The Full Cost of a Generational Copying Garbage Collection Implementation*

David Tarditi Amer Diwan†

September 10, 1993

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

1 Introduction

What is the cost of garbage collection? In recent years there have been a number of studies of the performance of generational copying garbage collection. The studies explore some of the costs of garbage collection such as cache performance [12, 13, 14, 8], pause times, tagging overhead [10], and write barrier overhead [10, 3]. However, no one has measured the *full cost* of a garbage collector. Most importantly, no one has measured the actual memory subsystem overhead. The novel point of this study is that it reports the entire cost of the Standard ML of New Jersey generational copying garbage collector on a DECStation 5000/200. The various costs are broken down by kind, such as memory subsystem overhead, tagging, etc.

With the data we collected, we can more authoritatively answer the following questions (for one point in the universe):

1. *Is GC expensive ?*

   We found the total cost due to garbage collection was from 22% to 40% of overall execution time [1].

2. *Is memory subsystem performance of the garbage collector worse than that of mutators ?*

   In general, no.

3. *Does intense heap allocation lead to poor cache performance?*

   No. On average the overall CPI penalty for the memory subsystem was 0.15 (less than a 15% overhead).

---

*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. 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 or the United States Government. David Tarditi was also supported by an AT&T PhD Scholarship.

†The authors can be reached electronically via Internet addresses {dtarditi,adiwan}@cs.cmu.edu. This work was done while Amer Diwan was on leave from University of Massachusetts.

[1] We assume that all instructions take one cycle and that the execution time of the program (in cycles) is the number of instructions plus the cycles lost to the memory subsystem.

Task	Penalty (in cycles)
Non page mode write	5
Page mode write	1
Read 16 bytes from memory	15
Refresh period	195
Refresh time	5
Write hit or miss	0

Table 1: Penalties of memory operations

4. *Does copying garbage collection disrupt the spatial locality of the mutator ?*

   No. On average, the CPI penalty for the data cache was 0.07, which is a generous upper bound on the performance penalty due to the disruption of spatial locality.

The following sections describe our methodology, important aspects of the SML/NJ garbage collector, the observed performance, and our conclusions.

2 Methodology

We used trace-driven simulations of six benchmarks to collect the data. In addition to memory references, the traces also contained "interesting events" such as write-barrier instructions, tag manipulation, etc. The traces were all for the MIPS R3000 architecture and the memory subsystem organization used in the simulations was similar to that of DECStation 5000/200 (which has been previously demonstrated to be favorable for allocation intensive programs [6]). All programs were written in Standard ML [7], a mostly-functional programming language, and were compiled using SML/NJ.

The DECStation 5000/200 has a 64K instruction and a 64K data cache. Both the instruction and data caches are direct mapped. The block size of the cache is 1 word but on a read miss, four words are fetched (aligned on a 4 word boundary). All writes (hits and misses) are written both to the cache and main memory[2]. This cache organization behaves very much like *subblock placement*. To prevent CPU stalls on writes, a 6 deep write buffer is provided. Also, the memory subsystem has page-mode smarts, which can retire subsequent writes to the same DRAM page in only 1 cycle, provided there are no intervening reads, writes to other pages, or memory refreshes. Table 1 gives the penalties used in our simulations; they are for the DECStation 5000/200.

Table 2 describes our benchmark programs. Table 3 the sizes of the benchmarks in terms of lines of SML code (excluding comments and blank lines), maximum heap size in kilobytes, size of the compiled code in kilobytes (does not include the garbage collector and other run-time support code which is about 60K)[3], and run time, in seconds, on a DECStation 5000/200.

3 Memory management in SML/NJ

SML/NJ 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. Thus, the performance of SML/NJ programs relies heavily on the efficiency of the garbage collector.

[2]Actually partial-word writes are treated differently, but since there are so few in our programs, we ignore them here without any loss of accuracy [6].

[3]The code size includes 207K for the standard libraries.

Program	Description
CW	The Concurrency Workbench [4] is a tool for analyzing networks of finite state processes expressed in Milner's Calculus of Communicating Systems.
Lexgen	A lexical-analyzer generator [2], processing the lexical description of Standard ML.
PIA	The Perspective Inversion Algorithm [11] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program [5].
VLIW	A Very-Long-Instruction-Word instruction scheduler.
YACC	An implementation of an LALR(1) parser generator [9] processing the grammar of Standard ML.

Table 2: 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
Lexgen	1224	2162	305	15.07	1.06
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 3: Sizes of Benchmark Programs

program	instruction count	allocation rate (words/instruction)	overall MCPI	gc time (%)
cw	523245987	0.11	0.20	25.58
lexgen	328422274	0.10	0.12	26.64
pia	122215199	0.11	0.08	15.62
simple	604610992	0.11	0.11	24.79
vliw	399812032	0.15	0.25	22.50
yacc	133043285	0.13	0.17	39.99
avg.	351891628.17	0.12	0.15	25.85

Table 4: Overall summary

SML/NJ uses a generational copying garbage collector to manage its heap [1]. The collector has two generations: an allocation area and an old generation. Objects are created in the allocation area. Objects that survive a garbage collection are promoted to the old generation.

To allow the garbage collector to find the pointers, all SML objects except integers have a header word which gives the kind (e.g., record, array, byte-array, etc.) and size of the object. In addition, all integers are tagged with 1 in the least significant bit and pointers with 0 in the least significant bit. Tag manipulation instructions are needed in many arithmetic operations. We use "header word overhead" to refer to the overhead of storing and copying the header words . We use "integer tag overhead" to refer to the overhead of integer tag manipulation.

SML/NJ uses a store list to keep track of pointer stores (which are needed to detect pointers from the old generation to the allocation area). Each pointer assignment requires a four-word record to be allocated and chained on to the store list. We use "write barrier overhead" to refer to the overhead of adding to the store list.

SML/NJ amortizes the cost of testing whether garbage collection is needed by placing a check at the beginning of each function rather than on every allocation[4]. These checks are also placed in functions that do no allocation since this mechanism is used to implement asynchronous signals. We use "heap-exhausted overhead" to refer to the overhead of testing to see if garbage collection is needed.

The SML/NJ interactive system puts all its code in the heap. Thus the code needs to be position independent since it may be relocated by the garbage collector. All addressing (of instructions) is done as offsets from a base register which needs to be adjusted every time a module is entered. We refer to the overhead of adjusting and addressing from the base register as "pic overhead". For our measurements, we placed the code in the text segment so there was no need to use the base register. However, we wanted to measure the overhead of using position independent code so we left it in.

4 Results

Table 4 summarizes the overall memory performance of the benchmark programs and the total overhead due to garbage collection. All memory subsystem penalties in this section are summarized as MCPI, the memory subsystem contribution to CPI. All percentages are in terms of total program cycle counts. We see that the average overall MCPI is 0.15, and that the average percentage of execution time spent doing garbage collection-related operations is 26%.

Table 5 separates the memory subsystem performance of the garbage collector from the mutator. It also lists the read miss rates for both the garbage collector and the mutator. Write miss rates are not given because they have no penalty on the DS5000/200. The miss rates indicate the data cache locality for the

[4]The compiler does an analysis to remove some of the checks for sequences of calls to known functions.

program	gc read miss rate	gc MCPI	mutator read miss rate	mutator MCPI	overall MCPI
cw	0.11	0.32	0.03	0.19	0.20
lexgen	0.06	0.12	0.03	0.12	0.12
pia	0.03	0.07	0.01	0.08	0.08
simple	0.16	0.43	0.02	0.08	0.11
vliw	0.06	0.19	0.02	0.26	0.25
yacc	0.06	0.12	0.04	0.19	0.17
avg.	0.08	0.21	0.02	0.15	0.15

Table 5: MCPI and locality of garbage collector and mutator

program	gc write buffer (MCPI)	gc read misses (MCPI)	gc if misses (MCPI)	mutator write buffer (MCPI)	mutator read misses (MCPI)	mutator if misses (MCPI)
cw	0.00	0.30	0.01	0.00	0.08	0.10
lexgen	0.00	0.12	0.00	0.00	0.08	0.04
pia	0.00	0.07	0.00	0.00	0.05	0.02
simple	0.02	0.40	0.00	0.00	0.06	0.02
vliw	0.00	0.14	0.04	0.02	0.05	0.19
yacc	0.00	0.12	0.00	0.00	0.11	0.07
avg.	0.00	0.19	0.00	0.00	0.07	0.07

Table 6: Breakdown of MCPI

garbage collector and the mutator. While the garbage collector consistently has a much higher read miss rate than the mutator, MCPI for the garbage collector is lower than that of the mutator for three of the benchmarks.

To explain why the MCPI of the garbage collector is often lower than that of the mutator, Table 6 breaks down MCPI by cause (write-buffer stalls, read misses, and instruction fetch misses). The garbage collector has much better instruction locality than the mutator. The penalty due to instruction fetch misses in the garbage collector is negligible for five of the six benchmarks. In addition, the effect of the high read miss rate in the garbage collector is lessened by the fact that the garbage collector often has fewer reads per instruction than the mutator. This is probably due to better register usage in the garbage collector which is written in C.

The penalty due to write-buffer stalls is almost negligible. This is not surprising, since the DS5000/200 has page-mode writes, and can retire writes to the same DRAM page at one per cycle.

Table 6 also gives an upper bound on the disruption to spatial locality caused by the garbage collector. Disrupting spatial locality worsens the MCPI due to data read misses. The total data read miss MCPI is 0.07, so disrupted spatial locality can increase CPI by at most 0.07. In other words, the best performance improvement we can get from improving spatial locality is 7%. Given the large size of even the smallest object (a cons cell, 3 words including a header word) relative to the block size (4 words), we can expect the gain from improving spatial locality to be less than half of that (*i.e.,* less than 3.5%). This is because improving spatial locality will at most halve the read misses by putting two objects in the same cache block.

Table 7 gives a breakdown of the costs of garbage collection. The column labeled “gc insts.” gives the

program	heap exhausted (%)	write barrier (%)	total tagging (%)	pic (%)	gc insts. (%)	gc memory (%)
cw	4.92	0.93	5.40	3.45	8.27	2.61
lexgen	5.66	0.00	10.73	4.16	5.42	0.67
pia	5.00	0.00	3.85	1.27	5.12	0.37
simple	4.11	0.31	6.40	2.03	8.34	3.59
vliw	3.98	0.07	8.97	2.64	5.77	1.07
yacc	3.86	0.12	6.59	2.38	24.13	2.91
avg.	4.59	0.24	6.99	2.65	9.51	1.87

Table 7: Breakdown of garbage collection costs

percentage of time spent executing garbage collector instructions, excluding memory subsystem costs. The column "gc memory" gives the memory subsystem costs for the garbage collector. On average, memory subsystem overheads comprise about 20% of the execution time of the garbage collector, although that varies considerably. The "heap exhausted" column gives the cost of checking for free space in the heap. The "write barrier" column gives the cost of implementing the write barrier (each of which takes 8 or 9 instructions). These programs simply do not do many pointer assignments. Total tagging gives the cost of all tag-related instructions in the code, including tagging integers, writing out header words, and copying header words[5]. Figure 1 gives the same information in graphical form.

Using the information in Figure 1 and Table 7, we can speculate about the cost of garbage collection on a different memory subsystem organization and about the cost of a different garbage collector. Regarding memory subsystem organizations, we have demonstrated in [6] that heap allocation performs well on the DECStation 5000/200 organization because it has a write-allocate policy and has no penalty for write-misses. Thus, we expect that on a less favorable memory subsystem organization (such as the SPARCStation II), the cycle count of some garbage collection operations will be greater. In particular, the contribution of write barriers and header-word tagging will be higher in the less favorable memory subsystem organizations because they both contribute to initializing writes which usually miss in the cache[6]. However, the percentage of time spent in garbage collection and related operations will decrease because the SML code writes at a much higher rate than the garbage collector.

Regarding different garbage collectors, Table 7 gives an upper bound on the performance of a tag-free garbage collection scheme: at most, going to a tag-free garbage collector would improve performance on average by 7%

The cost of tagging is broken down in Table 8. Eliminating header words from records may improve performance by about 3.5%. The cost of copying header words is an estimate. It was estimated by taking the number of writes done by the garbage collector, and dividing by 4; when copying a cons cell (the typical small object), the collector must make one write to install a forwarding pointer, and 3 writes to copy the cons cell, only one of which involves copying the tag.

5 Conclusions

We have measured the full cost of garbage collection for the SML/NJ system running on a DECStation 5000/200. In particular, we have also measured the memory subsystem overhead of garbage collection.

[5] As noted earlier, the overhead of copying header words is an approximation.

[6] Using the analytical model in [6], we can predict that on a SPARCStation with a 8 word cache size and a 24 cycle miss penalty, the memory subsystem will add 12 cycles to each write barrier

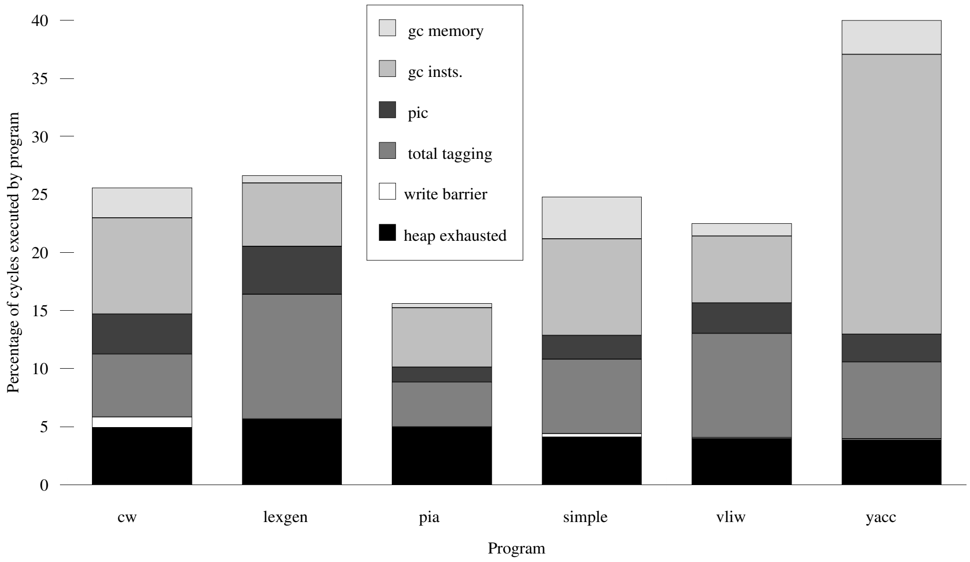

Figure 1: Breakdown of garbage collection costs as a percentage of overall cycles

program	integers (%)	header words (%)	copying header words (%)
cw	1.92	3.38	0.09
lexgen	7.50	3.14	0.10
pia	0.40	3.34	0.12
simple	2.83	3.38	0.19
vliw	4.69	4.19	0.10
yacc	2.46	3.68	0.44
avg.	3.30	3.52	0.17

Table 8: Tagging breakdown

We found the overall cost of garbage collection to be high: 22% to 40% of overall execution time. Most surprisingly we found the memory subsystem performance of the garbage collector to be quite good; in any case, it is no worse than that of the mutator. The garbage collection components which contributed most to the program execution time were instructions executed in the collector (9.5%), tagging overhead (7%), and checks to see if garbage collection is needed (5.6%). The write barrier overhead was low since our benchmarks do few pointer assignments.

We found that copying garbage collection did not significantly disrupt the spatial locality of programs. Our measurements suggest that changing the garbage collection scheme to preserve (or improve) the spatial locality of programs can speed up programs by at most 3.5%.

Finally, we found the overall memory subsystem performance to be good despite the intensive heap allocation.

References

[1] Andrew W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[2] Andrew W. Appel, James S. Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[3] Craig Chambers. Cost of garbage collection in the self system. Appeared in OOPSLA workshop on garbage collection, 1991.

[4] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[5] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

[6] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for publication, July 1993.

[7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[8] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[9] David Tarditi and Andrew W. Appel. ML-YACC, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[10] David Ungar. *The design and evaluation of a high performance Smalltalk system*. ACM Distinguished Dissertation. MIT Press, 1987.

[11] Kevin G. Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, August 1990.

[12] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection: a case for large and set-associative caches. Technical Report EECS-90-5, University of Illinios at Chicago, December 1990.

[13] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, June 1992.

[14] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.