Measuring the Cost of Storage Management

David Tarditi Amer Diwan [1]

May 3, 1995

CMU-CS-94-201

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for publication. This paper is also published as Fox Memorandum CMU-CS-FOX-94-08.

Abstract

We study the cost of storage management for garbage-collected programs compiled with the Standard ML of New Jersey compiler. We show that the cost of storage management is not the same as the time spent garbage collecting. For many of the programs, the time spent garbage collecting is less than the time spent doing other storage-management tasks.

[1] Authors' addresses: David Tarditi, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213-3891. e-mail: dtarditi@cs.cmu.edu. Amer Diwan, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. e-mail: diwan@cs.umass.edu.

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 8313, and monitored by ESD/AVS under contract F19628-91-C-0168. David Tarditi is also supported by an AT&T PhD Scholarship. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the United States Government, or AT&T.

Keywords: dynamic storage management, performance of systems, measurement techniques, applicative (functional) programming, copying garbage collection, Standard ML, simulation

1 Introduction

We study the cost of storage management for garbage-collected programs compiled with the Standard ML of New Jersey compiler [3]. There are two motivations for conducting this study. First, we want to better understand the cost of storage management. Since costs due to storage management occur throughout the entire execution of a program, it is not adequate to measure only the time spent garbage collecting. Moreover, since memory-system performance has an increasing effect on program performance, it is important to understand the memory-system cost of storage management. Second, we want to identify bottlenecks in the storage-management strategy of the SML/NJ compiler and suggest potential improvements.

We measure the cost of storage management for eight programs on a DECstation 5000/200 [12]. The measurements include most of the instruction-level and memory-system costs of storage management. We measure instructions spent garbage collecting, allocating, checking if garbage collection is necessary, tagging, implementing a write barrier, and making code relocatable so that it can be placed in the heap and garbage-collected. In addition, we measure the memory-system cost incurred during garbage collection and the cost incurred during the rest of program execution. We also measure the effect of garbage collection displacing instructions and data used during the rest of program execution from the cache. We estimate upper bounds on the memory system cost due to the disruption of spatial locality by storage management, header words occupying space in the data cache, and instruction-cache misses from storage-management instructions.

The measurements show that the time spent doing storage-management tasks other than garbage collection is greater than the time spent garbage collecting. Thus, measuring a storage-management scheme using only the time spent garbage collecting is misleading because it may ignore most of the cost. The measurements also identify bottlenecks in the storage-management strategy of the SML/NJ compiler and indicate that SML/NJ programs spend 19% to 46% of their execution time doing storage management.

We made the measurements using trace-driven simulation. This allowed us to count the instructions spent performing various tasks, such as tagging integers and implementing the write barrier. The memory simulator modeled the entire memory system of the DECStation 5000/200 [12], which is favorable to programs which heap allocate intensively. A less favorable memory-system organization would increase the cost of storage management by increasing the cost of allocation [14, 15].

The remainder of the paper is organized as follows. Section 2 introduces terminology and describes the storage-management strategy used by the SML/NJ compiler. Section 3 describes the measurement techniques and benchmark programs. Section 4 presents measurements for eight SML/NJ programs. Section 5 reviews related work. Section 6 concludes.

2 Background

The following sections introduce terminology and describe SML, the SML/NJ system, and the storage-management technique used by the SML/NJ system.

2.1 Terminology

Storage management refers to the management of memory by an individual program. In a garbage-collected program, the part of the program that is not the garbage collector is called the *mutator*. Execution of the mutator is called *mutation*. Storage management has two components in garbage-collected programs. The first component, which is obvious, is the execution of the garbage collector. The second component comprises tasks done outside the garbage collector to support storage management. The cost of these tasks is called *the storage-management cost during mutation*.

The number of instructions executed to perform a task is the *instruction-level cost* of that task. The time spent by the processor waiting for memory while performing a task is the *memory-system cost* of that task.

2.2 Standard ML and the SML/NJ system

Standard ML (SML) [23] is a call-by-value, lexically scoped language with higher-order functions, garbage collection, polymorphic static typing, provable safety properties, a sophisticated module system, and a dynamically-scoped exception mechanism.

The SML/NJ compiler [3] is a state-of-the-art compiler for SML. We used version 0.91. The compiler concentrates on making allocation cheap and function calls fast.

2.3 Storage management in the SML/NJ system

Storage management in the SML/NJ system has many components. One component is garbage collection. However, there are many additional components: checking whether garbage collection is needed, allocating new objects, tagging, implementing a write barrier, and implementing position independent code.

The SML/NJ system uses heap-only allocation: *all* allocation is done on the heap. In particular, all activation records are allocated on the heap rather than on a call stack. The heap is managed automatically using generational copying garbage collection [22, 1, 2].

In copying garbage collection [17, 7], an area of memory is reclaimed by copying the live (non-garbage) data to another area of memory. All data in the garbage-collected area becomes garbage and the area can be reused.

The SML/NJ system uses a simple variant of generational copying garbage collection [1]. Memory is divided into an old generation and an allocation area. New objects are created in the allocation area. When the allocation area becomes full, the live data in the allocation area is copied to the old generation in a *minor collection*. When the size of the old generation becomes sufficiently large, the entire heap is collected in a *major collection*. Live objects are copied using a Cheney scan [7], which copies objects in a breadth-first order. The criteria for when to collect the whole heap is described in Section 3.5. Generational garbage collection is efficient because most allocated objects die young (about 99% [3, p. 206]) and few objects are copied from the allocation area.

Before an object can be allocated, the mutator must check whether there is sufficient space on the heap to allocate the object. If not, a garbage collection is needed. Instead of inserting a check before every allocation, the SML/NJ compiler places a check at the beginning of most extended basic blocks[1]. This means that the cost of a check may be amortized across across several allocations. Checks are placed only on some of the extended basic blocks because for other extended basic blocks the checks are redundant; there are checks along all paths to those blocks which are sufficient. Checks are placed on many extended basic blocks that do not allocate, since these checks are also used to implement asynchronous signals [27].

Allocation is done in-line, except for the allocation of arrays and strings. Since the entire allocation area is always reclaimed, objects can be allocated sequentially from the allocation area in only two instructions. Figure 1 gives an example of pseudo-assembly code for allocating a `cons` cell. `ra` contains the `car` cell contents, `rd` contains the `cdr` cell contents, `alloc` is the address of the next free word in the allocation area, and `top` contains the end of the allocation area[2]. We do not regard initializing newly allocated storage as being part of allocation.

All objects are tagged, so that garbage collection can find all live objects and copy them. All objects except integers have a header word which gives the kind and the size of the object. The kind tells whether the object is a record, array, byte-array, etc. In addition, integers are tagged with 1 in the least significant bit and pointers are tagged with 0 in the least significant bit. This means that for integer arithmetic operations, tag manipulation instructions are needed.

```
% check for heap overflow
cmp alloc+12,top
branch-if-gt call-gc
% write the object
store tag,(alloc)
store ra,4(alloc)
store rd,8(alloc)
% save pointer to object
move alloc+4,result
% add 12 to alloc pointer
add alloc,12
```

Figure 1: Pseudo-assembly code for allocating an object

The *write barrier* tracks all pointers from the old generation to objects in the allocation area. The objects tracked by the write barrier must be regarded as live when only the allocation area is collected; otherwise collection of the allocation area could create dangling pointers. The write barrier is implemented using a *store list*. Since pointers from the old generation to the allocation area can be created only by assignment, at each assignment where the source value [3] could be a pointer, the target is added to the store list. The store list is processed when a minor collection occurs.

Code is position independent, since the SML/NJ interactive system places code in the heap and code may be relocated by garbage collection. Position independence is implemented by doing all addressing of instructions using base-offset addressing. The base register is adjusted every time a module is entered.

3 Methodology

We used trace-driven simulation to measure the cost of storage management. This allowed us to measure the cost of storage management precisely, including the memory-system cost, and to separate the cost into its components.

In the following subsections, we describe what we measured for each component of the cost of storage management, the traces and trace-generation mechanism, the

[1] An extended basic block is a block of code with only forward jumps.

[2] This figure originally appeared elsewhere [15].

[3] Given an assignment `x := t`, we say that `x` is the target of the assignment and `t` is the source of the assignment.

memory system simulated, the benchmark programs, and garbage collection sizing parameters.

3.1 Measurement methodology for each component

Table 1 lists what we measured for each component of the cost of storage management. The first three entries are the cost of garbage collection. The remaining rows are the storage management costs in the mutator.

The one instruction-level cost of storage management that we do not measure is the effect of storage management on program optimization [6]. Diwan *et al.* [13] have presented techniques that allow extensive optimization even using copying collection with unambiguous roots. However, we do not measure this cost.

Storage management also affects the memory-system cost incurred during mutation. We were unable to measure this effect directly. A detailed discussion of this effect and how we measured it is deferred to Section 4.3.

We measured the cost of position-independent code as the number of instructions spent updating the base register and the additional instructions that have to be executed relative to position-dependent code. In particular, jump tables are more expensive in position-independent code. For position-dependent code, the table address is an absolute address, while for position-independent code the table address must be computed. In addition, for position-dependent code, the table gives absolute addresses; whereas in position-independent code the table gives relative offsets and the address of the target must also be computed.

3.2 Trace generation

We extended QPT (Quick Program Profiler and Tracer) [5, 21, 20] to produce memory traces for SML/NJ programs. QPT rewrites an executable program to produce compressed trace information; QPT also produces a corresponding regeneration program that expands the compressed trace into a full address trace. Because QPT operates on the executable program, it can trace both the SML code as well as the garbage collector, which is written in C.

We extended QPT in two ways. First, we modified QPT and the SML/NJ system to produce traces for SML/NJ programs. Second, we added an event tracing facility to QPT. The changes to QPT and the SML/NJ system are described elsewhere [14, 15]. One important change we made to the SML/NJ system was to place code outside the heap so that it was not moved by garbage collection. In the original system, code was placed in the heap and it was moved by garbage collection. Allowing code to be moved makes tracing programs extremely difficult.

An *event* is a specially marked instruction. We have extended QPT so that when an event is encountered during the execution of a traced program, an event marker is inserted into the trace. The event marker identifies the event and also contains parameters to the event (if any). The simulator consuming the trace can take whatever actions are necessary when it encounters an event marker. For example, when an integer-tag event is encountered during run time, an integer-tag event marker is inserted into the trace. When the simulator sees the marker, it increments the count of instructions spent doing integer tagging.

We also used the event-tracing mechanism to mark different phases of garbage collection. For example, the first and last instructions for the Cheney scan are events. When the simulator encounters one of these events, it outputs a message noting the event and the memory-system statistics. The statistics are the total numbers of reads, writes, data-cache read misses, data-cache write misses, instruction-cache misses, and write buffer stalls through that point of program execution. From these statistics, we can compute the memory-system cost incurred during each phase of garbage collection.

We identify events to QPT by adding *event tables* to executable files. Each event table corresponds to one kind of event and lists the locations of all instructions at which that kind of event occurs. In addition, each event table also specifies the values of the parameters to the event for each instruction. When invoked on an executable file, QPT searches the executable file for these tables. We modified the SML/NJ compiler to emit these tables for "interesting" events. The event tables for the garbage collector, which is written in C, were produced manually by editing the assembly file produced by a compiler.

The tracing mechanism is non-intrusive: the traces produced by QPT correspond to addresses in the original programs rather than those in the instrumented programs.

3.3 Memory system simulation

We simulated the DECstation 5000/200 memory system using an extended version of Tycho [19], a cache simulator. The extensions to Tycho, described in [14, 15], allow us to measure costs due to the entire memory system, not just the cache.

Table 2 summarizes the memory system of the DECstation 5000/200. The DECstation 5000/200 has a split instruction and data cache. The instruction cache is direct mapped and is composed of blocks of 16 bytes each. The data cache is also direct mapped, but has a block size of 4 bytes. However, on a read miss, 16 bytes aligned on a

Root processing	Instructions to process the store list and registers, including copying objects immediately reachable from the store list and registers. Also any memory-system cost incurred while executing those instructions.
Cheney Scan	Instructions to do the breadth-first copying of objects. Also any memory-system cost incurred while executing those instructions.
Moveback	Instructions to move the old generation to one end of the free region. Also any memory-system cost incurred while executing those instructions.
Allocation	Instructions to increment the allocation pointer and initialize registers to point to newly-allocated objects. This does not include instructions to initialize newly-allocated storage before it is used.
GC Check	Instructions to check whether garbage collection is needed and to jump to the garbage collector entry code if garbage collection is needed.
Integer Tagging	Instructions to tag and untag integers.
Record Tagging	Instructions to write header words of records.
Position-Independent code	Instructions to update base register. Also, additional instructions needed relative to position dependent code to compute jump table addresses from the addressing register.
Store List	Instructions to add a record to the store list.

Table 1: Measurements for each component of the cost of storage management

Instruction cache
64K, direct mapped
Block size 16 bytes
On miss fetch aligned 16 bytes
Data cache
64K, direct mapped
Block size 4 bytes
Write through
On write miss, write word to cache
On read miss, fetch aligned 16 bytes
Write buffer
Six 4-byte entries

Table 2: Summary of the DECstation 5000/200 memory system

Task	Penalty (in cycles)
Write hit or miss	0
Read miss	15
Instruction-fetch miss	15
Non-page-mode write	5
Page-mode write	1

Table 3: Penalties of memory operations

16 byte boundary are fetched from memory. Because the DECstation 5000/200 has a block size of 4 bytes, a write miss can write to the cache immediately without fetching a block from memory[4]. The DECstation also has a write buffer to avoid stalling the CPU on a write; the CPU needs to stall on a write only when the write-buffer fills up. This memory system is favorable to allocation-intensive programs [14, 15].

3.4 Benchmark Programs

The material in this section originally appeared elsewhere [15] and is repeated here to keep this paper self-contained. Table 4 describes the benchmark programs[5]. *Knuth-Bendix, Lexgen, Life, Simple, VLIW*, and *YACC* are identical to the programs measured by Appel [3][6]. Table 5 gives for each program its source code size in lines, its maximum heap size in kilobytes, its compiled code size in kilobytes, [7] and its running time in seconds on a DECstation 5000/200. The source code size excludes comments and blank lines. The maximum heap size is the largest size of the heap during the execution of the program. It includes both live data and garbage. The compiled-code

[4]Partial-word writes are treated differently, but since there are so few in our programs, we can ignore them in our discussion without loss of accuracy.

[5]Available from the authors.

[6]The description of these programs has been taken from [3].

[7]This includes 207 kilobytes of code for the standard libraries.

size excludes the size of the garbage collector and other run-time support code. The excluded code is about 60 kilobytes in size. The running time is the minimum of five runs. All programs were compiled and run using the default compiler settings, which enable many optimizations.

Table 6 characterizes the memory references of the programs. All numbers for each program are percentages of the total number of instructions executed by the program. The *Reads*, *Writes*, and *Partial writes* columns list the reads, full-word writes, and partial-word writes for each program. These are for the *entire* program. The *assignments* column lists the non-initializing writes done only by the SML code; it excludes those done by the run-time. The *Nops* column lists the percentage of null instructions executed by each program. Note that the programs have long traces; most related works use traces that are an order of magnitude smaller. Also, note that the programs do few assignments; the majority of the writes are initializing writes.

Table 7 gives the allocation statistics for each program. All allocation and sizes are reported in 32-bit words. The *Allocation* column lists the total allocation done by each program. The remaining columns separate the allocation by kind: closures for escaping functions, closures for known functions, closures for callee-save continuations[8], records, and others (includes spill records, arrays, strings, vectors, ref cells, store list records, and floating-point numbers). For each allocation kind, the first column gives the total words allocated for objects of that kind as a percentage of total allocation and the *Size* column gives the average size in words, including the 1 word tag, of an object of that kind. The percent of total allocation in the other column for PIA and Simple is large because those programs are floating-point intensive.

Stefanović and Moss [29] find that the allocation of callee-save continuation closures on the heap has a profound impact on the young-object dynamics of ML programs. In the programs they measured[9], most objects are short-lived. They attribute the high mortality rate to the allocation of callee-save continuation closures on the heap. The age at which most objects die is program dependent, as is the percentage of objects that die at that age. In particular, YACC has a relatively high object survival rate when compared to Knuth-Bendix. We would expect YACC to have a higher garbage collection cost than Knuth-Bendix.

[8] Closures for callee-save continuations can be trivially allocated on a stack in the absence of first class continuations.

[9] This includes many of the programs we measured. However, they give data for only Knuth-Bendix and YACC in their paper.

3.5 Garbage collection sizing parameters

We used the default strategy for sizing the allocation area and the old generation [1]. The heap is sized as `r` times the size of the old generation after the old generation is collected, where `r` is the desired *ratio* of heap size to live data. We used the default system value (`r=5`). The allocation area is sized as one-half of the free space (the heap space not occupied by the old generation). As the old generation grows after each collection of the allocation area, the free space decreases and the allocation area decreases. The old generation is collected when the remaining free space is less than the original size of the old generation (less than 1/5 the size of the heap).

In addition to the ratio, the garbage collector is controlled by the *softmax* and the *initial heap size*. The softmax is a desired upper limit on the heap size. It is exceeded only to prevent programs from running out of space. The softmax was 20 megabytes; the benchmark programs never reached this limit and were able to always resize their heaps to maintain the desired ratio of 5. The initial heap size was 1 megabyte.

4 Results

In this section, we present our results. First, we give the breakdown of the cost of storage management and demonstrate that it provides insight into improving the performance of storage management by identifying potential areas of improvement. Second, we show it is important to measure as much of the cost of storage management as possible; measuring only some of the cost may be misleading. Third, we identify the different components of the memory-system cost of storage management and explain why it is difficult to measure these components. We then give measurements for most of these components and estimate upper bounds for the remaining components. These estimated costs range from a few percent to negligible on the DECStation 5000/200.

4.1 The cost of storage management

Figure 2 gives a breakdown of the cost of storage management. Table 8 gives these numbers as a table. For garbage collection, we measured the cost of processing the store list and registers (roots), scanning and forwarding reachable objects (Cheney), and moving the heap back at the end of a major collection (moveback). We also measured entry and exit costs, which are the costs of entering the collector from ML and returning from the collector to ML. These costs were insignificant and are omitted from the graph. All the costs of garbage collection include memory-system costs. For mutation we measured

Program	Description
CW	The Concurrency Workbench [8] is a tool for analyzing networks of finite state processes expressed in Milner's Calculus of Communicating Systems. The input is the sample session from Section 7.5 of [8].
Knuth-Bendix	An implementation of the Knuth-Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyzer generator, implemented by James S. Mattson and David R. Tarditi [4], processing the lexical description of Standard ML.
Life	The game of Life, written by Chris Reade [25], running 50 generations of a glider gun. It is implemented using lists.
PIA	A perspective inversion algorithm [32], deciding the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [9], translated into ID [16], and then translated into Standard ML by Lal George.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
YACC	An LALR(1) parser generator, implemented by David R. Tarditi [30], processing the grammar of Standard ML.

Table 4: Benchmark programs

	Size			Run time	
Program	Lines	Heap size (K)	Code size (K)	Non-gc (sec)	Gc (sec)
CW	5728	1107	894	22.74	3.09
Knuth-Bendix	491	2768	251	13.47	1.48
Lexgen	1224	2162	305	15.07	1.06
Life	111	1026	221	16.97	0.19
PIA	1454	1025	291	6.07	0.34
Simple	999	11571	314	25.58	4.23
VLIW	3207	1088	486	23.70	1.91
YACC	5751	1632	580	4.60	1.98

Table 5: Sizes of benchmark programs

Program	Inst Fetches	Reads (%)	Writes (%)	Partial Writes (%)	Assignments (%)	Nops (%)
CW	523,245,987	17.61	11.61	0.01	0.41	13.24
Knuth-Bendix	312,086,438	19.66	22.31	0.00	0.00	5.92
Lexgen	328,422,283	16.08	10.44	0.20	0.21	12.33
Life	413,536,662	12.18	9.26	0.00	0.00	15.45
PIA	122,215,151	25.27	16.50	0.00	0.00	8.39
Simple	604,611,016	23.86	14.06	0.00	0.05	7.58
VLIW	399,812,033	17.89	15.99	0.10	0.77	9.04
YACC	133,043,324	18.49	14.66	0.32	0.38	11.14

Table 6: Characteristics of benchmark programs

Program	Allocation (words)	Escaping %	Escaping Size	Known %	Known Size	Callee Saved %	Callee Saved Size	Records %	Records Size	Other %	Other Size
CW	56,467,440	4.0	4.12	3.3	15.39	67.2	6.20	19.5	3.01	6.0	4.00
Knuth-Bendix	67,733,930	37.6	6.60	0.1	15.22	49.5	4.90	12.7	3.00	0.1	15.05
Lexgen	33,046,349	3.4	6.20	5.4	12.96	72.7	6.40	15.1	3.00	3.7	6.97
Life	37,840,681	0.2	3.45	0.0	15.00	77.8	5.52	22.2	3.00	0.0	10.29
PIA	18,841,256	0.4	5.56	28.0	11.99	25.0	4.69	12.7	3.41	33.9	3.22
Simple	80,761,644	4.0	5.70	1.1	15.33	68.1	6.43	8.3	3.00	18.5	3.41
VLIW	59,497,132	9.9	5.22	6.0	26.62	61.8	7.67	20.3	3.01	2.1	2.60
YACC	17,015,250	2.3	4.83	15.3	15.35	54.8	7.44	23.7	3.04	4.0	10.22

Table 7: Allocation characteristics of benchmark programs

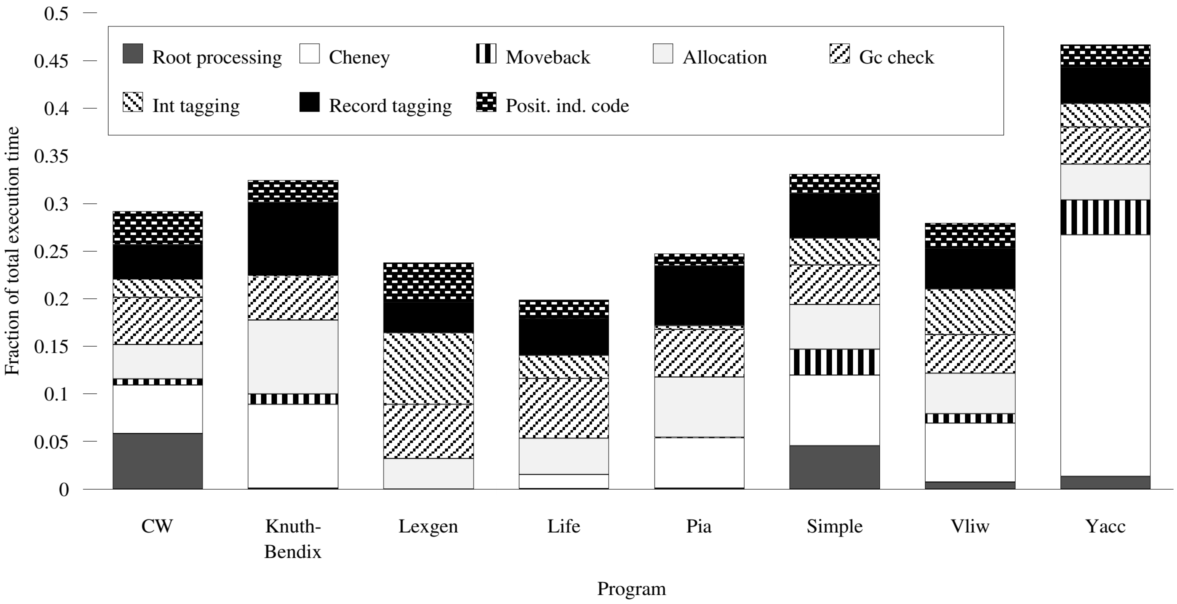

Figure 2: Breakdowns of storage management cost for benchmark programs.

Program	Roots	Cheney	Moveback	Allocation	Gc check	Int tag	Record tag	Posit. ind. code	Store list
CW	0.06	0.05	0.01	0.04	0.05	0.02	0.04	0.03	0.01
Knuth-Bendix	0.00	0.09	0.01	0.08	0.05	0.00	0.08	0.02	0.00
Lexgen	0.00	0.06	0.01	0.03	0.06	0.08	0.03	0.04	0.00
Life	0.00	0.01	0.00	0.04	0.06	0.02	0.04	0.02	0.00
PIA	0.00	0.05	0.00	0.06	0.05	0.00	0.06	0.01	0.00
Simple	0.05	0.07	0.03	0.05	0.04	0.03	0.05	0.02	0.00
VLIW	0.01	0.06	0.01	0.04	0.04	0.05	0.04	0.03	0.00
YACC	0.01	0.25	0.04	0.04	0.04	0.02	0.04	0.02	0.00

Table 8: Breakdowns of storage management costs for benchmark programs. All numbers are fractions of total execution time

the instruction counts for allocation (allocation), checking whether garbage collection is needed (gc check), tagging integers (int tagging), storing header words (record tagging), and position-independent code (posit. ind. code). The cost of adding elements to the store list was negligible for most programs, so it is omitted from the graph.

Figure 2 suggests several opportunities for improving the performance of storage management in SML/NJ programs. First, eliminating position-independent code can reduce program execution time by 1% to 4%. While placing code in the garbage-collected heap is useful in an interactive setting, it is not useful for stand-alone executables.

Second, improving integer tagging, which accounts for 0% to 8% of overall execution time, ignoring the negligible instruction cache penalty, may reduce execution time. Integers are tagged with 1 and pointers are tagged with 0 in the least significant bit. By tagging integers with 0 and pointers with 1 and using displacement addressing, which is available on many architectures, many tagging operations can be removed. Representation analysis may also eliminate some tagging operations.

Third, the store-check mechanism can be improved by using a hash-based scheme. The cost of adding elements to the store list during mutation is 0% to 1% of overall execution time, while the cost of processing the store list during garbage collection is 0% to 6% of overall execution time (Table 8, columns *Store list* and *Roots*). In a hash-based scheme, a table is used instead of a list, and duplicate entries are eliminated when elements are added to the table during mutation. The cost of adding entries during mutation would be similar for both schemes, since the cost of adding an element to the store list is already high (8 or 9 instructions). The cost of processing the table during garbage collection would be lower than the cost of processing the list, since the table has no duplicate entries.

Fourth, scanning and forwarding reachable objects takes 1% to 25% of overall execution time. This can be reduced by coding the inner loops of the collector carefully and by increasing the size of the allocation area. Increasing the size of the allocation area allows more objects to die, and thus fewer objects need to be copied during garbage collection. However, this involves a trade-off: increasing the size of the allocation area may reduce copying time but increase the memory-system cost. For the DECStation 5000/200 memory-system organization, which is favorable to heap allocation, increasing the allocation area size is unlikely to change the memory-system cost. Halving the cache size for the DECStation 5000/200 organization affects performance little [14, 15]. Some other memory-system organizations, such as the SPARC-StationII [10], are more sensitive to cache size. Increasing the allocation area size can increase the memory-system cost greatly.

An interesting point to note about Figure 2 is that the cost of checking whether garbage collection is needed (gc check) is larger than the cost of allocation (allocation) for CW, Lexgen, and Life. This is despite the fact that the check and allocating an object both take two instructions on the MIPS, and that a check is sometimes for multiple allocations. We speculate that this is because the SML/NJ compiler overloads checks to implement asynchronous signals [27]. This results in checks in extended basic blocks which do no allocation, so that the allocation cost is not an upper bound on the cost of checking whether garbage collection is needed.

4.2 Mosts cost are incurred during mutation

To illustrate how inaccurate it can be to measure only some of the cost of storage management, Figure 3 compares the cost of garbage collection against the cost of instructions executed during mutation to support storage management. The garbage-collection cost includes the memory-system cost, while the storage-management cost during mutation does not include the memory-system cost. Still, the storage-management cost during mutation is larger than the garbage-collection cost for seven of the programs. This shows that regarding the cost of garbage collection as the cost of storage management is inaccurate; most of the cost is often elsewhere.

4.3 The memory-system cost of storage management

Because the cache is a global shared resource, it can be difficult to pinpoint the exact cause of a cache miss. One instruction may cause a cache miss which knocks data out of the cache that is used by a subsequent instruction. Thus, the subsequent instruction will also have a cache miss. This makes it difficult to correlate misses with their actual source: an instruction at which a miss occurs may not be the actual cause of the miss.

Storage management may incur memory-system cost

1. during garbage collection.

2. by collection displacing mutator data and instructions from the cache.

3. by allocating memory which is not resident in the cache. When the mutator initializes or uses the memory, cache misses may occur.

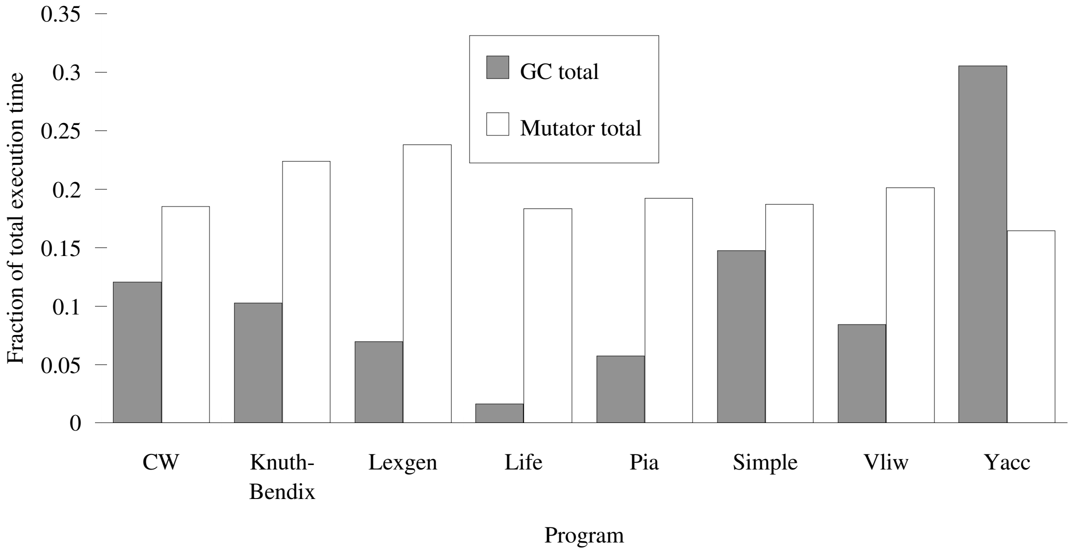

Figure 3: Comparison of garbage collection and mutation storage-management costs

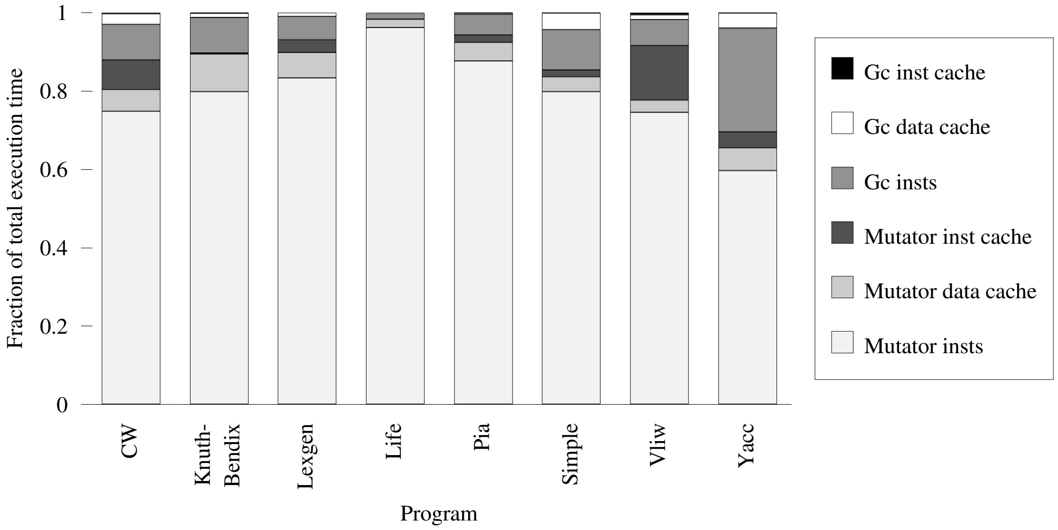

Figure 4: Breakdown of memory-system cost during collection and mutation.

Program	Before (miss rate)	After (miss rate)
CW	0.03	0.02
Knuth-Bendix	0.04	0.04
Lexgen	0.03	0.03
Life	0.01	0.01
PIA	0.01	0.01
Simple	0.01	0.01
VLIW	0.02	0.01
YACC	0.03	0.04

Table 9: Data-cache miss rates before and after garbage collections

4. by increasing the size of data by adding header words. This reduces spatial locality and the effective size of the cache.

5. by reducing the spatial locality of mutation by rearranging the layout of data in memory. This may also improve spatial locality.

6. by changing the code size of the mutator by requiring code for tagging, the write barrier, and position-independent code. This may increase instruction-cache misses.

We measured some of the components of the memory-system cost of storage management, and placed upper bounds on the remaining components

We measured the memory-system cost incurred during garbage collection. Figure 4 presents the memory-system costs incurred during mutation and collection. The memory-system cost that occurred during collection is 0.1% to 4.3% of overall execution time. The cost of instruction-cache misses during collection is negligible.

We also measured the effect of garbage collection displacing instructions and data used during mutation from the cache. The effect was negligible. The cost of instruction misses during mutation caused by garbage collection is bounded from above by the cost of instruction misses during garbage collection, since the machine has a split instruction and data cache. Since the cost of instruction misses during garbage collection is low (see Figure 4), this effect is negligible. To measure the effect of data-cache misses during mutation caused by collection, we collected memory-system statistics for intervals before and after each collection. The arithmetic mean of the sizes of the intervals measured before each collection was 143,000 instructions. The mean of the sizes of the intervals after each collection was 166,000 instructions. Table 9 shows the average cache-miss rates for the intervals for each program; the `Before` column gives the data-cache read-miss rates for the intervals just before each garbage collection and the `After` column gives the data-cache read-miss rates for the intervals after each garbage collection. If garbage collection were disturbing the cache locality of mutation, we would expect the miss rates after garbage collection to be noticeably higher than those before garbage collection. We see only a slight variation. This suggests that data references during collection do not cause significant data-cache misses during mutation.

Program	Upper Bound (% execution time)
CW	2.8
Knuth-Bendix	4.8
Lexgen	3.2
Life	1.1
PIA	2.4
Simple	1.9
VLIW	1.6
YACC	2.9

Table 10: Upper bound on disruption of spatial locality by storage management

Because the DECStation 5000/200 memory system has no penalty for write misses, besides write-buffer penalties which are small enough to be negligible, there is no cache penalty for initializing writes that miss. If there were a penalty for write misses[10], the programs would run 24% to 72% slower than they do now [14, 15]. In other words, with a penalty for cache write misses, most of the cost of storage management would be for initializing newly-allocated objects.

While we have been unable to measure the remaining components of the memory-system cost of storage management exactly, we have placed upper bounds on these components. Recall that header words and copying during garbage collection may worsen the spatial locality of the program. To bound this, note that obtaining perfect spatial locality could change the cache misses by at most a factor of 2. This is because without header words, at most two of the smallest objects (2 word cons cells) can fit in 16 bytes (which is the amount fetched on a read miss). Table 10 demonstrates that the upper bound on disruption of spatial locality by garbage collection and other storage-management tasks is small. The `Upper Bound` column gives an upper bound of the percentage of total execution time lost to disrupted spatial locality. Of course, the cost of disrupted spatial locality may be more substantial with larger fetch sizes or larger cache-miss penalties.

Header words may also increase the memory-system cost of mutation by decreasing the effective cache size.

[10] In particular, if write misses were non-blocking and required the cache block to be fetched from main memory.

Program	Upper Bound (% execution time)
CW	1.8
Knuth-Bendix	3.7
Lexgen	2.6
Life	1.0
PIA	1.8
Simple	0.8
VLIW	0.7
YACC	1.0

Table 11: Upper bound on data cache costs due to smaller effective cache size

Program	Est. cost (% execution time)
CW	1.8
Knuth-Bendix	0.1
Lexgen	0.9
Life	0.0
PIA	0.4
Simple	0.4
VLIW	3.8
YACC	1.1

Table 12: Estimate of instruction cache costs due to storage management instructions

Since a header word occupies 1/3 of the space used for the typical smallest object, a list cell, at most 1/3 of the space in the cache is being occupied by header words. We would need a cache which is 50% larger in practice to achieve the same effective cache size. We can give a generous estimate of the cost of header words by subtracting the data-cache cost for a 128K cache from the cost for 64K cache, that is, generously assuming that without header words the cache size is effectively doubled. Table 11 gives the improvement in going to a 128K cache; this is an upper bound on the memory-system costs due to header words decreasing the effective cache size.

The impact of effectively increasing the size of data depends on cache boundary conditions. If data fits well within the cache, then increasing the size does not matter much. However, if data just fits in the cache, then increasing the size of the data may cause cache misses. If data does not fit in the cache, increasing the size of the data leads to proportionately more cache misses. The effect of cache size on SML/NJ programs is explored thoroughly elsewhere [15]. There were no dramatic boundary conditions for SML/NJ programs on the DECStation 5000/200.

Just as header words increase the size of data, storage-management instructions increase the size of a program. This may cause additional instruction-cache misses. The fraction of instruction misses during mutation which are on average due to storage management is the fraction of instructions executed during mutation in support of storage management. Figure 12 gives an estimate of the cost due to these extra instruction-cache misses for each of the programs.

5 Related work

This study is more comprehensive in its measurements than previous works studying the cost of storage management in garbage-collected systems. Ungar [31] measures the time spent garbage collecting and the cost of integer tagging in a Smalltalk system, but does not measure other costs incurred during mutation. Zorn [34] compares the cost of two simulated garbage-collection algorithms. In contrast, we measure an actual implementation. He measures the memory-system cost using the cache-miss ratio, which is an inaccurate indicator of performance because it does not separate the cost of read and write misses [15]. Wilson *et al.*[33] and Peng and Sohi [24] also measure the memory-system cost of garbage collection using the cache-miss ratio. They do not measure the instruction-level cost of garbage collection or costs incurred during mutation. Reinhold [26] measures the cost of garbage collection for a Scheme system, including the change in memory-system performance of entire programs, but does not measure costs incurred during mutation.

Steenkiste [28] studies ways to reduce the cost of tagging in Lisp. He also studies instructions used for stack allocation. He is primarily concerned with hardware support to improve tag checking required for dynamic typing. He finds that tag insertion and removal costs about 4.5% with the best software scheme.

There have been several studies of the cost of storage management in languages with explicitly-managed heap storage and stack allocation of procedure activation records. Detlefs [11] measures time spent in allocation and deallocation routines, but does not measure the cost of managing the stack. Grunwald *et al.* [18] finds that the implementation of explicit heap management can affect the performance of allocation-intensive C programs significantly.

6 Conclusion

We have studied the cost of storage management for programs compiled with the SML/NJ compiler. Modern programming languages, such as SML, LISP, and object-oriented languages, use dynamic heap allocation extensively. Thus, the cost of storage management can have a major effect on program performance. Unlike other work measuring the cost of storage management, we measured both the time spent garbage collecting and costs incurred during mutation.

We believe that the design of high-performance garbage collectors and the debate about the cost of garbage collection should be based on measurements like those presented here.

We used trace-driven simulation to measure the cost of storage management. This allowed us to measure the cost of storage management precisely, including instruction-level and memory-system costs. Moreover the tools allowed us to separate the cost of storage management into its components. We measured eight programs compiled with the SML/NJ compiler running on a DECStation 5000/200.

We found that most of the cost of storage management did not occur during garbage collection; rather it occurred during mutation. We also found that SML/NJ programs spent 19% to 46% of their execution time doing storage management. These measurements indicate that it is necessary to measure all the cost of storage management; merely measuring the time spent garbage collecting is not adequate.

7 Acknowledgements

We would like to thank Anurag Acharya, Urs Hoezle, Peter Lee, Mark Leone, Greg Morrisett, Kathryn McKinley, and the anonymous referees for Lisp and Functional Programming '94 for comments on drafts of this paper.

References

[1] APPEL, A. W. Simple generational garbage collection and fast allocation. *Software — Practice and Experience 19*, 2 (Feb. 1989), 171–184.

[2] APPEL, A. W. A Runtime System. *Lisp and Symbolic Computation 3*, 4 (Nov. 1990), 343–380.

[3] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] APPEL, A. W., MATTSON, J. S., AND TARDITI, D. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[5] BALL, T., AND LARUS, J. R. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1992), ACM.

[6] CHASE, D. R. Safety considerations for storage allocation optimizations. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 1–10.

[7] CHENEY, C. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–678.

[8] CLEAVELAND, R., PARROW, J., AND STEFFEN, B. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 36–72.

[9] CROWLEY, W. P., HENDRICKSON, C. P., AND RUDY, T. E. The SIMPLE code. Tech. Rep. UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, Feb. 1978.

[10] CYPRESS SEMICONDUCTOR, ROSS TECHNOLOGY SUBSIDIARY. *SPARC RISC User's Guide*, second ed., Feb. 1990.

[11] DETLEFS, D., DOSSER, A., AND ZORN, B. Memory allocation costs in large C and C++ programs. Tech. Rep. CU-CS-665-93, University of Colorado, 1993.

[12] DIGITAL EQUIPMENT CORPORATION. *DS5000/200 KN02 System Module Functional Specification*.

[13] DIWAN, A., MOSS, J. E. B., AND HUDSON, R. L. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), SIGPLAN, ACM Press, pp. 273–282.

[14] DIWAN, A., TARDITI, D., AND MOSS, E. Memory subsystem performance of programs with intensive heap allocation. Tech. Rep. CMU-CS-93-227, School of Computer Science, Carnegie Mellon University, Dec. 1993. Submitted for publication.

[15] DIWAN, A., TARDITI, D., AND MOSS, E. Memory subsystem performance of programs with copying garbage collection. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming languages* (Portland, Oregon, Jan. 1994), ACM, pp. 1–14.

[16] EKANADHAM, K., AND ARVIND. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[17] FENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM 12*, 11 (Nov. 1969), 611–612.

[18] GRUNWALD, D., ZORN, B., AND HENDERSON, R. Improving the cache locality of memory allocation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June 1993), ACM, pp. 177–186.

[19] HILL, M., AND SMITH, A. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38*, 12 (Dec. 1989), 1612–1630.

[20] LARUS, J. R. Abstract Execution: A technique for efficiently tracing programs. *Software Practice and Experience 20*, 12 (Dec. 1990), 1241–1258.

[21] LARUS, J. R., AND BALL, T. Rewriting executable files to measure program behavior. Tech. Rep. Wis 1083, Computer Sciences Department, University of Wisconsin-Madison, Mar. 1992.

[22] LIEBERMAN, H., AND HEWITT, C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6 (1983), 419–429.

[23] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[24] PENG, C.-J., AND SOHI, G. S. Cache memory design considerations to support languages with dynamic heap allocation. Tech. Rep. 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[25] READE, C. *Elements of Functional Programming*. Addison-Wesley, Reading, Massachusetts, 1989.

[26] REINHOLD, M. B. *Cache Performance of Garbage-Collected Programming Languages*. PhD thesis, Laboratory for Computer Science, MIT, Sept. 1993.

[27] REPPY, J. H. Asynchronous Signals in Standard ML. Tech. Rep. 90-1144, Department of Computer Science, Cornell University, Aug. 1990.

[28] STEENKISTE, P. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford,CA 94305, Mar. 1987.

[29] STEFANOVIĆ, D., AND MOSS, E. Characterisation of object behavior in Standard ML of New Jersey. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming* (1994).

[30] TARDITI, D., AND APPEL, A. W. ML-YACC, version 2.0. Distributed with Standard ML of New Jersey, Apr. 1990.

[31] UNGAR, D. *The design and evaluation of a high performance Smalltalk system*. ACM Distinguished Dissertation. MIT Press, Cambridge, Massachusetts, 1987.

[32] WAUGH, K. G., MCANDREW, P., AND MICHAELSON, G. Parallel implementations from function prototypes: a case study. Tech. Rep. Computer Science 90/4, Heriot-Watt University, Edinburgh, Aug. 1990.

[33] WILSON, P. R., LAM, M. S., AND MOHER, T. G. Caching considerations for generational garbage collection: a case for large and set-associative caches. Tech. Rep. EECS-90-5, University of Illinios at Chicago, Dec. 1990.

[34] ZORN, B. G. *Comparative Performance evaluation of garbage collection algorithms*. PhD thesis, University of California, Berkeley, CA 94720, Dec. 1989.