Garbage Collection in MultiScheme (Preliminary Version)

James S. Miller
Brandeis University
Waltham, MA 02254-9110
USA

Barbara S. Epstein
Encore Computer Corp.
Marlboro, MA 01752-3089
USA

Abstract

On stock hardware, our garbage collector commonly takes under 4% of a system's time and less than 0.2 seconds at each occurrence. It provides an unsurpassed mechanism for reclaiming memory and processing resources in a parallel system. Furthermore, the elementary algorithm is easily extended to a parallel implementation that achieves significant performance improvement without slowing down the operation of user programs.

The garbage collector is also an important component of the system when investigating speculative computation. The requirement that the garbage collector recycle tasks that are no longer performing useful work impacts not only the design of internal data structures, but also affects the semantics of continuations in a parallel system.

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014–83–K–0125, N00014–84–K–0099, N00014–86–K–0180, and MDA903–84–C–0033. Additional funds and resources were provided by BBN Advanced Computers Inc., and the Hewlett-Packard Corporation.

1 Introduction

On stock hardware, MIT Scheme's[11] garbage collection commonly takes under 4% of a system's time and less than 0.2 seconds at each occurrence. It provides an unsurpassed mechanism for reclaiming memory and processing resources in a parallel system. Furthermore, the elementary algorithm is easily extended to a parallel implementation in MultiScheme that achieves significant performance improvement without slowing down the operation of user programs.

Our recent work is a direct extension of the work on parallel garbage collection described by Courtemanche[7] and Miller[14]. We report results achieved by a number of

researchers at BBN Advanced Computers, Inc., the Massachusetts Institute of Technology, and Brandeis University. Collectively, this work has been directed at both the practical problems related to achieving a high performance parallel garbage collector on stock hardware (primarily the work of Rozas, Courtemanche, Miller, and Johnson) and at exploring ways in which the garbage collector can support novel uses of parallelism at the user level (Miller and Epstein).

On some Lisp systems, garbage collection has been so slow that it has forced users to turn to other implementation languages; this is unfortunate, since an efficient algorithm has been well known for over twenty years. This compacting two-space algorithm, published in 1969[9], is described in Section 2 along with some variations that improve performance in a serial implementation. Creating a parallel variant of this algorithm is straightforward, but tuning it to perform well is somewhat more difficult. Section 3 describes both the algorithm and some of the performance issues that arise in actual practice.

An efficient garbage collector is an important goal in and of itself, but by adding one new data type (the weak pair) that provides a link between the user and the garbage collector, we acquire an entirely new kind of power. The implementation of the weak pair is simple (see Section 4.1) and forms the basis for a number of interesting features:

- The system automatically discards tasks that are no longer performing useful work.
- Hash functions on arbitrary Scheme objects can be implemented without forcing retention of the hashed objects.
- We can create populations[17] to store sets of objects without forcing retention of the objects.
- The system can detect and correct an important set of common errors (failure to release locks, close files, etc.).

The first of these features, the garbage collection of useless tasks, was suggested as long ago as 1977[4], but in a more restricted setting. In a system supporting Halstead's `future` construct[10], the system's data structures be carefully constructed from weak pairs in order to garbage collect tasks. As part of this structuring, MultiScheme separates the notion of a task (an active computational entity, similar to an actor[12]) from a continuation (the actor's script). This separation is *not* arbitrary, and Section 4.2 provides the rationale, based on a requirement that tasks be automatically garbage collected. The section also discusses the slightly thorny issue of choosing a root for garbage collection that simultaneously retains tasks of interest to the user and discards tasks that are actively but uselessly computing.

We close (Section 5) with a discussion of a form of parallel processing, called speculative computation, that becomes viable with the effective garbage collection of tasks. Speculative computation uses "spare" computational power to effectively explore a range of options, some of which may not be essential to the problem at hand. We have extended MultiScheme to provide a notion of task priority as a means for controlling the scheduling

of computational resources. In addition to merely using the priority when choosing the next task, MultiScheme dynamically modifies priorities as a program runs. In conjunction with the garbage collection of tasks this prevents a useless task from hogging resources, while preventing deadlocks from the premature removal of a useful task.

2 A Serial Two-Space Collector

The basic garbage collection algorithm used in MultiScheme is the well-known[9] two space compacting algorithm of Minsky, Fenichel, and Yochelson (for a simple but complete description and implementation see [2]). The algorithm works by copying all of active memory, compacted and relocated, into a new region of memory. We position a scan pointer at the start of the new area of memory, copy the root of the garbage collection from old space to new space, and position a free pointer to the word after the root (i.e. to the first free location in the new area of memory). Then we repeatedly relocate the object to which `scan` points, advancing `scan` to the next object after each relocation. (Recall that `scan` always points into the new part of memory, and hence at a copy of some object from the old space.) Relocating an object is easy:

- If the object isn't a pointer, it is left unchanged. Fixnums[13], for example, just cause the `scan` pointer to be incremented. For long objects consisting of non-pointer (binary) data (eg. strings, long integers, or machine language code), there is a header word indicating how far the `scan` pointer is to be advanced to skip over the data.
- If the object is a pointer, it will point into the old address space. Under most circumstances, the object to which it points is copied into the new space and `free` is advanced past the copy. The original version (in old space) is modified so that its first word contains a forwarding pointer to the newly created copy. (A forwarding pointer is sometimes called a broken heart.)
- If the object we want to relocate is a pointer, but the first word of the object it points to is a forwarding pointer, the object has already been copied into new space. The forwarding pointer specifies where the copy is; rather than make another copy the pointer we are relocating is made to point to the existing copy.

This process of relocating objects repeats until `scan` meets `free`, at which time all the contents of the old memory that can be referenced have been relocated into the new area of memory. By copying into a spare heap the same size as the active one we are assured that all of the active memory can be copied, and the algorithm will terminate. A proof of correctness is based on noting the invariant that memory between the start of new space and the current value of `scan` contains pointers only to copies of objects (in the new space), memory between `scan` and `free` contains copies of active objects, and memory beyond `free` is not referenced.

Readers familiar with Baker's incremental garbage collector[4] or the more recent generation-based[19] garbage collectors will also find this algorithm familiar, since it forms

the basis of all of these techniques. The two space collector has a running time proportional to the amount of memory in active use, unlike the more familiar mark-sweep garbage collector that takes time proportional to the maximum size of the heap.

2.1 Static Spaces Improve Performance

This garbage collector is quite efficient, since the code fits in most instruction caches and memory reference to the new space is highly localized. Unfortunately, a simple implementation will scan and copy large quantities of memory that are permanently needed by the system. This is a well-understood problem that has prompted the implementation of "generational" garbage collectors[19].

MIT's CScheme system (on which ours is based) takes an intermediate approach. It provides two static areas of memory, known as the pure and constant areas. The pure area contains only non-pointer objects and immutable pointers into the static areas — thus its contents are never modified by garbage collection and it need not be scanned. The constant area contains arbitrary (including mutable) objects; while this area must be scanned in order to relocate objects to which it points, the contents of the area are never copied.

The user has the ability to move arbitrary objects into either of these two areas, and moving an object recursively moves all objects to which the initial one points. It is important to guarantee that objects in the pure area never point into the heap. This is done by moving those subcomponents that can be mutated by operations internal to the Scheme system into the constant area instead of the pure area. The user-visible mutation operations (eg. `set-car!`, `set-cdr!`, and `vector-set!`) specifically test for mutation of objects in the pure area and produce an error rather than store a pointer to the heap into the pure area . Users can intercept this error in order to move the object from the pure area into the constant area and then perform the mutation.

These two areas provide a degree of flexibility that users can exploit to cut down the amount of time taken to perform a garbage collection. By moving immutable objects (such as large data structures constructed at load time or by initialization procedures) into the pure area, these objects are neither scanned nor copied at garbage collection time. They are also removed from the heap itself, so the frequency of garbage collection is reduced. Mutable objects can be moved to constant space, where they will be scanned but not copied. Again, objects in constant space do not occupy space on the heap and decrease the frequency of garbage collections.

As an example of the usefulness of the pure and constant areas, consider the runtime system which is written in Scheme. The majority of the code and its data structures are moved into one or the other of the static areas when Scheme is initially bootstrapped. The result is dramatic — we often observe a factor of 7 improvement in the speed of garbage collection over the same system without the code moved into the static area. This number falls as the running program generates active data on the heap since the speed of a garbage collection is determined by the amount of active storage.

The implementation of these areas is a very simple variant on the previous algorithm.

We add a new "circumstance" under which an object isn't copied: if it is located in either the pure or constant area. In addition, before copying the root of the garbage collection to new space, we place the `scan` pointer at the beginning of the constant area and the `free` pointer at the start of the new heap space. We then scan all of constant space. This will copy into the new space any objects referenced from the constant area (recall that the pure space has no such references). The root is then copied into new space wherever `free` points, and the algorithm proceeds as above.

There are two additional factors that arise as a consequence of compilation:

1. Interpretation of a program causes large quantities of data structure to be `cons`ed for passing arguments (i.e. to create environments or closures). A great deal of the compiler's analysis is specifically aimed at reducing the need for these data structures. A good compiler can eliminate the vast majority of this `cons`ing by making use of registers and a stack. One brief test based on MIT's compiler indicated that compiling a program reduced the `cons`ing to 44% of the amount performed by an interpreted version of the program. By compiling both the program and the runtime system procedures it called, the reduction was to 12% of the original amount. The pretty printer, a `cons` intensive program, was used for this test, so these results are extremely encouraging.

2. In MIT's system an interpreted program consists of list structure, while a compiled program is largely binary data. While the compiled code is sometimes as much as 30% larger than the interpreted version, only about 28% of the compiled code is scanned by the garbage collector. For simplicity (see [16]) compiled code isn't placed in the pure area. Overall, the entire MIT runtime system requires about twice the amount of storage to be scanned when it is compiled as when it is interpreted. There is reason to believe, however, that this particular program has an anomolously large amount of scanned code. Several other large programs exhibit much smaller expansion.

These two factors tend to act in opposite directions. The first dramatically reduces the frequency of garbage collection without noticeably changing the time required for an individual garbage collection pass. The second increases the time required for each individual pass, but doesn't alter the frequency of the passes. In practice we find that our system with a compiled runtime and no user code takes 0.85 seconds for a garbage collection pass on our HP9000/320 system based on a 16MHz Motorola 68020. The same system when interpreted requires only 0.58 seconds – a factor of 1.4 degradation imposed by compiling the system. On the other hand, the compiled system may garbage collect only one-half as often as the interpreted system. Just to get a sense of the balance between these two factors, we tried a very simple test program. The system spent 2% of its time garbage collecting with an interpreted runtime system and 4% when the runtime system was compiled. Clearly, other test programs would change this balance.

3 A Parallel Garbage Collection Algorithm

Despite the efficiency of the two-space collector on simple programs, many programs generate large active data structures on the heap and consequently undergo large numbers of relatively slow garbage collections. The CScheme compiler itself can spend as much as 70% of its time garbage collecting when processing a large program with a small heap size. One solution, of course, is to increase the heap size thereby reducing the frequency of garbage collections. This does nothing to speed up an individual garbage collection pass, and may degrade performance on a virtual memory system.

Fortunately, the garbage collection algorithm itself is easily modified to operate in parallel. MultiScheme contains an implementation of such a parallel garbage collector, and it provides an interesting case study in practical parallel algorithm design. An initial parallel implementation was proposed by Miller[14] and subsequently implemented, described, and measured by Courtemanche[7]. The basic idea is to divide the work of scanning the new space equitably between the processors.

A garbage collection is initiated by a processor that has noticed the need for more space. It alerts the other processors and starts a barrier synchronization that is complete when all other processors agree to stop running user code and begin the garbage collection phase (see [15] for MultiScheme's support of garbage collection initiation). Each processor is responsible for scanning part of the constant space and copying part of the garbage collection root, then begins scanning part of the data copied into the new heap space. The partitioning of constant space occurs every time an object is moved into either pure or constant space so it need not be recomputed when the garbage collection starts. The root of the garbage collection includes the processor-private stack areas used by the Scheme interpreters; each such stack is scanned by its associated processor[1]. The remainder of the root is shared by all of the processors, and is scanned by the processor that initiated the garbage collection.

This initial scanning causes some objects to be copied into the new heap space, and the algorithm must continue by scanning these objects. This raises two problems: how did the scanning (which occurs in parallel on all of the processors) manage to copy objects into a single shared heap area without interspersing their contents and destroying their integrity? Furthermore, how can we divide the work of scanning these copied objects? In Courtemanche's collector this was accomplished by dividing the heap space into fixed-sized consing partitions when the system first starts. At garbage collection time, each processor grabs one of these partitions as an output partition, used as the destination for objects copied while it is scanning. Whenever copying an object would over-fill the partition, it is released to a queue of filled partitions that will eventually be scanned and a new output partition is taken from the new heap space. When the initial scan phase concludes, all processors enter a loop that attempts to dequeue and scan a filled partition. If no partition is available from the queue, the current output partition is scanned instead. The algorithm completes when there are no partitions on the queue and all processors

[1]For architectural reasons, the MultiScheme implementation on the BBN Butterfly makes a copy of all compiled code in each interpreter's local memory. In addition to its stack, each processor scans its own copy of this code area.

have finished scanning their own output partitions.

3.1 Static Memory Allocation

Courtemanche's system was based on pre-allocating heap space to specific processors. That is, when the system was booted the memory was divided into fixed sized regions and each processor was allocated one region to be used solely by that processor as a heap. This gave the system convenient control over the binding of processors to the memory that they used for space allocation. It allowed for co-locality of a processor and its heap[2], but at the expense of increasing garbage collection frequency when the processors allocated space at different rates. That is, if only one processor actively allocated heap space it would require a garbage collection as soon as it filled up its own heap.

There is an obvious solution to this problem of uneven space allocation, but at the cost of losing some control over locality. The current implementation of Butterfly Scheme[5] dynamically allocates memory to processors. It continues to statically divide the memory between the processors, but maintains a globally visible data structure that indicates the amount of each partition that is in actual use. At system initialization (and after each garbage collection) each processor grabs a fixed quantum of memory from its own partition. As it needs to allocate memory it does so by using part of this initial quantum. When it runs out of space it goes back to the global data structure to see if its partition has sufficient space to fulfill the request; if so, it bites off another chunk of its own partition and continues running. Otherwise it checks all of the other processors' partitions. Only if no space is available on *any* processor does it initiate a garbage collection.

Even with this change, however, a hidden penalty arises when the processors have vastly different amounts of memory on their heaps. The cost of searching through the global data structure to find a processor whose consing partition has sufficient space is significantly higher than the cost of merely allocating space from the processor's own consing partition. Both Courtemanche's system and the current BBN Scheme system try to alleviate this problem by having the garbage collector divide the active memory equally among the processors' heaps. They do this by subdividing each of the consing partitions into multiple heaplets. At the start of a garbage collection, each processor takes the first heaplet from its own consing partition to use as its output partition. The remaining heaplets are considered to be organized in a spiral consisting of the lowest remaining heaplets in all of the consing partitions, then the next higher heaplets, and so on. When a processor fills its output partition it takes the next available heaplet in the spiral for its new output partition. At the end of a garbage collection, therefore, all of the processors' consing partitions are filled to roughly the same extent – they differ by no more than one in the number of filled heaplets in the partition.

This does, however, lead to a phenomenon that Courtemanche called partition wastage (sic). At the end of a garbage collection each processor will use as its `free` pointer the `free` pointer of the highest heaplet used during the garbage collection from its partition. The portion of memory that is free in any other heaplets within the partition is thus

[2]On the Butterfly, MultiScheme deliberately didn't aim for co-locality. Instead, it tried to randomly distribute memory references across the Butterfly switch.

wasted – Courtemanche's partition wastage. His measurements show that, as might be expected, partition wastage rises linearly with the number of processors in the system.

3.2 Dynamic Memory Allocation

In addition to the partition wastage problem there is another problem with the memory allocation systems described above. As the number of processors grows, the consing partition must shrink unless the virtual address space grows proportionately. And here we encounter a problem with current architectures: they support 32 bit pointers to byte addresses but have no data type bits. Even on today's commercial parallel processors this is barely sufficient: a 512 processor machine with a mere 4 megabytes per node requires 31 bits to address all of the available memory. This is a particularly severe problem in a Lisp system that uses a two space collector since it will use half the address space for each of the two semi-spaces.

But the practical limits are considerably smaller. The current implementation of MultiScheme (using 8 bits for type codes) has a maximum heap size of 4 megapointers. On a 128 processor machine this means that the individual consing partition is only 32K pointers, and from this we must take space for the static and program areas. One well-known problem of a two-space collector is that it divides this small amount of space in half by dividing the heap into two parts. Minsky's original implementation did not suffer from this problem: his second space was allocated on the magnetic drum. BBN Butterfly Scheme uses a similar technique – the second space is allocated in virtual memory that isn't accessible through ordinary Scheme pointers because it lies outside of the 4 megapointer reference range. At the end of the garbage collection this inaccessible memory is copied back down into the Scheme heap. Since the Butterfly typically has far more physical memory than is available in a single shared virtual address space, this mechanism tends to keep both the old and new space in physical memory. As a result the copy operation is relatively fast. It can be even faster under an operating system like Mach, where it can be replaced by a set of page map operation that use the virtual memory hardware instead of performing memory operations.

Despite this, any reasonable initial quantum taken by all processors from their 32K pointer consing partition virtually guarantees that the system is constantly garbage collecting. Worse yet, no vectors can be allocated that exceed the length of a processor's consing partition since no processor will ever have this much free space. The problem is quite dramatic.

Consider the following (uninteresting) program:

```
(define (test)
  (define (loop n name)
    (if (= n 0)
        'done
        (begin (vector-cons 100000 'junk) ; Make a large (garbage) vector
               (write name)               ; Show who's working
               (loop (- n 1) name)))))    ; Loop around again
```

```
(for-each touch                          ; Start and wait for 6 tasks
          (list (future (loop 10 'a)) ... (future (loop 10 'f)))))
```

This program has six tasks (named `a` through `f`) regardless of the number of processors used to run the program. Running on a machine with a maximal heap size of 2.6 megapointers we would expect this to require three garbage collections since it allocates (and discards) 6 megapointers of memory. Using ten processors this is precisely the observed result. Increasing this to twenty processors, however, caused in excess of 60 garbage collections! The reason, upon investigation, turned out to be that the size of a consing partition with twenty processors was just under 100K pointers. Thus each attempt to `vector-cons` caused a garbage collection[3]!

The solution is again simple, and was implemented by Kirk Johnson and Jim Miller. In this solution there is a single global `free` pointer used by all processors to allocate large chunks of heap space. When a processor runs out of room in its current heap, it allocates a new chunk using the global pointer. It then allocates smaller amounts as required during execution. When a processor needs to allocate a chunk that exceeds the upper limit of the actual shared heap it uses a privately allocated "emergency area" instead and immediately requests a garbage collection.

This mechanism both reduces the frequency of garbage collections when the load on the system is unbalanced, and entirely eliminates the partition wastage. The implementation has proven efficient, provided that the "chunks" are large enough to reduce contention for the global `free` pointer.

3.3 Performance Bottlenecks

Courtemanche showed that his algorithm achieved linear speed-up to five processors, but then performance remained constant. He identified two specific problems with his garbage collection mechanism that accounted for this phenomenon: inadequate partitioning of the constant space, and processor starvation. The first was the result of an apparently harmless simplification of the partitioning algorithm for constant space, and was easily repaired.

Processor starvation is more serious. Courtemanche attributes it to a lack of active storage available for scanning. As the number of processors increases there is no work for the additional processors to perform. Ultimately, as the number of processors becomes very large they generate contention just trying to locate scanning to be performed and the system, in fact, slows down. This problem was the subject of recent work at BBN by Guillermo Rozas. Rozas examined the sources of contention and processor idle time during garbage collection. He identified eight different problem areas:

[3]The observant reader may point out that the algorithm described here would, in fact, prevent the system from ever allocating a vector of 100K elements. In fact, the BBN implementation has a special case for handling allocation of large objects that allows this case to succeed once at the end of each garbage collection.

1. Idle time waiting for the initial synchronization barrier. This barrier is required to guarantee that all processors have stopped allocating space on the heap and "flipped" into the new space.

2. Contention for the data structure used to allocate partitions to be scanned.

3. Contention for the data structure used to allocate output partitions.

4. Contention for moving a single object. This occurs when two (or more) processors are actively scanning memory and both need to move the same object into new space.

5. Idle time because no partition is available to be scanned. This occurs when other processors are still scanning memory, since they might cause a new partition to be available at a later time.

6. Idle time waiting for the synchronization barrier indicating termination of the scanning operation.

7. Idle time waiting for the synchronization barrier indicating that all post-scanning operations have been completed.

8. Idle time waiting for the processor that initiated the garbage collection to complete Scheme level code needed by the system to complete the garbage collection cycle.

Of these, by far the least common is the conflict between multiple processors for moving the same object. In fact, to lessen the impact of this problem to a completely negligible level we carefully arrange to store the forwarding pointer in old space *before* the copy operation begins. Thus, another processor will see a completed relocation as early as possible. The location in old space that holds the forwarding pointer is used as a lock to guarantee the atomicity of the copy operation.

The remaining seven items occur often and can be a source of significant performance degradation as the number of processors grows. To help eliminate the contention generated by using a simple spin to detect synchronization or lock release, Rozas added a set of tunable delays. By making some of the delays proportional to the number of interpreters, the contention is reduced at the cost of a lengthened minimum delay.

4 Garbage Collection of Tasks

MultiScheme's model of parallelism, based on the `future`, is directly inherited from Halstead's Multilisp[10]. A `future` is composed of two parts: a task representing work to be performed and a placeholder representing a value being computed by one or more tasks. Tasks are created for the explicit purpose of calculating a value, and each of these values allows for the synchronization of the task computing the value (producer) and any tasks that need the value to continue their own computation (consumers). We refer to

```
(define (make-tree f low high)
  (let ((delta (- high low)))
    (if (zero? delta)
        (f low)
        (let* ((interval (if (even? delta) delta (- delta 1)))
               (mid-point (+ low (/ interval 2))))
          (cons (future (make-tree low mid-point))
                (future (make-tree (+ mid-point 1) high)))))))

(define (search tree value) ...)

(search (make-tree square 1 10) x)      ;  For some value x
```

Figure 1: Parallel Tree Search

this as value-oriented parallelism as opposed to the more common task-oriented parallelism generally found in parallel versions of languages like C, PASCAL, and FORTRAN.

Because placeholders act just as the values they represent, the synchronization supported by MultiScheme is highly dynamic and resembles the mechanisms used in dataflow computation models. Storing a placeholder in a widely visible data structure is a reasonable (and common) activity, even in a side-effect free program like the one in Figure 1. In this example, `tree` is probably *not* fully constructed when the call to `search` starts. Thus, the code within `search` must be responsible for synchronizing with the tasks that are creating the tree. This synchronization is dynamic, since the tasks have not even been created before the search commences.

The parallel tree search of Figure 1 raises another problem in addition to that of dynamic synchronization. If the function `f` is hard to compute, we should abort the building of sub-trees as soon as the correct value has been located by the `search` procedure. This problem is well-known in serial computing, and has led to lazy evaluation and the stream-based model of computation. But a parallel system, unfortunately, gains its speed advantage from eager evaluation. One way out of this dilemma is to remove tasks from the system as soon as they are known to be unneeded, an approach suggested by Hewitt and Baker in [4]. Another commonly proposed alternative (often in conjunction with a construct like Common Lisp's `catch`) is to kill tasks when they become useless. These two approaches strongly resemble the same two options for memory allocation – garbage collection vs. explicit release.

Just as prematurely releasing memory in an explicit memory management system causes problems arising from dangling pointers, deliberate"task homicide" can result in unexpected and undesirable premature reclamation of tasks. The dangling pointer problem of memory allocation has its direct analog in a form of deadlock in the parallel processor system. Imagine that we modify the program of Figure 1 by adding the following

code:

```
(let ((tree (make-tree 2 20)))
  (if (search tree 3)
      '(FOUND 3)
      (search tree 27)))
```

Recall that `make-tree` creates the tree using parallel tasks, and thus the call to `(search tree 3)` may begin before the tree is fully constructed. If `search` kills off "useless" tasks — those constructing parts of the tree that the search didn't have to examine — then the subsequent `(search tree 27)` may well try to search a part of the tree that has not been constructed and whose tasks have been killed by the earlier search. The program deadlocks.

While MultiScheme provides procedures to kill tasks[15], their use is generally discouraged. Instead, the standard mechanisms for creating tasks (the `future` construct and the ones described below for speculative computation) interact with the garbage collector to provide automatic reclamation of unneeded tasks. In order to perform this reclamation, MultiScheme requires mechanisms to reference tasks without preventing their reclamation and to identify tasks that must continue to run. The former is supported by weak pairs, discussed in Section 4.1. The latter is handled by our choice of task-related data structures and the root of garbage collection, as discussed in Section 4.2.

4.1 Low-Level Support

The MultiScheme system incorporates two modifications originally intended to support the garbage collection of tasks. Both have since been merged into the sequential implementation of Scheme, since they provide useful services beyond their original design goal.

The first modification is the addition of a new data type, the weak pair. This resembles a standard cons cell[4], but with the property that the car of the cell reverts to `'()` if there are no references to the object in the car other than through the cars of weak pairs. Support for these objects[5] is easily implemented in a two-space garbage collector, since it provides space for four pointers at garbage collection time for each pair that is retained (the original two in old space, plus two more in new space). These are used as follows:

1. One old space pointer is used as usual to contain a forwarding pointer indicating the new location of the cell.

2. The second old space pointer would ordinarily be unused. It is used to store the original car of the cell.

[4]For purely aesthetic reasons, the standard operations for use with cons cells do not operate on weak pairs. A separate set of operations (`weak-car`, etc.) are provided instead.

[5]Objects similar to the weak pair may exist in other Lisp systems. But the authors are unaware of any implementation descriptions elsewhere in the literature.

3. One new space pointer is used as usual to contain a copy of the cdr of the cell.

4. The second new space pointer allows each processor to maintain a list, headed by the processor-private variable `weak-list`, of all the weak pairs copied by this processor into new space during the garbage collection. The type code on this cell indicates that it does *not* contain a pointer to be traced by the garbage collector when it is subsequently scanned.

At the end of the garbage collection, the cdrs of the weak pairs in new space are correct. The cars, however, must be located and corrected. This is done in parallel, by having each processor follow the chain of weak pairs headed by its `weak-list` variable. For each cell, the original car (still available in old space, cell 2) is examined. If the type code indicates a non-pointer, or the address indicates an object in pure or constant space, the new car is just the same as the old car. Otherwise, the object originally stored in the car is examined to see if a forwarding pointer is present: if so, the new car becomes a pointer to the relocated object; if not, the new car is set to `'()`.

There are three things worth noticing about this algorithm. First, it is performed in parallel by having each processor responsible for updating the weak pairs that it copied during the regular scan phase. Second, it does *not* itself allocate memory, and hence can be safely performed during the garbage collection operation. Finally, it must have access to both old and new space, and hence is best considered a part of the garbage collection operation itself rather than as a Scheme procedure that must run immediately after the garbage collection is complete.

The weak pair mechanism has proven quite useful. It is a convenient base on which to implement the weak set (formerly population) abstraction of T[17] and the object hash tables of MIT Scheme[11]. These, in turn, provide the mechanism upon which to build a variety of user conveniences including the automatic closing of files reclaimed through garbage collection.

A second mechanism, precious objects, has been proposed to augment the weak pair, and is only now beginning to be explored. The goal of the weak pair mechanism is to allow the garbage collector to reclaim certain objects despite references to them. By contrast, precious objects allow the garbage collector to report its *desire* to reclaim certain objects without actually reclaiming them. Precious objects are implemented by allowing the user to designate an additional root for garbage collection, called the precious object list. The garbage collector first performs its normal operations without considering this list. After the collection is complete (either before or after handling weak pairs, depending on the desired interaction between the two features), the precious object list is scanned and split in two: one list contains those objects that are "alive" (i.e. those containing a forwarding pointer indicating that they were relocated, plus those in constant space and non-pointer objects) and the other list contains those that would otherwise be reclaimed. This second list is then used as the root to commence another garbage collection pass — thus preserving the precious objects. This list is made available to users after the garbage collection.

Precious objects can form the base of a number of user services, including the automatic release of resources when they are no longer in use. This is very important for

catching and reporting certain user errors (eg. forgetting to release a lock). It is also useful for interfacing with external services. The MIT Scheme interface to the X Windows package, for example, will use this mechanism to release resources in the X server when a Scheme program ceases to need them.

The precious objects mechanism is desirable but it has not yet been exploited since it can be mimicked using the (pre-existing) weak pair mechanism. MIT Scheme, for example, automatically calls the Unix `close` operation when all references to a file (through the `port` data objects) are lost. This is done by building a weak list of all the ports in the system. After each garbage collection, this list is scanned for entries whose `car` has become `'()` indicating that the port is no longer in use. A parallel list is also maintained containing the information needed to locate the actual operating system resource that must be released, and the appropriate call to `close` is issued. The entry is then spliced out of both lists. This trick, providing parallel structures for user-accessible and system internal data structures, allows any system with weak pairs to mimic a system with precious objects (albeit in a slightly inelegant manner).

4.2 System Issues

Garbage collection of tasks seems easy once we have made the small changes to the garbage collector that provide the weak pair mechanism. We need only arrange the system's data structures to use a carefully chosen combination of ordinary and weak pairs. We examine every pointer to a task and ask whether it provides a reason to retain the task (in which case it must be an ordinary cons cell) or if it is for purely internal uses (and is implemented with a weak pair).

MultiScheme's value-oriented parallelism leads directly to one kind of reference: when a value becomes available, all of the tasks that are waiting for its value must be reactivated. The connection from the placeholder receiving the value to the tasks that are activated when it arrives is maintained using weak pairs. This allows the garbage collector to reclaim tasks whose computation has been halted awaiting the arrival of a value. Two issues, however, remain: how do we reclaim tasks that are *not* waiting for values (for example, tasks that are useless but are in an infinite loop), and what prevents all tasks from being garbage collected?

These questions force us to consider carefully the nature of the graph traversed by the garbage collector. We must examine both the edges of the graph and the root used to locate the "useful" portion of the graph. Let us begin by assuming that we can choose an appropriate set of root tasks and memory objects, thus concentrating on the construction of the graph itself. Since we have adopted a value-oriented approach we make the flow of a data value through the graph the dominant consideration. That is, we retain a task only if it is computing a value that is still needed by the on-going computation. In terms of MultiScheme, this means that we require a task precisely when the placeholder for the value it is computing is required — the graph must have an edge connecting each placeholder with the task[6] computing its value. At first glance, this seems simple enough: when we encounter a `future` we create a placeholder and a task and simply make the

[6]When we discuss speculative computation in Section 5 we will see that there may be multiple tasks

placeholder reference the corresponding task. In fact, MultiScheme has a slot, called the motivated task[15], in the placeholder for just this purpose.

But MultiScheme inherits from Scheme the ability to reify a continuation into a procedure by using `call-with-current-continuation`[18]. What is the nature of the continuation for a task newly created by `future`? Clearly, when the task completes its computation it must store the computed value in the appropriate placeholder and stop execution. But what, exactly, is the "appropriate placeholder"? There are two equally legitimate possibilities, hinging critically on the desired relationship between a continuation and a task:

1. The continuation for a newly created task has a specific placeholder built into it. In this model, the *continuation* contains the information connecting a task with a placeholder. The user can be completely unaware of the existence of tasks. The only role tasks play is as structures internal to the scheduling of the system itself.
2. The continuation for a newly created task stores the computed value into whatever placeholder is associated with the task that is executing. Thus the *task* (not the continuation) contains the information linking a task to a placeholder.

To make the difference clearer, we can write the Scheme procedure used to implement the `future` macro for each interpretation. Assume the transformation:

`(future` *exp*`)` $\rightsquigarrow$ `(spawn-task (lambda () exp))`

In our first interpretation the code would be roughly:

```
(define (spawn-task thunk)
  (let ((placeholder (make-placeholder)))
    (make-task              ;  Task holds continuation only
      (lambda (value)       ;  Continuation
        (determine! placeholder value)))
    placeholder))           ;  Value of  future
```

In our second, we would use:

```
(define (spawn-task thunk)
  (let ((placeholder (make-placeholder)))
    (make-task placeholder ;  Task holds placeholder (its  goal), too
      (lambda (value)       ;  Continuation
        (determine! (task.goal (current-task))
                    value)))
    placeholder))           ;  Value of  future
```

computing the value of a single placeholder. Thus, there may be multiple edges emanating from a single placeholder.

The difference in code, clearly, is minor. But the impact on the semantics of both continuations and tasks is major, as is the impact on the garbage collection of tasks. We can see the difference in semantics clearly by considering the interaction with MultiScheme shown in Figure 2. Consider our first semantics. In this case, when `(define value-2`

```
(define cont '())
(define value-1 '())
(define value-2
  (future (call-with-current-continuation
           (lambda (continuation)
             (set! cont continuation)
             (let loop-forever () (loop-forever))))))))
;  Wait for  cont to have a non-null value
(set! value-1 (future (cont 'STOP)))

;  What are the values of  value-1 and  value-2?
```

Figure 2: The Semantics of a Task

...) is evaluated, `value-2` receives a placeholder as its value. The continuation for the newly created task explicitly references this particular placeholder. After exporting this continuation to the variable `cont` the task goes into an infinite loop (hopefully to be reclaimed by the garbage collector).

When `(set! value-1 ...)` is evaluated, a second task and placeholder are created. But this new task calls the continuation created for the original task (stored in `cont`). Since this continuation explicitly references the particular placeholder stored in `value-2`, the termination of the task causes `value-2` to have the value `STOP`. `Value-1` contains a placeholder, but it will never receive a value.

The situation is similar in the second semantics, but the result is very different. Neither continuation references a specific placeholder. Instead they both look to the task that is executing at the time the continuation is invoked in order to locate the placeholder. It is the *second* task that ultimately calls the task termination continuation, and that task was created with the intention of supplying a value to the placeholder stored in `value-1`. Thus it is `value-1` that receives the value `STOP` and `value-2` will never receive a value.

From a language design point of view, either semantics can be defended. There is perhaps a slight preference for the first semantics, since extending this choice allows us to build a MultiScheme variant that supports the equivalent of or-parallelism in PROLOG. Extension in this direction requires that the act of supplying a value to a placeholder *not* allow the placeholder to be replaced by the value. Instead it must remain to record the tasks that have used the value so that a new value arriving later will cause the same tasks to be resumed again — the familiar forking of processes when multiple values are available at a choice point.

From the system design point of view, however, there is a marked advantage to the second semantics related directly to our desire to garbage collect tasks. By making the task an explicit object and the repository for the placeholder that will receive a value, we make the user responsible for maintaining the information needed to correctly reclaim tasks. When a task is created with a placeholder as its goal, we add the task to the list of motivated tasks that is part of the placeholder data structure. This information is essentially static — initialized when a task is created and not subsequently modified.

By contrast, the first semantics requires this information to be updated whenever a task invokes a continuation. By itself, this change is minor; but the ramifications are severe. Since we cannot predict the behavior of an arbitrary task, we must assume that it might invoke *any* continuation that is visible to it. Since this invocation would lead to supplying a value for the placeholder associated with the continuation, we must assume that *any* task might be responsible for computing the value of *any* (visible) placeholder. But this means that we can never reclaim a task! For this reason, we have chosen to promote the task to a user-visible object. The resulting system has three independent components related to parallelism:

Placeholder
: The representative of a value not yet computed. These are essentially invisible to the computation, but provide the value synchronization of the underlying model of parallelism.

Continuation
: The representation of work to be performed. At any time a computation can create a procedure representing the work it has remaining (by calling `call-with-current-continuation`).

Task
: The representation of an active computational agent, similar in spirit (but not detail) to an actor. A task has a placeholder whose value it is computing, and a continuation (the actor's script) representing the work it is performing to compute that value.

We now understand how to construct the edges of the graph traced by the garbage collector. Every placeholder has an edge (called the motivated task) connecting it to the task that is computing its value. The connections indicating the tasks that must be awakened when a value has been computed, however, are *not* edges of the graph traversed by the garbage collector.

There are two obvious choices for the root of garbage collection. We could either use all tasks active at the time of garbage collection or use those reachable from the global naming environment. Both of these have defects: the former fails to reclaim useless tasks that are in infinite loops (and hence always running when a GC begins), and the latter fails to retain tasks spawned by user input. Our solution is relatively simple and seems to work well in practice. The garbage collector uses as its root ...

1. ... the global naming environment. This is essential in any case, and serves to retain any processes calculating globally visible data structures.

2. ... the stack of any task running with a reduced interrupt mask at the time of the garbage collection. Tasks that are in a critical section change interrupt levels to lock out user interrupts. This root serves to prevent them from being reclaimed until the critical section is complete.

3. ... the set of tasks representing active read–eval–print (`rep`) loops. Each invocation of a rep loop creates a new task and this set of tasks is maintained in a global variable. Thus, this is really handled through the use of the global naming environment.

5 Speculative Computation

The ability to garbage collect tasks allows us the freedom to explore uses of parallelism in which we do not know *a priori* that the tasks we are creating are necessary — an area known as speculative computation. Based on the mechanism used to gain performance, we distinguish two kinds of speculative computation: eager evaluation and multiple approaches,

In eager evaluation, we gain speed by overlapping the execution of an essential task with that of one or more other tasks, and use the value of the essential task (when it is finally computed) to determine which of the other tasks is valid. This approach arises from a natural desire to evaluate the predicate of an `if` statement concurrently with both the consequent and the alternative. The performance improvement is determined by the difficulty of computing the predicate. It is easy to define a `speculative-if` macro that provides precisely this power:

```
(speculative-if predicate consequent alternative) ⇝
(let ((conseq (future consequent))
      (alt (future alternative)))
  (if predicate (touch conseq) (touch alt)))
```

For some problems, the appropriate predicate to use with eager evaluation is essentially "use whichever one computes an answer fastest." For example, problems involving complicated searches (such as those arising naturally in artificial intelligence and theorem proving) tend to have this characteristic. For these kinds of problems it may be productive to devote processing power to following several alternative problem solving strategies, in the hopes that at least one of them will ultimately provide a viable solution. We refer to this as multiple approach speculation, and it is accessible to MultiScheme users through the procedures `disjoin` and `await-first`[15].

In both cases, of course, we must worry about possible side-effects that might lead to non-determinism or errors in one of the speculative branches of the computation. Because the garbage collector will remove any tasks that aren't working toward a useful goal, however, we needn't worry about explicitly killing "competitive" branches once one of them is resolved. On the other hand, there is no reason to squander processing resources if this can be easily avoided. We have implemented an intermediate solution[8] to this dilemma by adding priorities to MultiScheme tasks and using a timer to cause task preemption.

The model of process scheduling described in [15] is based on an undifferentiated pool of tasks from which some architecture-dependent mechanism (encapsulated in the procedure `get-work`) selects tasks to run. We have modified this model to allow the user to specify a priority for a task and we require `get-work` to choose a higher priority task before any lower priority tasks that may be available. By combining this with a timer interrupt mechanism that forces tasks to relinquish the processor after a finite interval, we provide a service that over time causes higher priority tasks to take precedence over lower priority tasks.

Using this basic mechanism we have added two higher-level features, called promotion and downgrading. Promotion occurs when a task must suspend execution while waiting for the value of a placeholder to be computed. Since this task can do no further work until the value of the placeholder is known, we promote the priority of the tasks that are computing the value of the placeholder to that of the waiting task. If these tasks are waiting for some other placeholder's value, then the tasks computing that value must be promoted as well, and so on. The act of `touch`ing a placeholder (i.e. waiting for its value to be computed) thus causes the tree of tasks rooted at the placeholder and generated by the motivated task and waiting for slots to be promoted to the maximum of their current priority and the priority of the task that performed the touch.

Downgrading operates similarly, but "in reverse." When the placeholder created by a call to `disjoin` receives a value (because one of the multiple problem solving approaches has yielded a value), the priorities of any sibling tasks (i.e. tasks whose values were alternatives to the one that has succeeded in computing a value) are lowered to a very small value. This change in priority is also propagated down the tree of tasks in the same way. With this in mind, we can rewrite our earlier `speculative-if` macro:

```
(speculative-if predicate consequent alternative) ⇝
(let ((conseq (future consequent (speculative-priority)))
      (alt (future alternative (speculative-priority))))
  (if predicate
      (begin
        (reduce-priority alt (ultra-low-priority))
        (touch conseq))
      (begin
        (reduce-priority conseq (ultra-low-priority))
        (touch alt))))
```

The details of the propagation mechanism are of some interest, and are described in detail in [8]. In essence, we try to maintain an invariant that a task's priority must be at least as high as the priority of any task waiting for the value it is computing. A change in priority is propagated down this tree until either a task is found that is actively computing (i.e. we reach a leaf of the tree), **or** we encounter a task whose priority already violates this invariant, **or** modifying the task's priority would violate the invariant.

5.1 Toy Example: A Rule-Based System

To demonstrate the speculative computation mechanisms we will examine a simplified interpreter for a rule-based system. The example is based on a program originally developed for teaching an introductory undergraduate course in signal processing[1]. It is a rule-based expression simplifier supplying enough power to handle a set of rules for simple symbolic algebra. Like most rule interpreters it consists of a pattern matcher, a rule firing mechanism, and a control structure component. Both the pattern matcher and the control structure can utilize speculative parallelism, but for simplicity this example deals only with the control structure. For a more complete description of the other components of the system, see [1] and [14].

The rule interpreter has an inner loop that takes an expression and tries to match it against all of the rules. If a rule matches the expression, the rule firing mechanism is invoked to produce a modified (presumably simpler) expression. If no rule matches, then the expression is already in its simplest form. The problem we are attempting to solve, then, is converting the sequential control structure of "multiple solution techniques augmented with a default result" into a corresponding speculative (parallel) control structure. In doing so, we will assume that rules may be fired *in any order* that terminates and will still arrive at a correct final result — the only restriction is that we will not "give up" on simplifying an expression until all rules have failed to simplify it. This assumption is *not* valid in general for rule sets and poses an important practical limitation to the solution described here.

The primitive mechanisms available for speculative computation directly from MultiScheme (`disjoin` and `await-first`) provide the ability to wait for any one of a set of placeholders to receive a value. The problem we have here, however, is more complex. We must detect the case in which none of the standard solution approaches provides a result (i.e. none of the rules match) and provide a default behavior. We must code the rule matcher in a way that allows this to be detected, and we have chosen a rather simple but perhaps inelegant technique: the rule matcher returns a specific result indicating failure and we loop until either a match succeeds or all matches fail. This is packaged together in the following procedure:

```
(define (fail) fail)
(define (first-value placeholders default-value)
  (define (failed? object)
    (and (not (placeholder? object))     ;  Has a value, and...
         (eq? object fail))              ;  the value is a failure
  (if (null? placeholders)
      default-value                      ;  No alternative
      (let ((result (await-first placeholders)))
        (if (failed? result)
            (first-value                 ;  Failed: try again
             (filter-out failed? placeholders) default-value)
            result))))                   ;  Success
```

With this procedure, we can rewrite the sequential control structure component:

```
(define (scan rules expression)
  (if (null? rules)
      expression
      (match-rule (car rules)
                  expression
                  (lambda () (scan (cdr rules))))))
```

into the speculative parallel version:

```
(define (scan rules expression)
  (first-value
    (map (lambda (rule) (future (match-rule rule expression fail))
         rules))
    expression))
```

If we are able to estimate the likelihood of success for any given rule (either statically or dynamically), we can consider supplying a priority with each rule. The only change that is needed to `scan` is to add this priority to the `future` expression:

```
(future (match-rule rule expression fail)
        (calculate-priority rule))
```

MultiScheme then attempts to schedule the matching tasks for the rules on a priority basis.

6 Conclusions and Future Work

We are happy with the performance of our garbage collector on stock serial hardware; with improvements in hardware performance we anticipate achieving garbage collection in "the blink of an eye[7]" Since garbage collection need not interfere with either data collection or control signalling, a serial Lisp system using this collector can be seriously considered for use in hard real-time systems.

Unfortunately, the parallel implementation does not perform as well. This is due in part to the technological lag between serial processors used in inexpensive desktop workstations and those in current parallel processors. Presumably this will change with time. Also, the contention generated during parallel garbage collection remains significant. We plan to complete a set of measurements (currently underway) aimed at quantifying this contention with the goal of tuning the garbage collector as a function of the number of processors available.

[7]Psychologists put this figure of merit — the maximum amount of time that can elapse without altering the perception of effective control — at 0.05 seconds.

Our initial work in taking these measurements has shown us the difficulties involved in both gathering statistics and visualizing them. We are using a standard set of tools available on the BBN Butterfly, and by practical experience are gaining insight into their uses and limitations. We plan to capitalize on this experience by generating a similar set of tools within the MultiScheme environment. We expect to provide a dynamic (rather than our current post-mortem[3]) visualization tool, triggered by a measurement task that runs periodically in the spirit of Clamen's original work[6].

Our work on measuring the garbage collector raises another issue that has long been a concern. Having a substantial program of genuine practical interest provides tremendous motivation and the insight needed to develop tools appropriate to the problem at hand. Our interest in the garbage collector has helped spur the development of tools for visualization of parallel programs written in C (since that is the implementation language for the garbage collector itself). By developing a high performance Lisp environment on a parallel processor, we hope to attract implementors of one or more major Lisp applications. In conjunction with these implementors, we will have both the necessary resources and (more importantly) the necessary motivation and experience to capitalize on the technological base provided by parallel processors.

References

[1] Hal Abelson and G. J. Sussman. Procedural abstractions in lisp programming. Tentative title, submitted to *Byte* Magazine, August 1987.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] Laura Bagnall et al. The butterfly lisp user interface. Submitted to *Workshop on Parallel and Distributed Debugging*, 1987.

[4] H. Baker and C. Hewitt. The incremental garbage collection of processes. AI Memo 454, Mass. Inst. of Technology, Artificial Intelligence Laboratory, December 1977.

[5] BBN Advanced Computers Inc., Cambridge, MA. *Butterfly Common Lisp and Butterfly Scheme Release Notes*, February 1989. Release 1.0.

[6] Stewart Michael Clamen. Debugging in a parallel lisp environment. Bachelor's thesis, Mass. Inst. of Technology, 1986.

[7] Anthony James Courtemanche. Multitrash, a parallel garbage collector for multiScheme. Bachelor's thesis, Mass. Inst. of Technology, 1986.

[8] Barbara S. Epstein. Support for speculative computation in multiScheme. Bachelor's thesis, Brandeis, 1989.

[9] R. Fenichel and J. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Comm. of the ACM*, 12(11):611–612, 1969.

[10] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[11] Chris Hanson et al. MIT Scheme reference manual. Technical Report 1281, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, MA, January 1991.

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[13] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[14] James Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, Mass. Inst. of Technology, August 1987. Available as MIT LCS/TR/402.

[15] James Miller. Implementing a Scheme-based parallel processing system. *International Journal of Parallel Processing*, 17(5), October 1988.

[16] James Miller and Guillermo Rozas. Free variables and first-class environments. *Journal of Lisp and Symbolic Computation*, 1991.

[17] Jonathan Rees, Norman Adams, and James Meehan. The t manual. Technical report, Yale University, January 1984. Fourth edition.

[18] Jonathan Rees and William Clinger (*editors*). Revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12), December 1986. Also available as MIT AI Memo 818a.

[19] David Ungar. Generation scavenging: A non-distruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, May 1984.