On Bounding Time and Space for Multiprocessor Garbage Collection

Guy E. Blelloch Perry Cheng

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-6245
{guyb,pscheng}@cs.cmu.edu

Abstract

This paper presents the first multiprocessor garbage collection algorithm with provable bounds on time and space. The algorithm is a real-time shared-memory copying collector. We prove that the algorithm requires at most $2(R(1+2/k)+N+5PD)$ memory locations, where P is the number of processors, R is the maximum reachable space during a computation (number of locations accessible from the root set), N is the maximum number of reachable objects, D is the maximum depth of any data object, and k is a parameter specifying how many locations are copied each time a location is allocated. Furthermore we show that client threads are never stopped for more than time proportional to k non-blocking machine instructions. The bounds are guaranteed even with arbitrary length arrays. The collector only requires write-barriers (reads are unaffected by the collector), makes few assumptions about the threads that are generating the garbage, and allows them to run mostly asynchronously.

1 Introduction

Baker, in his classic paper on real-time collection, derived the first bounds on time and space for uniprocessor garbage collection [3]. In particular he described an algorithm that would never require more than $2R(1+1/k)$ space, where R is the reachable space and k is an adjustable parameter. Furthermore the client program would never be stopped for a collection cycle for more than k constant-time steps. Since Baker's work there have been several designs for multiprocessor garbage collection algorithms [15, 2, 8, 12, 16, 17, 26, 10, 30]. Unfortunately none of the algorithms come with any bounds on the time and space. These algorithms either do not limit space, potentially requiring unbounded memory, or do not limit time, potentially sequentializing the collection. In fact we know of no multiprocessor collector that can claim to be "real-time" in the sense that they can guarantee scalable bounds on how long client threads can be stalled (*i.e.*, that do not grow linearly with the number of processors). In addition to time, we believe that bounding space is important since programs can and do fail due to memory exhaustion, making it potentially as critical to reason about space as about correctness.[1] This problem is exacerbated on multiprocessors since it is hard to experimentally verify that a program will always run within space limits. Because of the nondeterminism in parallel implementations and changes in the number of processors, one successful execution of a program within certain memory bounds does not imply that the program will always run within these same bounds.

In this paper we derive provable bounds on both time and space for a real-time shared-memory multiprocessor garbage collector. As with Baker's algorithm the space bounds are given in terms of properties of the memory graph. We also guarantee that each client process never waits for more than constant time per memory operation for the garbage collector, and our bounds are valid even with arbitrary length objects. The only synchronization primitives we use are `TestAndSet`, `FetchAndAdd`, and a barrier interrupt (only used when starting and stopping the collector), which all have parallel implementations. In addition to proving bounds, we have made an effort to make the collection algorithm practical by minimizing the number of synchronizations, avoiding read-barriers, and incurring almost no overhead when the collector is off (only a test to detect that it is off). We are currently working on an implementation of the ideas within the context of a threaded version

[1]Even for machines with "unbounded" virtual memory it is necessary to keep an application within physical memory to maintain a real-time guarantee.

```
TestAndSet(ptr A) {           FetchAndAdd(ptr A, int N) {
  if (*A == 0) {                Tmp = *A;
    *A = 1;                     *A = *A + N;
    return 0;                   return Tmp;
  }                           }
  else return 1;
}
```

Figure 1: Definition of the **TestAndSet** and **FetchAndAdd** instructions. These instructions operate atomically and have some hardware support in most shared-memory multiprocessors. All code we will show uses a C-like syntax and semantics. Unless marked by **shared**, variable declarations are local.[3]

of the TILT runtime system [6].

The basic algorithm we describe is a copying collector with two spaces. All processors are involved as both mutators (client threads) and collectors, and the two spaces are shared among the processors. The main challenges in achieving the time and space bounds were properly balancing the collection work among the processors, guaranteeing that no operation needs to wait on a synchronization for more than constant time, and incrementally copying arbitrary length objects in a parallel setting. None of these issues have been properly dealt with in previous multiprocessor collectors. The load balancing is needed to guarantee that processors do not have to wait for other processors to complete when the collector starts. The algorithm is incremental and, as with Baker's real-time collector [3], when the collector is running, the mutators (mutator in Baker's case) execute a constant k steps of the collector for each location they allocate. To avoid the expensive read-barriers in Baker's collector, we use a replication scheme similar to the one suggested by Nettles and O'Toole [25], which completely replicates the reachable memory without affecting the memory seen by the mutators.

1.1 Machine model

We assume P processors sharing a single memory. The processors run asynchronously, and each has a constant number of local registers and executes a standard set of single-processor instructions (*e.g.*, register-to-register instructions, branches, and reads and writes to the shared memory). In addition the processors can execute a set of synchronization instructions. For the collector we will only need atomic **TestAndSet** and **FetchAndAdd** instructions—defined in Figure 1—and a simple interrupt that is only used when the collector is started or stopped. In designing our collector we assume that the **TestAndSet** instruction is almost as cheap as a memory reference. On the other hand, **FetchAndAdd** is somewhat more expensive so its use is minimized. These assumptions are true for most shared memory machines. We discuss the choice of these synchronization in Section 8.

We assume the memory is organized as a contiguous set of *memory locations* addressable from $[2 \ldots M + 1]$ where M is the maximum memory size (pointers with value 0 and 1 have special meanings), and assume each memory location can hold at least a pointer. We assume all locations can fit in physical memory, and that memory accesses are sequentially consistent [21]—although accesses to a given location can overlap in time, each access appears to take place atomically sometime between its invocation and its return. The algorithm must run correctly with arbitrary finite delay on any instruction. However, for timing analysis, we use the longest time taken by any instruction as the cost of all instructions (interrupts only occur between instructions and are not counted towards this time). Our machine assumptions are meant to model standard shared-memory multiprocessors such as the Sun Enterprise Servers (up to 64 processors), the SGI Origin (up to 128 processors) or the HP/Convex Exemplar (up to 256 processors).

1.2 Application Model

The application model makes the same assumptions about the processors as the machine model except that the shared memory is accessed through the following four instructions:

Allocate(n)	allocates a new object with n locations and returns a pointer to it.
InitLoc(v)	initializes the next location of the last allocated object of the processor with v.
Read(s, i)	returns the i^{th} location of object s.
Write(s, v, i)	writes v into the i^{th} location of object s.

We require that on each processor every **Allocate**(n) is followed by n **InitLoc**'s, which fill the object starting at location 0. Any number of instructions including **Read**'s and **Write**'s can appear between the **InitLoc**'s, but a processor must fully fill an object before it uses the object or executes the next **Allocate**. We distinguish between a concurrent write and exclusive write model for applications. In the *exclusive write model* we assume no two writes to a given location can overlap in time, while in the *concurrent write model* we allow for an arbitrary number of writes to overlap. Our basic collector algorithm assumes the exclusive write model. For many applications this is reasonable since it avoids the inherent nondeterminism of concurrent writes.

[3]The variant of **TestAndSet** we use leaves the value unchanged if the value is nonzero. This can be implemented with a compare-and-swap if the machine **TestAndSet** always sets the value to 1.

We refer to the contents of all processor registers as the *roots*. We refer to the objects accessible from the roots as the *reachable objects*, and all the locations of those objects as the *reachable locations*. We refer to the number of reachable objects as the *object count* and the number of reachable locations as the *reachable space*. Memory can be viewed as a directed graph with objects as nodes, pointers as edges, and the registers as the roots. Edges are weighted by the length of the object they point to. We define the *depth* of each object by the length of the shortest weighted path from any root to the object, and the depth of the graph is the greatest depth of any object in the graph. We define the *maximum reachable space* of a program by maximizing the reachable space over every program point of all possible executions. We similarly define the *maximum object count* and *maximum depth*.

1.3 Collector Assumptions

We assume the collector can access information from the application through the following instructions:

`Length`(p)	returns the length of the object at position p.
`IsPointer`(p, i)	returns whether position i of p is a pointer.
`v = REG[`i`]`	returns value of application register i of the running processor.
`REG[`i`] = v`	writes value into application register i of the running processor.
`Interrupt`(f)	temporarily interrupts application on all processors to execute procedure f. This is only used to start and stop the collector.

The `Length` and `IsPointer` routines might be implemented by examining a tag for the object. We assume that both take constant time. We designed the algorithm so that it only needs to get pointer information from the object and never directly from a particular location—this avoids requiring tags on every memory location. We permit concurrent `Interrupt`(f) instructions but require that they all specify the same interrupt handler. We do not permit nested interrupts. A processor that issues an interrupt executes the handler immediately. Other processors execute the handler immediately after their current memory instruction. Since all memory instructions take constant time, the interrupts are not indefinitely delayed.

We also assume that interrupts can be turned off by a processor when executing the collector code. In this case the interruption is delayed on that processor until the collector code is completed. The collector code only runs for constant time, so interrupts are not delayed indefinitely.

1.4 Summary of Results

Given an application with maximum reachable space R, maximum object count N, and maximum depth D, we show a collector algorithm that will require at most $2(R(1 + 2/k) + N + 5PD)$ memory locations, for any positive integer constant k. Furthermore, each of the four memory instructions will take at most ck time, for some (small) constant c, and an application process will never be interrupted for more than ck time. Here we define a time unit as the maximum time taken by any machine instruction. As with Baker's collector [3], the parameter k controls the tradeoff between space and time. Baker showed a space bound of $2(R(1 + 1/k))$ for his sequential read-barrier collector for fixed-sized objects. The factor of 2 comes from the two spaces and the $1/k$ comes from the incremental nature of the algorithm. He also described an extension to arbitrarily sized objects that requires an extra header word and hence requires $2(R(1 + 1/k) + N)$ memory.

Baker's algorithm uses read barriers. A write-barrier version using replication requires $2(R(1 + 2/k) + N)$ memory since objects allocated while the collector is on have to be allocated twice. The additional $5PD$ term arises in our collector from three effects: the stack we use to store copy pointers (PD), allocations of large objects can happen before the incremental steps that are counted against the allocation $(2PD)$, and the extra time it takes to traverse the memory graph $(2PD)$. These are described further in the proof of Lemma 4.5. Although the first effect could possibly be removed by overlapping the stack with the data, and the second effect does not appear for small objects, we do not see any way to remove the last. Some PD term seems inherent in any multiprocessor collector. We note, however, that for many parallel applications PD is likely to be much less than R since rapid traversal of data structures in a parallel program implies a relatively shallow memory graph.

In many applications both nondeterminism and the number of processors can affect the maximum reachable space, object count, and depth. One might wonder, therefore, about the utility of being able to bound memory in terms of reachable space if determining the reachable space itself might be nondeterministic. As a partial solution to this problem, however, elsewhere we have shown bounds on the maximum reachable space of parallel programs [4]. This work showed that if S_1 is the maximum reachable space of a serial implementation then the maximum reachable space of the (nondeterministic) parallel version is $S_1 + O(P\mathcal{D})$ where $\mathcal{D}$ is the depth of the computation (*i.e.*, the length of the longest chain of dependency). Similar bounds hold for object counts and depth.

2 The Algorithm

In this section we describe and prove bounds on our basic algorithm, and show the complete code for the algorithm. In practice there are several very important optimizations that should be made to the basic algorithm, which are described in the next section. We leave them out of this section for the sake of simplification.

We assume the memory is divided into two equal-sized spaces, called *fromspace* and *tospace*, which are shared by all processors. Memory is always allocated in tospace and when the memory runs out the garbage collector is started. First, the collector *flips* the roles of fromspace and tospace, and starts a new *collector cycle* which copies the objects that are reachable at the time of the flip from fromspace to tospace. When all reachable objects have been copied the garbage collector turns itself off.

The job of the collector is shared by all the processors. When the collector is running, the `InitLoc` instruction can invoke the collector which will then execute a constant amount of work towards the collection. Since objects are not necessarily of constant size, each object is copied incrementally. When copying an object, space is first allocated for it in tospace, and then its locations are copied over incrementally. For describing our algorithm we use a tri-color abstraction (white, grey, and black) to describe the state of each object during the collection. When a collection starts all reachable objects are white. A white object becomes grey when space is allocated for it in tospace. A grey object becomes black when all its locations have been copied to tospace. When copying or updating a location the collector checks if it is a pointer to a white object, and if so turns the object grey. All black objects therefore can only point to other black or grey objects. If we consider collection as traversing the memory graph, then the black nodes are the fully visited nodes, the grey nodes are the *frontier* nodes used to keep track of what still needs to be traversed, and the white nodes are unvisited.

During each collection the collector makes a complete replica of the memory graph in tospace. We will refer to the original graph as the *primary* graph and the copy as the *replica* graph. Similarly we will refer to the two copies of objects within the graphs as the primary and replica copies. When the collector ends a cycle, the replica graph becomes the primary graph. Both graphs are fully self referential—no object from either contains a pointer to the other, excepting the header words described below. Furthermore only the primary graph is reachable from the registers of the processors. These properties are important for the correctness of the algorithm, and for avoiding read barriers. We refer to these two properties, which are kept throughout the algorithm, as the *replication invariant*. In the work of Nettles and O'Toole [25] this invariant was called the fromspace invariant, but in our version this name is not appropriate since some of the primary objects will reside in tospace. To maintain the replication invariant in light of allocations that occur while the collector is on, two copies are made of all objects that are allocated, one belonging to the primary graph and the other to the replica. Since fromspace is full when the collector starts, both copies are allocated in tospace.

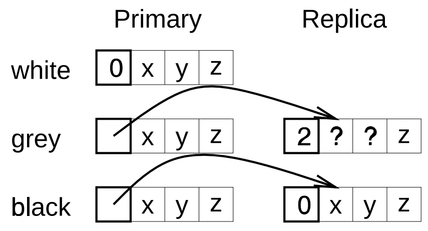

Figure 2: Illustration of an object when it is white, grey and black. For the grey version one of its three locations has been copied.

The algorithm maintains the property throughout the collection that all reachable white objects are reachable from a grey object. This will be referred to as the *reachability invariant*. It is inadequate, for example, for a white object to only be reachable from a register since we need to make sure that the mutators cannot hide part of the reachable graph in their registers (this would prevent us from properly balancing the work). Our basic algorithm also has the property that everything that is reachable when a collector cycle starts will be replicated during the cycle, even if it becomes unreachable from the registers during this time. This property of an incremental collection algorithm is sometimes referred to as a snapshot-at-the-beginning algorithm [19] and is true of various previous incremental collectors [33, 1, 32].

The collector adds an extra location, the *header*, to the front of every object. Given an object pointer A, it can be accessed by reading location $A[-1]$. This header will be used for four purposes: a synchronization variable to determine which processor generates the replica, a *forwarding pointer* from the primary to the replica, a count of how much remains to be copied in the object, and a synchronization variable between writers and the copier. The first two purposes are only used in the primary copy and the second two only in the replica. In time, when an object is white there is only a primary copy and its header contains 0. Once the object turns grey and space for the replica copy is allocated for it in tospace the header of the primary copy points to the replica copy, and the header of the replica copy contains how many locations remain to be copied. When the object turns black (is fully copied) this counter returns to zero. Figure 2 illustrates these properties. Objects allocated while the collector is on are allocated black (*i.e.*, two copies are allocated and the header of the primary points to the replica).

The collector maintains a shared stack called the *copy stack* with pointers to all the grey objects.[4] This is used for sharing the work of collecting among the processors and is kept in the top part of tospace. The memory in tospace is organized as follows.

```
  |     allocated     |   unused   |    stack     |
ToBot                 A            S            ToTop
```

The area between ToBot and A is used for all objects that are copied or allocated in tospace. This area grows upwards from ToBot. New objects are allocated at A by incrementing A by the size of the object using a FetchAndAdd. The area between S and ToTop is used for the copy stack. The stack grows downwards from ToTop and S marks the top of stack. When A = S the collector has run out of memory. If the collector is off when this happens then the collector is started, otherwise a fatal out-of-memory error is reported. In our code the variables ToBot, ToTop, FromBot, and FromTop are local variables denoting the two spaces. In contrast, A and S are shared variables. The motivations for using a stack instead of a standard Cheney style scan queue [5] are given in Section 8.

Figure 3 gives the code for copying a location from the primary to the replica of an object. It takes a pointer, to the primary copy of a grey objects (which either comes from the copy stack or from a local stack as described later), and copies the location specified by the current count, stored in the replica. As well as copying the location, CopyLoc is responsible for turning the object pointed to by the contents of the location grey (by calling MakeGrey), and for decrementing the count. Before CopyLoc actually copies the location, it places a lock on R by setting the header to the negative of $i+1$. This lock prevents $R[i]$ from a mutator update by another processor while it is being copied, and is further discussed when we describe the Write instruction. The lock is unnecessary if the object is immutable.

The MakeGrey function turns an object grey if it is white, and it returns the pointer to the replica copy. Its argument must be a pointer to a primary copy. A TestAndSet instruction is used on the header to check if the object is white. This needs to be atomic since many processors could try to turn an object grey simultaneously, and it is important that only one processor is selected otherwise multiple replicas of the object could be allocated in tospace. The one processor which obtains a 0 from the TestAndSet is the *designated copier*.

[4]Actually, not quite all since the pointers to some grey objects reside in local stacks on each processor.

```
shared int GcOn = 0;
shared ptr A, S;
ptr *ToBot, *ToTop, *FromBot, *FromTop;

void CopyLoc(ptr P)              // P is the prim. copy
{                                // of a grey object
  R = P[-1];                     // Get pointer to
                                 //   the replica copy
  i = R[-1]-1;                   // Get index of location
                                 //   to be copied
  R[-1] = -(i+1);                // Lock loc. to prevent
                                 //   write while copying
  if (IsPointer(P,i))            // If i^th loc. is pointer,
    R[i] = MakeGrey(P[i]);       //   grey before copy loc.
  else R[i] = P[i];              // Else just copy
  R[-1] = i;                     // Unlock object with
                                 //   decremented index
  if (i > 0)                     // If not fully copied,
    PushStack(P);                //   push back on stack
}

ptr MakeGrey(ptr P) {
  if (TestAndSet(&P[-1]))        // Commit to copy F?
    while(P[-1] == 1);           // If uncommitted,
                                 //   wait for forward. addr.
  else {
    L = Length(P);               // Length of object
    R = NewSpace(L+1)+1;         // Alloc. space for replica
    R[-1] = L;                   // Set counter for replica
    P[-1] = R;                   // Set forward. addr.
    PushStack(P);                // Push command to copy
                                 //   object onto stack
  }
  return P[-1];
}

ptr NewSpace(int n) {
  P = FetchAndAdd(&A,n);
  if (P + n > S) {               // Is tospace exhausted?
    if (GcOn) error();           // Out of memory
    Interrupt(CollectOn);        // Start next collection
    NewSpace(n);                 // and try again
  }
  return P;
}
```

Figure 3: Code for copying locations.

We refer to this synchronization as a *copy-copy synchronization*. There are actually three cases when running MakeGrey.

1. The TestAndSet returns 0, in which case the processor becomes the designated copier and allocates space in tospace for the replica, sets the header of the replica copy to the length of the object, sets the header of the primary copy to point to the replica, pushes the object onto a local stack, and returns a pointer to the new replica.

2. The TestAndSet returns nonzero, and the value in the header is a valid pointer (not 1), in which case MakeGrey returns this pointer to the replica.

3. The TestAndSet returns nonzero, but the value in the header is 1. This means that another processor is the designated copier but has not set the forwarding pointer yet. In this case MakeGrey must

```
 1  ptr lA;                   // pointer to last allocated object
 2  ptr lL;                   // length of last allocated object
 3  ptr lC;                   // count of number of locs. filled

 4  val Read(ptr P, int i) { // Reads are uneffected
 5    return P[i];
 6  }

 7  ptr Allocate(int n) {
 8    P = NewSpace(n+1)+1;// Allocate space for primary
 9    R = NewSpace(n+1)+1;// Allocate space for replica
10    P[-1] = R;             // Set primary header to replica
11    R[-1] = 0;             // Set replica header to 0
12    lA = P;                // Set last allocated
13    lC = 0;                // Set count
14    lL = n;                // Set length
15    return P;
16  }

17  void InitLoc(val v) {
18    lA[lC] = v;                  // Init. primary
19    if (IsPointer(lA,lC))
20      (lA[-1])[lC++] = MakeGrey(v); // Init. replica
21    else (lA[-1])[lC++] = v;// Init. replica
22    Collect(k);                  // Execute k copy steps
23  }

24  void Write(ptr P, ptr v, int i) {
25    if (IsPointer(P,i))      // Grey old value
        MakeGrey(P[i]);
26    P[i] = v;                // Write new value into primary
27    if (P[-1]) {             // Check if obj. is forwarded
28      while (P[-1] == 1);    // Wait for forward. addr.
29      R = P[-1];             // Set pointer to replica
30      while (R[-1] == -(i+1)); //Wait if currently
                                 //  copying this loc.
31      if (IsPointer(P,i))
32        R[i] = MakeGrey(v);// Update repl; grey new value
33      else R[i] = v;         // Update repl; no greying
34    }                        //   and write into replica
35    Collect(k);              // Execute k copy steps
36  }
```

Figure 4: Code for the memory instructions assuming the collector is on.

wait until it is set before returning the forwarding pointer. In Section 6 we will show that for short objects this waiting can be avoided by using a compare-and-swap instead of a `TestAndSet`. In any case, as we later show, the wait requires at most constant time.

Figures 4 gives the code for the application's memory instructions when the collector is on. The `Allocate` operation uses `NewSpace` to grab space for the primary and replica copies and sets some local variables specifying where `InitLoc` should write. In particular the variable `lA` keeps track of the most recently allocated object and `lC` specifies how many locations have already been filled. The `InitLoc` instruction fills the next location of both the primary and replica copies and increments `lC`. It also needs to turn the value being written grey since we need to keep the invariant that black objects only point to grey objects. The `Collect(k)` is the incremental step that guarantees that we copy `k` words for every word we allocate. We note that our algorithm is designed so the collector can start while an object is only partially initialized (*i.e.*, when a newly allocated object has not yet been filled by `InitLoc`). This is critical for the real-time constraint.

The `Write` instruction first turns the pointer that is going to be overwritten grey (to maintain the reachability invariant), and then writes the value into the primary copy, and into the replica if it exists. When writing into a grey object the processor responsible for copying the object and the processor writing might need to synchronize, which will be referred to as a *copy-write synchronization*. A potential problem arises when writing into a location that is currently being copied and is illustrated by the following ordering of memory instructions.

Mutator	Copier	replica	primary
	read primary	–	a
write primary		–	b
write replica		b	b
	write replica	a	b

However, this ordering is impossible because line 30 in the `Write` instruction waits for the copy to complete before writing the value. Both `while` statements in `Write` will wait at most constant time.

There are several differences between `InitLoc` and `Write` that make `InitLoc` cheaper. First since `InitLoc` is writing over uninitialized data, it need not (and must not) turn the old pointer in the location grey, otherwise it could capture some unreachable data or try to dereference a non-pointer. Second `InitLoc` need not check whether the object has a replica since newly allocated objects always have a replica. Third, the collector algorithm is designed so that if the collector starts while an object is being filled, only the initialized locations are copied (see Figure 7 and lines 17–21 in Figure 6). This avoids capturing unreachable data and eliminates the need for a copy-write synchronization in `InitLoc`.

Figure 5 gives the code for the `Collect(k)` function. It executes k `CopyLoc` operations. A critical part of our algorithm is sharing the copy work among the processors and hence the use of the shared copy stack. However to reduce the use of the `FetchAndAdd`, allow for more local optimization, and increase locality, most of the work is taken from a local per-processor stack (`LS`). As long as there is work on the local stack `Collect` need not go fetch from the shared copy stack. After copying `k` locations the `Collect` routine places any remaining work back into the shared copy stack. The most involved process is proper insertion and deletion from the shared copy stack. To do this we separate in time removing from the stack from inserting into it by guaranteeing that no two processors can simultaneously execute code between lines 4–16 (the `SynchStart` and `SynchMid`), and between lines 16–21 (the `SynchMid` and `SynchEnd`). However, we permit multiple processors

```
ptr LS[k];                       // local stack
int li = 0;                      // local stack pointer

void Collect(int k) {
  SynchStart();
  for(i = 0; i < k; i++){
    if (li == 0) {               // If local stack empty,
      L = FetchAndAdd(&S, 1);    //   try to grab
                                 //   from shared stack.
      if (L >= T) {              // If shared stack empty,
        FetchAndAdd(&S, -1);     //   readjust stack
        break;                   //   and break
      }
      else LS[li++] = *L;        // Move job to local stack
    }
    CopyLoc(LS[--li]);
  }
  SynchMid();
  P = FetchAndAdd(&S,-li)-li;    // Move copy jobs
  for (i = 0; i < li; i++)       //   from local stack
    P[i] = LS[i];                //   to shared stack
  li = 0;
  SynchEnd();
}

void PushStack(ptr P) {
  LS[li++] = P;
}
```

Figure 5: The `Collect` function.

in each section. The implementations of **SynchStart**, **SynchMid**, and **SynchEnd** are described in Section 3. **SynchEnd** is also responsible for turning the collector off when nothing remains to be copied.

Figure 6 shows the code for starting and stopping the collector. The **synch** routine, which uses two shared variables **cnt** and **round**, is a barrier synchronization that blocks a processor until all processors have reached the barrier. We surround modifications to the shared variables **GcOn**, **A**, and **S** with **synch** to ensure that there is a consistent view of these variables. As mentioned earlier, when the collector is started, only the initialized locations of the last allocated object need to be copied. We do this by setting the replica header to the last initialized location (**lC**) rather than the object length (**lL**). The last **synch** in **CollectOn** is needed to ensure that all copy jobs are in the shared copy stack before collection begins. When the collection is over, the registers and the last allocated object are replaced by their replicas by examining their headers. The reachability invariant guarantees that all objects, including those referenced by registers, are replicated since no grey (and therefore white) objects can exist when the shared and local copy stacks are empty.

In designing the collector, we have separated the handling of the copy stack (pointers to the current grey objects) from the rest of the algorithm. This separation allows for different implementations of the copy stack. For example, on a small number of processors, using a lock for every push and pop operation from the copy stack might be sufficient.

```
shared unsigned int cnt = 0, round = 0;

void synch() {
  curRound = round;
  self = FetchAndAdd(&cnt,1);
  if (self == (numProc-1)) {
    cnt = 0;
    round++;        // assumes modular arithmetic
  }                 // round must have at least 3 states
  while (round == curRound);
}

void CollectOn() {
  synch();
  GcOn = 1;
  swap(&ToBot,&FromBot); // Flip from and to space
  swap(&ToTop,&FromTop); //
  A = ToBot; S = ToTop;
  synch();
  R = NewSpace(lL+1)+1;   // Allocate space for the replica
                          //   of the last allocated obj.
  lA[-1] = R;             // Set forwarding addr
  R[-1] = lC;             // Set # of locs. to be copied
  if (lC > 0) PushStack(lA); // Push job onto local stack
  for (i = 0; i < rsize; i++)
    if (IsPointer(REG,i))
      MakeGrey(REG[i]);  // Make root objects grey
  P = FetchAndAdd(&S,-li)-li; // Move jobs from local
  for (i = 0; i < li; i++)    //   stack to shared stack
    P[i] = LS[i];
  synch();
}

void CollectOff() {
  synch();
  for (i = 0; i < rsize; i++)
    if (IsPointer(REG,i))  // If register is a pointer,
      REG[i] = REG[i][-1]; //   replace with replica
  lA = lA[-1];
  GcOn = 0;
  synch();
}
```

Figure 6: Code for starting and stopping the collector.

3 Implementing the Shared Stack

To access the shared copy stack without conflicts the collector separates in time pushing onto the stack and pulling off of it. We see no practical way to just use the **FetchAndAdd** to push and pop simultaneously since the **FetchAndAdd** does not permit incrementing the stack counter and inserting the stack element atomically, nor decrementing it and checking for underflow atomically. This leads to severe push-pop synchronization problems. We note that the goal is to design a technique that scales to a large number of processors as long as our machine implements the **FetchAndAdd** in parallel (see Section 8). Locking the queue, or using an opportunistic synchronization like load-linked/store-conditional would both sequentialize access. For a small number of processors, however, such synchronizations would probably work fine. Assuming the **FetchAndAdd** is parallel, the technique we describe scales to any number of processors without sequentializing access.

To separate the pushing and popping in time our algorithm uses a gated synchronization to synchronize among the processors that executed a collector step. We use two gates that lead to two separate regions, the first of which is used for pulling values off of the copy stack and executing the copies (the code between the `SynchStart` and `SynchMid` in Figure 5) and the second for pushing values back onto the stack (the code between the `SynchMid` and `SynchEnd`). The synchronization guarantees that although any number of processors can be executing code simultaneously within either of the regions, no two processors can be executing code in the two separate regions simultaneously. The two *gates* hold processors from continuing past `SynchStart` or `SynchMid` until appropriate. Each gate alternatively opens and closes during a round of the collection. Any subset of the processors that enter the first gate while it is open will participate in that round of the collection. Any processor that tries to enter while the gate is closed must wait until it reopens and will participate in a future round.

For the first gate we imagine a turnstile (`Tr1`) followed by a gate (`Gt`) which opens into the first region. The `SynchStart` is implemented by having the processor look past the turnstile to see if the gate is open and if it is the processor passes through the turnstile incrementing it (using a `FetchAndAdd`). When passed the turnstile it checks the gate again and if it is still open it enters the region, otherwise it backs out through the turnstile decrementing it. The code is given in Figure 8. Repeated backing out and retrying is necessary for correctness since a processor may require multiple rounds before successfully completing `SynchStart`. For timing analysis, however, the retry code can only happen once to a processor if we assume that lines 5 through 15 of `Collect` take longer than one iteration of line 7 of `SynchStart`.

Once inside the first region (between the `SynchStart` and `SynchMid`) any number of processors can take elements from the stack and do other work. In our case the code executes k copies and stores any new grey objects into its local stack. These may be eventually moved back into the shared stack.

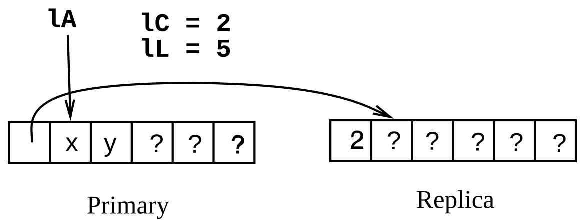

Figure 7: Illustration of how the object that is currently being filled is forwarded when the collector starts. The picture illustrates the primary and replica copies immediately after `CollectOn`.

```
 1  shared int Gt = 0, Tr1 = 0, Tr2 = 0;

 2  int SynchStart() {
 3    while(Gt);                  // Wait until gate opens
 4    FetchAndAdd(&Tr1, 1);       // Increment turnstile
 5    while(Gt) {                 // If gate closed,
 6      FetchAndAdd(&Tr1, -1);    //   decrement turnstile
 7      while(Gt);                //   and try again
 8      FetchAndAdd(&Tr1, 1);
 9    }
10  }

11  void SynchMid() {
12    Gt = 1;
13    FetchAndAdd(&Tr2, 1);
14    FetchAndAdd(&Tr1, -1);
15    while (Tr1 > 0);
16  }

17  void SynchEnd() {
18    i = FetchAndAdd(&Tr2, -1);
19    if (i == 0) {
20      if (*S == ToTop) {          // If no grey objects left
21        Gt = 0;                   //   open gate and
22        Interrupt(CollectOff);    //   turn collector off
23      }
24      Gt = 0;                   // Open gate
25    }
26  }
```

Figure 8: Synchronization operations

The `SynchMid` is implemented by having the processor close the first gate, increment the second turnstile (`Tr2`), decrement the first turnstile (`Tr1`), and wait until there are no processors left in the first region (`Tr1 = 0`). Note that the first processor that executes the `SynchMid` code closes the first gate, after which no more processors can enter the first region until the gate is reopened. If each processor spends constant time in the region, in constant time after the gate is closed all processors have left the first region, at which point all processors can enter the second region. Once in the second region (between the `SynchMid` and `SynchEnd`) the processors can push the grey objects they are holding on their local stack onto the shared stack.

The `SynchEnd` is implemented by decrementing the second turnstile and checking if the count has gone down to 0, which if true indicates that the processor is the last to leave the region. If a processor is the last to leave, it reopens the gate to the first region and the collector is ready for the next round. Again, if each processor spends constant time in the region, in constant time after the second gate is opened everyone has left the second region and the gate to the first region is reopened (effectively closing the gate to the second region). A round of the collector (from when the first processor enters the first region until the last one leaves the second region) therefore takes constant time (assuming the code executed in each region takes constant time).

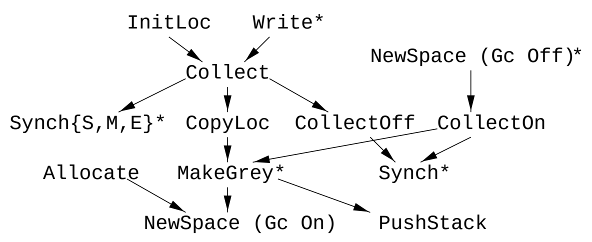

Figure 9: The call graph for the collector routines. The transitive edges have been left out.

4 Time and Space Bounds

We first consider the time bounds. Figure 9 shows the call graph for all the collector routines. The functions at the bottom (`PushStack` and `NewSpace` when the collector is on) take constant time since they only call primitives, or potentially `error`, which will terminate the program. The routines marked with a * are the only ones which have control structures that might loop (or possibly recurse) for more than a constant number of iterations. In the following discusssion recal that any interrupt received while executing a collector function is delayed until the function completes.

Lemma 4.1 *The* `while` *loops in* `MakeGrey` *and* `Write` *take constant time, and therefore* `MakeGrey` *and* `CopyLoc` *take constant time.*

Proof: A header of the primary copy of an object will remain with value 1 for constant time since the processor that sets it to 1 (the designated copier) only executes about 10 instructions before resetting it to the forwarding address. Other processors can therefore only wait constant time in the `while` loop in `MakeGrey` (Figure 3, line 17) and the first `while` loop in `Write` (Figure 4, line 28). This implies that `MakeGrey` and hence `CopyLoc` take constant time. For the second `while` loop in `Write` (Figure 4, line 30), `R[-1]` can have a particular negative value only for constant time since only `CopyLoc` can set it to a negative value and `CopyLoc` takes constant time. ∎

Lemma 4.2 `Collect(k)` *takes at most ck time, for some constant c.*

Proof: Executing k copy steps (calls to `CopyLoc`) takes time proportional to k by Lemma 4.1. Putting the up to k values back onto the copy stack takes time at most proportional to k since it just involves a single `FetchAndAdd` and a loop to put them in. Finally, we use properties of the shared stack described in Section 3 to bound the time of the synchronizations. A processor will wait at `SynchStart` for no more time than required by another processor to copy k locations, plus the time to put the values back on the copy stack. Also, it will wait at `SynchMid` for no more time than required by another processor to copy k locations. Finally, `SynchEnd` requires no waiting at all assuming the normal case in which the collector does not finish. We show below that when the collector does finish, the interrupt handler code will take at most constant time. Therefore the total time for `Collect(k)` is proportional to k ∎

Theorem 4.3 *Each of the memory instructions will take at most ck time, for some constant c.*

Proof: The function `Write` takes at most ck time since it calls `Collect(k)` which takes time proportional to k (Lemma 4.2) and its two while loops take constant time (Lemma 4.1). Similarly for `InitLoc`. `Allocate` takes constant time since it has no loops and only calls `NewSpace`. `Read` does a direct read. ∎

Theorem 4.4 *A processor will never be interrupted by* `CollectOn` *or* `CollectOff` *for more than ck time, for some constant c.*

Proof: When an interrupt is initiated the interrupt handlers `CollectOn` and `CollectOff` on each processor will have to wait until all processors complete their current memory instruction (if running one) because of the barrier synchronization (`synch`) at the start of each handler. This will take time at most proportional to k. Since we assume there are only a constant number of registers, turning them all grey in `CollectOn` and forwarding them in `CollectOff` will also take at most constant time. ∎

Theorems 4.3 and 4.4 together guarantee that the client (mutator) processors never have to wait for more than constant time for the collector. If concerned that the "constant" for the interrupt might be large (depending on the number of registers), it is easy to show that no more than 2 interrupts will occur every M/P instruction cycles, where M is the memory size and P is the number of processors.

We now consider bounds on space. In the following discussion lower case r, n, and d will refer to the properties of the memory graph at a particular time, while upper case R, N, and D refer to the maximum over time.

Lemma 4.5 *If when the collector starts, the memory graph has r reachable space, n objects, and d depth, then the maximum space required in tospace before the collector is turned off is bounded by $r(1 + 2/k) + n + 5Pd$, where P is the number of processors.*

Proof: The replicas of the reachable objects along with their header require $r+n$ locations. We show that at most $2r/k + 4Pd$ locations can be allocated while copying, and that the copy stack requires no more than Pd space. We will count each pair of locations that are allocated (the primary and replica copies) against the `InitLoc` that fills them. Since the memory is actually claimed at the `Allocate` before the `InitLoc`, and each processor can in the worst case allocate d locations before filling them, this accounting will allow $2Pd$ extra locations to be created beyond our count. This will account for $2Pd$ out of the $2r/k + 4Pd$ mentioned above.

Now consider the s collector rounds taken while the collector is on (a round is one pass of `Collect(k)` from `SynchStart` to `SynchEnd`). We denote by p_i ($p_i \leq P$) the number of processors that are executing the `Collect(k)` during round i ($0 \leq i < s$). On round i at most $2p_i$ locations will be "allocated" since only some of the processors executing the `Collect(k)` come from an `InitLoc` and each of those allocates a pair of locations (recall that they are allocated in an amortized sense since the space is actually allocated by the previous `Allocate`) .

Consider the memory graph in which we view each object as a chain of nodes each representing one of its locations. Since our algorithm copies the last location of an object first, we will view the chain as starting at the end of the object. Each node (location) in this new graph will have as its descendents the previous location in the object, and the first node of the chain of the object pointed to by the location (assuming it contains a pointer). Define all the locations at depth l in this graph as being on level l (depth is defined as the shortest path assuming unit weight edges from any root of the graph). It is not hard to show that the graph has at most d levels. We say a level is completed if all the locations on that level have been copied. After a round i define l_i as the level such that level l_i and all previous levels are completed, but level $l_i + 1$ is not completed. Since all locations at level l_i are copied, all uncopied locations in level $l_i + 1$ must be in the copy stack as the next location to be copied by their respective object. To see this, consider the two cases. If the node is not the first in a chain then its parent in the memory graph is the previous location in the chain, and since the corresponding location has been copied, the location in question must be in the copy stack as the next to be copied (see line 12 in Figure 3). If the node is the first in a chain and its parent has been copied, then the `MakeGrey` executed on this parent would have put this object in the copy stack (line 22 in Figure 3).

Since all the uncopied locations in level $l_i + 1$ are in the copy stack, this implies that on round $i + 1$ the collector will either complete level $l_i + 1$ by processing all the elements in the copy stack, or copy at least kp_i locations (since each p_i will copy k locations unless the copy stack empties). Out of all the rounds, at most d of them can finish a level since there are only d levels. These d rounds can allocate at most $2P$ locations each for a total of $2Pd$. For the rest of the rounds every kp_i locations copied will allocate at most $2p_i$ locations, so for copying a total of r locations, at most $2r/k$ locations will be allocated. The total allocated will therefore be $2r/k + 2Pd$. The bound on the size of the stack can be made by a related argument.

The total space is therefore $(r + n) + 2Pd + (2r/k + 2Pd) + Pd = r(1 + 2/k) + n + 5Pd$. ∎

Theorem 4.6 *Given an application with maximum reachable space R, maximum object count N, and maximum depth D, our collector algorithm requires at most $2(R(1+2/k)+N+5PD)$ memory locations, for any positive integer constant k.*

Follows from Lemma 4.5 which bounds the space for each collector cycle.

5 Correctness of the Write

Here we prove the correctness of the `Write` instruction since it involves one of the most delicate synchronizations in the algorithm, the copy-write synchronization, and because the proof sheds light on the particular ordering of the instructions we use.

Theorem 5.1 (Copy-write) *When the collector finishes, all `Write` instructions are correctly duplicated in the replica.*

Proof: In the following discussion we consider a `Write` as having occurred when the primary copy is written. This write is atomic. We consider a replication as having occurred when the forwarding pointer is written into the primary's header. This write is also atomic. Under these definitions of occurrences, all pairs of a write and a replication are strictly ordered. We consider the two cases in which a write occurs before the object being written into is replicated, and when the write occurs after the object is replicated. In the first case, the eventual copying of the object into the replica will happen strictly after the primary object is written to, so the copy will properly duplicate the value of the write.

The second case is more involved since copying the object and writing to it can overlap in time, as discussed previously. We show that all interleavings of instructions between the `CopyLoc` and `Write` will properly update the replica. When writing to an object that has already been replicated, the forwarding pointer has been installed since by definition the forwarding pointer was set before the write on line 26. Therefore we can assume the `if` on line 27 is entered. Also the `while` on line 28 always falls through and the value of `R` set on line 29 is always a pointer to the replica, so we can ignore how these instructions are interleaved with the `CopyLoc` instructions. For the case that the location contains a pointer, the `CopyLoc` and `Write` functions reduce to the following:

```
CopyLoc                        Write

(a) R[-1] = -(i+1);            (x) P[i] = v
(b) R[i]  = MakeGrey(P[i]); (y) while (R[-1] == -(i+1));
(c) R[-1] = i;                 (z) R[i] = MakeGrey(v);
```

We analyze the interleavings of the above lines with the relations $\rightarrow$ and $\Rightarrow$. We define $\alpha \Rightarrow \beta$ to mean α finishes execution before β begins execution and $\alpha \rightarrow \beta$ to mean α begins execution before β begins execution. Both relations are transitive. In addition, $\alpha \Rightarrow \beta \rightarrow \gamma$ implies $\alpha \Rightarrow \gamma$.

If (a) $\Rightarrow$ (y), then (y) cannot complete execution until (c) executes since (c) is atomic. However, (y) $\Rightarrow$ (z) so that (c) $\Rightarrow$ (z). Thus, the mutator's write is last and not lost.

Otherwise, we have (y) $\rightarrow$ (a). Since (x) $\Rightarrow$ (y) and (a) $\Rightarrow$ (b), we have (x) $\Rightarrow$ (a) $\Rightarrow$ (b). If (x) precedes (b), that means the copier reads the updated value. Therefore, whether the mutator or the copier performs the final write is not important. That is, whether (c) or (z) executes last, the replica will contain the same updated value. ∎

6 Optimizations and Issues of Practicality

Here we describe several optimizations to the basic algorithms, many of which should certainly be used in a real implementation.

Block Allocations. Instead of using a `FetchAndAdd` instruction to get every object from the shared space (Figure 3, line 28) in practice it would make more sense for each processor to grab a block of space at a time and allocate small objects from that block until it runs out. In addition to reducing the number of `FetchAndAdd`'s, it will take advantage of spatial locality (*i.e.*, put related cells on the same cache line). Less obviously we can avoid spin-waiting for the forwarding pointer in the `MakeGrey` routine. If the processor knows the location in which it is going to place an object if it does become the designated copier, then it can use a compare-and-swap instead of a `TestAndSet`. This optimization comes at a small cost of some wasted space since blocks might not be filled to capacity.

Avoiding tests in `InitLoc`. In many languages that use GC, such as Lisp or ML, the `InitLoc` is so common that we would not want to include in the code a test for whether the GC is on along with a function call for when it is. This could seriously increase code size, and would slow down the client even when the collector is off. To avoid the test (at least for small objects), a blocking scheme in conjunction with the block allocations optimization can be used. The idea is to only `Allocate` and `InitLoc` a primary copy into the current block no differently than when the collector is off. When the block runs out, the processor then allocates the replica, copy the contents from the primary, and run the `Collect` routine. By controlling the size of the blocks the time for copying the blocks can be bounded.

Reducing the cost of double allocations. When the collector is running it allocates two copies of all objects (primary and replica). The double allocation can be avoided for objects that do not contain pointers and for small, immutable objects whose descendants have the same property. Some modifications to `CopyLoc` are required when used on such objects. Finally, even with double allocating, primary copies can be allocated to a third space which can be discarded when the collector stops. This scheme reduces the factor in the space bound from $(1 + 2/k)$ to $(1 + 1.5/k)$ since this third space is now shared between the two other spaces.

Improving the depth bounds. The bounds we defined were based on the depth being the weighted sum of edges. If a program has long arrays the PD term could be a bottleneck in the space. The following modification allows one to weigh each edge by the logarithm (base 2) of the size of the object it points to when calculating the depth. It comes at the cost of requiring an additional location for every k mutable locations, and extra space on the copy stack. For such objects, the copy stack stores not only a pointer to the object to copy, but also a region (offset and length) to copy. Such jobs from the copy stack are directly copied as before if the region is less than k long. Otherwise, the region is subdivided and the two resulting jobs are pushed back onto the stack. Since many processors can now be copying the same object, a single write lock for the whole object does not suffice. Instead, we now need a lock for each region that is sequentially copied. This increases the size of large arrays by a factor of $1/k$.

Avoiding the header word. In certain cases we can avoid the header word. Nettles and O'Toole's suggested in their replicating algorithm that if all objects have immutable tags, these tags can be overloaded to serve as the forwarding pointer from the primary to the replica. In particular when an object is copied the tag is replaced with the forwarding pointer, and when the client reads the tag it reads it indirectly through the pointer. In our multiprocessor algorithm, however, the header serves many more purposes—a counter for the number of locations that remain to be copied, and a location used for copy-copy and copy-write synchronizations. This makes the overloading more difficult. However, the count can be stored on the copy stack with the pointer (surely an overall saving of memory). The copy-copy synchronization can be implemented by reading the tag, writing it to the potential replica (assuming we are using block allocation), and using a compare-and-swap to replace it and check if the processor actually became the designated copier. For small objects the copy-write synchronization can be implemented by reserving a few bits in the tag. So for all but large objects, where the relative cost is insignificant, the header word can be avoided.

Copying small objects. Copying small objects can clearly be specialized to the object by possibly copying all the locations at once, avoiding the pointer check, and avoiding the copy-write synchronization on locations within the object which are known to be immutable.

7 Related Work

Real time collection was introduced by Steele [29] and independently by Dijkstra [9]. Both suggested methods based on mark-and-sweep collection. Baker suggested the first real-time copying collector and the first bounds on memory use and time [3]. It was a real-time version of Cheney's copying collector [5] and used a read-barrier. He proved that if each allocation copied k locations then his collector would require only $2(R(1+1/k))$ locations. He also suggested extensions to arbitrarily sized objects, although this required both a read and write barrier and requires an extra link-word for every object (as ours does). Baker did not consider any multiprocessor extensions to his algorithm.

As part of the Multilisp system Halstead developed a multiple-mutator multiple-collector variant of Baker's basic algorithm for shared memory multiprocessors [15]. In the algorithm every processor maintains its own from and tospaces and traces its own root set to move objects from *any* fromspace to its tospace. This algorithm has several properties that make it neither time- nor space-efficient, both in theory or in practice. Firstly, 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 property is problematic in his Multilisp implementation. 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 fromspace until all processors have completed scanning, all processors might have to wait for one processor to completes its unfair share of the copy. Another problem with the Multilisp algorithm is that it uses locks to make sure that every object is copied only once. In the worst case this could effectively sequentialize the collection. In practice they did not observe any performance degradation based on the locks, but they only ran their experiments for up to 8 processors and had no real-time constraints. Finally, they did not discuss how to handle arbitrarily sized objects.

Herlihy and Moss described a lock-free variant of the Multilisp algorithm [16]. The locks are avoided by keeping multiple versions of every object in a linked list with the most current at the end. Although this algorithm is lock-free it still makes no guarantees about space or time. In particular, reading or writing an object might require tracing down an arbitrarily long linked list. Like the Multilisp algorithm, this algorithm does not not properly share the space or work of the collectors.

Lins described a parallel cyclic reference counting collector for a shared memory architecture [22]. Although his algorithm is concurrent, it is not work efficient. It is possible for one processor to perform all the recursive deletes while the remaining processors do not participate. This can potentially result in violation of space bounds or in stalling of processors that are waiting for free space.

Doligez and Gonthier described a multiple-mutator single-collector algorithm [10]. They give neither bounds on time or space. In fact the collector can generate garbage faster than it collects it, thus requiring unbounded memory. More fundamentally, no single collector algorithm can hope to scale beyond a few processors since the single collector cannot keep up with an increasing 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 a small numbers of processors.

More recently, Endo has proposed a parallel mark-sweep algorithm [11]. Although his algorithm uses work-stealing to distribute the load, it is neither concurrent nor space-efficient.

Many collectors for distributed memory parallel machines have been suggested. Jones and Lins give a good summary [19]. To our knowledge none has tried to prove any bounds on time or space.

Our replicating algorithm is loosely based on the replicating scheme of Nettles and O'Toole [25, 26]. Beyond the basic idea of replication, however, there are few similarities. This is mostly because they do not consider multiple collector threads and do not incrementally collect large objects.

8 Discussion of Design Decisions

We went through several variants of the algorithm and model before settling on the one described. Here we discuss some of the issues.

Using a stack. Baker's algorithm, as well as most uniprocessor copying collectors, use a Cheney style scan queue instead of a stack to maintain the grey objects (the current frontier of the graph). The idea is to keep the tospace ordered with black objects first and grey objects next. The algorithms can then just iterate by taking a grey object off the bottom of the grey area (adjacent to the last black object), turn it black, and put any new grey objects that are created at the top. We originally tried to use such a queue since it would avoid the extra space required by our copy stack. We however found that it was important to separate the copy stack from the allocated space for several reasons. First, properly synchronizing the Cheney queue so that processors taking jobs from one end and processors placing data on the other do not conflict is more complicated than using a copy stack. Second, it is very hard to make Cheney algorithm work properly when each processor allocates blocks of memory and the processors need to share the work. The problem is that the blocks leave holes (each is only partially filled), and it is not clear how to have other processors find valid grey objects among the holes. Third, to avoid per-location tags we have organized our algorithm such that pointer testing is based on the object not the location. We found it very hard to maintain this property when using Cheney collection along with parallelism and incremental copying of large objects. Another possible reason for using a stack instead of a queue is that it might lead to better locality [24, 7]. In our block allocation scheme along with our use of a local stack, however, we would expect to get this benefit independent of how the global set of frontier nodes are stored.

The copy-write synchronization. We tried several ways to remove the copy-write synchronization, but could not find a satisfactory way. The Nettles and O'Toole's algorithm [25] uses a mutation log, which keeps a list of all the locations that are written instead of writing them to the replica copy immediately. This log is then processed separately after the copying is finished. This means that there is no longer a chance for copy-write conflicts. To maintain the real-time constraint, however, in our algorithm we need to continue running the mutators while processing the mutation log. The algorithm then has a write-write conflict between a mutator and the jobs processing the log. This conflict is actually worse than the copy-write conflict since this can involve multiple processors handling the mutation log instead of a single copier handling the object. Another problem with a mutation log in our algorithm is that there is no simple way of bounding its size. We note, however, that it is extremely unlikely for a `Write` instruction to get stuck waiting for a copy since the copy lock is on a specific location rather than on an entire object.

Choosing synchronizations. Since we wanted a practical collector, we chose synchronization primitives that can be parallelized. The `TestAndSet` and `FetchAndAdd` primitives can be implemented in parallel using a combining network, and it has been argued that they can be made no slower than a memory reference to the shared memory [14, 28]. Furthermore several machines have supported these primitives in hardware [13, 27, 20, 31, 18]. Most current symmetric multiprocessors support the `TestAndSet` operation directly, but not the `FetchAndAdd`. On the Sun Enterprise Server, for example, `TestAndSet` only requires about the same time as a memory reference (15 cycles if in second level cache and 60 cycles if in shared memory) [23]. Although the `FetchAndAdd` is not implemented directly it is still quite fast using a hardware load-linked/store-conditional or compare-and-swap, and its use in our algorithm (with optimizations) is minimal. The only concern would be to ensure that `FetchAndAdd` is implemented fairly so that all processors make progress. It does not appear that a multiprocessor collector can be implemented time and space efficiently without a synchronization primitive as powerful as the `FetchAndAdd`. Finally, the interrupt might be implemented with period polls. An interrupt is issued by setting a flag at a specified memory location. Processors periodically poll to check if the flag is set (*e.g.*, after every block of code). The cost of implementing interrupts is small if the polls can be effectively inserted.

References

[1] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, Sept. 1987. Elsevier-North Holland.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] H. 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.

[4] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

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

[6] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Lanugage Design and Implementation*, pages 162–173, June 1998.

[7] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, Sept. 1988.

[8] J. Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988.

[9] E. W. Dijkstra, L. 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.

[10] D. Doligez and G. Gonthier. Portable, unobstrusive garbage collection for multiprocessor systems. In *Proceedings of the ACM Symposium on Principals of Programming Languages*, pages 70–83, Jan. 1994.

[11] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[12] I. Foster. A multicomputer garbage collector for a single-assignment language. *International Journal of Parallel Programming*, 18(3):181–203, 1989.

[13] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing a MIMD, shared-memory parallel machine. *IEEE Transactions on Computers*, C–32:175–189, 1983.

[14] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), Apr. 1983.

[15] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

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

[17] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, May 1993.

[18] C. R. Inc. Cray T3D: Technical summary, Sept. 1993.

[19] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[20] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, pages 967–974, Feb. 1986.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[22] R. D. Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 34:31–35, Sept. 1991.

[23] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-protocol active messages on a cluster of smp's. In *Proceedings of SC97: High Performance Networking and Computing*, Nov. 1997.

[24] D. A. Moon. Garbage collection in a large LISP system. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, Texas, Aug. 1984. ACM Press.

[25] S. M. Nettles and J. 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*, Albuquerque, New Mexico, June 1993. ACM Press.

[26] J. W. O'Toole and S. M. Nettles. Concurrent replicating garbage collection. In *ACM Symposium on Lisp and Functional Programming*, June 1994.

[27] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings International Conference on Parallel Processing*, pages 764–771, Aug. 1985.

[28] A. G. Ranade. How to emulate shared memory. In *Proceedings Symposium on Foundations of Computer Science*, pages 185–194, 1987.

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

[30] K. Taura and A. Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 264–275, June 1997.

[31] Thinking Machines Corporation. Model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, Apr. 1987.

[32] M. Wallace and C. Runciman. An incremental garbage collector for embedded real-time systems. In *Proceedings of the Chalmers Winter Meeting*, pages 273–288, Tanum Strand, Sweden, 1993. Published as Programming Methodology Group, Chalmers University of Technology, Technical Report 73.

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