# Cache Performance of Fast-Allocating Programs

Marcelo J. R. Gonçalves and Andrew W. Appel
Princeton University
Department of Computer Science
{mjrg,appel}@cs.princeton.edu

Princeton CS-TR-482-94
December 21, 1994

**Abstract**

We study the cache performance of a set of ML programs, compiled by the Standard ML of New Jersey compiler. We find that more than half of the reads are for objects that have just been allocated. We also consider the effects of varying software (garbage collection frequency) and hardware (cache) parameters. Confirming results of related experiments, we found that ML programs can have good cache performance when there is no penalty for allocation. Even on caches that have an allocation penalty, we found that ML programs can have lower miss ratios than the C and Fortran SPEC92 benchmarks.

Topics: 4 benchmarks, performance analysis; 21 hardware design, measurements; 17 garbage collection, storage allocation; 46 runtime systems.

## 1 Introduction

With the gap between CPU and memory speed widening, good cache performance is increasingly important for programs to take full advantage of the speed of current microprocessors. Most recent microprocessors come with a small on-chip cache, and many machines add a large second level off-chip cache to that. It is therefore important to understand the memory behavior of programs, so that computer architects can design cache organizations that can exploit the locality of reference of these programs, or programmers can modify the programs to take advantage of cache properties, or both. This work studies the cache performance of programs with intensive heap allocation and generational garbage collection, such as ML programs compiled by the *Standard ML of New Jersey* (SML/NJ) compiler[4].

Dynamic heap allocation is used in the implementation of many programming languages, but the SML/NJ implementation is characterized by a much higher allocation rate than most other implementations: a typical program allocates one word for every six machine instructions. SML/NJ allocates so frequently because it allocates function activation records (closures) on the heap, instead of using a stack as do most language implementations[2]. In order to make heap allocation efficient it is essential that both allocation and deallocation be very fast[1]. Efficient allocation can be achieved by allocating sequentially from a large contiguous memory area, the *allocation space*. We only need to keep two pointers, one to the next free address, the *allocation pointer*, and one to the last usable address, the *limit pointer*. Before the allocation of a new object, if the allocation pointer plus the size of the object is greater than the limit pointer, a garbage collection must be performed to reclaim the space of objects that are no longer needed. Otherwise, the object is allocated and the allocation pointer incremented by the size of the object. Efficient deallocation can be achieved with the use of a generational garbage collector. Early versions of SML/NJ used a simple two-generational garbage collector. Future releases will use a multi-generational collector, which we have measured in this work.

It might seem that allocating closures sequentially on the heap would be terrible for cache performance. After all, stacks provide a simple and effective way of recycling memory, and have very good spatial locality. With sequential heap allocation, on the other hand, the first allocation in each cache block can be a cache miss (*allocation miss*), which could result in bad cache performance.

We have comprehensively studied the cache performance of a set of ML programs. We consider the effects of varying both software and hardware (cache) parameters. Our main contributions are the following:

- We examine in detail the memory reference patterns of a set of ML programs. We find that more than half of the read references are to objects (typically closures) that have just been allocated. This means that heap allocated closures can have good locality; on suitable cache architectures, stack allocation should not be preferred over heap allocation *just for reasons of locality.* Diwan et al. [10] suggest that ML programs tend to read objects soon after they have been written, based on indirect evidence from cache behavior on different architectures, and we have confirmed this by direct experiment.

- Varying minor garbage collection frequency (i.e., changing the size of allocation space) may have an impact on cache performance. In the case of caches without an allocation miss penalty, the impact is very small and is largely offset by changes in garbage collection overhead. On caches with a penalty for allocation misses, however, making the allocation space fit in the cache can result in a significant improvement in performance. We show quantitatively for what cache sizes and miss penalties fitting the allocation space in the cache improves performance.

- We report a comprehensive set of miss ratios, for varying cache sizes, block sizes, associativity, and write miss policies. We find that ML programs can have a very good cache performance on some architectures, comparable to or even better than that of the C and Fortran SPEC92 benchmarks. We also found that allocation misses depend mostly on block size, and if blocks are not too small (at least 32 bytes), even on architectures that do not eliminate allocation misses, ML programs can have much lower miss ratios than the C and Fortran programs.

- On the DEC3000 Alpha workstation, we find that fitting the allocation space in the secondary cache can result in significant performance improvement, even though the machine has a deep write buffer that should eliminate penalties for allocation misses. The ineffectiveness of the write buffer is related to the fact that so many reads are to recently allocated objects: some of those reads cause the write buffer to be flushed prematurely.

- The frequency of major collections and the number of active generations can have a significant impact on cache performance. We found that a large number of active generations can increase the number of conflict misses.

The problem of the cache performance of programs with sequential allocation has been addressed previously. Peng and Sohi [18] studied the cache performance of a set of Lisp programs. They show that conventional cache memories were inadequate for these programs and proposed the use of an *allocate* instruction to eliminate allocation misses. They also show that the LRU replacement policy commonly used in associative caches is bad for this class of programs and proposed an alternative replacement policy. Finally they proposed the use of a control bit to avoid writing back cache blocks filled with garbage. Wilson et al.[26] and Zorn[27, 28] proposed fitting the allocation space in the cache as an alternative, software-based method for eliminating allocation misses on large caches. Wilson et al. also show that modestly set-associative caches can achieve a significant performance improvement over direct mapped caches. In essence, the main conclusion of these papers is that programs with dynamic heap allocation tend to have bad cache performance—and either hardware techniques, or software techniques, or a combination of both, must be used in order to improve cache performance.

More recent work, by Diwan et al. [10] and Reinhold [20, 21] has shown that some cache design features, already available on current machines, can eliminate all of the allocation misses. Moreover, Reinhold shows that sequential allocation, due to the fact that it tends to spread memory references uniformly across memory, is naturally suited to direct-mapped caches.

Following Jouppi [17] we classify cache architectures as follows. A *fetch-on-write* cache is one that allocates a block and fetches the contents of the block from memory on a write miss. A *write-validate* cache allocates a block, but does not fetch the contents from memory on a write miss (and thus, must mark each word of every cache line as valid or invalid). And a *write-around* cache bypasses the cache entirely on a write miss.

## 2 The SML/NJ Runtime System

The SML/NJ runtime system uses a generational garbage collector. The two-generation collector used up to version 0.93 proved inadequate on long-running programs, so John Reppy has implemented a multi-generation collector[22]. Our measurements reported here are of an early prototype of Reppy's collector, which will be distributed with future versions of SML/NJ.

The heap is divided into an *allocation space* and from one to seven *generations*. The generations are named from one to seven, and each higher numbered generation contains objects that are older (i.e., have been allocated at an earlier time) than the objects in the previous generations. In addition, within each generation, objects are divided into two age groups.

Most objects (closures and user data structures) are allocated in the allocation space. The only exception is that large objects (more than 512 words) are allocated in the first generation. Each generation (but not the allocation space) is divided into five *arenas*, one for each class of objects: *records, pairs, arrays, strings,* and *code*. All objects are preceded by a one-word descriptor (tag), except pairs copied into the first or older generations. Removing the tag from pairs not only saves space, but should improve cache performance, since more pairs can fit into a single block, and pairs can be aligned to block boundaries.

There are two types of garbage collections, minor and major. Minor collections are very frequent but brief, and copy live objects from the allocation space to the first generations. Major collections are longer, but much less frequent, and collect from the older generations. A $k$th-generation major collection promotes the older objects of each generation $i$ into generation $i+1$, for $1 \leq i \leq k$; a fixed number of $k$th-generation collections occur for each $(k+1)th$-generation collection.

The early prototype we measured lacks certain features of the collector Reppy describes [22], particularly the use of a mark-and-sweep collector for machine code and other large objects. This change is likely to decrease major garbage collection overhead (from the amount we measured) on some of the benchmark programs, particularly when a small allocation space is used.

## 3 Methodology

We used a benchmark suite of ten ML programs, which were compiled and run in SML/NJ, version 0.93, with the new runtime system and generational collector. We compiled each program with the default parameters of the compiler, and saved a heap image of the system. We then used a MIPS instruction-level simulator, written by E. Gün Sirer, to simulate execution of the entire SML/NJ system—including the runtime system, which loads the heap image and executes the programs. We have extended the MIPS simulator with a cache simulator, so that cache misses are determined on the fly, with no need to generate reference traces. Our simulation results do not include operating system data.

We also ran some experiments on an DEC3000 Alpha workstation. For these experiments we used SML/NJ version 1.05a, since version 0.93 was not ported to the Alpha architecture. The main difference in version 1.05 from version 0.93 is the use of more efficient closure representations[23],

Table 1: General information about the benchmark programs.

| Program | Description |
|---|---|
| Barnes-Hut | The Barnes-Hut N-body simulation program [7, 6], translated into ML by John H. Reppy. |
| Boyer | An ML implementation of the Boyer benchmark [13]. |
| Knuth-Bendix | An implementation by Gerard Huet of the Knuth-Bendix completion algorithm translated into ML by Xavier Leroy. |
| Lexgen | A lexical-analyzer generator, written by James S. Mattson and David R. Tarditi[5], processing the lexical description of Standard ML. |
| Life | Reade's implementation of the game of Life [19]. |
| Mandelbrot | A program to generate Mandelbrot sets. |
| MLYACC | A LALR(1) parser generator, written by David Tarditi [25], processing the grammar of Standard ML. |
| Ray | A ray tracer, written by Don Mitchell, and translated into ML by John H. Reppy. |
| Simple | A spherical fluid-dynamics program [8, 12], translated into ML by Lal George. |
| VLIW | A VLIW instruction scheduler written by John Danskin. |

Table 2: Total, user, garbage collection, and system times in seconds for the benchmark programs, measured on a DEC5000/240 workstation, with an allocation space of 512K bytes.

| Program | Time (sec.) | | | |
|---|---|---|---|---|
| | Total | User | GC | Sys. |
| Barnes-Hut | 26.41 | 24.96 | 1.46 | 0.42 |
| Boyer | 2.14 | 1.32 | 0.82 | 0.18 |
| Knuth-Bendix | 12.49 | 9.25 | 3.24 | 0.70 |
| Lexgen | 10.87 | 9.97 | 0.89 | 0.25 |
| Life | 11.66 | 11.50 | 0.16 | 0.05 |
| Mandelbrot | 11.15 | 11.12 | 0.04 | 0.04 |
| MLYacc | 4.50 | 3.52 | 0.98 | 0.33 |
| Ray | 22.34 | 22.21 | 0.12 | 0.71 |
| Simple | 37.69 | 29.14 | 8.55 | 1.39 |
| VLIW | 21.17 | 20.43 | 0.75 | 0.41 |

which reduces the amount of closure allocation. Version 1.05a also uses the newest version of the multigenerational collector, which uses a mark-and-sweep collector for large objects.

The benchmark programs are described in Table 1. Some or all of these programs have been used as benchmark programs in other works on the performance of SML/NJ programs [2, 23, 10, 24]. Each of these programs is run in its entirety by the simulator. Four of the programs—Barnes-Hut, Mandelbrot, Ray, and Simple—use floating-point intensively; the other six use only integer instructions.

Table 2 shows the run time of the programs, measured on a DECstation 5000/240, and time breakdowns by user code, garbage collection (GC) time, and system time. We ran each program a number of times, and took the run with minimum total time (as recommended by the SPEC consortium [14]). Table 3 shows instruction counts and number of reads and writes for user and garbage collection code. Garbage collection time and instruction counts can vary considerably if the size of the allocation space is changed. User times can also vary, as the size of the allocation space may have an impact on cache performance. User code instruction counts are not affected by the size of the allocation space. The data shown in these tables are for a 512-Kbyte allocation space.

Table 3: Instruction counts, reads and writes for user code and garbage collection. Garbage collection counts are for an allocation space of 512K bytes. All numbers are in millions.

| Program | User | | | GC | | |
|---|---|---|---|---|---|---|
| | instr. | reads | writes | instr. | reads | writes |
| Barnes-Hut | 617.5 | 179.8 | 125.5 | 39.7 | 6.1 | 4.7 |
| Boyer | 42.9 | 9.5 | 9.0 | 27.1 | 5.0 | 3.7 |
| Knuth-Bend. | 281.0 | 57.2 | 67.0 | 73.8 | 14.2 | 10.2 |
| Lexgen | 296.0 | 55.4 | 38.2 | 25.6 | 4.7 | 3.3 |
| Life | 422.9 | 50.0 | 38.3 | 4.9 | 0.9 | 0.6 |
| Mandelbrot | 367.6 | 82.7 | 53.1 | 1.2 | 0.2 | 0.1 |
| MLYACC | 97.8 | 21.3 | 18.7 | 32.1 | 5.8 | 4.3 |
| Ray | 679.2 | 192.4 | 141.3 | 2.7 | 0.5 | 0.3 |
| Simple | 764.3 | 174.3 | 130.3 | 232.0 | 38.0 | 27.5 |
| VLIW | 427.2 | 82.1 | 73.1 | 19.7 | 3.3 | 2.2 |
| Total | 3664.6 | 877.0 | 694.4 | 458.8 | 78.7 | 56.9 |

### Performance Model

We use a simple model to predict the performance of the programs under different cache organizations. We assume that the machine can issue one instruction per cycle, with no pipeline stalls, except for memory accesses. Loads or stores that hit the cache take one cycle, and cache misses take a fixed number of cycles. We allow for different read and write miss penalties. We assume that a cache miss stalls the instruction pipeline. The total number of cycles of a program is given by:

$$\begin{aligned} TotalCycles &= IC + IMiss * R_{penalty} + \\ & \quad RMiss * R_{\mathrm{penalty}} + WMiss * W_{\mathrm{penalty}} \end{aligned}$$

where $IC$ is the instruction count, $IMiss$ the number of instruction fetch cache misses, $RMiss$ the number of read misses, $WMiss$ the number of write misses, $R_{\mathrm{penalty}}$ the penalty for read misses in cycles, and $W_{\mathrm{penalty}}$ the penalty for write misses in cycles.

Table 4 compares the run time of each program, measured on a DEC5000/240 workstation, with the time predicted by the model. Most of the error probably comes from the assumption that each instruction takes one cycle. The program with highest relative error, VLIW, executes many integer division operations, which take several cycles to complete. Barnes-Hut, Mandelbrot and Simple execute many floating point instructions, which may take many cycles to complete.

Table 4: Measured time on a DECstation 5000/240 compared to the time predicted by the simple machine model. All times are in seconds. The size of the allocation space is 512K bytes. The value on the column labeled *Difference* is given by $((Measured\ time/Predicted\ time) - 1) * 100$.

| Program | Measured time | Predicted time | Difference |
|---|---|---|---|
| Barnes-Hut | 26.41 | 21.54 | 23% |
| Boyer | 2.14 | 2.17 | -1% |
| Knuth-Bendix | 12.49 | 10.63 | 17% |
| Lexgen | 10.87 | 10.65 | 2% |
| Life | 11.66 | 11.35 | 3% |
| Mandelbrot | 11.15 | 9.33 | 20% |
| MLYacc | 4.50 | 4.18 | 8% |
| Ray | 22.34 | 22.72 | -2% |
| Simple | 37.69 | 32.76 | 15% |
| VLIW | 21.17 | 15.72 | 35% |

Table 5: Total allocation (in words), allocation rate (total allocation divided by number of user instructions), and distribution of allocation and non-allocation writes for the benchmark programs.

| Program | Allocation writes | Allocation rate | Fraction of alloc. writes |
|---|---|---|---|
| Barnes-Hut | 124,368,910 | 0.201 | 99.07% |
| Boyer | 8,802,524 | 0.205 | 98.25% |
| Knuth-Bendix | 66,837,173 | 0.238 | 99.74% |
| Lexgen | 33,024,196 | 0.112 | 86.52% |
| Life | 38,104,645 | 0.090 | 99.59% |
| Mandelbrot | 48,712,383 | 0.133 | 91.81% |
| MLYACC | 17,611,127 | 0.180 | 94.22% |
| Simple | 129,215,509 | 0.169 | 99.18% |
| Ray | 139,506,521 | 0.205 | 98.72% |
| VLIW | 67,076,362 | 0.157 | 91.75% |
| Average | | 0.169 | 95.885% |

Because we are interested in the performance of the programs with different settings for garbage collection parameters, which may affect the instruction count of the programs, we also use a performance metric that takes this change into account. This metric is the number of cycles per user program instruction, *CPUI*. The number of user program instructions is the total number of instructions minus garbage collection instructions. That is, we charge garbage collection and cache miss cycles to user program instructions. CPUI provides a simple way of evaluating the total cost of cache and garbage collection overhead. We define CPUI as:

$$CPUI = \frac{\text{TotalCycles}}{\text{number of user instructions}}$$

## 4 Memory Reference Patterns

### 4.1 Write References

We can divide writes into two groups: allocation writes and non-allocation writes. Most writes from all programs are for allocation. Table 5 shows the number of allocated words for each program, the allocation rate, and the fraction of all writes that are for allocation (these numbers do not include garbage collection writes). The allocation rate is the number of words allocated divided by the number of user instructions. All programs allocate at a very fast rate, about one word for each six user program instructions. On average for the 10 programs, approximately 96% of all writes are allocation writes. User-code non-allocation writes update user data structures; but most non-allocation writes are performed by runtime system.

Table 6: Allocation profile for closures and user data.

| Program | Closures | | User data | |
|---|---|---|---|---|
| | Fraction of allocation | Avg. size | Fraction of allocation | Avg. size |
| Barnes-Hut | 38.12% | 8.65 | 61.88% | 3.24 |
| Boyer | 73.38% | 7.01 | 26.62% | 3.30 |
| Knuth-Bendix | 86.59% | 5.36 | 13.41% | 3.07 |
| Lexgen | 54.75% | 7.94 | 45.25% | 3.98 |
| Life | 43.23% | 6.03 | 56.77% | 3.97 |
| Mandelbrot | 60.30% | 6.70 | 39.70% | 3.02 |
| MLYACC | 60.09% | 9.15 | 39.91% | 3.76 |
| Ray | 30.07% | 5.71 | 69.93% | 3.79 |
| Simple | 51.67% | 6.03 | 48.33% | 3.24 |
| VLIW | 64.36% | 6.97 | 35.64% | 3.32 |
| Average | 56.26% | 6.96 | 43.74% | 3.47 |

Allocation writes are sequential, and there is a cyclic pattern with a period that is equal to the size of the allocation space. Figure 1 shows this cyclic pattern for the program Lexgen, with an allocation space of 512K bytes. The graph plots the address written, modulo 512K, versus time, where time is measured in words written. The plot shows the write patterns of Lexgen from time 2000K to 5000K. We can see clearly each allocation cycle, followed by a vertical line, which is a minor garbage collection. One of the minor collections, near time 2500K, is followed by a major collection, which appears as the unusual pattern in the graph. The horizontal lines are non-allocation writes. For this program, most non-allocation writes are concentrated on a few locations that are updated more or less uniformly throughout the program execution. The difference in the density of the lines means that some locations are written more frequently than others. We can see that one of the horizontal lines (the second from top) seems to move down after the major collection. This is probably because the collector relocates the object that is being updated. As the numbers in Table 5 show, Lexgen is the program with the most non-allocation writes. The horizontal lines that appear in Figure 1 do not appear on the patterns for other programs.

Objects allocated by each program can be divided into two groups: closures and user data structures. Table 6 shows the distribution of allocation by closures and user data structures, and the average sizes of objects in each group. Closures account for the greatest fraction of all words allocated, 56% on average for the benchmark programs. They are also larger, approximately seven words on average. User data, on the other hand, are much smaller in size, about 3.5 words on average. Most objects tend to have a very short life. Only a small fraction of objects survive a garbage collection, and most surviving objects are user data structures[15, 24]. All objects are preceded by a one-word tag, which is included in the size of the objects. In most cases, tags are only used by the garbage collector, and since most objects do not survive

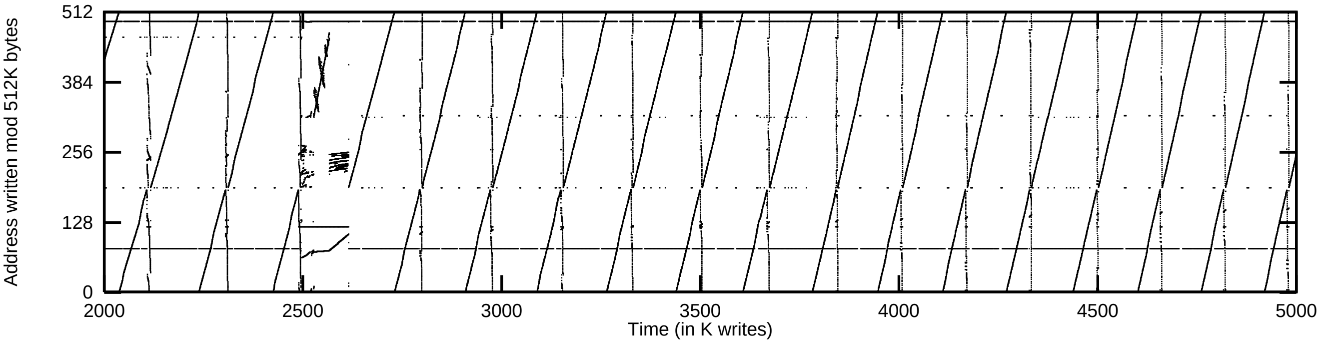


Figure 1: Write reference patterns for Lexgen.

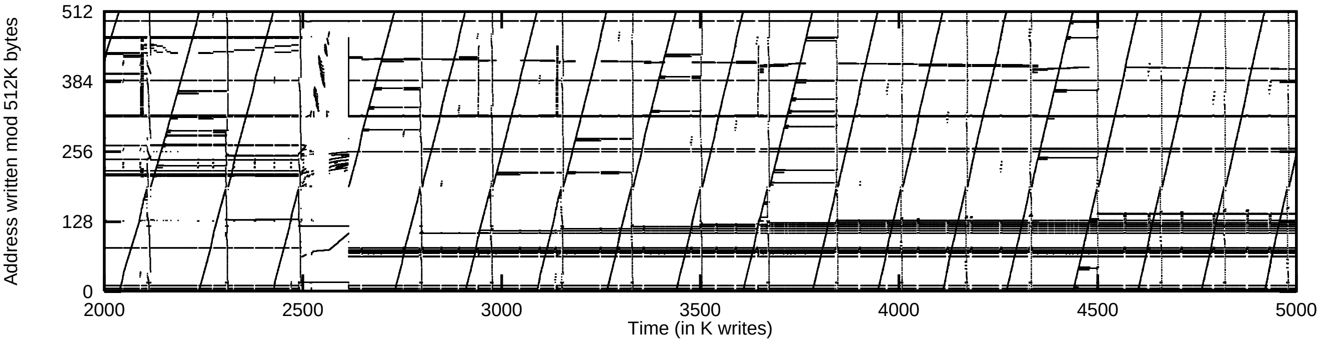


Figure 2: Read reference patterns for Lexgen.

Table 7: Read references by class.

| Program | Closure/ User data | Compile time/ runtime sys. | Runtime system |
|---|---|---|---|
| Barnes-Hut | 82.89 | 14.10 | 3.01 |
| Boyer | 64.53 | 33.90 | 1.57 |
| Knuth-Bendix | 95.64 | 4.08 | 0.28 |
| Lexgen | 80.82 | 7.54 | 11.64 |
| Life | 88.02 | 11.69 | 0.29 |
| Mandelbrot | 61.52 | 38.31 | 0.17 |
| MLYACC | 84.39 | 11.75 | 3.86 |
| Ray | 73.61 | 23.39 | 3.00 |
| Simple | 73.37 | 26.39 | 0.24 |
| VLIW | 79.09 | 15.85 | 5.06 |

to a garbage collection, most tags are never used. Tags account for approximately twenty percent of words allocated. Most objects are allocated in the allocation space. Large objects (more than 2K bytes) are allocated in the first generation, but they account for a negligible fraction of the allocation of the programs in the benchmark suite.

### 4.2 Read References

Non-garbage-collection reads are to closures, user data structures, user constants, and runtime system objects. Closures and user data structures are allocated on the allocation space. If they are still "alive" at the end of an allocation cycle, they are copied to the first generation, and they can be promoted to older generations after major collections. Floating-point and string literals reside in machine-code objects in older generations. Runtime system objects are located mostly outside the ML heap, but a few may be located in older generations. Garbage collection references are to objects in the heap that are being traced and collected, or to garbage collection data structures outside the ML heap.

Table 7 shows the distribution of reads by runtime allocated data (closures and user data structures), compile time allocated data (constants and old heap-allocated runtime-system objects), and references to locations outside the ML heap (static runtime-system objects). Most references are to closures and user data, but there is also a significant number of references to compile-time-allocated data. These numbers do not include garbage collection references.

References to closures and user data can potentially be spread across large areas of memory. The allocation space and the older generations can be several megabytes in size. But, in fact, the majority of these references do have some form of locality. This can be seen on Figure 3. The graphs show the cumulative distribution of reads by age. Here, we use allocation to measure time. The age of the target of a read reference is given by the number of words allocated since its allocation. For objects in the allocation space, the age is given by the distance of the object to the allocation pointer. The graphs show that about 60% of the references are to objects which are only 256 words (1K bytes) from the allocation pointer. Most of these references should be to closures. Most of the references that are farther then 256 words from the allocation pointer should be to user data structures. The remainder, the difference between 1 and the maximum of each curve, are references to constants and user data structures. Garbage collection references are not

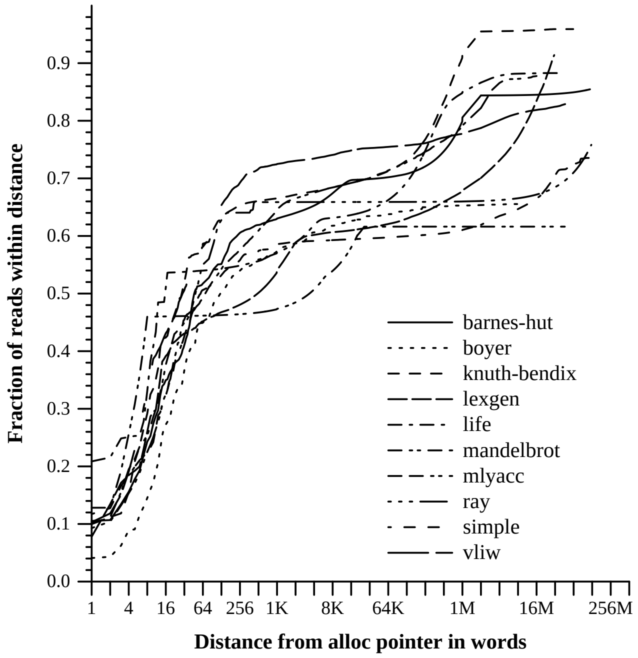


Figure 3: Cumulative distribution of reads as a function of the distance of the address referenced from the allocation pointer.

shown in this graph.

Read references also show a cyclic pattern, following the cyclic allocation pattern. Figure 2 shows a plot of address read (modulo 512K) versus time (measured in words written) for Lexgen. The graph shows a cyclic pattern similar to the one that can be seen in Figure 1. This pattern corresponds to closure reads. For this program, most other reads tend to be concentrated within a relatively small number of locations, which cause the horizontal stripes on the graph.

In summary, all programs in the benchmark suite have a very regular, cyclic write pattern. More than half of reads are to objects that have just been allocated.

## 5 Miss Ratios

In this section we report miss ratios for the programs in the benchmark suite. We vary cache size from 4K to 4M bytes, block size from 16 bytes to 256 bytes, and associativity from direct-mapped to 16-way associative. We report read and write miss ratios separately. Due to space limitations, we report only the arithmetic mean of miss ratios of the 10 benchmark programs. The complete set of miss ratios is available in [15].

Figure 4 shows read and write miss ratios for direct-mapped caches, with varying block sizes. The allocation space is 512K bytes, which explains the sharp drop in write miss ratios for caches of 512K bytes or more. When the allocation space fits in the cache, there are no allocation misses, except when there are conflicts between blocks in the allocation space and older generations, which are not too common. When the allocation space is larger than the cache, the write miss ratios depend mostly on block size, being approximately $1/block\_size\_in\_words$. The actual miss ratios tend to decrease slightly as cache size increases, because some blocks may be read again after they are evicted from the cache by the allocation cycle. For large blocks and small caches, the write miss rate is higher than expected, because of conflict misses. For reads, the lower miss ratios are seen for 32 and 64-byte blocks, on caches up to 128K bytes. For bigger caches, all block sizes from 32 to 256 bytes have similar miss ratios. When read and write miss ratios are combined (Figure 5), blocks of at least 64 bytes tend to have significantly lower miss ratios.

Figure 5 also compares the average miss ratios of the ML programs with those of the SPEC92 benchmarks[14]. In

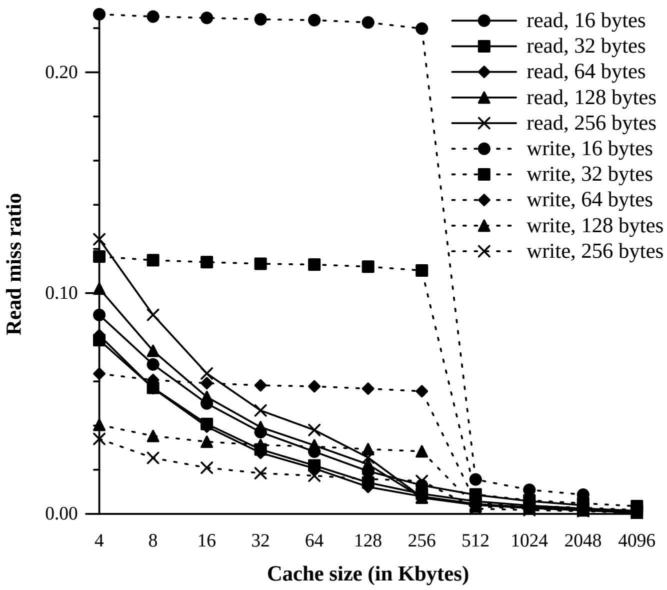


Figure 4: Read and write miss ratios on direct mapped *fetch-on-write* caches, for different block sizes. The allocation space is 512K bytes.

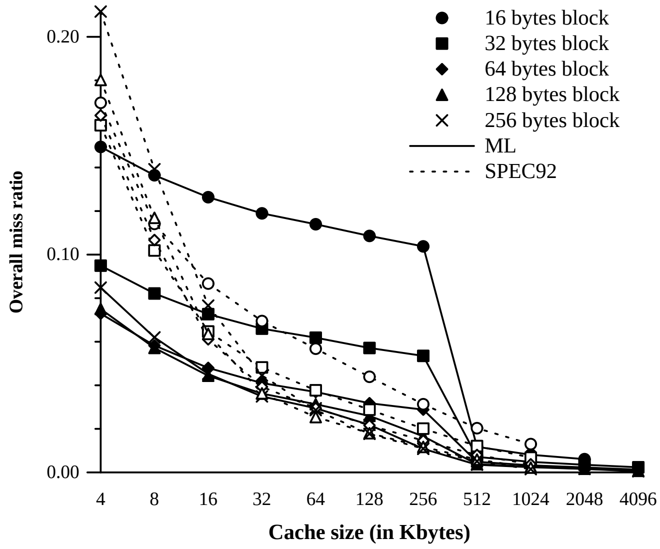


Figure 5: Overall miss ratios on direct mapped *fetch-on-write* caches, for different block sizes, for the ML programs and SPEC92 benchmarks. The allocation space for the ML programs is 512K bytes.

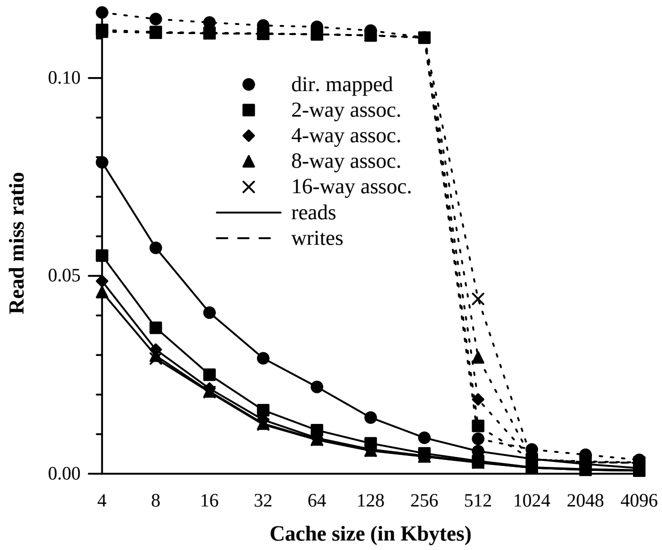


Figure 6: Read and write miss ratios for direct mapped and set-associative caches. Allocation space size is 512K bytes.

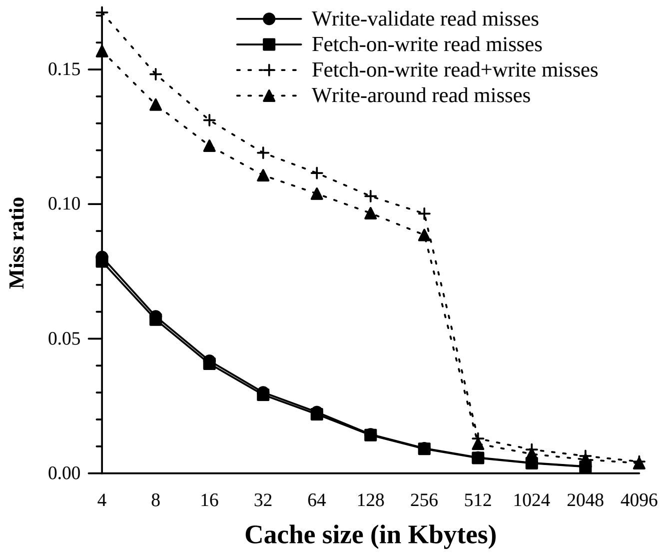


Figure 7: Miss ratios on write-validate and write-around caches, compared to the read miss ratios of *fetch-on-write* caches and with the sum of miss ratios of fetch-on-write caches when the write misses are counted as read misses.

many cases the ratios for the ML programs are lower than those of the SPEC programs. In particular, on very small caches (4K and 8K bytes), and 32-byte or bigger blocks, the miss ratios of the ML programs are significantly lower than those of the SPEC92 benchmarks. On very small caches, the ML programs suffer many allocation misses, but since they have very good locality (i.e., many references to addresses close to the allocation pointer), there are fewer of other types of misses. As cache size increases, since the number of allocation misses remains almost constant, the miss ratios of SPEC programs become lower. Even if we take into account the higher fraction of memory references for the ML programs, we can still say that they have better cache performance than the SPEC programs. For an 8-Kbyte cache, with 32 byte blocks and a 10 cycle miss penalty, and assuming one instruction per cycle, the SPEC programs spend an average of 35% of their time on cache misses, compared with 32% for the ML programs. For 64-byte block caches, the cache overheads are 36% for SPEC and 23% for ML.

Figure 6 shows the miss rates for direct-mapped and set-associative caches, with 32-byte blocks. The size of the allocation space is 512K bytes. Associativity improves miss ratios significantly, since it eliminates many conflict misses between objects in the allocation space and older generations. Four-way associativity seems to be good enough, however. There is only a very small decrease in miss ratios for 8-way and 16-way associative caches.

As expected, write miss ratios fall sharply when the allocation space fits in the cache. Associativity has unusual results for the case where the cache matches allocation space size (512K bytes). At this point there is an inversion, with higher miss ratios for higher associativity. Wilson et al. observed something similar [26] in their study of Lisp programs. The problem is that the LRU replacement policy of the associative caches is not good for cyclic reference patterns. For a 512K-byte cache there should be no allocation misses, because the allocation space is also 512K bytes. But there are many references to objects in the older generation and these references will cause some blocks to be removed from the cache. In this case, the LRU replacement policy would pick the block that is to be written next as the victim, which is obviously not a good idea. But why does this problem happen only for a cache that is the same size as the allocation space? The LRU replacement policy is actually good for objects from the older generations. These objects are likely to have a very long life, and be referenced many times. Thus the LRU policy would tend to keep these objects in the cache, which would reduce the number of read misses. For the caches that are larger than the allocation space, there is plenty of room in the cache to hold many objects from the older generation without evicting blocks from the allocation space. And for caches that are smaller than the allocation space, the write to the first word of each block will cause a miss in both the direct mapped and set-associative caches.

In summary, write miss ratios depend mostly on block size, and can be very high for very small blocks. If block sizes are not too small, miss ratios of ML programs can be lower than that of SPEC programs.

## 5.1 Write-validate and write-around caches

Write-validate caches always, and write-around caches usually, outperform fetch-on-write caches [17]. Write-validate and write-around avoid fetching data on write misses, but they may have to fetch data later on a read. In other words, write-validate and write-around do not have write misses, but they may have more read misses, although they never have more misses than the sum of read and write misses of a fetch-on-write cache. Figure 7 compares the miss ratios of the ML programs on write-validate and write-around caches to the miss ratios on fetch-on-write caches. This graph shows the read miss ratios of fetch-on-write, write-validate and write-around caches. The curve labeled "fetch-on-write read+write misses" is the sum of read and write misses on a fetch-on-write cache divided by the number of reads. This curve is shown to give an idea of the fraction of misses that are eliminated by write-validate and write-around caches.

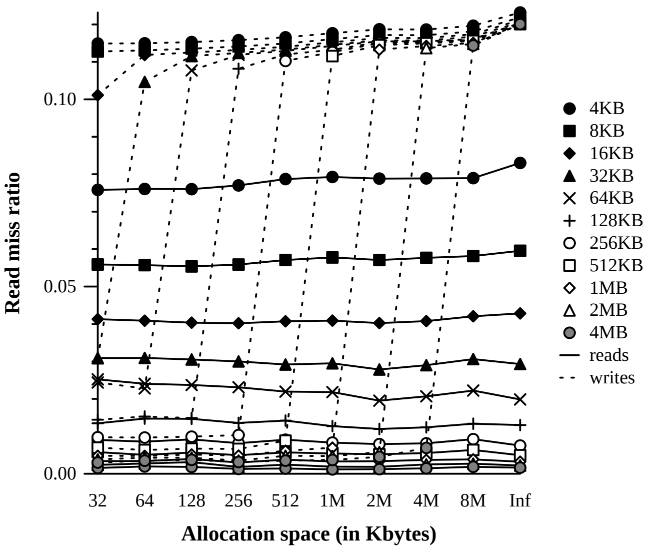


Figure 8: Effects of varying allocation space size on read and write miss ratios, for direct mapped caches from 4K to 4M bytes, 32-byte blocks.

As shown by Diwan el al.[10], write-validate is the best organization for ML programs. It eliminates all of the write misses without adding many read misses, as can been seem from the comparison with the read miss ratio of fetch-on-write caches. On the write-around caches, on the other hand, most write misses of the fetch-on-write cache simply become read misses. This is because most objects are read soon after they are allocated, and, the reads will cause a miss if the writes bypass the cache. So the miss ratio of write-around is only a little better than that of fetch-on-write.

Part of the difference between write-around and fetch-on-write misses comes from garbage collection. Since many objects are not read right after they are collected, bypassing the cache may save some misses. Some user data structures may also not be read soon after being allocated, and some misses may also be saved in this case.

## 6 Effect of garbage collection frequency

We varied the size of the allocation space from 32K bytes to "infinite" (i.e., the allocation space can hold all data allocated by the programs, with no need for garbage collections.) A small allocation space means very frequent garbage collections. As the size of the allocation space increases, the frequency of collections decreases. We consider the effects on direct mapped and 2-way associative caches. Figure 8 shows how miss ratios vary with the size of the allocation space for direct mapped caches, with 32-byte blocks. The size of the allocation space has a significant impact on write misses, but almost no effect on read misses. On small caches, there is a tendency for a slight increase in read miss ratios as the frequency of collections decrease, but this is not sufficient to compensate for the extra garbage collection overhead. There is a slightly higher increase in read miss ratios on small 2-way associative caches, but still not high enough to justify the use of an aggressive garbage collection policy.

The most unusual result was found for the program Sim-

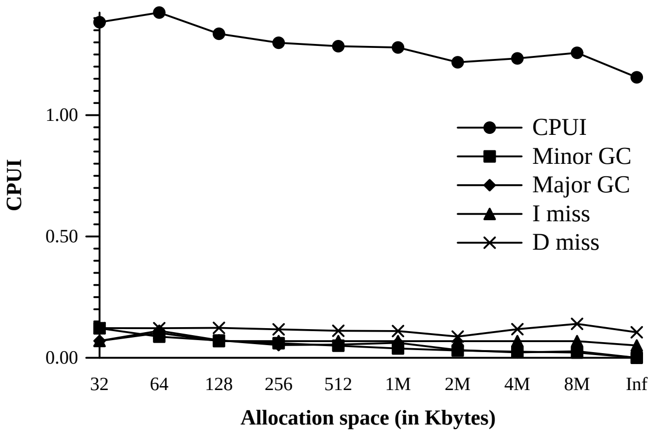


Figure 9: Cycles per user instructions for lexgen, on write-validate cache. Cache size is 64K bytes, 20 cycles read miss penalty.

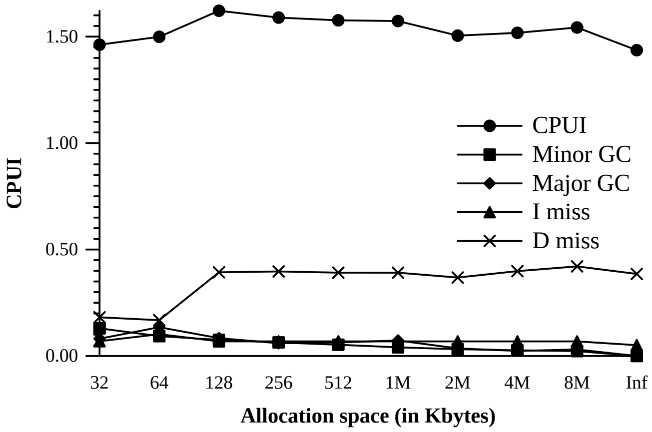


Figure 10: Cycles per user instructions for lexgen, on fetch-on-write cache. Cache size is 64K bytes, 20 cycles read/write miss penalty.

ple. Frequent collections increase miss ratios for this program significantly, especially on direct mapped caches. We found that the problem is that the program has a large working set, and doing frequent collections causes objects to be spread among many generations. These may cause many conflict misses, since it is possible that different generations may be aligned to the same cache blocks. When we ran the program with a large first generation, so that there is no need for major collections, we noticed a significant decrease in miss ratios. Moreover, the miss ratios become insensitive to frequency of minor collections, much like the other programs. So the problem here is frequency of major collections, and not frequency of minor collections.

Figure 9 shows CPUI for Lexgen on a 64-Kbyte direct-mapped write-validate cache[1] with a 20 cycle miss penalty. CPUI decreases with the increase in allocation-space size because of the decrease in garbage collection overhead. But when the allocation space becomes very large (4M and 8M bytes), CPUI begins to increase again, due to an increase

[1] In fact, the data is for a fetch-on-write cache with 0-cycle write penalty. As shown earlier, this should be a very good approximation of a write-validate cache.

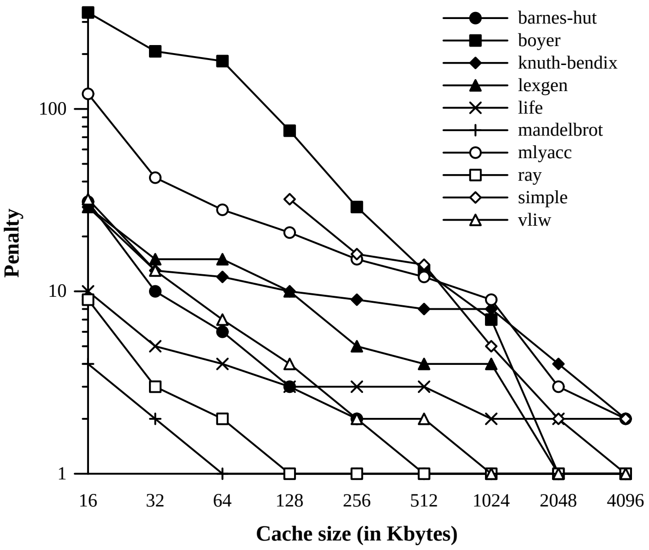


Figure 11: The minimum penalty for which it is better to fit the allocation space in the cache, instead of using a large allocation space of 8M bytes. Assuming separate I & D caches, fetch-on-write data caches, and the same penalty for reads and writes.

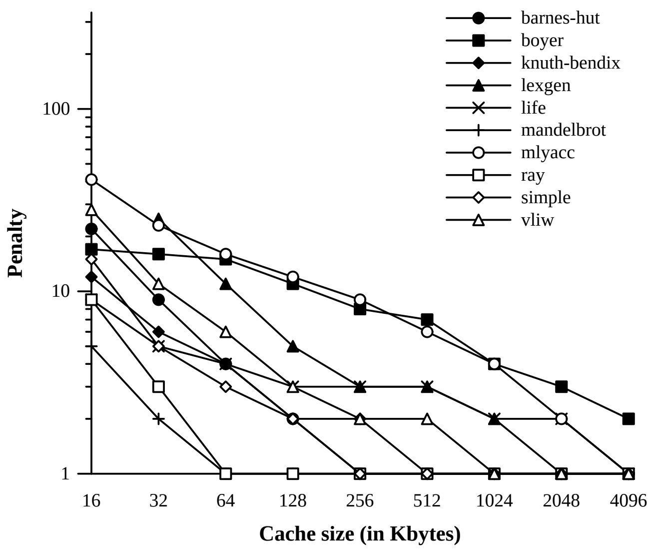


Figure 12: Same as Figure 11, but with the use of a very large first generation, so that there are no major collections.

in cache misses. Most of those misses are not caused by user data, since they do not appear in the case of an infinite allocation space. These misses occur on garbage collections, due to the increased volume of data copied per collection.

The situation is different for a fetch-on-write cache. Figure 10 shows CPUI for 64K bytes fetch-on-write cache, with 20 cycle miss penalty. Using an allocation space that fits in the cache gives better performance, because for this program, and this cache organization, the number of cycles saved by the elimination of allocation misses is offset by the extra garbage collection cycles.

In conclusion, the size of the allocation space, which determines garbage collection frequency, can have a significant impact on cache performance of caches that have allocation-miss penalty, such as fetch-on-write and write-around. The size of the allocation space has almost no effect on the performance of ML programs on write-validate caches.

### 6.1 Adapting to caches with allocation misses

Ideally, we would like to be able to run programs with intensive allocation on machines with caches that have no penalty for allocation writes. But there are many machines that have a cache miss penalty for allocation, and for those machines, it may be possible to eliminate allocation misses by fitting the allocation space in the cache. But doing this can increase garbage collection overhead, which may offset gains from eliminating cache misses. Whether it is good to fit the allocation space in the cache depends on the cache size, miss penalty, and garbage collection overhead of each program.

We performed the analysis for our benchmark programs. Figure 11 shows the results. Each curve plots the minimum miss penalty for each cache size for which fitting the allocation space in the cache gives better performance than using a very large allocation space (8M bytes). For a 512K-byte cache, for example, it is better to keep the allocation space in the cache if the penalty is a least 9 cycles. In most cases, a 16K-byte cache would be too small to justify fitting the allocation space in the cache, except if the penalty is very high. But for some programs it may be a good idea to do so for 32K-byte caches.[2]

One problem with the Reppy's generational collector is that the sizes of older generations depend on the size of the allocation space. This may not be a good idea when we want to reduce the size of the allocation space so that it fits in the cache. We may want to reduce the size of the allocation space, and increase the size of older generations, so as to compensate for the increase in minor collector overhead with a decrease (or at least a non-increase) in major collection overhead. We did the same experiments as above using a very large first generation, so that the programs run with no major collections. The results are shown in Figure 12. Because in many of the programs most garbage collection overhead comes from major collections, in most cases it is better to keep the allocation space inside the cache.

We conclude that reducing the size the allocation space to fit in large and medium-sized fetch-on-write or write-around caches can improve performance of most programs, even though it may increase garbage collection overhead. Reducing major collection frequency to compensate for the increase in minor collections can improve performance even more.

## 7 A case study: the DEC3000 Alpha workstation

The DEC Alpha 21064 microprocessor has 8-Kbyte on chip instruction and data caches, and support for a second level cache. The first level data cache is write-around, and write misses are non-blocking. There is a four-entry write buffer, with one 32-byte block per entry, and writes that miss the first level cache go to the write buffer. The second level cache is write-allocate (fetch-on-write), but if a full block is in the

[2] This figure assumes 32-byte blocks. The graph should be useful for $b$-byte blocks, as long as one first multiplies the miss penalty by $32/b$.

Table 8: User plus garbage collection time (in seconds) on an Alpha workstation, for different sizes of the allocation space. The version of the compiler used is 1.05a.

| Program | Allocation space size | | | | |
|---|---|---|---|---|---|
| | 128k | 256k | 512k | 1024k | 2048k |
| Barnes-Hut | 13.52 | 13.35 | 13.55 | 16.42 | 16.45 |
| Boyer | 1.25 | 0.98 | 0.90 | 0.90 | 0.88 |
| Knuth-Bendix | 4.09 | 4.03 | 4.63 | 4.93 | 4.63 |
| Lexgen | 3.73 | 3.67 | 3.69 | 3.99 | 3.93 |
| Life | 0.54 | 0.53 | 0.53 | 0.57 | 0.56 |
| Mandelbrot | 5.19 | 5.13 | 5.08 | 7.11 | 7.11 |
| MLYACC | 1.89 | 1.89 | 1.78 | 1.82 | 1.80 |
| Ray | 15.69 | 15.07 | 15.93 | 19.80 | 20.32 |
| Simple | – | 8.51 | 8.50 | 10.16 | 9.97 |
| VLIW | 6.22 | 6.32 | 6.26 | 6.86 | 6.97 |

write buffer, there is no need to fetch the block from main memory since the entire block is going to be overwritten.

Since ML programs allocate sequentially at a very fast rate, we may expect that very few partially filled blocks will be written to the second level cache, and thus we would not have a (second-level cache-miss) penalty for allocation misses. We ran the benchmark programs on a DEC3000 Alpha workstation[11] with a 512K-byte second level cache, varying the allocation space from 256K to 2048K bytes. The results are shown in Table 8. Surprisingly, in most cases the best performance occurs when the allocation space is 512K bytes or less, which suggests that there is a penalty being paid for allocation misses.

It appears that there are frequent flushes of the write buffer before a full block is allocated. There are several reasons to flush the write buffer [9], but we identify the following as the most likely cause of the problem. The first level cache policy is read-allocate, so when there is a read miss, the block containing the address must be fetched. If the block is in the write buffer, but is partially filled, the write buffer must be flushed, and if the block is not in the secondary cache, it must be read from main memory so that the contents of the missing words can be filled in. As we saw earlier, most objects are read very shortly after being allocated. Thus it is likely that there will be many reads to partially filled blocks on the write buffer. Using the data in Figure 3 we calculate that, on average, approximately 8.4% of the reads are to blocks that should still be partially filled in the write buffer.

We also computed the probability of a write buffer being flushed because of a read to a partially filled block. For the benchmark programs, on average, 69% of the write buffers would be flushed. Considering a penalty of 27 cycles to fetch a block from memory[11], this should add a cost of 0.41 CPUI (cycles per user instruction).[3] Keeping the allocation space in the cache can eliminate the memory fetch penalty, but it increases garbage collection time. On average, the increased garbage-collection time causes a penalty of about 4% in performance.

In summary, because ML programs tend to have so many reads to objects that have just been allocated, the Alpha's write buffer must be flushed very frequently, and therefore it is not effective in eliminating allocation misses.

## 8 Simulating write-allocate

Write-allocate caches with no write-miss penalty perform well for fast-allocating programs. On machines with fetch-on-write (with a nontrivial write-miss penalty) or write-allocate (where write misses will inevitably be followed by expensive read misses), what strategy can the software use? As we have discussed above, avoiding misses by keeping the allocation space in cache is impractical for small caches (under 100 kbytes).

Some machines (such as the IBM R/S 6000 [16]) have an instruction to allocate (and zero) a specified cache line. If this is done in advance of writing to the line, no allocate miss will occur. The compiler can easily insert such instructions when incrementing the allocation pointer: whenever $ap \leftarrow ap + c$, allocate and zero the cache line surrounding address $ap + 100$. Then, when allocation reaches that line, it will already be in the cache.

On write-around machines without a cache-line-allocate instruction, it is always possible to allocate a specified cache line: just read from it! Reads always allocate in the cache. If, in addition, read misses do not stall the processor, then the compiler can simulate cache-line allocate: whenever incrementing $ap$, just read from address $ap+K$ into a dead register. If the program allocates (for example) 4 bytes for every 6 instructions executed, then $K = 100$ will allow 150 instructions to execute before the attempt to write into that line—enough time for the read miss to complete.

On the DEC Alpha 21064, even though cache read misses are "semi-stalling" (they lock up the address box for 5 cycles, but allow the processor to issue and complete arithmetic instructions), this technique still improves performance for SML/NJ [3].

It appears that many machines of the near future will have write-around caches with fully nonblocking read misses, so this technique should become more useful and important.

## 9 Full-line write-allocate

Write-validate (write-allocate with partial fill, but not fetch-on-write) gives much better performance than write-around for fast-allocating programs. Even for "conventional" C programs, it is well known [17] that write-validate gives significantly better performance than write-around. Why, then, do the Alpha and several other modern processors use write-around?

One important reason is that write-validate makes multiprocessor cache coherence much more difficult to achieve. If Processor $P$ writes word 1 of a cache line, and Processor $Q$ writes word 2, then (with write-around) an external cache-coherence controller can order the writes appropriately. But with write-validate, the cache-coherence controller must somehow merge the valid portions of lines in $P$'s and $Q$'s caches. This appears to be very nasty.

The problem of sequential consistency is not solved by write-around: $P$ writing to (and hitting) word 1 of a cache line, and reading back word 2, will lead to inconsistency unless the writes stall until complete. This stall is unaccept-

[3] For this calculations, it was assumed that writes are strictly sequential, and there is at most one entry in the buffer (it also ignores non-allocation writes). But, in fact, when an object is allocated its fields are initialized in reverse order, from last to first, which means that is possible that more than one entry in the buffer is partially filled. Therefore, the probability of a write buffer being flushed before an entry is complete can be even higher.

able; so the Alpha does not guarantee sequential consistency unless a "memory barrier" instruction is executed.

We propose a solution: a new cache write-miss policy called *full-line write-allocate* that interacts gracefully with cache coherence protocols (but does not "solve" the sequential consistency problem any more than the Alpha does).

In our scheme, any cache write miss goes into the write buffer and not into the cache. But when the write buffer accumulates a full cache line, that line is allocated into the cache. (If the cache is write-through, then the line is also written to the next level of the memory hierarchy.)

Cache read misses that hit in the write buffer should be satisfied from the write buffer, without flushing it.

For sequential writes (such as heap allocations), this is as good as write-validate. It doesn't require "valid bits" on each word of every line in the cache (as only full lines are put there). It doesn't pose problems for cache coherence protocols. It doesn't require much extra hardware, and it doesn't add latency to critical execution paths. But we don't have enough room to justify these claims here.

## 10 Conclusion

We have studied the cache performance of ML programs compiled by the SML/NJ compiler. These programs have a very regular, sequential pattern for write references, and more than half of references are to objects that have just been allocated, which means that the programs have good temporal and spatial locality. Confirming the results of previous studies, we found that ML programs have good cache performance on write-validate cache architectures. On other architectures, the cache performance of the ML programs can be bad. However, for the cache architectures of many current machines — i.e. a small on-chip first level cache and a large second level cache — the cache performance of ML programs can be better than that of C and Fortran programs. Garbage collection frequency has little impact on cache performance for write-validate caches, but for large or mid-sized fetch-on-write and write-validate caches, fitting the allocation space in the cache can improve performance, even though a penalty can be paid for extra garbage collection overhead. For small fetch-on-write and write-around nonblocking caches prefetching can be used to simulate write-allocate and reduce the penalty for allocation misses.


## Acknowledgements

We thank E. Gün Sirer for writing the MIPS instruction level simulator that we use in our experiments, and John Reppy for writing (and helping us use) the multi-generational collector. We thank Doug Clark and Anne Rogers for their helpful suggestions.

The authors were supported in part by National Science Foundation Grant CCR-9200790. Marcelo Gonçalves was also supported in part by CNPq, Brazil.


## References


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

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel. Emulating write-allocate on a no-write-allocate cache. Technical Report CS-TR-459-94, Princeton University, June 1994.

[4] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

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

[6] Joshua Barnes and Piet Hut. Error analysis of a tree code. *Astrophysical Journal Supplement*, 389(70).

[7] Joshua Barnes and Piet Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 446(324).

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

[9] Digital Equipment Corporation, Maynard, Massachussets. *DECchip 21064 — AA Microprocessor Hardware Reference Manual*, first edition, October 1992. Order number EC-N0079-72.

[10] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with copying garbage collection. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 1–13, January 1994.

[11] Todd A. Dutton, Daniel Eiref, Hugh R. Kurth, James J. Reisert, and Robin L. Stewart. The design of the DEC 3000 AXP systems, two high-performance workstations. 4(4), 1992.

[12] K. Ekanadham 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.

[13] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, MA, 1985.

[14] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 1993.

[15] Marcelo J. R. Gonçalves. *Cache Performance of Programs with Intensive Allocation and Generational Garbage Colection.* PhD thesis, Princeton University, Department of Computer Science, 1995. (In preparation).

[16] William R. Hardell, Dwain A. Hicks, Lawrence C. Howell, Warren E. Maule, Robert Montoye, and David P. Tuttle. Data cache and storage control units. In *IBM RISC System/6000 Technology*, pages 44–50. IBM, 1990.

[17] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201, San Diego, California, May 1993.

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

[19] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[20] Mark B. Reinhold. Cache performance of garbage-collected programming languages. Technical Report 581, MIT Laboratory for Computer Science, September 1993.

[21] Mark B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[22] John H. Reppy. A high-performance garbage collector for Standard ML. Technical report, AT&T Bell Laboratories, 1994.

[23] Zhong Shao and Andrew Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 1994.

[24] Darko Stefanovic and J. E. B. Moss. Characterization of object behaviour in Standard ML of New Jersey. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

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

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

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

[28] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, EECS Department, 1989. Technical Report UCB/CSD 89/544.