Concurrent Garbage Collection on Stock Hardware

S.C. North
J.H. Reppy[†]

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

I. Introduction

This paper describes the design and implementation of a memory management system for **Pegasus**. **Pegasus** is a system that supports the implementation of programming environments and other interactive applications on single-user workstations[RG86]. It is designed to run on conventional von Neumann processors, such as the Motorola MC68000 or MicroVAX-2[1]. It provides a concurrent functional programming language, called the **Pegasus Meta-Language (PML)**; run-time support for **PML**; and a core library, written in **PML**, of useful services such as a window manager. Applications are written in **PML** and are usually built on the services of the core library. One of the important features of **Pegasus** is that it supports multiple processes (or *threads*) in a shared address space.

PML is derived from **ML**[Milner85][MacQ85]. It is a strongly typed, polymorphic language, with functions as "first class" values, and multiple processes which communicate via typed channels. Because it is a high-level language, **PML** requires extensive run-time support. One of the most crucial aspects of this support is the memory management system. Since it is inherently a heap-based language, the cost of memory management has a strong effect on the efficiency of any **PML** implementation. For example, because of the polymorphic type system of **PML**, functions always take exactly one argument; a function with multiple parameters takes a tuple object as its single argument. This means that most function applications involve memory allocation. Thus the speed of memory allocation and reclamation has a significant effect on the speed of **PML** programs. Furthermore, because **Pegasus** is used for interactive applications, it is important that memory operations complete in a bounded amount of time (i.e., that they are *real-time*).

† Consultant.

1. MC68000 is a product of Motorola Inc. MicroVAX-2 is a product of Digital Equipment Corporation.

The **Pegasus** memory management system is based on semispace copying[Baker78][Brooks84] and uses a separate *Garbage Collector* (GC) process to reclaim storage. By having a separate process for garbage collection, we can off-load some of the cost of memory management to otherwise unused cycles or *dead-time*[2]. The system does not rely on hardware tag bits or specially microcoded instructions. We have also implemented a version of our memory management system based on generations[LH83][Ungar84].

We first present an overview of memory management techniques. Then we give some important details about **Pegasus**, followed by the description of the **Pegasus** memory management system. We then describe the modifications we made to our original design to support generations. This is followed by our preliminary performance results, a discussion of future work and directions, and conclusions.

II. Memory Management Techniques

Since the invention of Lisp, programmers have sought ways to efficiently manage dynamic memory[Stoyan84]. We survey some of the noteworthy techniques, after introducing a few useful terms. For a more complete survey see [Cohen81] and [Ungar84].

Objects are allocated in the *HEAP*. The set of objects that are transitively reachable from the stack, registers, and static data is called the *LIVE* set. Objects that are not reachable form the *DEAD* set until they are reclaimed, at which time they become part of the *FREE* set. The *HEAP* is the union of *LIVE*, *DEAD*, and *FREE*.

There are three aspects to the design of a memory management system: the representation of objects and the organization of the heap; the method of allocation; and the method of reclaiming *DEAD* objects. The simplest and potentially cheapest method of storage management is to require the user to explicitly free objects when they become *DEAD*. Given *a priori* knowledge of the sizes of objects, it is possible to realize very fast constant time allocation and deallocation by maintaining a free list for each object size. Other methods are preferable when sizes are unpredictable. In applications with shared data structures, it is often difficult to know when an object becomes *DEAD*. For these reasons, this technique is not applicable to implementations of higher-level languages, such as **Lisp, ML,** or **PML**.

The method of reclamation can be characterized as: *batch*, where other computation is suspended while *DEAD* objects are reclaimed; *incremental*, where reclamation is interleaved with

2. In interactive programming environments there are considerable periods of time when no processing is going on, either because of interprocess synchronization, or because the user has taken a coffee break.

other computation; or *concurrent*, where a separate, asynchronously scheduled process is responsible for reclamation. Incremental and concurrent methods are considered *real-time* if user computation, ignoring pathological cases such as heap overflow, is not suspended for an unbounded amount of time because of memory management (either allocation or reclamation). For interactive systems, such as **Pegasus**, real-time behavior is essential.

II.1 Reference counting

One solution to the problems of explicit storage management is to count the number of references to an object. When the count reaches zero, then the object is *DEAD* and can be reclaimed. This technique has been used in **Smalltalk-80**[3] and other systems[Goldberg83][Cohen81].

Although reference counting is an incremental technique, most implementations do not guarantee real-time behavior. When an object is freed, the reference counts of any objects it references are decremented and may also become zero. This recursion can lead to an unbounded amount of work[4]. The overhead of updating and testing reference counts can be high[Ungar84]. Another disadvantage of reference counting is that it cannot reclaim cyclic structures.

II.2 Deferred reference counting

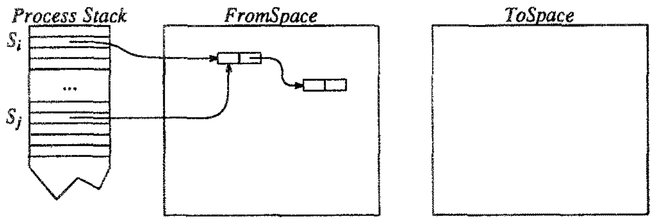
Empirical studies of **Lisp** programs have lead to the observation that most *LIVE* objects have exactly one reference to them[CG77]. Deferred reference counting, also known as Deutsch-Bobrow reference counting, takes advantage of this to reduce the cost of reference counting[DB76]. Reference counts are not stored with the objects. Instead, two tables of objects are maintained: the Zero Count Table (ZCT), which contains those objects with reference count zero, and the Multiple Reference Table (MRT), which contains those objects with reference counts greater than one together with their counts. Objects with reference counts of one, usually over 80% of all objects, are not recorded. Also, only references from heap objects are counted; references from stack variables are not recorded. This saves time in updating references from the stack.

Increment and decrement transactions to be applied to the ZCT and MRT may be executed immediately or buffered. When the ZCT becomes full, it is time to reclaim *DEAD* objects. Those objects that are in the ZCT and not referenced by the stack or by static data are *DEAD*. If the transaction buffer is keyed by object address it is possible to cancel and combine transactions. For example, allocation followed by an increment cancels out. This kind of cancellation can be done at compile time[Barth77][JM81].

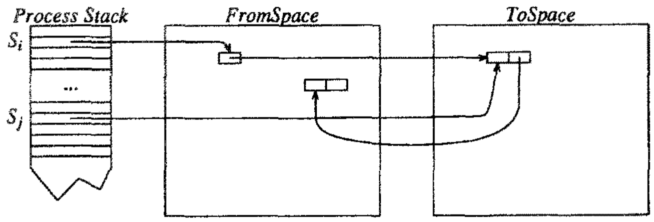
3. **Smalltalk-80** is a trademark of Xerox Corporation.
4. Real-time behavior is possible if de-allocation is interleaved with other computation, for instance by using a work list.

II.3 Semispace copying

Semispace copying is a technique that separates *LIVE* objects from *DEAD* objects[FY69]. It has the advantages of reclaiming dead cyclic data structures, compacting *LIVE* objects, and requiring only O(|*LIVE*|) work. Memory for dynamic objects is divided into two spaces, *FromSpace* and *ToSpace*. New objects are allocated in *ToSpace*. *FromSpace* is initially unused. When allocation reaches a threshold, garbage collection begins with a *flip* of *FromSpace* and *ToSpace*, exchanging their names. The plan is to free *FromSpace*, reclaiming its storage, by moving all *LIVE* objects to *ToSpace*. The operation of moving a *FromSpace* object to *ToSpace* is called *forwarding*. When an object is forwarded, a *forwarding pointer* to the object is left in its place. Later, if another reference to a forwarded object is discovered, the forwarding pointer is followed to find the object. The operation of examining an object reference and forwarding the object if it is in *FromSpace* is called *scavenging*. Scavenging begins with global variables and the stack, and continues until all *LIVE* objects have been forwarded and scavenged.



(a) Before scavenging S_j.



(b) After scavenging S_j.

Figure 1. Semispace copying.

Figure 1a shows a pair object to which there are two references from the stack, S_i and S_j, before either has been scavenged. The second element of the object contains a pointer to another pair. Figure 1b shows the same object after S_j has been scavenged by the garbage collector. To dereference S_i, the forwarding pointer in *FromSpace* must be followed.

II.4 Real-time semispace copying

Baker's incremental, real-time version of semispace copying[Baker78] interleaves garbage collection with other computation. Allocation operations (e.g., **cons**) scavenge the initial object contents, and accessing operations (e.g., **cdr**) scavenge their result. New objects are allocated at one end of *ToSpace* and forwarded objects at the other; the garbage collector only needs to sweep the copied objects. Sweeping of *ToSpace* and the user data (stack and registers) is interleaved with user computation by scavenging a bounded number of words each time an object is allocated.

Most implementations of semispace copying use a tag bit to distinguish forwarded objects. Unfortunately, operating on tagged data is expensive on stock hardware. Usually several instructions are needed to examine a tag and take the correct branch when dereferencing a pointer to an object. Also, although the scheduling of incremental semispace copying ensures real-time response, there is garbage collection overhead on allocation operations.

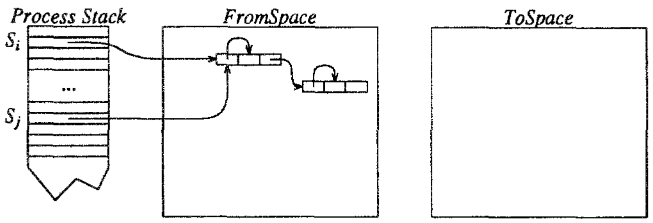
Brooks' modification of Baker's real-time semispace copying[Brooks84] trades space for time on stock hardware by eliminating tag checking for forwarding pointers, and allowing programs to reference *FromSpace* objects. The cost is one word per object and an extra level of indirection for object references. Every object contains an explicit forwarding pointer that is always followed to find the object's contents. The forwarding pointer is set when the object is allocated, and updated when it is forwarded. Run-time code simply follows the forwarding pointer to access an object. Under Brooks' version, *FromSpace* objects may still be referenced; the only restriction on user processes is that they may not store a *FromSpace* reference in an existing object. This saves time when operations are open-coded on a machine without hardware data tags. For example, on the Motorola MC68000 processor, **cdr** with explicit forwarding pointers takes only about 60% of the time of the standard Baker version (see [Brooks84] for a more detailed analysis).

To illustrate, figure 2 shows the same situation as figure 1 above. The objects have an extra word each for their forward pointers. Initially these point to the respective contents of each object. In figure 2b we see the objects after S_j has been scavenged. Note that the forwarded object may be found from either S_i or S_j by following the extra level of indirection.

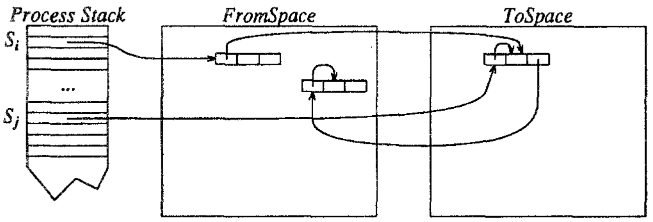
One problem with real-time copying garbage collectors is *floating garbage*[Dawson82]. This consists of those objects in *ToSpace* that become *DEAD* while scavenging is in progress. If the amount of floating garbage is large, then the garbage collector cannot reclaim memory efficiently.

II.5 Generation-based garbage collection

Generation-based garbage collection[LH83] is a form of copying garbage collection that exploits two observations about heap objects: one, that new objects have a high death rate, whereas old objects have a low death rate; and two, that new objects tend to refer to old objects, whereas old objects rarely refer to new objects. Garbage collection efficiency can be improved by dividing objects into generations, and reclaiming the storage of younger generations more frequently. References from older objects to younger objects are recorded in small *entry tables*, one for each



(a) Before scavenging S_j.



(b) After scavenging S_j.

Figure 2. Brooks semispace copying.

generation. To reclaim the storage of a generation, the GC need only scavenge its entry table and newer generations, thus avoiding the work of scavenging older generations.

Berkeley Smalltalk and the SOAR system use a generation-based batch approach to storage reclamation[Ungar84]. The oldest generation of objects is considered permanent. Objects that reference newer objects are kept in the *remembered set* for their generation, whose function is similar to that of the entry table. The garbage collector reclaims storage of newer generations by tracing live objects, starting from the remembered set and registers (stack frames are heap allocated). An object that survives enough scavenges eventually migrates to the oldest generation. Storage for ‘‘permanent’’ objects is reclaimed off-line, using mark-sweep garbage collection. Although scavenging suspends all other computation, it runs at high rates on small sets of objects, so that pauses are quick (usually around 100 ms.) and total garbage collection overhead is only 1-2%. This is very low overhead, but a 100 ms. pause in mouse tracking every ten seconds is definitely noticeable.

II.6 Concurrent garbage collection

Because garbage collection is not tightly coupled to user computation, it is an attractive target for concurrency. One approach is to use a dedicated processor for garbage collection; [Steele75] describes a copying garbage collection algorithm for two processors. Several papers have been written describing concurrent mark-sweep collection algorithms and proving their correctness[DL78][Ben-Ari84]. The major thrust of these papers, however, is correctness; no implementations have been reported.

A concurrent garbage collector has been built for **Cedar**[Rovner85]. Its target environment is very similar to **Pegasus** — multiple lightweight processes and interactive applications. The garbage collector is based on deferred reference counting and uses a back-up mark-sweep collector for reclaiming cyclic data. The implementation relies on special microcode and data-tags. Two interesting features are the use of "snap-shots" to scavenge the stack, and the support for object finalization.

III. Overview of Pegasus

To describe the design and implementation of our memory management system, we must first present an overview of **Pegasus**. In particular we are concerned with **PML**, described above, and the **Pegasus Virtual Machine (PVM)**, which provides the run-time support for **PML**, including memory management. For a more complete description of **Pegasus** see [RG86].

The **PVM** is organized into three levels: level-0 is the nucleus, providing the lightweight processes and low-level synchronization primitives; level-1 is a collection of services including the channel primitives and memory allocation; and level-2 consists of a small collection of system processes including the GC process. On top of this run the user processes, written in **PML**. Each process, including level-2 processes, has a *process stack*, and a copy of the machine's registers called the *process registers*. One of the services provided by level-1 is access to the process registers and stacks.

The implementation of the **PVM** imposes several constraints on the memory management system:

i) Since process scheduling is pre-emptive, access and updating of shared data must be atomic. Level-1 provides primitives to make the execution of a region of code atomic for a small cost (6 μsec. for our test machine; see Section V).

ii) An important goal of **Pegasus** is to insure smooth interactive response. To guarantee this, it is necessary to avoid locking up the processor for long periods (more than 50 msec.) This means that the execution time of atomic regions must be bounded. It also means that memory management must be real-time.

iii) The **PVM** is implemented on top of the resident system and uses many of its services. Because these routines are written and compiled by someone else, we have no control over them. Thus we cannot rely on our own stack frame or register allocation conventions. We do rely on the minimal assumption that stack frames are word aligned[5].

IV. The Pegasus Memory Management System

Our memory management system is based on copying objects with explicit forwarding pointers[Brooks84]. Garbage collection (GC) is done by a separate lightweight process that runs in the background. Our expectation is that there will be enough spare cycles for the GC to run unobtrusively. Our strategy, then, is to shift memory management overhead from user processes to the GC. This should improve the performance of foreground processes, as compared to incremental GC which requires the user processes to do the work.

We first describe the representation of objects and the data structures for managing the heap. Then we describe the allocator, the GC process, and the implications of concurrency. Finally we describe an extension to support finalized objects.

IV.1 Object representation and heap organization

Heap objects consist of one or more words, where a word is an address-sized quantity. Objects are either *tuples*, which consist of object references and other data, or are *atoms*[6]. In addition, objects are divided into several classes by size. So we have tuples which can be singletons, pairs, triples, or n-tuples (with an additional size word), and atoms which can be one-word atoms, two-word atoms, or n-word atoms (with an additional size word).

To keep track of the classes of objects, the system maintains a Big Bag of Pages (BIBOP) index[Steele77]. A page in the heap is either *FromSpace* or *ToSpace* and contains only one class of object[7]. The BIBOP index is used to map object references to the class and space of the object. Thus when the GC scavenges an object reference, it can determine if the object should be forwarded and how many words to copy.

As in [Brooks84], we associate a forward pointer (FP) with each object. Object references are represented by pointers to the associated FP. We follow the FP to access the object's contents. By using explicit forward pointers we avoid the cost of checking tag bits when accessing objects, which is expensive on stock hardware.

Because we don't have hardware tags and because **Pegasus** includes and supports code generated by different compilers, there are some subtleties in scavenging the process stacks and registers. Although we know that object references are word aligned in the stack, we cannot verify that an aligned word in the stack is an object reference and not something else (e.g., a floating point

5. Note that we do have control over any code that manipulates heap objects. Programs that manipulate **Pegasus** objects, but written with foreign compilers, use **Pegasus** macros or libraries.
6. As shown below, this distinction could be avoided, but it allows the GC to avoid unnecessary scavenging of atomic data.
7. These memory allocation "pages" should not be confused with the hardware-defined virtual memory pages.

number). Therefore, we adopt a conservative strategy that we call ***heuristic scavenging***. The GC assumes that every word in the heap, process stacks, registers, and channels is an object reference unless proven otherwise. This ensures that all *LIVE* objects are copied.

On the other hand, the GC may misinterpret a word as a *FromSpace* object reference. When there is such a false reference, the GC would corrupt the word if it updated it. To prevent this, we keep the forwarding pointers in a separate space (*FP-space*); this is similar to the object tables of some **Smalltalk** implementations[BS83]. A reference to an object is always an address in *FP-space* and only FPs point directly to objects, except for short-lived temporaries (see below). To scavenge a word, the GC first uses the BIBOP index to check if it is a valid reference to an FP. If so, the GC checks if the FP contains a *FromSpace* reference. If both these tests succeed, the GC forwards the object. Since only the FP is updated, the GC never corrupts the process stacks or registers.

Figure 3a shows our example under the **Pegasus** memory system. Both stack references point to the object's FP, and the second element of the first object points to the second object's FP. In figure 3b, after S_j has been scavenged, the object has been forwarded. Neither stack reference was updated, only the FP.

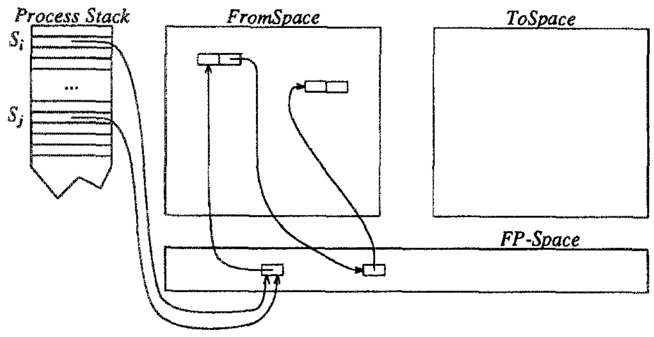
An advantage of keeping the FPs separate from their objects is that only one FP per object is needed, as opposed to two in Brooks' method (one *FromSpace* and one *ToSpace*). The penalty of heuristic scavenging is that a few *DEAD* objects may be accidentally forwarded, but the probability of this is small. A *DEAD* object forwarded because of a false reference can be reclaimed later, if the false reference is destroyed.

Managing *FP-space* is an important aspect of our system. FPs must be allocated and reclaimed. An incremental reclamation system such as is used in [BS83] is inappropiate for our system, both because of the overhead of tag bits and because it places the cost of FP reclamation on the allocating process. Our first approach was a simple linear scanning of *FP-space* at the end of each GC pass; any FP containing a *FromSpace* reference was reclaimed. This, however, involves $O(|HEAP|)$ work and turned out to be a real bottleneck in the system.

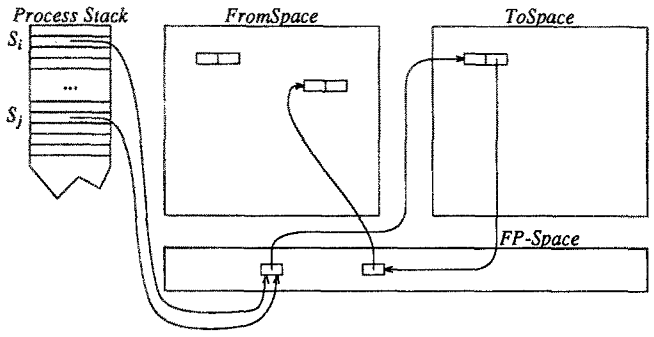
Our solution to this problem is to divide the *FP-space* into small *forward pointer groups* (FPGs). We keep a list of those FPGs that are in use (one or more *LIVE* FPs), and those that are free. In addition, there is the *allocation* FPG, which is where user processes get new FPs from. When the allocation FPG is used up it is placed on the in-use list and a new allocation FPG is aquired from the free list. At the end of each pass the GC process sweeps those FPGs that were on the in-use list at the start of the pass. This sweeping involves $O(|HEAP - FREE_0|)$ work, where $FREE_0$ is the set of free objects at the start of the pass.

IV.2 The allocator

Each object class has its own allocator, which follows the same basic form. This involves allocating storage for the object's contents from the appropriate page, allocating a new FP, and



(a) Before scavenging S_j.



(b) After scavenging S_j.

Figure 3. Pegasus GC.

initializing the FP and object contents. Unlike [Baker78] and [Brooks84], allocation does not involve any scavenging; the contents of new objects are scavenged by the GC process. Because the heap data structures are shared, allocation is an atomic operation.

The expected case is fast; the exceptional case arises when the free FPG list is empty. When all the FPGs are in use, the caller blocks until some are reclaimed. This is the only time that allocation does not exhibit real-time behaviour, but this only happens if the heap is full or the rate of storage reclamation is too low.

IV.3 The garbage collector

The basic structure of the GC process follows conventional lines. When memory usage reaches some threshold the GC condemns *ToSpace* (making it *FromSpace*) and creates a new *ToSpace*. The

GC first scavenges static data, and then user data and *ToSpace*. The channel message queues are the last objects scavenged. After scavenging is completed the FPs are swept (as described above), and finally *FromSpace* is reclaimed. The GC process performs this pass as a series of small atomic operations — small to insure real-time behaviour, and atomic because the heap data structures are shared.

The GC interleaves the scavenging of user process data and *ToSpace*. If there are unscavenged objects in *ToSpace* then the GC works on them, otherwise it scavenges process data. Because allocation does not scavenge the initial object contents, all *ToSpace* objects must be scavenged by the GC. Scavenging of process stacks is done incrementally bottom-up. When a process' stack is completely scavenged, then its registers are scavenged. The strategy is to scavenge the more volatile user data last. Because user processes continue to execute while the GC is scavenging, there are some subtleties to be aware of. We discuss these in the next section.

IV.4 Concurrency implications

Process scheduling in **Pegasus** is pre-emptive, which means that special care must be taken when operating on shared data. The data shared by the GC and user processes falls into three categories: the heap data structures (FPG lists etc.), the heap (*FP-space*, *FromSpace*, and *ToSpace*), and the user process data (stacks and registers). As noted above, the integrity of the heap data structures is perserved by restricting operations on them to atomic regions. The other shared data is more difficult to deal with.

One difficulty with the heap is that the GC process might forward a *FromSpace* object while a user process is accessing the object. This isn't a problem for non-destructive accessing operations, since the *FromSpace* copy of the object is identical to the *ToSpace* copy. If the user process is updating the object, however, there are two problems: one, that the user process might modify the obsolete *FromSpace* version of the object, and two, that the user might store a *FromSpace* reference into a *ToSpace* object that has already been scavenged by the GC process. We deal with these problems by requiring that object updating be done atomically, and that the new value being stored in the object be scavenged. We feel that the extra cost on object updates is acceptable, especially since we are trying to encourage applicative-style programming.

The other problem with the heap is that the user might have an obsolete pointer to the *FromSpace* copy of the object when *FromSpace* is reclaimed. For this reason we restrict the use of pointers into *FromSpace* and *ToSpace* by user processes. User processes cannot keep such pointers for long or unbounded amounts of time (e.g., across function calls). Thus, once scavenging is complete, if every non-blocked user process runs for at least one time-slice, then there will be no pointers into *FromSpace*, and we can safely reclaim it. The time that it takes the GC process to sweep the FPGs usually is sufficient to satisfy this condition, but, just in case, the GC examines the top stack frame and registers of the user processes before reclaiming *FromSpace*. If it sees anything suspicious, it waits a while before reclaiming *FromSpace*.

As mentioned above, scavenging of user process stacks is done incrementally. Each stack frame is scavenged atomically (large frames can be broken up to preserve real-time behaviour), but user processes may run between the scavenging of frames. This means that the GC needs to be aware of possible changes in the stack it is scavenging. The GC keeps track of its progress by use of *swept-stack* marker stored in each stack frame[8]. Stacks are swept in round-robin order, working on the bottom-most set of unswept stack frames each time a stack is visited. If a process has run since the last time the GC worked on its stack, then the GC must find the highest marked stack frame in the stack to use as a starting point.

Care must be taken by the GC to ensure that all areas are completely swept. When the GC has swept all objects in *ToSpace* and the top frames of all processes, it enters *completion* phase. In this phase the GC tries to finish scavenging without yielding the processor. This is necessary to ensure that no process register or top stack frame contains an unscavenged *FromSpace* reference. In *completion*, the GC resweeps the top stack frame of each process, and then sweeps the channels. If the GC finds previously unscavenged *FromSpace* references to large data structures during *completion*, the GC may not be able to complete in a single atomic action (because of real-time behaviour constraints). If this is the case, it will have to try later on. Usually, however, *completion* can be done atomically. Our experience with our system suggests that this is the area that needs the most work; in particular, we have problems with processes that exhibit volatile stack behaviour. We discuss a new design that we are working on in section VII.

IV.5 Finalized objects

It is often useful to have some action taken when an object dies; for example, **Pegasus** currently runs on top of the resident graphics library. A **Pegasus** bitmap object contains a pointer to bitmap storage that is managed by the library, so when a **Pegasus** bitmap object dies it is necessary to call a library routine to free the bitmap storage. This is called *object finalization*[Rovner85].

The **Pegasus** memory management system supports finalization of objects, by supporting a class of finalized objects. Each finalized object has an associated finalization procedure. The finalization procedure takes an object reference as an argument, and contains actions to be executed when an object is known to be *DEAD* but before its storage is reclaimed.

To distinguish finalized objects, we use keep a separate allocation FPG and in-use FPG list for finalized objects. The first word of a finalized object is used to store a pointer to the finalization procedure. The GC recognizes the death of a finalized object when sweeping the finalized object

8. This word is set to zero upon procedure entry as part of the standard Apollo calling convention. We realize that it is non-portable (in fact future versions of the Apollo compilers will not support it), and discuss an alternative implementation in section VII.

FPGs. When a finalized object dies, we resurrect it by copying it to *ToSpace* and put it on a work list. There is a special finalization process that removes objects from this work list and calls the appropriate finalization procedure. Once an object has been finalized, we zero its first word and destroy any references to it. This allows the GC to tell if a finalized object has been finalized.

V. Generation based garbage collection

As discussed in section II.5, there is an advantage to grouping objects into generations[LH83][Ungar84]. We have implemented a version of our system that partitions the heap into a small fixed number of generations, each with its own *FromSpace* and *ToSpace*. New objects are allocated in the youngest generation's *ToSpace*. After an object survives a fixed number of scavenges in a generation, it is *promoted* to the next older generation. Objects in the oldest are not promoted. The changes we made to our system to support object generations were mostly in the heap data structures, some changes to the way we scavenged *ToSpace* were also needed. The allocation routines remained unchanged.

Each generation has its own heap pages. We extend the BIBOP index to store the generation number of each page. The FPG lists provide a natural mechanism for recording the age (or *sub-generation*) of an object. If the objects of generation i (G_i) need to survive n scavenges before being promoted, then we maintaing n in-use FPG lists for G_i.

To handle references from objects in older generations to objects in younger generations, we use *remembered object lists* (after [Ungar84]). Each G_i has a list R_{ij}, for all G_j older than G_i, of those objects in G_i that are referenced (*remembered*) by objects in G_j. Objects are added to these lists as a side-effect of object updating. We assume that the number of object updates is small compared to the number of object allocations, so no attempt is made to avoid duplicate entries or to remove obsolete entries when adding objects to the list. Storage for these lists is reclaimed during scavenging (see below).

The advantage of generation-based scavenging is that the GC does not work on the entire heap every pass, but, instead, concentrates on the younger generations where the density of *LIVE* objects is lowest (recall that copying is $O(|LIVE|)$). Our system starts each pass by choosing the oldest generation that it will work on during that pass. It then condemns that generation and all younger generations. After this is done, the objects of the oldest sub-generations of the condemned generations are promoted. Promotion involves forwarding the objects into their new generation's *ToSpace*, and and adjusting the in-use FPG lists.

After object promotion, the GC starts scavenging with the remembered object lists. For each condemned G_i and uncondemned G_j the GC scavenges R_{ij}. This covers all the references from uncondemned generations to condemned generations. Note that any reference added to these lists by a user process during the GC pass will be scavenged by the update operation. The lists R_{ij}, where both G_i and G_j are condemned, are rebuilt, removing obsolete entries, while scavenging condemned

ToSpace. The rest of scavenging follows the form described in section II.

Our method for dealing with remembered objects is simple and avoids expensive operations by user processes when updating objects. It has the disadvantage that duplicate and obsolete entries can fill the remembered lists with useless information. We do not have any data yet that suggests that this is a problem, but if it is, there are several possible solutions. Two that we are considering are: removing duplicate entries when rebuilding the lists during scavenging (this could, in fact, be done on all the lists); and using hash tables to represent the lists, thus avoiding duplicate and obsolete entries entirely. The first of these has the advantage that is avoids any extra work by user processes, but it might not sufficient to solve the problem.

VI. Performance

To evaluate our system we made some preliminary measurements of its performance. The benchmarks test burst rate of allocation and scavenging, and overall throughput under various patterns of heavy allocation[9].

We used the generation-based version of our system for these tests, with four generations, each composed of four sub-generations. Our FPG size was 32 FPs per FPG. We limited the number of objects by restricting the number of FPs to 256K. This, in effect, limited the size of *FromSpace* and *ToSpace* each to 256K times the object size.

The benchmarks were written in C++ and run with our prototype implementation of the **PVM** on an Apollo DN3000 workstation[10]. The machine we used has a 12 MHz MC68020 processor with 4 megabytes of physical memory. This amount of memory is not large for a system such as **Pegasus**, especially since the Apollo operating system and display manager underlying the **PVM** consume a large fraction of the available memory. As a result, the paging behavior for some of our tests was fairly bad. Consequently we analyzed performance in terms of CPU-time, which does not include paging costs. It should be emphasized, however, that the results below are preliminary.

VI.1 Burst rate performance

We measured burst rate allocation and scavenging for various kinds of objects. The sample size for the 256-word atoms was 1024 (1K) objects, the sample size for all other objects was 16K objects. These measurements give a good indication of the base cost per object in our system.

The allocation rate measurements (see table 1) are the most meaningful numbers we have. This is because allocation is unaffected by system load, unless the heap overflows.

9. Other results on garbage collection performance can be found in [Ungar84] and [Gabriel85].
10. DN3000 is a product of Apollo Computer Inc.

Class	TIME Real-time (sec)	CPU-time (sec)	ALLOCATION RATE (in CPU time) 1,000's bytes/sec	1,000's objects/sec	μsec/object
singletons	0.83	0.82	79.9	20.0	50
pairs	0.92	0.91	144.0	18.0	56
triples	0.92	0.90	218.5	18.2	55
16-tuples	3.94	3.49	319.2	4.7	213
one-word atoms	0.87	0.85	77.1	19.3	52
two-word atoms	0.90	0.91	144.0	18.0	56
16-word atoms	4.01	3.49	463.7	4.7	213
256-word atoms (1K)	2.67	2.27	452.9	0.5	2,217

TABLE 1. Allocation burst rate.

To measure the burst rate of object scavenging, we allocated our sample of objects as a linked structure with the GC stopped. We then started the GC and measured the time it took to complete a single pass. Since all the objects we allocated were *LIVE*, the measurements in Table 2 represent worst case cost per allocated object for copying garbage collection. On the other hand, these numbers don't reflect the overhead of other processes running, so they are only useful for establishing a bound on the performance of the system. One interesting point is that the cost of scavenging atoms is less than that for tuples of the same size. This is expected since the contents of atoms don't need to be scavenged.

Class	TIME Real-time (sec)	CPU-time (sec)	SCAVENGING RATE (in CPU time) 1,000's bytes/sec	1,000's objects/sec	μsec/object
singletons	1.32	1.28	51.2	12.8	78
pairs	1.44	1.40	93.6	11.7	85
triples	1.60	1.59	123.7	10.3	97
16-tuples	20.90	9.33	119.4	1.8	569
one-word atoms	1.22	1.18	55.5	13.9	72
two-word atoms	1.22	1.18	111.1	13.9	72
16-word atoms	12.95	6.47	172.2	2.5	394
256-word atoms (1K)	6.69	4.03	261.2	0.3	3936

TABLE 2. Scavenging burst rate.

VI.2 Overall performance

To gauge the overall performance of our system we ran four sets of benchmarks. Each benchmark set used a different kind of allocator process. Table 3 summarizes the three allocator processes, giving the number of objects allocated and the maximum size of *LIVE* for a single

Benchmark	*max.* |*LIVE*|	# of objects allocated	# of bytes / object
Ackermann (3, 7)	1,025	693,964	8
Lists	16,384	1,048,576	8
N-tuples	273	17,472	68
Quick-sort[11]	1,941	336,784	8

TABLE 3. Benchmarks

allocator process. In our tests we varied the number of allocator processes to test the sensitivity of our system to the number of user processes.

We also measured the performance of other kinds of memory management. We made measurements of two kinds of explicit memory management — free lists and **malloc**, an "off the shelf" C library allocator[Vo86][12]. And we implemented and measured a package of reference counted objects. These measurements were made using the same allocator processes as with our GC[13]. Although, as noted in Section III.1, explicit management is insufficient for supporting languages like **PML**, these allocators do give us a base-line for comparison of our system's performance.

The allocation process for the first set of tests computes the Ackermann function $A(3,7)$ using a **PML**-style calling convention where the arguments are passed as a heap-allocated pair. The maximum size of *LIVE* in this test is much smaller than the number of objects allocated, thus the number of objects forwarded each pass is small. Most of the GC time was spent dealing with the deep, volatile stacks of the allocator processes. Figure 4 gives the results of this benchmark. As expected, the free list allocator is the fastest by far. What is interesting is that our system is faster than the "off the shelf" allocator and reference counting.

The second set of tests measures memory management performance under heavy allocation of large lists. The allocator process iteratively creates long lists (16K elements) and then destroys the list headers, creating *DEAD* objects as fast as it can. The results of this test can be found in figure 4. The GC process runs almost continuously and forwards large amounts of floating garbage.

The third set of tests concern large objects (see figure 6). An allocation process recursively creates a complete 16-ary tree of three levels, and then destroys the root node to make *DEAD* 16-

11. There is an additional 512 element list that is shared among all the sorting processes.
12. We also tried the standard Unix 4.2BSD allocator, based on linked lists of free blocks sorted in ascending order of address and coalesced while searching for free space during allocation, but found its performance to be so poor that we were unable to run some of our tests to completion.
13. Note that these other implementation avoid the level of indirection provided by FPs.

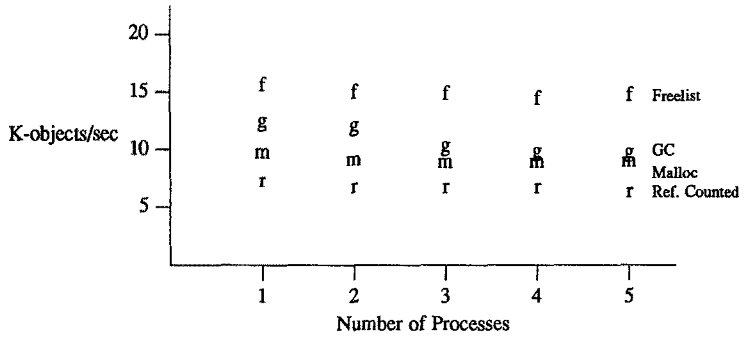


Figure 4. Ackermann benchmarks

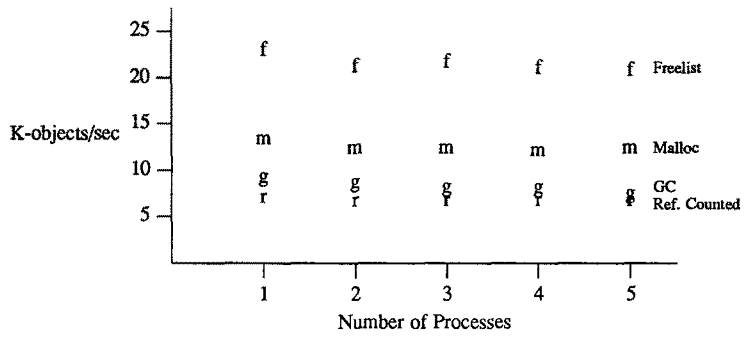


Figure 5. List benchmarks.

tuples. Each allocation process builds and destroys 64 trees.

Our final benchmark is an implementation quick-sort on lists of integers. The implementation is completely recursive and uses the **PML**-style calling convention of heap-allocated arguments. A single, randomly generated, list of 512 elements is shared by the sorting processes. Each sorting process sorts this list 16 times. This benchmark is interesting for several reasons: it is more typical of a real application than our other benchmarks; it intermixes allocation of objects for list construction with allocation of objects for argument passing; and it is an example of a program where explicit storage management is not possible because of object sharing. For this last reason, we did not run either the free-list or **malloc** tests. We also measured the FPG fragmentation rate for quick-sort, and found it to be surprisingly low (less than 5% with an FPG size of 32 FPs). Since quick-sort is fairly typical of **PML** code in the intermixing of object allocation for data structures

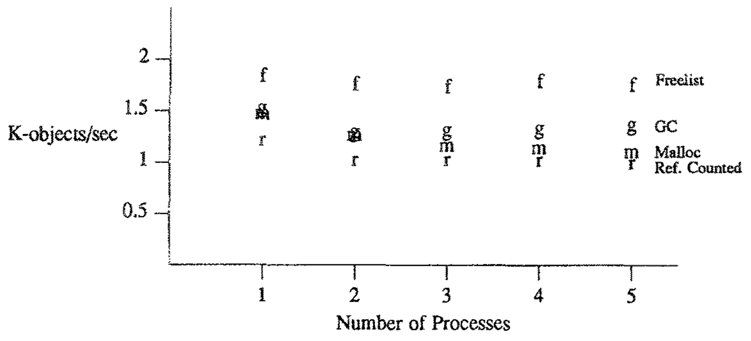


Figure 6. N-tuple benchmarks.

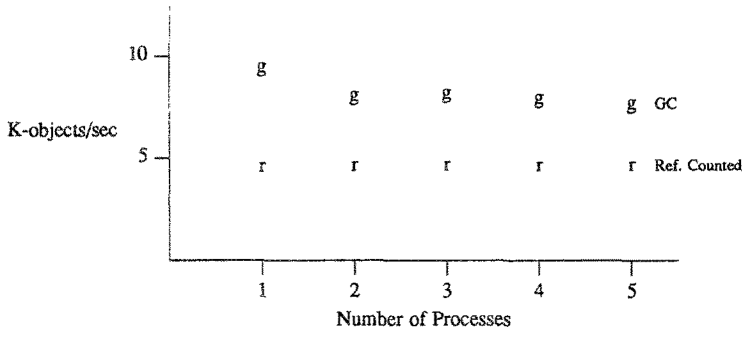


Figure 7. Quick-sort benchmarks

and object allocation for argument passing, we feel that FPG fragmentation will not be a problem.

VI.3 Discussion

The burst mode measurements show that allocation and scavenging are fast. Of the overall performance numbers, we feel that the Ackermann and quick-sort benchmarks are the most valid. Though all of the benchmarks involve much heavier allocation rates than we expect under typical usage, they do point out some trouble spots. In particular, the GC has problems with volatile user stacks, and as a result requires a long time to complete scavenging. This results in a great deal of floating garbage, further hurting system performance. We discuss some solutions to these problems in the next section. On a positive note, our system is consistently faster than reference counting.

VII. Future work

Although the performance of our system is quite satisfactory, having met the major goal of building a system that can utilize dead-time to reclaim memory, there are areas for future work.

i) Our current strategy for scavenging user data and *ToSpace* is unsatisfactory; our tests have shown that the GC has trouble completing scavenging in the presence of volatile user stacks. We are currently reimplementing the GC to solve this problem. The new implementation maintains a work list of FPs for the GC to scavenge. The GC alternates between scavenging FPs from the work list and scavenging *FromSpace*. When it runs out of things to do, the GC takes a ''snap-shot'' of a process' stack and registers, adding any *FromSpace* references to the work list. In order to minimize the disruption caused by this, we have introduced a new operation in the **Pegasus** kernel that allows the GC process to suspend a process while working on its stack. This means that only those processes with very deep stacks will be suspended for noticeable periods. Another advantage of this new design is that it does not use the swept-stack marker of our current design.

ii) Allocation is faster than garbage collection, so the GC can only keep up because other computation is interleaved with allocation. Under heavy allocation, however, the GC sometimes falls far behind, forcing user processes to block until the GC completes a pass. One way to avoid this may be an adaptive scheduling policy where the GC increases its scheduling priority in response to heavy allocation rates. Before we try this kind of tuning, however, we need a better understanding of the GC's behaviour in typical applications.

iii) Another way to improve performance is to code time-critical parts of the system in assembly language. The most likely targets are the allocation routines. They are a good choice for several reasons: they are a part of the system that must be executed by user processes and thus cannot be off-loaded to dead-time; allocation is a common activity of **PML** programs, thus any improvement in allocation performance may reap significant benefits in overall system performance; and the code for allocation is fairly simple, thus avoiding significant debugging problems.

iv) We have not considered the implications of garbage collection on virtual memory paging behavior. Pegasus currently runs inside a single heavyweight process, so a page fault suspends the entire PVM. We might be able to improve the paging behavior of the GC by altering the scavenging strategy to use approximately-depth-first order instead of breadth-first[Dawson82][Moon84].

v) We intend to move our system to a shared address-space multi-processor computer, where each processor has its own local memory and the ability to access other processors memory. There are a number of issues to consider with regards to our memory management system. Some of these are: how many GC processes should we have, one for each processor, or just one running on a dedicated processor; how do we distribute the heap data structures; and how do we control access to heap data structures?

vi) We need to do more benchmarking of our system. In particular, we need to study the effect of generations, measure "real" programs, and find out how effectively our system uses dead-time.

VIII. Conclusions

We have demonstrated a practical design for memory management in a concurrent system running on stack hardware. Under our modification of Brooks' forwarding pointers, the only run-time costs, owing to storage reclamation, incurred by user processes are an extra level of indirection when accessing object contents and the need to scavenge when updating mutable objects. Therefore, we believe that our system can successfully off-load much of the cost of memory management to, otherwise unused, dead-time.

Beyond **Pegasus**, this work has application to any system based on lightweight processes and heap allocation. We believe the ease with which the memory management system was implemented is a testimony to the soundness of **Pegasus** as a foundation for building programming environments.

References

[Barth77] Barth, J. "Shifting Garbage Collection Overhead to Compile Time," *Communications of the ACM*, V. 20, Nr. 7, July 1977, pp. 513-518.

[Baker78] Baker, H.G. "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, V. 21, Nr. 4, April 1978, pp. 280-294.

[Ben-Ari84] Ben-Ari, M. "Algorithms for On-the-fly Garbage Collection," *ACM TOPLAS*, V. 6, Nr. 3, July, 1984, pp. 333-344.

[BS83] Ballard, S., Shirron, S. "The Design and Implementation of VAX/Smalltalk-80," in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, ed., Addison-Wesley, Reading, Mass., 1983, pp. 127-150.

[Brooks84] Brooks, R.A. "Trading Data Space for Reduced Time and Code Space in Real Time Garbage Collection on Stock Hardware," *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Aug. 6-8, 1984, pp. 256-262.

[Cohen81] Cohen, J. "Garbage Collection of Linked Data Structures," *Computing Surveys*, V. 13, Nr. 3, September 1981, pp. 340-367.

[CG77] Clark, Douglas W., and Green, C. Cordell. "An Empirical Study of List Structure in Lisp," *Communications of the ACM*, V. 20, Nr. 2, pp. 78-87.

[Dawson82] Dawson, J.L. "Improved Effectiveness From a Real Time Lisp Garbage Collector," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, August 15-18, 1982, pp. 159-167.

[DB76] Deutsch, L.P., Bobrow, D. "An Efficient, Incremental, Automatic Garbage Collector," *Communications of the ACM*, V. 19, Nr. 9, September 1976, pp. 522-526.

[DL78] Dijkstra, E.W., Lamport, L., et al. "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, V. 21, Nr. 11, November 1978, pp. 966-975.

[FY69] Fenichel, R., Yochelson, J. "A LISP Garbage Collector for Virtual Memory Computer Systems," *Communications of the ACM*, V. 12, Nr. 11, November 1969, pp. 611-612.

[Gabriel85] Gabriel, R.P. *Performance and Evaluation of Lisp Systems*, The MIT Press, Cambridge, Mass., 1985.

[Goldberg83] Goldberg, A. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[JM81] Jones,, N.D., Muchnik, S.S. "Flow Analysis and Optimization of LISP-like Structures," in *Program Flow Analysis: Theory and Applications*, S.S. Muchnik and N.D. Jones, eds., Prentice-Hall, Englewood Cliffs, N.J., pp.102-131.

[LH83] Lieberman, H., Hewitt, C. "A Real-time Garbage Collector Based on the Lifetime of Objects," *Communications of the ACM*, V. 26, Nr. 6, June 1983, pp. 419-429.

[Milner85] Milner, R. "The Standard ML Core Language," *Polymorphism*, V. 2, Nr. 2, October 1985.

[MacQ85] MacQueen, D. "Modules for Standard ML," *Polymorphism*, V. 2, Nr. 2, October 1985.

[Moon84] Moon, D.A. "Garbage Collection in a Large Lisp System," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 6-8, 1984, pp. 235-246.

[Motorola85] Motorola Inc. *MC68020 32-bit Microprocessor User's Manual, 2nd Ed.*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[Rovner85] Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language," *Xerox PARC Report CSL-84-7*, July 1985.

[RG86] Reppy, J.H., Gansner, E.R. "Pegasus: A Foundation for Programming Environments," *AT&T Bell Laboratories Technical Memorandum*, December 1986. An earlier version of this appeared in the *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, December 9-11, 1986, pp. 218-227.

[Steele77] Steele, G.L. Jr., "Data Representations in PDP-10 MacLISP," *Proceedings of the 1977 MACSYMA Users' Conference*, NASA, Washington, D.C., July 1977, pp. 203-214.

[Stoyan84] Stoyan, H. "Early LISP History (1956-1959)," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 6-8, 1984, pp. 229-310.

[Ungar84] Ungar, D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 157-167.

[Vo86] Vo, K.P. AT&T Bell Laboratories, Murray Hill, N.J. Unix memory allocator, personal communication.