Barrier Methods for Garbage Collection

Benjamin Zorn

Department of Computer Science

University of Colorado at Boulder

CU-CS-494-90   November 1990

University of Colorado at Boulder

Barrier Methods for Garbage Collection

Benjamin Zorn

Department of Computer Science

University of Colorado at Boulder

November 1990

Abstract

Garbage collection algorithms have been enhanced in recent years with two methods: generation-based collection and Baker incremental copying collection. Generation-based collection requires special actions during certain store operations to implement the "write barrier." Incremental collection requires special actions on certain load operations to implement the "read barrier." This paper evaluates the performance of different implementations of the read and write barriers and reaches several important conclusions. First, the inlining of barrier checks results in surprisingly low overheads, both for the write barrier (2–6%) and the read barrier ($< 20\%$). Contrary to previous belief, these results suggest that a Baker-style read barrier can be implemented efficiently without hardware support. Second, the use of operating system traps to implement garbage collection methods results in extremely high overheads because the cost of trap handling is so high. Since this large overhead is completely unnecessary, operating system memory protection traps should be reimplemented to be as fast as possible. Finally, the performance of these approaches on several machine architectures is compared to show that the results are generally applicable.

1 Introduction

Garbage collection is a technique of automatic heap storage reclamation that has been used in Lisp runtime systems for many years. More recently defined block-structured languages such as Modula-3 [7] and Cedar [19] also include automatic storage reclamation. Even C, widely used as a systems programming language, has been extended to include garbage collection of heap-allocated objects [5].

Because of the widespread use of garbage collection in programming languages, careful evaluation of implementation techniques for these algorithms is extremely important.

1

Unfortunately, there has been little empirical data published about the behavior of Lisp systems and even fewer measurements comparing alternative implementations of garbage collection algorithms. As a result, many different methods have been published and implemented but there is very little consensus about the relative effectiveness of these methods. As a response to this lack of comparative measurement, this paper provides measurements of several different implementations of two very important garbage collection techniques.

Over the years, garbage collection algorithms have changed with the technology of the systems for which they are implemented. Copying collection algorithms, which copy objects back and forth between semispaces, have proven to be the most effective and widely implemented collection algorithms. In the last decade, two enhancements to copying algorithms have increased their effectiveness. Henry Baker proposed a technique called "incremental garbage collection" which greatly shortens the pauses associated with stop-and-copy algorithms [4]. In 1983, Lieberman and Hewitt introduced the idea of generation garbage collection, where only a fraction of the entire heap needs to be considered for most collections [16]. This enhancement both reduces the memory reference disruption during garbage collection and shortens collection-related pauses.

This paper considers several different implementation techniques for generation and incremental collection, and reaches important and suprising conclusions about the effectiveness of these implementations. In particular, inline techniques are shown to have excellent performance, even in the implementation of incremental garbage collection, which is generally though to be too inefficient without special hardware support. Furthermore, the use of operating system traps to implement these algorithms is shown to be highly inefficient due to the unnecessarily high execution cost of the trap handlers.

Before describing the implementations and evaluations, the basics of the algorithms are covered. The rest of this section introduces the fundamental ideas of copying garbage collection and the terminology used to describe the incremental and generation enhancements.
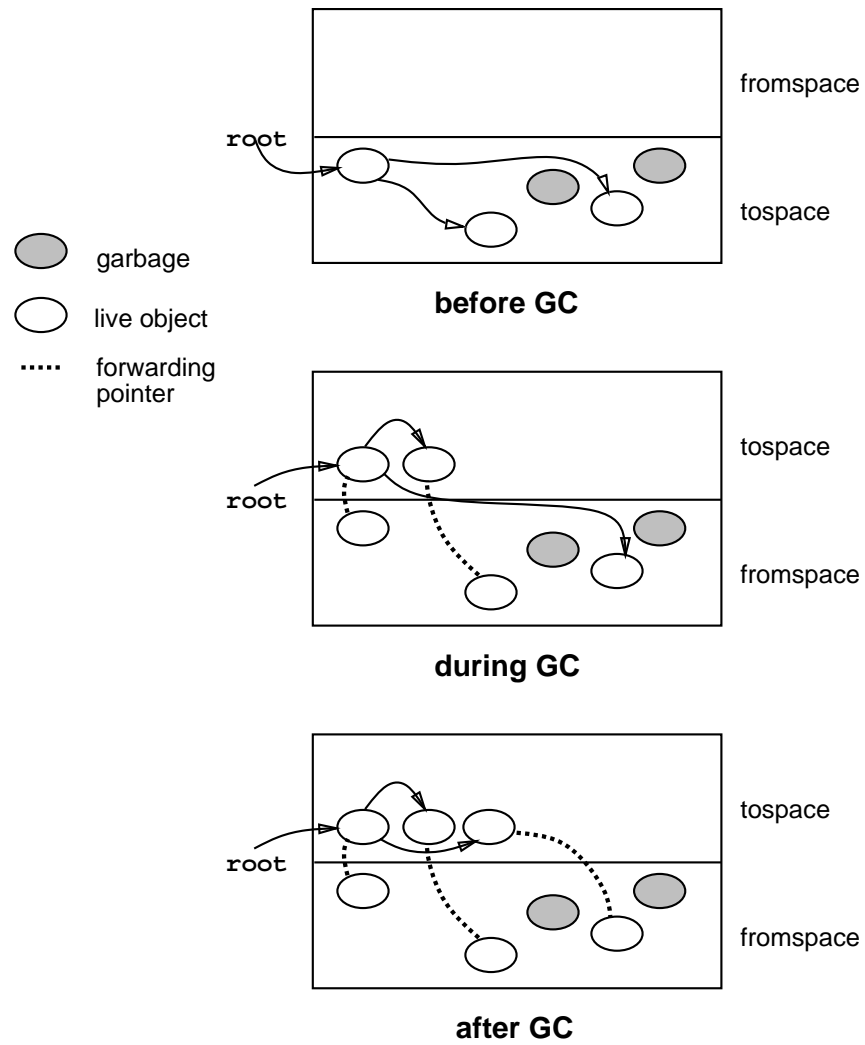
1.1 Copying Garbage Collection

The goal of garbage collection is to identify objects in the heap that are no longer accessible to the program and to reclaim them for reuse. "Garbage" objects are distinguished from "live" objects (objects still in use) by starting from a special set of objects (the *root set*) and transitively visiting all objects reachable from that set. By visiting all live objects, all other objects are known to be garbage and can be reclaimed.

The basic idea of copying garbage collection can be attributed to Fenichel and Yochelson [11], and to Cheney, who eliminated the need for auxiliary stacks in the algorithm [8]. Copying garbage collection reclaims garbage by dividing the address space into two semispaces (*fromspace* and *tospace*), only one of which is in use most of the time. During normal program operation, all objects are located in one semispace (tospace) in which new objects are also allocated. When the semispace is entirely consumed, a garbage collection cycle is invoked. The first operation performed is to exchange the names of the semispaces, resulting in a garbage collection *flip*. During the collection cycle, objects starting with the root set are visited and copied from fromspace (previously tospace) into the new tospace (previously fromspace). As objects are copied, a forwarding pointer is left in the old version of the object so that other pointers to the object can be correctly updated to point to its new location. After all reachable objects have been copied, program execution resumes. Only the live data has been copied, so there is now extra space in the new tospace in which to allocate new objects.

Figure 1 illustrates the location of objects in the semispaces before, during, and after a collection cycle. Note that the labels of the two semispaces change during the garbage collection flip. Also note that copying the live objects into tospace also compacts them, resulting in more localized memory references and better memory system performance.

Traditionally, a copying collection cycle is initiated when there is no more room to allocate in the current tospace. An alternative policy is to initiate a collection cycle when a fixed amount of memory has been allocated (the *allocation threshold*). This approach requires

Figure 1: Organization of a Simple Copying Garbage Collection Algorithm.

that there is sufficient virtual memory to allow the semispaces to grow. Basing collection on an allocation threshold, instead of using a fixed semispace size, has an important advantage. Fixing the semispace size and invoking garbage collection when a semispace fills leads to thrashing. Thrashing occurs when most of the memory in a semispace is allocated to live objects—as the semispace fills, garbage collection occurs more frequently and recovers less garbage each time. Invocation based on thresholds avoids thrashing because semispaces are allowed to grow as needed.

1.2 Generation Garbage Collection

Generation garbage collection enhances the standard copying algorithm by further dividing the address space into different regions or *generations*. Each generation is used to store objects that have been alive for different lengths of time. With generation garbage collection, individual generations are collected independently, greatly reducing the memory disruption and pause length normally associated with copying collection. Generation collection also takes advantage of the well-documented empirical fact that most heap-allocated objects become garbage shortly after they are allocated [20, 29]. By frequently collecting the youngest objects, the efficiency of collection is considerably increased.

In a generation algorithm, objects are allocated in the youngest generation (*newspace*). If an object remains alive long enough, it is eventually copied to the next older generation. There may be one or more older generations, collectively referred to as *oldspace*. The act of copying an object to an older generation is referred to as *promotion*. Objects are promoted when they become older because empirical evidence shows that older objects are less likely to become garbage. Objects are promoted based on the *promotion policy*. A typical policy, copy-count promotion, promotes objects after they have been copy-collected K times (where K is a small number).

To correctly collect a single generation (such as newspace), all live objects in the generation must be identified. In particular, if an object in a generation is pointed to by an object in another generation, that *intergeneration reference* must be included in the root set of the

5

generation being pointed into. In practice, generations are ordered by age, and only pointers forward in time (i.e., from older generations to younger ones) are recorded. With this implementation, when a generation of a particular age is collected, all younger generations must be collected at the same time. Figure 2 illustrates how generation collection can be used to enhance a copying collection algorithm.
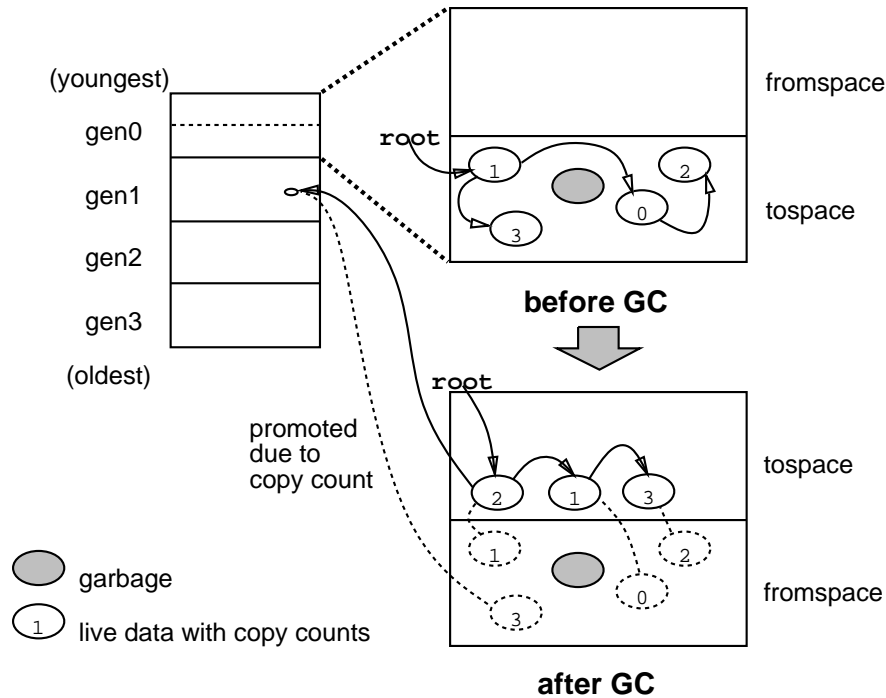
Figure 2: Organization of a Simple Generation Stop-and-Copy Collection Algorithm. Copy count promotion is used, where objects are promoted after they have been collected three times.

The record of pointers from older generations into younger generations is called the *remembered set*. The remembered set is maintained by identifying stores that create intergeneration pointers (from older to newer generations), also called maintaining the *write barrier*. When an intergeneration pointer is created, a *write barrier* or *generation trap* occurs and the intergeneration pointer is noted in the remembered set. Stores to the heap are a relatively uncommon operation in programs, typically accounting for about 1% of all instructions executed. Because maintaining the write barrier requires that all such stores are checked, the overhead associated with the write barrier can be substantial. This paper

considers several alternative implementations of the write barrier to determine the most efficient method.

1.3 Incremental Garbage Collection

Henry Baker introduced an enhancement of the standard copying collection algorithm that greatly reduces pauses associated with garbage collection [4], making garbage collection potentially suitable for real-time applications. The actions performed by Baker's incremental algorithm are essentially identical to those of the traditional copying algorithm—only the time at which the copying is performed changes. The important idea behind incremental garbage collection is that the copying of objects into tospace can be interleaved with the program's operations, thus completely eliminating collection pauses.

Baker's algorithm copies only objects directly reachable from the root set during the garbage collection cycle, leaving most objects remaining to be copied. The incremental algorithm maintains the illusion that <u>all</u> objects were copied during the garbage collection cycle. To maintain this illusion, all memory references to objects remaining in fromspace (not yet copied) must be avoided. Since memory references can only take place using addresses in machine registers, the key to Baker's algorithm is to prevent pointers into fromspace from getting into machine registers. Thus every time a pointer is loaded from memory, it is first checked. If it points into fromspace, the object is immediately copied to tospace (just as would have been done in a normal collection cycle), a forwarding pointer is left behind in its place, and the pointer into fromspace is updated to point to the tospace copy of the object. This check on pointer loads is sometimes referred to as the *read barrier* and is in many ways analogous to the write barrier of generation garbage collection. To insure that all objects are eventually copied, Baker's algorithm also associates copying with allocation—every time a new object is allocated K objects are copied from fromspace. This insures that tospace will not fill before fromspace is evacuated.

For Lisp systems with large address spaces, Baker's algorithm greatly reduces the pauses associated with garbage collection. This algorithm, coupled with generation collection,

has been used successfully for many years by Lisp machines such as the Symbolics [17] and Explorer [9]. Studies of the Explorer's algorithm by Courts suggest that the copy-on-demand strategy of incremental collection significantly improves the memory reference locality of Lisp programs by dynamically compacting the working set of the data [9]. Lisp machines provide hardware support for the read and write barriers, substantially reducing the overhead. Unfortunately, high performance RISC microprocessors do not provide any hardware support for barriers. An important contribution of this paper is that it examines implementation alternatives and determines the most effective implementation of barrier methods on RISC architectures.

1.4   Outline

The rest of this paper describes the evaluation of specific implementations of the read and write barriers. Section 2 discusses the evaluation methods used. Sections 3 and 4 provide an in-depth evaluation of several implementations of the write and read barriers, respectively. Section 5 discusses related research and Section 6 summarizes the results.

2   Evaluation Methods

This paper compares the performance of three possible implementations of the read and write barriers: hardware support, inline software tests, and methods using operating system memory protection mechanisms. The actual details of each implementation are described in the following sections. In all cases the cost of maintaining the barriers is considered as a fraction of the total program execution time (without including other garbage collection costs). For example, a 10% write barrier overhead means that the program takes 10% longer to execute if it performs the tests necessary to maintain the write barrier. Copying garbage collection typically add another 10–20% overhead depending on the program, but we do not consider those costs in this paper. By measuring the costs of implementations

8

in terms of relative overheads, implementations on different architectures can be compared fairly because processor speed is factored out.

To estimate the CPU overhead of different implementations, two measures are combined. Each implementation of interest requires particular actions to be taken when specific events occur. For example, to implement the write barrier with inline software tests, each store instruction must be accompanied by several instructions testing for intergeneration pointers. The CPU cost of an implementation can be estimated by measuring the number of particular events that occur in the execution of a program and multiplying that number by the measured CPU cost of the event. By factoring the overhead into number of events and cost per event, the CPU overhead on different machine architectures can be estimated simply by measuring the event cost on each architecture.

The event counts are determined using trace-driven simulation. Just as the performance of cache and virtual memory systems has been effectively evaluated with trace-driven simulation based on address traces, the performance of garbage collection algorithms can be evaluated using trace-driven simulation at a higher level. Object-level tracing has been used to investigate many aspects of garbage collection performance [26, 30]. The events that form the trace used to simulate garbage collection algorithms are: object references (loads and stores), allocations, and deallocations.

MARS (Memory Allocation and Reference Simulator) is the simulator that collected the event counts used in this paper [29]. MARS is attached to a commercial Common Lisp system (Franz Allegro Common Lisp), and large Lisp programs drive the algorithm simulation. In order to simulate the behavior of garbage collection algorithms, the simulator maintains its own view of how objects are organized in memory. The simulator translates references to objects into references to addresses in the "shadow memory." By maintaining a shadow memory, the simulator does not interfere with the execution of the program in any way, except that execution is slowed.

9

This paper uses four large Common Lisp applications for its evaluation. These programs, which represent a variety of programming styles and application areas, are summarized in Table 1.

Resource	RL	Prolog Comp.	PMA	RSIM
General Comments	Microcode compiler for a class of signal processing. architectures. Modern style, many structures.	Prolog compiler for RISC architecture. Modern style, many data types.	Microcode assembler for PERQ machine. Multiple passes.	Circuit simulator. Builds a network of nodes, propagates floating point values.
Source lines	10,200	4,700	5,100	2,700
SPARC time (sec)	407	40.4	89.7	505
MIPS* time (sec)	288	28.6	63.5	358
MC68020 time (sec)	720	91.9	174	1260
Heap references ($\times 10^6$)	110	42	20	38
Objects allocated ($\times 10^6$)	7.8	3.9	1.1	6.9
Bytes allocated ($\times 10^6$)	82	35	15	57

Table 1: General Information about the Test Programs. The times provided are the execution times of the programs (not including garbage collection) on the following computers: SPARC (Solbourne Series 4), MIPS (Decstation 3100), and MC68020 (Sun3/260). The Lisp system measured is Kyoto Common Lisp (KCL) with enhancements provided by Edward Wang of U.C. Berkeley. *Because this version of Lisp was not available on the Decstation 3100, the MIPS execution times are derived from the SPARC times using the Dhrystone benchmarks to determine the relative processor speed.

The other performance measure needed for this evaluation is the CPU cost of handling each of the events that were counted using MARS. The paper estimates the barrier costs on three different architectures: the SPARC, MIPS R2000, and MC68020. The instruction sequences required for each event were programmed for each architecture and then executed repeatedly (10^8 times) to provide an accurate fine-grained measure of the execution cost of each of the sequences. To insure that the resulting times were reproducible, measurements were repeated. Given an accurate measurement of the cost per event and exact count of the number of events, the cost of maintaining the read and write barriers for each application can be estimated. To determine the barrier overhead, the total execution time of the applications on each architecture was also measured.

Before considering the individual implementations of the read and write barrier, the general memory reference characteristics of the application programs are presented. Table 2 breaks down the heap references made by each application into loads and stores, and further

Lisp Application	Fraction of References (%)				
	loadp	load	storep	store	storei
RL	71.5	0.4	9.0	0.1	18.9
Prolog Compiler	68.2	0.3	10.4	0.04	21.0
PMA	72.0	1.1	7.4	1.2	18.3
RSIM	56.3	0.00	5.7	0.00	38.0

Table 2: Object References by Instruction Type for the Test Programs. Loadp represents a pointer load (that is, the load of a pointer), load is a non-pointer load (the load of a non-pointer), storep is a non-initializing pointer store, store is a non-initializing non-pointer store, and storei is an initializing store (both pointers and non-pointers).

subdivides these categories into pointer and non-pointer operations. Also, stores are divided into initializing stores (which are always written to newspace and therefore never need to be checked for a generation trap) and non-initializing stores, which require generation checks. The table indicates the reason why the read and write barriers may require different handling. The read barrier potentially adds overhead to every pointer load, which account for 56–72% of all memory references in the applications. Non-initializing pointer stores, however, only account for 5–10% of memory references, and adding a small overhead per reference may only have a minor impact on performance.

3 The Write Barrier

This section will first consider the implementation of three different approaches to implementing the write barrier: special hardware, software tests, and operating system protection faults. After describing the implementations, the costs of the different methods on different machine architectures will be compared.

Write barrier overhead comes from two sources: checking to see if a pointer store creates an intergeneration pointer (the generation check) and recording the location of the intergeneration pointer in the remembered set when one is created (the generation trap). Because the actual creation of intergeneration pointers is quite infrequent, the largest source of overhead from the write barrier results from the generation test.

Before discussing specific implementations of the generation test and trap, I will first describe the organization of generations that I assume and indicate how a pointer's generation is determined. The generation organization simulated with MARS has four generations. Newspace, the youngest generation, is located at physically higher addresses than other generations (oldspace). With this organization, a pointer's generation can be determined by testing the pointer's address against the addresses representing the boundaries of the generations. Since newspace occupies the highest addresses, a pointer can be determined to point into newspace or oldspace with one comparison. Determining which generation in oldspace a pointer actually points into requires further tests.

After an intergeneration pointer has been discovered, its location must be recorded in the remembered set. Simple representations of the remembered set maintain a sequence of addresses, where each generation trap adds the address of the intergeneration pointer to the sequence [3, 27, 31]. One problem with a sequence representation is that the same address can occur in the sequence many times. Such redundancy wastes space and also time when the sequence is scanned during garbage collection. The advantages of a sequence representation are simplicity of implementation and speed of trap handling. Sobalvarro describes an organization, called word marking, where a bitmap is used to indicate the memory locations of intergeneration pointers [22]. In this case, the generation trap handler simply sets a bit in the bitmap. Word marking avoids the redundancy of a sequence representation at the memory cost of the bitmap. Another alternative, which Sobalvarro calls card marking, uses a bitmap to indicate if an intergeneration pointer is stored in a region of memory (the card). The Symbolics uses such a method where the card is a hardware page [17]. Card marking

12

is less desirable than word marking because at collection time each marked card must be scanned to locate the intergeneration references.

Sobalvarro describes a 10 instruction sequence for the MC68020 architecture that implements write barrier word marking. The same sequence in the SPARC and MIPS architectures requires 19 and 22 instructions, respectively, due to the simpler instructions and lack of indirect addressing modes. Because word marking avoids the redundancy of the sequence representation and because its handler is sufficiently fast, I assume word marking is used to record intergeneration pointers throughout this section. Because all implementations of the write barrier will use the same representation of the remembered set, the rest of this section will focus on different methods of implementing generation checks efficiently.

3.1 The Hardware Write Barrier

With a hardware write barrier, the generation test is performed in parallel with the store. If a generation trap occurs, a trap handler records the intergeneration pointer in the remembered set and continues. Several architectures provide hardware write barriers, including SPUR [13] and the Symbolics [17]. Because the generation check is performed in parallel with the store, the total overhead using this method occurs during the handling of generation traps. While the number of generation traps depends on the promotion policy and allocation threshold, the fraction of instructions causing generation traps is typically small (Zorn reports typically $< 0.01\%$ of all instructions [29]).

Using a remembered set implemented with word-marking, the number of instructions to mark a word is small, 22 at most as indicated above. The only additional overhead is the instruction sequence to handle the machine trap and return from it. The cost of fielding a trap differs with architectures, but without careful design, can amount to 100–200 instructions, as Johnson points out [15]. If we assume a well-designed trap architecture allowing fast traps (as we should if there is hardware support for the write barrier), the cost of fielding a trap is also small, as few as 10 instructions as suggested by Johnson. With approximately 30 instructions ($10 + < 22$) to implement a generation trap and a frequency

13

of 0.01%, the total overhead of a hardware supported write barrier can be estimated at approximately 0.3%, a negligible amount. I conclude that hardware support allows for a very low overhead write barrier. Furthermore, hardware support for maintaining the remembered set (for example, as provided by the Symbolics, which supports page marking in hardware) is unnecessary because the cost of maintaining the remembered set information using software is already small.

3.2 The Software Write Barrier

The most common approach to implementing the write barrier on stock hardware, used in commercial Lisp systems [12], is to implement the generation test inline with every non-initializing pointer store (NIP store). Because these stores are quite frequent (5–10% of all references), this approach requires careful design in order to minimize the cost per store. Furthermore, because inline tests expand the code around every NIP store, reducing the resulting code expansion is another important consideration. With four generations, a code sequence that identifies generation traps by determining the generation of the pointer being stored and the object being stored into (the container) requires about 30 SPARC instructions, far too many to include with every NIP store.

A more efficient approach is to perform a test for the most likely cases inline (e.g., test to see if the container is in newspace), and check the more unlikely cases in a subroutine call. To determine what cases are likely and unlikely, some empirical data is required. Table 3 shows the relative frequency of NIP store operations, where the generation of the pointer and the container was noted in a particular application (the RL compiler). From the table, we see that 67% of NIP stores are to containers in newspace. A single compare will determine if the container is in newspace and if the test succeeds, then the store can proceed immediately, since stores into newspace cannot create intergeneration pointers. Of the remaining 33%, which are stores to oldspace, 15% are stores of integer immediate values, which also cannot cause a generation trap. If the inline test checks that either the container

Fraction of Pointer Stores (%)				
Pointer	Container Generation			
Generation	gen0	gen1	gen2	gen3
gen0 (youngest)	21.3	2.2	0.0	0.4
gen1	0.2	0.1	0.0	0.7
gen2	0.0	0.0	0.0	0.0
gen3 (oldest)	27.2	3.5	0.0	8.2
fixnum (immediates)	17.9	2.7	0.0	15.6
total	66.6	8.5	0.0	24.9

Table 3: Breakdown of non-initializing pointer stores by generation of pointer stored and object stored into (container). The fixnum (immediate) row represents the storage of integer immediates into containers. Because immediates are not allocated in memory, they do not reside in a generation, and storing them should never cause a generation trap. Note that in this example, no data was promoted to generation two, and hence it is empty. Data in generation three represents very long-lived system data present from the start of program execution.

is in newspace or the pointer is actually an integer constant, over 80% of the NIP stores will only require a few additional instructions of overhead.

The write barrier sequence for the SPARC architecture is presented in Figure 3. This

```
        ! Perform the store first
        st      %object, [%container + %offset]

        ! Check for storing a fixnum, and continue if it is
        andcc   %object, 3, %g0
        bzero   done

        ! Compare container and newspace boundary, and continue if above
        cmp     %container, %newspace_boundary
        bgu     done

        ! Otherwise call writeBarrierTest to check other cases
        mov     %object, %o0            ! %o0 is first arg
        add     %container, %offset, %o1   ! %o1 is second arg
        call    _writeBarrierTest
done:
```

Figure 3: SPARC Instruction Sequence for Software Write Barrier Test. Some instruction sequences have been rearranged to avoid confusion associated with delayed control flow.

sequence illustrates the two tests that are performed inline and the call to the function `writeBarrierTest`, which checks for all the other possible generation traps. The pointer is assumed to use the two lower bits of the address as a type tag (low-tagging) as has been used in several Lisp systems [12, 28].

To correctly estimate the overhead of this implementation of the write barrier, a CPU cost must be determined for each possible value for the generation of the pointer and the container. The cost for each configuration multiplied by the number of times each configuration occurs gives the total cost of this implementation of the write barrier. Table 4 gives the CPU times for each of the different generation configurations measured on the

CPU Time Per Generation Check (microseconds)				
Pointer Generation	Container Generation			
	gen0	gen1	gen2	gen3
gen0 (youngest)	0.385	5.20	5.20	5.20
gen1	0.382	4.46	6.06	5.39
gen2	0.385	4.04	4.04	5.39
gen3 (oldest)	0.386	3.25	3.25	3.25
fixnum (immediates)	0.130	0.130	0.130	0.130

Table 4: Breakdown of CPU costs for different configurations of pointer and container generation as measured on the SPARC architecture.

SPARC architecture. Note that the fastest times occurred when a fixnum is stored into a container. The next fastest occurs when a store is made to a container in newspace. The other configurations, which require a function call, often take more than ten times as long as the inline tests. Similar data was gathered for the MIPS R2000 and MC68020 architectures.

The code sequence associated with an inline write barrier test replaces a one instruction store with eight instructions. Measurements of several large SPUR Lisp programs [24] indicate that the static frequency of store instructions is typically less than 4%. By expanding each of these instructions into eight instructions, the overall code expansion should be on the order of 20–30%. Some researchers consider this code expansion to be potentially very costly and suggest *unconditional* generation traps that record information about every NIP

16

store [3]. The actual impact of this code expansion depends on the size and configuration of the machine memory hierarchy, and in particular the instruction cache. The effect of this expansion on performance of Lisp programs has been considered by Steenkiste, who shows that a code expansion of 30% has negligible impact on performance ($< 2\%$) if instruction caches are 4 kilobytes or larger [23].

3.3 The Write Barrier Using Write Protection Faults

Another approach to implementing the write barrier differs significantly from the previous two. Instead of testing stores as they occur, this technique uses operating system page protection mechanisms to identify stores into oldspace. A simplistic version of this method would write protect pages in oldspace and catch each protection fault when a location in oldspace is written. The fault handler could identify generations of the pointer and container associated with the store and record intergeneration pointers as they are created. This approach has two advantages: overhead is only added to the pointer stores into oldspace, instead of all pointer stores, and no code expansion is required. The severe disadvantage of the simplistic approach is the high cost of the operating system handling a protection violation (typically several thousand instructions). Because, in the past, protection faults have been associated with program errors or security violations, operating system designers have not attempted to implement these faults with great efficiency. Faulting on <u>every</u> store into oldspace is clearly too expensive.

A more clever implementation of this approach only faults once per oldspace page that is stored into. After the first faulting store, the page is made writable, and subsequent stores do not fault. If each page in oldspace that has been written between garbage collections is recorded, those pages can be scanned for intergeneration pointers when a collection cycle starts. This approach was first described by Shaw, who suggests that small modifications to the virtual memory interface of traditional operating systems (essentially providing user-access to "dirty-bits") would allow this information to be maintained very cheaply [20].

Without operating system changes, the collection algorithm must maintain its own "dirty-bit" information about oldspace pages.

My version of this method behaves as follows. If the operating system allows the user to make specific pages in his address space unwritable, then pages in oldspace can be write-protected after each garbage collection. When program execution continues after the collection, pages in oldspace are occasionally written. Each write causes a protection violation that is handled by a Lisp-defined fault handler. The fault handler records the oldspace page that was written (effectively maintaining a "dirty bit" for the page) and makes the page writable. When the next collection is initiated, the dirty oldspace pages are scanned for newspace pointers. The locations of the newspace pointers that are found are added to the remembered set.

The cost of this approach is directly related to the number of pages in oldspace marked as dirty between collections. For each such page an operating system fault occurs and later the page must be scanned for intergeneration pointers. Unlike other software methods, these two operations are quite expensive. Table 5 shows the costs of a write protection

CPU Time per Operation (microseconds)			
	Architecture		
Operation	SPARC	MIPS	68020
Write protection fault	1805	304	1683
Scan 4096-byte page	1239	569	1279

Table 5: Cost of handling a write protection fault and scanning a 4096-byte page on three architectures. The SPARC machine was a Solbourne Series 4 processor running version OS/MP 4.0C of the operating system. The MIPS machine was a Decstation 3100 running Ultrix Worksystem V2.2. The MC68020 machine was a Sun3 machine running SunOS Release 4.0.

fault (including making the page writable) and the cost of scanning a 4096-byte page on the three architectures of interest. Unlike the cost of a generation test or trap, these overheads are very large, almost all more than a millisecond. The effectiveness of the protection fault approach depends heavily on the number of dirty pages in oldspace, and this in turn depends on other system parameters as we shall see in the next section.

18

3.4   Performance Evaluation

In this section, we consider the CPU overheads of these implementations for each of the four test applications, executing on the SPARC architecture (Figure 4). In the figure, overhead is plotted as a function of allocation threshold, a parameter that is easy for users to vary and one that has a strong influence on the performance of garbage collection methods. As the allocation threshold increases, more data is allocated between collections and collections occur less frequently. By delaying collection in this way, more objects have an opportunity to become garbage between collections, and a smaller fraction of live data is copied during the collection cycle, increasing the efficiency of collection. Furthermore, if collections occur less frequently, object promotion to older generations occurs less frequently if promotion is based on surviving a fixed number of collections.[1] Decreased promotion results in fewer objects being located in oldspace, and hence fewer stores into oldspace. The overall effect is that the write barrier methods that are heavily affected by stores into oldspace are significantly more efficient with larger allocation thresholds. This result is clearly evident in the figure, where operating system methods of implementing the write barrier improve dramatically with larger thresholds. One disadvantage of larger allocation thresholds is that more data is copied during each collection cycle (since more has been allocated). When the allocation threshold becomes large enough, typically above 500 kilobytes, the pauses associated with garbage collection become noticeable to an interactive user, and are therefore much more disruptive.

Each plot in the figure shows the overhead of the three described methods of implementing the write barrier, and a fourth line representing the overhead if the operating system fault handler mechanism was speeded up by a factor of ten (not an unreasonable assumption considering the times measured were on the order of one millisecond, or approximately 10,000 instructions). The improved OS trap line is intended to give an idea of what a well-designed operating system might provide.

---

[1]In all cases shown, objects were promoted to the next generation after surviving three collections.
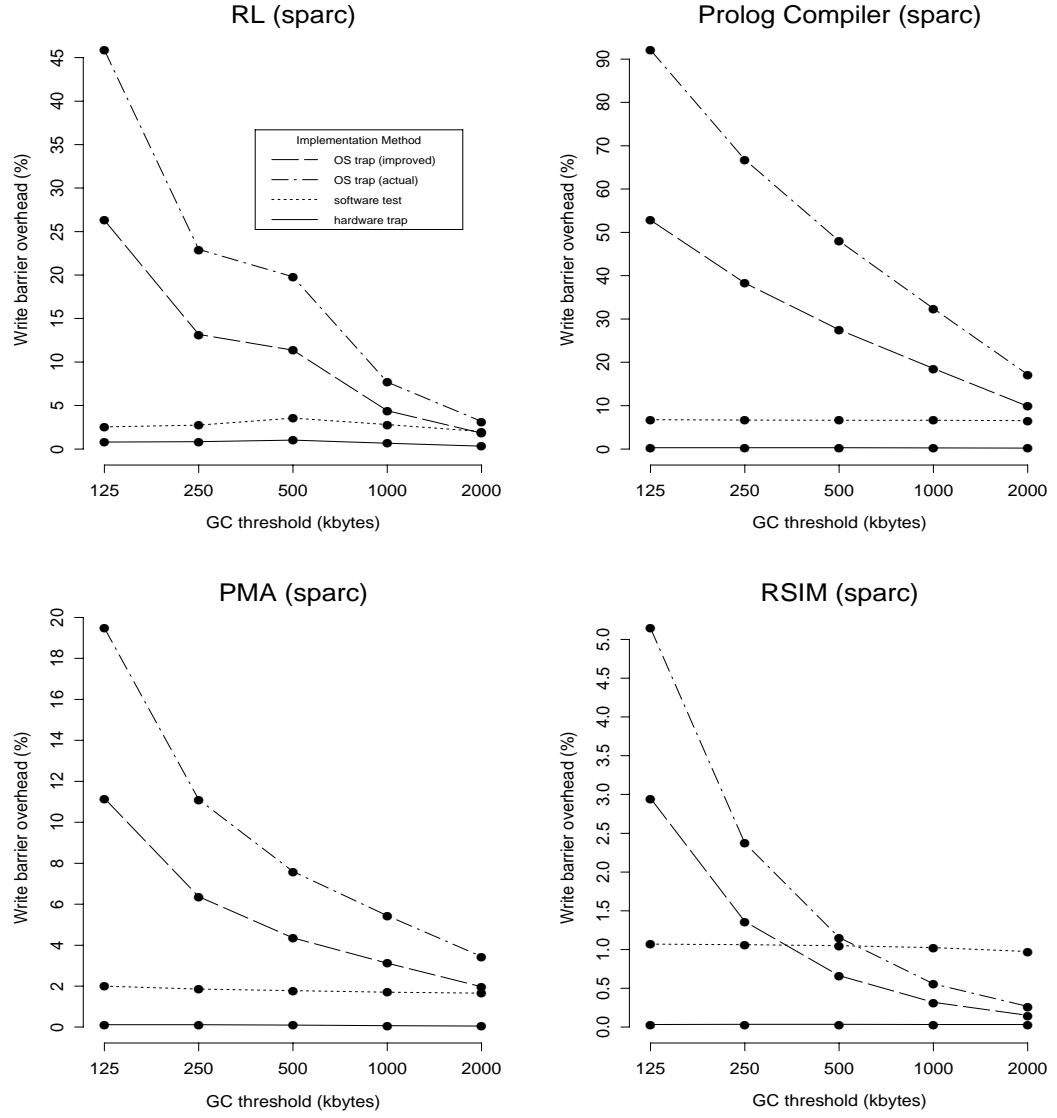
Figure 4: CPU Overhead for Write Barrier Implementations.

20

The figure indicates that an implementation based on inline software tests shows uniformly good performance without requiring either hardware support or a well-designed operating system. The implementation based on protection faults shows reasonable performance at large threshold sizes, but the pause length associated with very large thresholds makes this configuration less desirable. Using the operating system page protection mechanism also introduces additional implementation dependencies that reduce portability, making this alternative even less attractive. Even with a relatively fast fault handler, the overhead of the protection fault method is substantial due to the high cost of scanning a 4096-byte page for intergeneration pointers. With operating system support, as suggested by Shaw, the protection fault component of the overhead could be totally eliminated, but the page scanning cost would still remain and result in overheads generally higher than the inline software test approach.
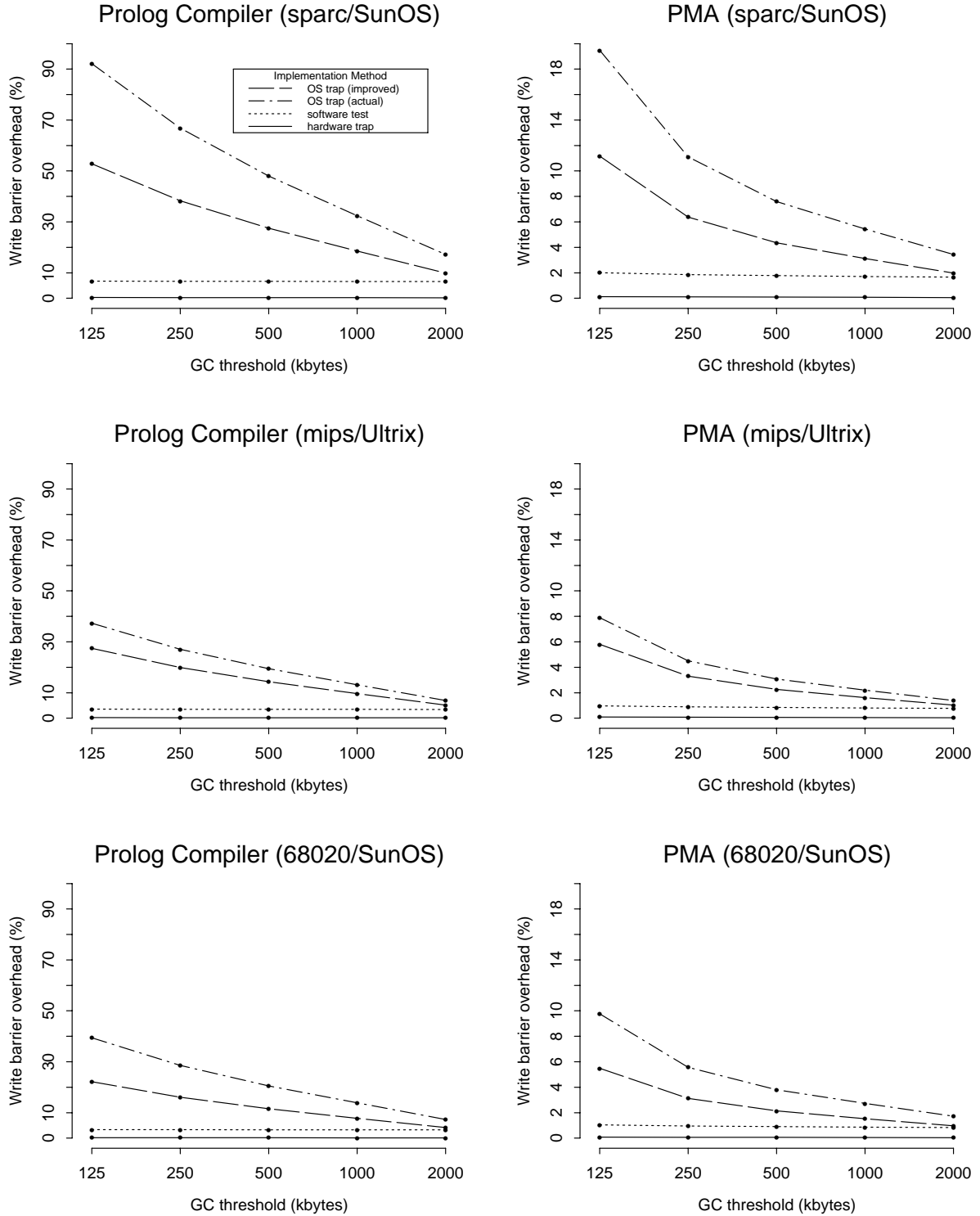
To show that the results presented are not specific to the SPARC architecture, Figure 5 compares the performance of the write barrier methods using two applications and three different architectures. In general, the results are virtually identical, except that the overhead of the methods is lower on the MIPS and 68020 architectures than it is for the SPARC architecture. The MIPS architecture also shows less difference between the actual and improved protection fault method because the write protection fault handler on the Decstation is substantially faster than the handler for the Solbourne SPARC implementation, even when the relative machine speeds are normalized.

In conclusion, short of special hardware, carefully designed inline software tests appear to be the most effective way to implement the write barrier and result in overheads of 2–6% in a variety of applications executing on a number of different architectures.

4 The Read Barrier

This section evaluates implementations of the read barrier much as the write barrier was evaluated in the previous section. While other approaches to incremental collection have

Prolog Compiler (sparc/SunOS)

Write barrier overhead (%)

Implementation Method
OS trap (improved)
OS trap (actual)
software test
hardware trap

GC threshold (kbytes)

PMA (sparc/SunOS)

Write barrier overhead (%)

GC threshold (kbytes)

Prolog Compiler (mips/Ultrix)

Write barrier overhead (%)

GC threshold (kbytes)

PMA (mips/Ultrix)

Write barrier overhead (%)

GC threshold (kbytes)

Prolog Compiler (68020/SunOS)

Write barrier overhead (%)

GC threshold (kbytes)

PMA (68020/SunOS)

Write barrier overhead (%)

GC threshold (kbytes)

Figure 5: CPU Overhead for Write Barrier Implementations on Different Architectures. The overhead of different write barrier approachs for the Prolog compiler and microcode assembler applications are presented.

22

been attempted, most notably work by Brooks [6] and Appel [2], this section will consider implementations of Baker-style incremental collection, where pointers into fromspace are relocated as they are loaded into registers. Three implementations of the read barrier are described: special hardware, inline software, and read protection faults.

Before discussing the implementations, I will outline the memory organization assumed. Incremental collection requires that all pointers loaded into registers are checked first to determine if they point into fromspace. This check is normally performed when the pointer is loaded from memory and the test consists of two comparisons with the base and top of the current fromspace area. Two tests are necessary (unlike the newspace test for the write barrier) because the position of fromspace in memory "flips" back and forth during garbage collection cycles. This organization allows only one fromspace region to be collected at a time, which could be a limitation if incremental and generation collection methods are combined, and multiple generations are incrementally collected simultaneously (possibly requiring the presence of multiple fromspaces).

4.1   The Hardware Read Barrier

With a hardware read barrier, the test for fromspace pointers is performed by the architecture in the cycle after a pointer has been loaded from memory. Because the test must wait for the pointer to be loaded, operations on the loaded pointer must be delayed one cycle. In general, instruction reorganization can often fill this delay slot, and the cost of the hardware read barrier, like the hardware write barrier, is negligible. Such a barrier has been successfully implemented in several generations of Lisp machine architectures [10, 18].

4.2   The Software Read Barrier

As an alternative to the hardware read barrier, the read barrier tests can be performed inline after the pointer is loaded. Unlike the write barrier, where numerous checks must be performed to detect a generation trap, the read barrier requires two simple comparisons. The SPARC instruction sequence to implement the software read barrier is given in Figure 6.

23

The simplicity of this sequence suggests its execution cost will not be large. Furthermore,

```
        ! Load the pointer from memory into register dest
        ld      [%pointer], %dest

        ! Test if below the bottom of fromspace (in a global register)
        cmp     %dest, %fromspace_base
        blu     done

        ! Test if above the top of fromspace (in a global register)
        cmp     %dest, %fromspace_top
        bgeu    done

        ! Copy object from fromspace to tospace
        mov     dest, %o0
        call    _readBarrierMove

        ! New dest is in %o1
        mov     %o1, %dest
done:
```

Figure 6:   SPARC Instruction Sequence for Software Read Barrier Test. Some instruction sequences have been rearranged to avoid confusion associated with delayed control flow. Some nop instruction have been left out for the same reasons.

consider a generation collection algorithm with the organization of generations in Figure 7. Notice that all loads of pointers into oldspace will fail on the first read barrier test (since
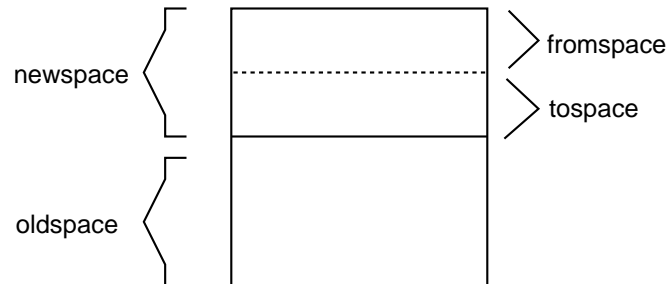
Figure 7: Generation Organization to Facilitate Rapid Read Barrier Testing

oldspace is always below the bottom of fromspace). With this organization, at least half of all read barrier tests will fail in the first test, adding only two instructions to the cost of a

load. This implementation suggests that the read barrier can be implemented using inline tests at a relatively small execution cost.

We can estimate the cost using a rough calculation. The precise costs will be presented later. Memory references to the heap (as opposed to the runtime stack), account for an average of 12% of all instructions over a variety of applications (as measured by Zorn [29]). Of these, approximately 70% are pointer loads (see table above). With inline tests, three or five additional instructions are added to each load (and we will assume the split is 50/50). The result is that approximately 33% (= $0.12 \times 0.70 \times (3 + 5)/2$) more instructions are executed to implement the software read barrier. This estimate is conservative (i.e., too large) because the added instructions are all register (i.e., not memory) instructions, and so will all execute in one cycle (unlike memory operations). Furthermore, more than 50% of the loads will only require one comparison if memory is organized as it is in the previous figure. The result, confirmed by measuring the test applications, is that the read barrier can be implemented using line software tests with less than 20% overhead. While this overhead is still high, the substantial interactive advantages of incremental collection suggest it might be an attractive alternative, even with stock hardware.

One potentially serious drawback of using inline tests to implement the read barrier is that the code is significantly expanded. Measurements of SPUR Lisp programs indicate that pointer load instructions account for 13–15% of the static code. Assuming the inline test adds seven instructions to every load, the associated code expansion would double the size of the code. Doubling the code size will definitely have negative effects on performance, but these effects are very memory system dependent and hard to quantify. With small caches the expansion could cause catastrophic degradation, while with large caches, the degradation could be negligible.

4.3   The Read Barrier Using Protection Faults

While the inline tests appear to provide a reasonable overhead, implementing the read barrier with protection faults can avoid the per pointer load overhead and the code expansion

caused by the inline test implementation. This method removes the invariant that fromspace pointers may not be loaded into registers. Instead of checking for fromspace pointers as they are loaded, the pages of fromspace are read and write protected, and objects are transported only when their contents are accessed and an operating system fault occurs. When the fault occurs, the object in fromspace is relocated to tospace, its pointer is updated, and the program continues. In this way, objects in fromspace are "faulted" into tospace as they are referenced.

This approach is made more complex because fromspace pointers are allowed in registers, hence a copy of a fromspace pointer can be made before the pointer is dereferenced and updated to point to the tospace copy of the object. In this way, pointer *aliases* can be created, where a pointer to the tospace copy and a pointer to the fromspace copy of an object (the fromspace alias) exist simultaneously. These fromspace aliases can cause problems if they are not detected. In particular, when aliases of the same object are compared (with the Lisp "EQ" operation), they must be considered to be equal. Also, fromspace aliases must be prevented from being stored in memory, where they would be hard to locate and update.

If fromspace aliases are allowed, the EQ operation must be modified as follows. First, the pointers are checked for being identical. If they are identical, no further action is needed because no aliases were involved. If the test fails then there are two possibilities. First, if one argument is known to be nil or a fixnum immediate (as generated by the compiler), then the test fails and no aliases were involved because nil and immediates cannot have aliases.[2] The only case where aliases need be be checked occurs when both arguments are not recognized by the compiler and the EQ test fails. In this case, the pointers must be checked for being aliases to each other. Table 6 indicates the relative frequency of EQ tests and heap references and the frequency of outcomes of EQ tests for several large test programs.

---

[2]Nil is stored in the oldest generation and not incrementally transported.

26

| Operation	RSIM	Weaver	SLC	PMA	Average
Eq true, total (%)	52.1	36.6	27.3	36.2	38.1
Eq false, total (%)	47.9	63.4	72.7	63.8	61.9
Eq false, with nil (%)	24.6	27.4	41.4	31.0	31.1
Eq false, with immediate (%)	8.6	25.1	0.8	7.8	10.6
Eq false, 2 unknown args (%)	14.8	10.8	30.6	25.0	20.3
Eq tests / heap access (%)	11.4	42.0	36.9	19.7	27.5

Table 6: Relative Frequency of Outcomes of EQ Tests. RSIM and PMA have been described. Weaver is a layout program implemented in an OPS5 system built on top of Common Lisp and SLC is the SPUR Lisp compiler.

The table indicates that while a majority of EQ tests fail, only a relatively small fraction of the tests (10–30%) would require tests for object aliases. In addition to a modified EQ test, fromspace aliases require that all pointer stores (including stores to initialize) are modified to include a test for a pointer into fromspace (just as the loads are checked in the previous section). The net result is overhead from a number of sources, but the largest source of overhead is the expense of the operating system protection fault.

As we shall see in the next section, current operating system overheads make implementing the read barrier with protection faults prohibitive. Another implementation variation, suggested by Doug Johnson and not explored here, would also be promising if faults were more efficient. Instead of allowing copies to be made of potential fromspace pointers, an implementation could choose to always dereference a pointer as soon as it was loaded into memory. If the pointer pointed into fromspace, the resulting trap would relocate the object immediately. This approach would prevent fromspace aliases from occurring at the cost of an additional load per pointer load. Unfortunately, the next section shows that any approach that uses operating system protection faults is unlikely to be sufficiently efficient.

4.4 Performance Evaluation

Figure 8 compares the performance of these three implementations for the test programs executing on a MIPS-based computer. In this case, the MIPS machine was used because
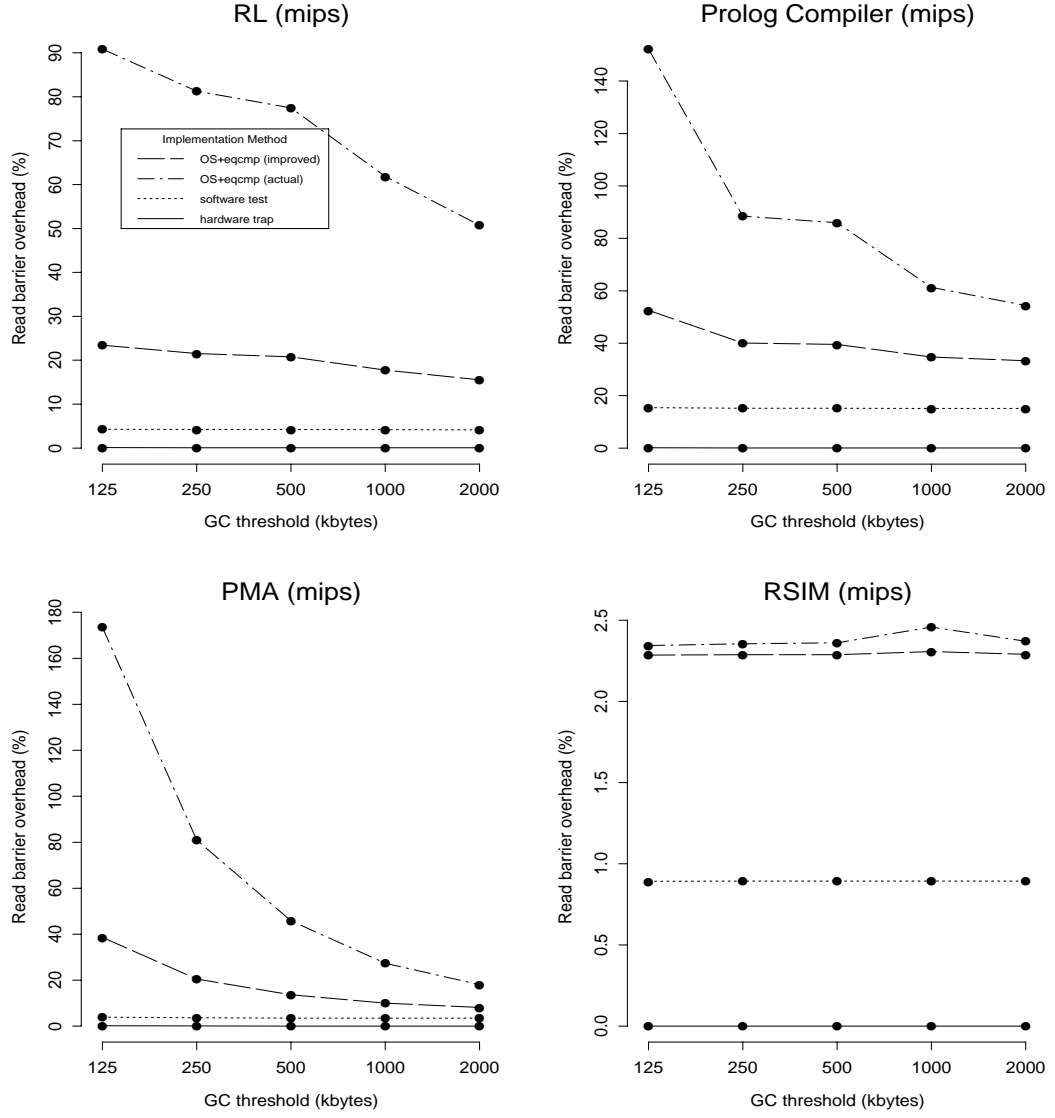
Figure 8: CPU Overhead for Read Barrier Implementations. The results indicate the overhead of the read barrier for an incremental copying algorithm.

the overhead associated with the SPARC architecture's protection faults makes the SPARC figure less informative. Again, the cost of the protection fault approach is considered for the actual operating system execution costs and for a hypothetical improved operating system with fault handlers that are ten times as fast.

The results suggest that implementing Baker-style incremental collection using operating system protection faults is clearly a bad idea because of the high cost of handling a fault (approximately one millisecond). Even with much faster fault handlers, the overhead using this approach is considerably larger than the overhead of inline software tests. On the other hand, the CPU overhead of inline tests appears to be reasonably small (on the order of 20%). Again, the overhead of the hardware implementation is negligible.

Figure 9 shows the relative costs of the different read barrier implementations on three different machine architectures. From this figure, we see that the SPARC architecture had by far the highest overhead of the three architectures. In particular, the MIPS system, with its significantly more efficient fault handlers, shows performance for the fault protection implementation that approaches the cost of the inline test approach (with large thresholds). The overhead of the inline test method is quite small on the MIPS machine, making this implementation even more attractive.

In summary, the overhead of implementing the read barrier is substantially higher than that of the write barrier, but the cost of an inline test implementation is not unreasonable (from 10–20%), especially on the MIPS architecture. A serious drawback of the inline approach is that it will probably double the code size of applications, which may or may not dramatically affect memory system performance. Implementing the read barrier with protection faults is very costly, due to the unnecessarily high costs of operating system protection fault handlers.
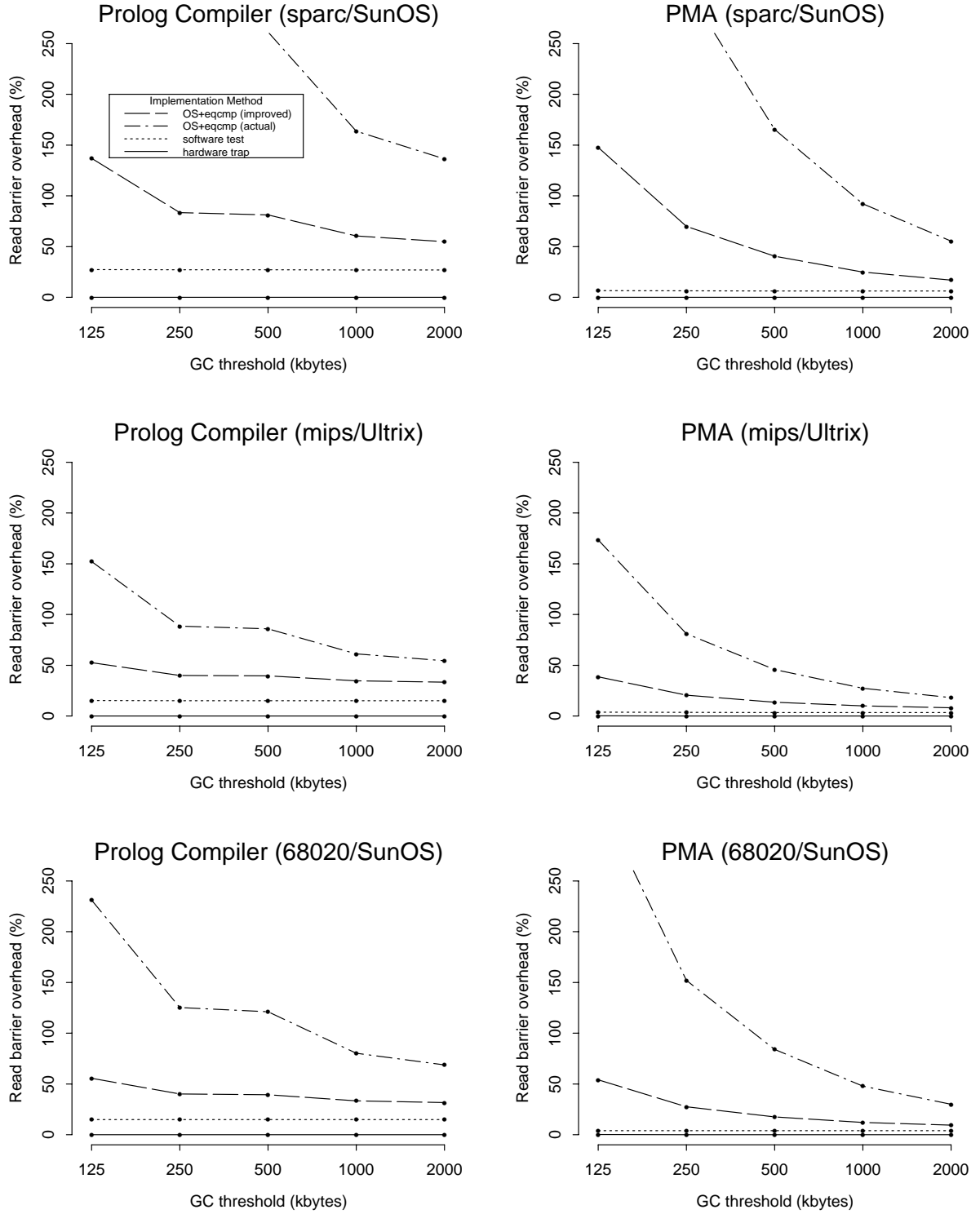
Figure 9: CPU Overhead for Read Barrier Implementations on Different Architectures. The overhead of different read barrier approachs for the Prolog compiler and microcode assembler applications are presented.

5 Related Work

Other researchers have suggested implementations of the read and write barriers, but none have attempted to compare the costs of different approaches.

Appel describes an implementation of the Standard ML generation collector where every NIP store to the heap adds an item to the remembered set (implemented as a list) [3]. At collection time, the list is scanned for intergeneration pointers. The two advantages of an unconditional store are reduced code expansion (because it avoids a conditional test) and global registers are not required to hold the generation boundaries. This approach has the disadvantage of significantly increasing the number of stores performed by the program (producing more dirty cache lines and pages) and substantially increasing the size of the remembered set. These disadvantages are relatively minor in ML, where stores are discouraged (because ML is a mostly-functional language), but would be significant in a Lisp system, where stores are more common.

Johnson describes hardware modifications that increase the speed of user trap handlers [15]. In the paper, he estimates the cost an inline software write barrier as 7–10% based on Steenkiste's measurement of stores to the heap. Moon estimates that implementing the Symbolics write barrier with software would result in a 10–20% overhead [17]. My results show that hardware support for the write barrier trap can almost eliminate overhead, and that hardware support for the remembered set (such as is provided by the Symbolics) is unnecessary. Furthermore, my estimates of the the cost of a software write barrier indicate Moon's estimates of 10–20% are too high.

Shaw compares software and "dirty-bit" implementations of the write barrier [21]. Shaw estimates that inline software implementation would add 7–14% overhead. As an alternative, he suggests that the operating system virtual memory interface should be extended to let the user know about dirty pages. He concludes that for thresholds larger than 160 kilobytes, the dirty-bit implementation will result in lower overhead than inline tests. My results, which avoid modifications to the operating system, indicate that a dirty-bit im-

plementation will result in high overheads due to the high cost of the operating system fault. Even with very fast faults, however, my results show that the page scanning overhead makes a dirty bit implementation of the write barrier more expensive than the inline software implementation for all the observed threshold sizes (125–2000 kilobytes).

Several other approaches have been taken toward providing incremental garbage collection on stock hardware. Brooks proposes a technique that adds an extra pointer per object and an extra indirection per object reference in order to allow incremental garbage collection on stock hardware [6]. While this technique reduces the code expansion of inline tests, the data expansion associated with adding an extra pointer per cons cell (33% increase) is probably more significant. Furthermore Brooks does not attempt to quantify the overhead of his technique or compare its efficiency with other techniques.

Appel, Ellis, and Li describe an approach to incremental collection that protects pages in tospace and relocates the pointers on protected pages as they are referenced [2]. This method avoids the per-load costs of maintaining the read barrier in software, but also results in longer pauses than Baker's method. This method also prevents the use of the read barrier to increase data locality, as described by Courts [9]. Nevertheless, Appel's method provides an alternative to the costly Baker collection algorithm.

Johnson discusses the value of the read barrier in improving the locality of reference in a very large Lisp program [14]. Citing Zorn [29] and Johnson [15], he concludes that the read barrier is too expensive to implement in software.

In work that predates the present evaluation, Zorn uses the same evaluation techniques to arrive at larger costs for the inline implementations and smaller costs for the fault protection implementations of the barriers [29]. Zorn's previous results are pessimistic in predicting the inline costs because he used instruction counts to determine CPU costs. The actual execution time of the sequences, measured here, show that instruction counts are not accurate predictors of execution time due to the non-uniform execution time of different instructions and different combinations of instructions.

Similarly, Zorn's originally estimates of the cost of protection fault handlers were optimistic because actual operating systems do not have particularly efficient fault handlers. Where Zorn's estimates ranged from 200–1,000 instructions per fault, actual measurements suggest the actual number is closer to 10,000 instructions per fault.

6   Summary

This paper has evaluated the implementation of several different algorithms for garbage collection read and write barriers. Using trace-driven simulation in combination with instruction sequence measurements, the performance of hardware, software, and protection fault implementations was compared. In all cases, implementing the read and write barriers with hardware results in negligible performance degradation, as long as hardware traps are handled sufficiently fast. While several architectures have provided support for these barriers, including the Symbolics [18], Explorer [25], and SPUR [13], most new general-purpose computer architectures do not.

The most significant result of this paper is that carefully implemented inline software tests can result in relatively simple, low overhead implementations of the read and write barriers. I have shown that the write barrier can be implemented with 2–6% CPU overhead and a code expansion that is typically less than 30%. The read barrier requires more overhead, but typically less than 20%. This overhead, while substantial, is not prohibitive considering that the other costs of garbage collection often add 20% overhead. The code expansion associated with an inline read barriers, on the order of 100%, is potentially problematic. Because the performance effect of this expansion is very memory system dependent, implementors need to keep the expansion in mind when considering using the inline read barrier on a particular machine.

A second important result of this paper is the most operating systems do not provide sufficient support for fast memory protection faults. Using protection faults to implement the barriers is an attractive alternative to inline tests, because it avoids the high cost

33

associated with adding overhead to every memory reference to the heap and the potentially serious code expansions. Two approaches to garbage collection based on protection faults have proven to be successful [2, 20], but both required changes to the operating system kernel. Unfortunately, most operating systems are not designed to provide fast memory protection faults because they assume that these faults occur only in error situations. As we have seen in this paper, language runtime systems can use the operating system memory protection mechanisms to implement sophisticated garbage collection algorithms.

Because operating systems may now be used by language runtime systems in ways that were not envisioned at the time the operating systems were designed, a rethinking of operating systems design appears necessary. In particular, an operating system should provide efficient use of its mechanisms by program runtime systems, and more interaction between the runtime system and the operating system should be possible. Even now, some operating systems provide more efficient support than others. Consider the cost of a write protection fault on the three architectures used in this paper (Table 7). From the table

| | Architecture | | |
Operation	SPARC	MIPS	68020
Relative Dhrystone speed (SPARC = 1, larger is faster)	1.00	1.37	0.30
Write protection fault (microseconds)	1805	304	1683
Normalized protection fault (SPARC microseconds)	1805	434	510
Relative fault handler speed (SPARC = 1, larger is faster)	1.0	4.1	3.5

Table 7: Relative cost of handling a write protection fault on the three architectures. The SPARC machine was a Solbourne Series 4 processor running version OS/MP 4.0C of the operating system. The MIPS machine was a Decstation 3100 running Ultrix Worksystem V2.2. The MC68020 machine was a Sun3/160 machine running SunOS Release 4.0.

we see that the Decstation Ultrix and MC68020 SunOS 4.0 operating system provided trap handlers that were 3-4 times faster than the Solbourne handlers. This discrepancy may be due to sloppy design for the SPARC handler, or it may be a result of the SPARC register-window architecture, which can add overhead on context switches because the on-chip register windows must be written to memory. From this example, we see that the cost

of handling a write protection fault varies dramatically from operating system to operating system. What language runtime systems need is well-designed, efficient fault handlers available from every operating system and architecture. Without fast fault handlers, as we have seen in this paper, garbage collection techniques based on help from the operating system will be far too inefficient.

Interestingly, just as this paper questions the suitability of current operating systems designs for language runtime systems, very recently, Anderson et al have noted that the designs of modern operating systems and microprocessor architectures are mismatched [1]. Among other things, Anderson notes that recent architectures tend to provide less support for fast memory protection faults. This observation suggests that design for optimal execution requires the cooperation of computer architects, operating systems, and language designers.

Building on the results in this paper, it will be important in future research to quantify the effects of code expansion associated with inline barrier tests and to provide cost measurements of actual implementations. Nevertheless, these results should encourage future garbage collection implementors to provide incremental collection and implement both read and write barriers using inline tests.

References

[1] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. Technical Report Technical Report 90-08-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, August 1990.

[2] Andrew Appel, John Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, GA, June 1988. SIGPLAN, ACM Press.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, February 1989.

[4] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, pages 807–820, September 1988.

[6] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 256–262, Austin, TX, August 1984.

[7] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 report. Technical Report Research Report 31, Digital Equipment Corporation System Research Center, Palo Alto, CA, August 1988.

[8] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

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

[10] Bruce Edwards, Greg Efland, and Neil Weste. The symbolics I-machine architecture: A symbolic processor architecture for VLSI implementation. In *ICCD*, 1987.

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

[12] Franz Incorporated. *Allegro Common Lisp User Guide*, Release 3.0 (beta) edition, April 1988.

[13] Mark Hill, Susan Eggers, James Larus, George Taylor, et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1986.

[14] Douglas Johnson. The case for a read barrier. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 279–287, Santa Clara, CA, April 1990.

[15] Douglas Johnson. Trap architectures for Lisp systems. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 79–86, Nice, France, June 1990.

[16] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[17] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[18] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[19] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

[20] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, March 1987.

[21] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, Stanford, CA, February 1988. Also appears as Computer Systems Laboratory tech report CSL-TR-88-351.

[22] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general purpose computers. Bachelor's thesis, MIT, 1988.

[23] Peter Steenkiste. The impact of code density on instruction cache performance. In *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture*, pages 252–259, Jerusalem, Israel, May 1989.

[24] George Taylor. Static instruction counts of SPUR Lisp programs. Personal communication. Actual data will be published in forthcoming PhD thesis.

[25] Texas Instruments, Incorporated, Austin, TX. *Explorer Programming Concepts and Tools*, June 1985.

[26] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88 Conference Proceedings*, pages 1–17. ACM, September 1988.

[27] David M. Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, University of California at Berkeley, Berkeley, CA, March 1986. Also appears as tech report UCB/CSD 86/287.

[28] Steve Vegdahl and Uwe F. Pleban. The runtime environment of Screme, a Scheme implementation for the 88000. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 172–182, Boston, MA, April 1989. ACM.

[29] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.

[30] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990.

[31] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, October 1987.