/k^2)$, which, using $r = 0.2$ and $k = 2$ as before, yields 20.7%.

This 2-phase algorithm requires an extra global synchronization and root set computation. However, our experiments show that this cost is more than compensated for by the reduced copying. This optimization does not require a read barrier nor modifying the existing write barrier.

Rooms and Better Rooms. As explained in section 3.1, the shared stack needs a mechanism to ensure that pushes and pops do not occur at the same time. This can be accomplished by creating two *rooms*, a pop room and a push room. A processor may push onto the shared stack only when in the push room and pop from the shared stack only when in the pop room. The rooms mechanism guarantees that each processor may be in one of two rooms (or neither) but at most one of the rooms is non-empty. Thus, concurrent pushes and pops are avoided. The reader is referred to our paper on rooms synchronization [3].

In our original algorithm and formulation of the rooms, a round of collection work involves the following steps in order: entering the pop room, fetching work from the shared stack, performing the work, transitioning to the push room, returning work to the shared stack, and finally exiting the push room. The number and size of the rounds depend on the desired real-time bounds. This scheme has some shortcomings. The time for graying objects is considerable compared to fetching work and a processor trying to transition to the push room has to wait for all other processors already in the pop room to finish graying their objects. The idle time can be significant since there may be significant variation in the time for different processors to gray objects.

These problems can be eliminated by changing to a more relaxed room abstraction in which a processor can leave the pop room rather than having to transition to the push room. Since graying objects is not related to the shared stack, it can be performed outside the rooms. This greatly reduces the time in which the pop room is active and the potential wait time of any processor. This change relies on a new definition and efficient implementation of room synchronizations that does not require processors to go through the rooms in order [3].

A subtle termination problem, however, arises with the more asynchronous rooms. In the original algorithm, since processors can have gray objects only within the pop-work-push cycle, we can detect that there are globally no more objects by having the last processor to leave the push room check that the shared stack is empty. The new version of the rooms places no constraints on when a per-processor local stack is empty. To correctly detect global emptiness, the shared stack must maintain a borrowed counter, updated with `FetchAndAdd`, that indicates how many local stacks have borrowed objects from the global stack. Termination is now signalled when the last processor to leave the push room detects that the shared stack is empty while the borrowed counter is zero.

Generational Collection. Generational collectors were first proposed to reduce pause times [25]. The simplest generational collector divides memory into a *nursery* and a *tenured* space. Objects are allocated from the nursery. When the nursery is exhausted, a *minor* collection is triggered in which the live objects in the nursery are copied (or *tenured*) to the tenured area. When the tenured area is exhausted after repeated minor collections, a major collection is triggered in which the live objects of the tenured area are copied to an alternate tenured area. The advantage of such an arrangement is based on the generational hypothesis which asserts that most allocated objects die shortly after they are allocated. If this is true, then a minor collection is more productive at reclaiming space than a collection in a semispace collector since a smaller fraction of objects needs to be copied. Pauses from minor collections are indeed short but the inevitable major collection still makes generational collectors unsuitable for real-time applications. However, since collections are on average more productive, generational collectors are more efficient than semispace collectors when the generational hypothesis holds. In addition, they provide better data locality for the application.

Unfortunately, adding generations is not a straightforward extension to our collector. In a non-incremental generational collector, a minor collection involves copying all live objects in the nursery (*i.e.*, those objects reachable from the roots). In the presence of mutable data structures, objects in the tenured area may refer to objects in the nursery and such references must be considered roots as well. Since the collector is incremental, these tenured references may not be modified during the collection since the mutator is executing. Instead the referenced objects are copied during the collection and only at the end of the collection are the references themselves modified. In general, the number of references is unbounded and these references cannot all be atomically modified without breaking the real-time property.

The solution is to use two fields for each mutable pointer field. During any particular collection, one field is used by the mutator while the other field is updated by the collector. The roles of the fields are reversed at the end of each collection. This is similar to the technique we described for global variables. Our initial experiments show that this scheme allows the same real-time bounds to be met while giving most of the overall performance improvement of using generations. Further, the benchmarks show that even a modestly-sized array of 10,000 pointer values was enough to make these modifications necessary. The major drawback to this scheme is that field access by the mutator is slightly more expensive and more space is required for mutable pointer fields.

5. EVALUATION

To evaluate the algorithms, we implemented several collectors within the runtime system [6] for the TILT SML compiler [24]. SML is a statically typed, functional language with a module system. The prevalence of closures, algebraic datatypes, and tupling in most SML programs leads to a high allocation rate, providing a tough challenge for a collector. Further, the TILT compiler uses both global variables and stacks and thus requires the modifications described in section 4.1. For our experiments, the TILT compiler been extended with a parallel binding construct `pval and ...`. This construct is like SML's construct `val and ...` except that the bound expressions are executed in parallel.

The collector code consists of approximately 6000 lines of C code and 500 lines of assembly code (mostly glue code). Altogether, we implemented 6 collectors: semispace stop-copy, semispace parallel, semispace concurrent, generational stop-copy, generational parallel, and generational concurrent. Critical functions, determined with profiling, are inlined. Currently the TILT SML compiler as well as the runtime system and garbage collector runs on Alpha workstations running Digital UNIX and on UltraSparc-II workstations running Solaris.

Experiments were performed on a 6-processor Enterprise 3000 server and a 64-processor Enterprise 10000 server. Experiments were performed multiple times and on both machines to assure that the variance in timing was low. Interestingly, these two machines exhibited slightly but consistently different performance characteristics, probably because the machines have different memory subsystems. The data presented in this section comes from executions on the Enterprise 10000.

To obtain reasonable real-time behavior, the runtime system locks down all pages that are part of the heap at the beginning of execution. Even so, context switches or scheduling glitches can disrupt our timings. Because Solaris is not a real-time operating system, we can only cope with this by running at a higher priority and performing the experiments when the machine is not heavily loaded.

5.1 Benchmarks

The experiments were conducted on 15 benchmarks, some of which are standard for SML [24]. They include symbolic processing (life, knuth-bendix, boyer-moore, grobner polynomials, lexgen, frank - tree search with backtracking), numerical applications (fft, pia), large application (the TILT compiler itself), data intensive (merge sort, red-black tree), and parallel applications (convex-hull, barnes-hut, treap manipulation). Some of the benchmarks have large arrays (convex-hull and fft), deep stacks (knuth-bendix), and high mutation rates (frank - tree search, convex-hull, and fft). Figure 1 gives for each benchmark the number of instructions (in millions), the allocation rate (number of kilobytes per million instructions), and the mutation rate (number of kilobytes of modified heap objects per million instructions).

Benchmark	Instruction (Mi)	Alloc Rate (Kb/Mi)	Mutate Rate (Kb/Mi)
pia	1185	72.8	0.05
rbtree	1205	440.7	8.9
convex-hull	1754	45.7	28.6
leroy	1289	124.6	0.00
tyan	3111	68.4	0.29
fft	5747	76.7	11.1
boyer	408	167.2	0.08
pmsort	560	55.4	10.3
tree	6754	12.6	0.01
treap	17600	67.1	0.04
frank	4460	183.5	3.7
lexgen	1753	52.8	1.5
barnes-hut	4271	81.0	0.0
life	533	176.6	0.14
msort	165	225.4	0.20
tilt	16602	72.4	2.93

Figure 1: Benchmark characteristics.

5.2 Cost of Parallelism and Incrementality

To understand the overhead of using a parallel and/or incremental collector, we analyze the costs of various necessary mechanisms by measuring the cost of running on one processor a collector with various features enabled. The basis for our comparison is a non-parallel, non-incremental semispace collector which does not have an explicit data structure for maintaining the set of gray objects.

Figure 2 illustrates the relative time cost of these mechanisms with a stacked bar graph. The bottom component corresponds to the basic `Stop-Copy` collector. We successively take steps toward a parallel, real-time collector by adding in various features. To support multiple collectors, each processor must maintain a local pool of memory with a `space check` from which it allocates space for the replica objects. As infrastructure for load-balancing and traversal control, a `local stack` must be added. Multi-threaded programs support `multiple allocator`(s) with a two-level allocation scheme. `work tracking` is needed to provide both real-time bounds and parallelism. The `shared stack` component includes the costs of rooms and is needed for load-balancing. The `copy-copy` synchronization is necessary for correctness when there are multiple collectors. Finally, `incrementality` is required for real-time bounds.

The time costs of these components have been normalized for each benchmark so that the cost of the base collector is 1. On average, the cost of (unbalanced) parallelism (multiple allocators, the space check, support for multiple allocators, the local stack, and copy-copy synchronization) total 25%. Finally, load-balancing (room synchronization and communication between the local stacks and the shared stack) requires an additional 14%. Overall, the cost of scalable parallelism is 39%. If incrementality is required, an additional 12% overhead is incurred due to double allocation, more frequent context switches, and the write barrier. For each benchmark the total memory available was kept constant across the different collector variants. This means that part of the overhead of the incrementality is due to slightly more frequent collections.

Because of the use of a write barrier, one might wonder how efficient a concurrent collector would be in the context of a language that is more heavily based on mutation, such as Java. We don't have conclusive evidence to answer this question, but we have noticed that in the applications with a reasonable amount of mutations (frank, convex-hull, and fft), the cost of the write barrier was not significant.

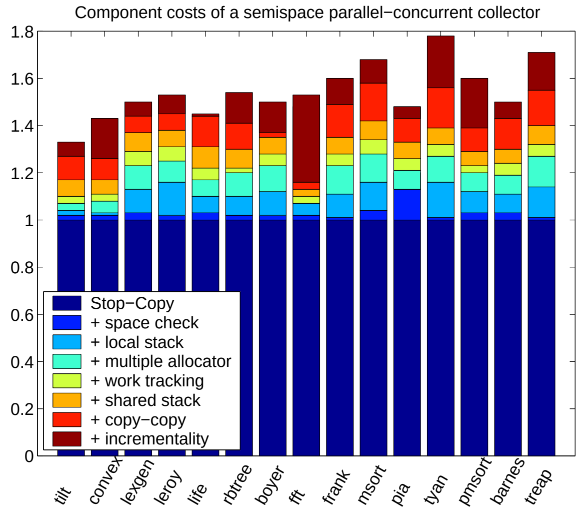


Figure 2: Time costs of parallelism and incrementality. The cost of the various mechanisms are displayed relative to the cost of a non-parallel, non-incremental collector which is normalized to 1.00. The components are listed in top-down order in the legend but bottom-up in the graphs.

As is typical with other parallel algorithms, the parallel version performs worse than its sequential counterpart when run on one processor. However, this overhead (39% in our case) for scalable parallelism occurs once and is more than compensated for even with the addition of a single processor. For example, at 4 processors, assuming a scalability of 3.8 which we do achieve, despite the overhead there is still a net speedup of 2.7 ($= \frac{3.8}{1.39}$).

5.3 Utilization and Maximum Pause Time

As discussed in Section 2.3, maximum pause time alone is too limited for quantifying the degree to which a garbage collector is real-time or incremental. It fails to take into account whether the mutator has reasonable access to the processor and is able to make sufficient progress towards its task. Instead we defined the more general notion of minimum mutator utilization (MMU).

Figure 3 shows the MMU of all benchmarks for the stop-copy collector and the parallel, concurrent collector for two values of k (the rate of collection). The minimum mutator utilization is linearly plotted against time-granularity which is shown logarithmically. As expected, the utilization curves generally increase as the granularity increases (though it is not strictly monotone). At the low end, the utilization falls to zero when the granularity is below the slowest collection for the stop-copy collector or, for the incremental collector, below the largest increment of work. Naturally, this point of minimum granularity is much lower for an incremental collector. At the high end of granularity, the stop-collector provides greater utilization than an incremental collector since the incremental collector has an overhead compared to the stop-copy collector. There is a granularity point below which, if utilization is of concern as in a real-time collector,

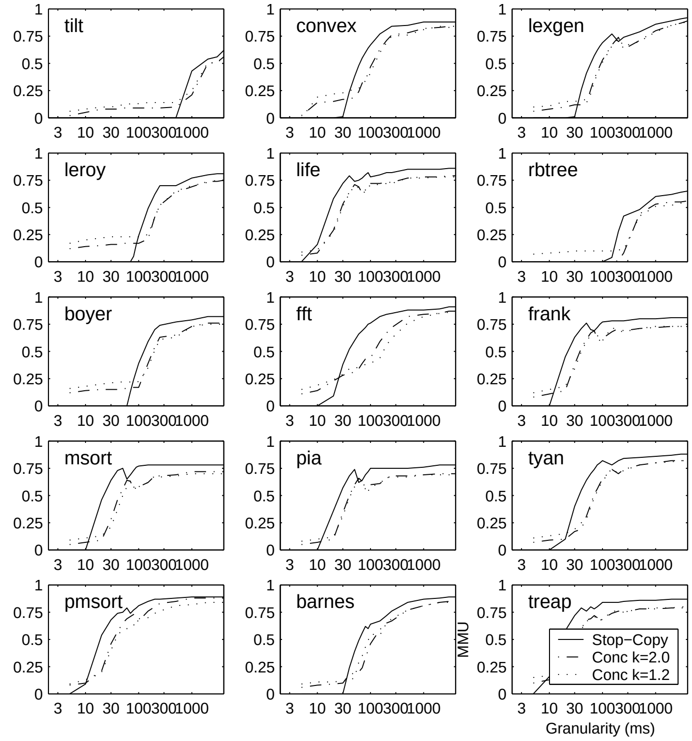


Figure 3: MMU vs granularity (ms) of semispace collectors: non-incremental, incremental (k=1.2), and incremental (k=2.0)

the incremental collector is preferable. This crossover point ranges from 3-10 ms for the smaller benchmarks to almost 1 s for more memory-intensive applications.

For each benchmark the total memory available is held constant for all three collector variants. The parameter k controls the rate of collection and effects the utilization level. Recall that k is the number of units of space collected for each unit allocated. When k is low, less collection work is performed relative to allocation and so the collector runs less often, leading to higher utilization. However, a lower value of k requires more memory. Given that we keep the available memory fixed, lowering k translates to more frequent collection and potentially lower overall performance.

In summary, at the 10 ms granularity, the incremental, concurrent collector provides a minimum utilization of about 10% for $k = 2.0$ and 15% for $k = 1.2$. On the other hand, the stop-copy collector provides 0% utilization at 10 ms granularity for 12 of the 15 benchmarks. The concurrent collector's maximum pause times range from 3 to 5 ms.

5.4 Generations

Next, we examine the effect of generations on utilization level. Figure 4 shows the MMU curve of 4 collectors which are differentiated by whether they are generational and/or incremental. Generally, the presence of generations do not greatly affect the MMU curve. On average, in the non-incremental collectors, generations only slightly lower the maximum pause time. Clearly, to gain significant (worst-case) responsiveness, adding generations is insufficient and incrementality is necessary.

We also note that although it appears from the diagrams that the generational concurrent collector is often less efficient than the semispace concurrent version, this is actually not the case. Because of the locality effects of the generational collector, the mutator runs significantly faster, so the overall time is faster although the utilization is sometimes worse. For efficiency and real-time bounds, both generations and concurrency are required.

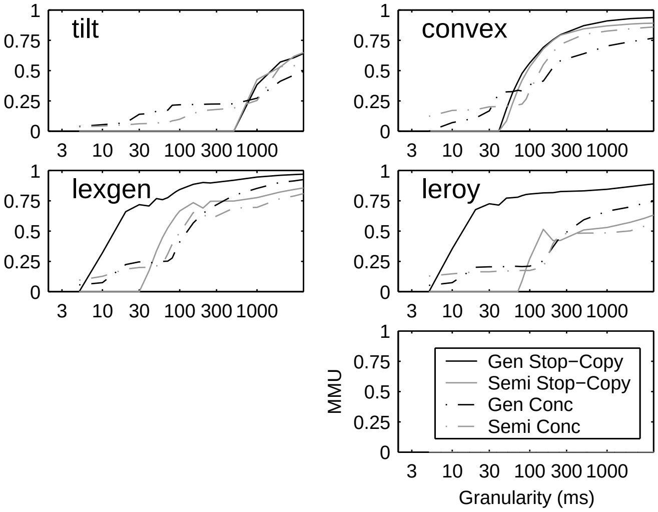


Figure 4: MMU vs granularity (ms) of 4 collectors. Generational collectors are marked with darker lines. Independently, the line for the incremental collectors are dashed.

5.5 Scalability

We consider two types of benchmarks for testing scalability. The first group includes coarsely parallel programs in which there are as many threads as processors, each thread running a sequence of non-parallel benchmarks in a different order. The second group consists of the benchmarks that are already (finely) parallel using a fork-join construct (convex-hull, barnes-hut, and treap). The semispace parallel non-concurrent collector is used for the experiments in this section.

Figure 5 show how the various time components of the application change as the number of processors varies. The top graph corresponds to the coarsely parallel benchmarks and the bottom graph to the finely parallel benchmarks. Each bar in the graphs represents the total normalized work (the normalized sum of the times across the processors) of that benchmark. The 4 components, from bottom to top, are garbage collecting (`GC`), idling during collection (`GC-Idle`), running the application (`Mutator`), and idling during application (`Mutator-Idle`). Linear scalability corresponds to a flat line starting at the single processor case.

For the coarsely parallel benchmarks (top graph of figure 5), the executions compare the effect of load-balancing as the number of processors vary from 1 to 8. All three benchmarks show similar behavior. With almost no interaction among the threads, it is unsurprising that the application scaled almost perfectly with almost no `Mutator-Idle` time. What little mutator idle time is present indicates that some threads finish slightly before others even though each is running the same program. On the other hand, the effect

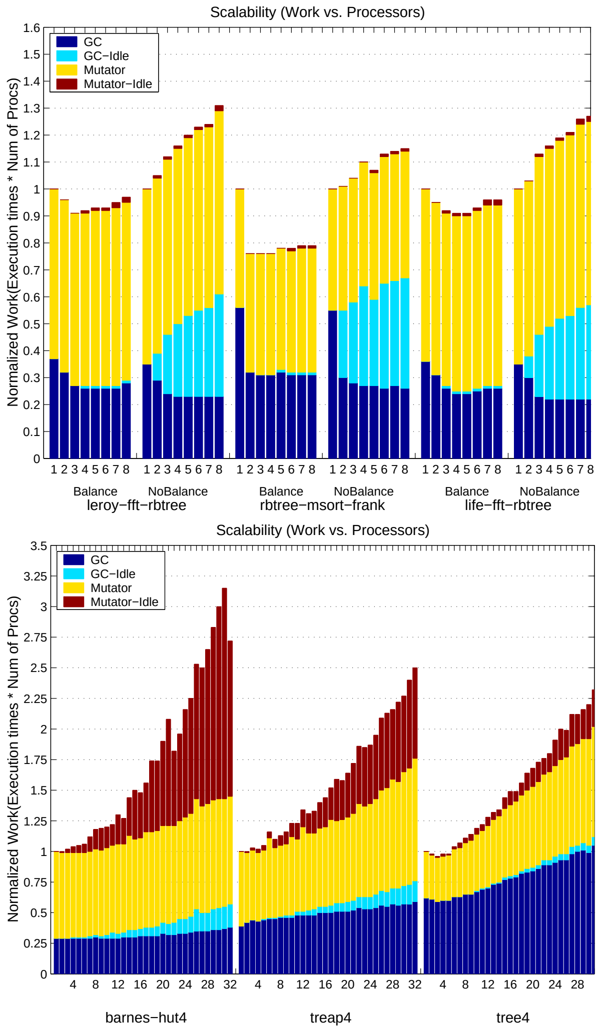


Figure 5: Total processor time spent in the mutator and collector for coarsely (top) and finely (bottom) parallel benchmarks. The components are listed in top-down order in the legend but bottom-up in the graphs.

of load-balancing on the scalability of the garbage collector is pronounced. With load-balancing, there is almost no `GC-Idle` time. However, without load-balancing, `GC-Idle` is comparable to `GC` so we are wasting half our resources without load-balancing. As expected, this effect is more evident as the number of processors increases.

For the finely parallel benchmarks (bottom graph of Figure 5), we examine the benchmarks only with load-balancing enabled. Up to 32 processors, the total collector times (`GC` + `GC-Idle`) increases on average by 86% while the corresponding mutator increase is 246%. In other words, the collector is scaling far better than the application. In terms of speedup, the collector and mutator have respective speedups of 17.2 and 9.2 at 32 processors. At 8 processors, the respective speedups are 7.8 and 7.2. The idle time of the mutator greatly varies depending on the parallel application while the collector shows more consistent behavior. We conclude that our collector scales almost perfectly at 8 processors and at slightly better than 50% linear speedup at 32 processors. In both cases, load-balancing is critical to achieving this.

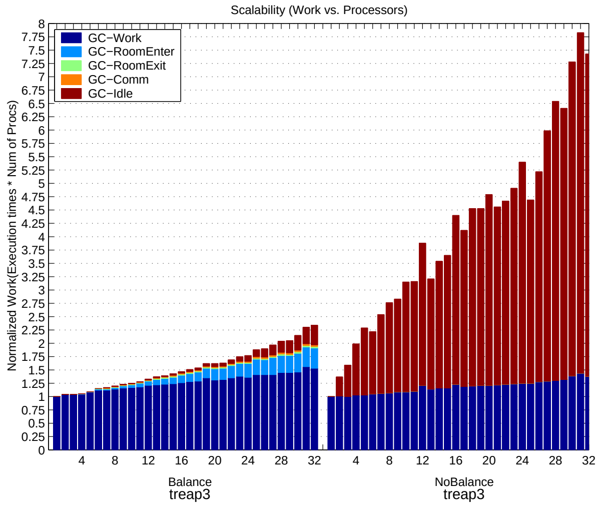


Figure 6: Total processor time spent in garbage collection for the treap benchmark with and without load balancing. The components are listed in top-down order in the legend but bottom-up in the graph.

Next, we examine the effect of two factors on collection scalability in Figures 6 and 7 by considering *only* the time components within the collection. In these graphs, the bottom component `GC-Work` corresponds to when a processor is actively doing collection work such as graying objects, allocating space, copying fields, and so on. The next component up `GC-RoomEnter` corresponds to time spent when a processor is waiting for access to a room. `GC-Comm` represents the time to transfer objects to and from the shared stack while in a room. `GC-RoomExit` is the time for exiting a room. Finally, `GC-Idle` measures time while a processor idles, waiting for work to appear on the shared stack.

Figures 6 shows the effect of load-balancing on the treap benchmark. Since the treap benchmark involves threads operating on the same tree, one might speculate that load-balancing is not needed or much less needed than in the coarsely parallel benchmarks where the threads are unrelated. In fact, load-balancing is still critical. Without load-balancing, the amount of work or time spent in collection is approximately 3 times higher. Other benchmarks (not shown) in this category show similar behavior.

Figures 7 illustrates that dataset size which is proportional to the amount of data copied has a significant impact on scalability. The four groups of data in this graph correspond, from left to right, to dataset sizes that double from one to the next. The times are normalized so that the single-processor case is 1.0 despite changes in total work. At 32 processors, scalability improves from 5.6 to 17.7 as we move from the smallest to the largest dataset.

6. RELATED WORK

Concurrent garbage collection was independently introduced by Steele [23] and Dijkstra [7], both of whom based their work on mark-and-sweep collectors. By extending Cheney's copying collector [4] with a read barrier, Baker pro-

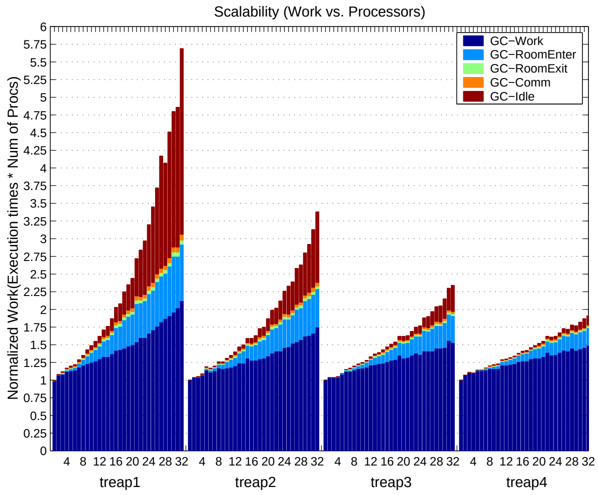


Figure 7: Total processor time spent in garbage collection for the treap benchmark at different dataset sizes. The components are listed in top-down order in the legend but bottom-up in the graph.

posed the first real-time copying collector with bounds on memory use and time [1]. He proved that if each allocation copied k locations then his collector would require only $2(R(1 + 1/k))$ locations. He also suggested how it could be extended to arbitrarily sized structures, although this required both a read and write barrier and also an extra link-word for every structure.

As part of the Multilisp system, Halstead developed a multiple-mutator multiple-collector variant of Baker's basic algorithm for shared memory multiprocessors [18]. In their scheme, every processor maintains its own from- and to-space and traces its own root set to move objects from any from-space to its own to-space. This algorithm has several properties that make it neither time nor space efficient, both in theory and practice. First, the separation of space among the processors can lead to one processor running out of space, while others have plenty left over. In addition to making it unlikely that any space bounds stronger than $2RP$ can be shown (*i.e.*, a factor of P more memory than the sequential version), Halstead noted that this problem actually occurred in practice. A related problem is that since each processor does its own collection from its own root set without sharing the work, one processor could end up doing much more copying than the others. Since no processor can discard its from-space until all processors have completed scanning, all processors will have to wait while the one processor completes its unfair share of the collection.

Herlihy and Moss [19] described a lock-free variant of the Multilisp algorithm. The locks are avoided by keeping multiple versions of every object in a linked list. Although the algorithm is lock-free it cannot make guarantees about space or time. For example, to read or write an object might require tracing down an arbitrarily long linked list.

Endo [12] solved the problem with work imbalance by using work stealing to share the work among processors. His collector was a mark-sweep conservative collector for C++. It was a stop-collect algorithm and therefore made no attempt to be real-time. Endo was able to show good scalability on up to 64 processors of a shared-memory machine (the same one we used), and furthermore noted, as we also found, that load-balancing is critical to achieving scalability. Endo's work was extended by Flood et. al. for a copying collector [15]. We use work-sharing instead of work-stealing because it allowed us to prove our time and space bounds [2]. We expect that we could replace our work-sharing with work-stealing with little practical effect.

Doligez and Gonthier describe a multiple mutator single collector algorithm [9]. They give no bounds on either time or space. In fact the collector can generate garbage faster than it collects it, requiring unbounded memory. More generally, no single collector algorithm can scale beyond a few processors since one processor cannot keep up with an arbitrary number number of mutators. On the other hand the Doligez-Gonthier algorithm has some properties—including minimal synchronizations and no overheads on reads—that make it likely to be practical on small multiprocessors. Domani et. al. describe a generational version of the collector [11], but it has the same scalability issues as the original.

Our replicating algorithm is loosely based on the replicating scheme of Nettles and O'Toole [22]. Beyond the basic idea of replication, however, there are few similarities. This is largely due to the fact that they do not consider multiple collector threads and do not incrementally collect large structures.

Incremental or real-time collectors are based on a variety of techniques, leading to varying worst-case pause times: 2 - 5 ms [13], 15 ms [20], 50 ms [22], and 360 ms [10]. However, direct comparison of worst-case pause times without considering mutator access to the processor can be misleading.

7. ONGOING WORK

Approximately half the loss in scalability arises from imperfect load-balancing. We believe that strategies that are sensitive to both the amount of work that is locally *and* globally available can maintain or improve load-balancing while decreasing room usage (which can have high synchronization costs if overused).

Providing better real-time behavior requires reducing the granularity of the collection increments and/or maintaining or increasing the MMU level. Reduction of granularity can be achieved through adjustments of parameters, further subdivision of stacklet-related work, and improving the context-switching code of the collector. Recently, we have found that scheduling the collection work by combining high-resolution timers with our notion of work can minimize throughput variations so that MMU can be improved. Preliminary figures indicate that we can improve MMU from 15% to about 25% and that pause times can be reduced from 3 to 1.5 ms.

A more systematic and thorough treatment of the work in this paper and our earlier paper [2] can be found in Cheng's doctoral thesis [5].

8. CONCLUSION

We have presented an implementation of a scalably parallel, concurrent, real-time garbage collector. The collector allows for multiple collector and mutator threads to run concurrently with minimal thread synchronization. We believe that our modified algorithm maintains the time and space bounds of our previous theoretical algorithm, which gives some justification for calling the algorithm "real-time". Our experiments and the analysis of the MMU across 15 benchmarks gives further justification.

Acknowledgements. Thanks to Toshio Endo and the Yonezawa Laboratory for use of their Enterprise 10000. Thanks go to the general support of Robert Harper and the FOX project. We are grateful to Doug Baker for his comments on an earlier draft.

9. REFERENCES

[1] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[2] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 99. ACM Press.

[3] Guy E. Blelloch, Perry Cheng, and Phil Gibbons. Room synchronizations. In *ACM Symposium on Parallel Algorithms and Architecture*. ACM Press, July 2001.

[4] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[5] Perry Cheng. *Scalable Real-time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, 2001.

[6] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[7] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[8] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[9] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[10] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[11] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of 2000 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 274–284, May 2000.

[12] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.

[13] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[14] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[15] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *USENIX JVM Conference*, April 2001.

[16] Seth Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California at Berkeley, Fall 1997.

[17] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequqnetial processors. *ACM Trans. on Programming Languages and Systems*, 5(2):164–189, April 1983.

[18] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[19] Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), May 1992.

[20] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 1–9, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[21] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[22] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[23] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[24] Tarditi, Morrisett, Cheng, Stone, Harper, and Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Programming Languages Design and Implementation (PLDI)*, pages 181–192, 1996.

[25] David M. Ungar and David A. Patterson. Berkeley Smalltalk: Who knows where the time goes? In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 189–206. Addison-Wesley, 1983.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.