Memory Subsystem Performance of Programs Using Copying Garbage Collection

Amer Diwan
David Tarditi
Eliot Moss*

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

Abstract

Heap allocation with copying garbage collection is believed to have poor memory subsystem performance. We conducted a study of the memory subsystem performance of heap allocation for memory subsystems found on many machines. We found that many machines support heap allocation poorly. However, with the appropriate memory subsystem organization, heap allocation can have good memory subsystem performance.

1 Introduction

Heap allocation with copying garbage collection is widely believed to have poor memory subsystem performance [31, 38, 39, 24, 40]. To investigate this, we conducted an extensive study of memory subsystem performance of heap allocation intensive programs on memory subsystem organizations typical of many workstations. The programs, compiled with the SML/NJ compiler [3], do tremendous amounts of heap allocation, allocating one word every to 4 to 10 instructions. The programs used a generational copying garbage collector to manage their heaps. To our surprise, we found that for some configurations corresponding to actual machines, such as the DECStation 5000/200, the memory subsystem performance was comparable to that of C and Fortran programs [10]: programs ran only 16% slower than they would have with an infinitely fast memory. This performance is similar to that for C and Fortran programs For other configurations, the slowdown was often higher than 100%.

The memory subsystem features important for achieving good performance with heap allocation are subblock placement with a subblock size of one word combined with write-allocate on write-miss, a write buffer and page-mode writes, and cache sizes of 32K or larger. Heap allocation performs poorly on machines which do not have one or more of these features; this includes most current workstations.

Our work differs from previous reported work [31, 38, 39, 24, 40] on memory subsystem performance of heap allocation in two important ways. First, previous work used *overall miss ratios* as the performance metric and neglected the potentially different costs of read and write misses. Overall miss ratios are misleading indicators of performance: a high overall miss ratio does not always translate to bad performance. We separate read misses from write misses. Second, previous work did not model the entire memory subsystem: it concentrated solely on caches. Memory subsystem features such as write buffers and page-mode writes interact with the costs of hits and misses in the cache and should be simulated to give a correct picture of memory subsystem behavior. We simulate the entire memory subsystem.

We did the study by instrumenting programs to produce traces of all memory references. We fed the references into a memory subsystem simulator which calculated a performance penalty due to the memory subsystem. We fixed the architecture to be the MIPS R3000 [23] and varied cache configurations to cover the design space typical of workstations such as DECStations, SPARCStations, and HP 9000 series 700. All the memory subsystem configurations we studied had a write buffer and page-mode writes. We studied eight substantial programs.

We varied the following cache parameters: size (8K to 128K), block size (16 or 32 bytes), write miss policy (write allocate or write no allocate), subblock placement (with and without), and associativity (one and two way). We simulated

*The authors can be reached electronically via Internet addresses diwan@cs.umass.edu, dtarditi@cs.cmu.edu, moss@cs.umass.edu. This work was done while Amer Diwan and Eliot Moss were on leave from University of Massachusetts.

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 is also supported by an AT&T PhD Scholarship.

only split instruction and data caches, *i.e.*, no unified caches. We report data only for write-through caches but the results extend easily to write-back caches (see Section 5.2).

Section 2 gives background information. Section 3 describes related work. Section 4 describes the simulation methods used, the benchmarks used, and the metrics used to measure memory subsystem performance. Section 5 presents the results of the simulation studies, and an analysis of those results. Section 6 concludes.

2 Background

The following sections describe memory subsystems, copying garbage collection, SML, and the SML/NJ compiler.

2.1 Memory subsystems

This section reviews the organization of memory subsystems. Since terminology for memory subsystems is not standardized we use Przybylski's terminology [32].

It is well known that CPUs are getting faster relative to DRAM memory chips; main memory cannot supply the CPU with instructions and data fast enough. A solution to this problem is to use a *cache*, a small fast memory placed between the CPU and main memory that holds a subset of memory. If the CPU reads a memory location which is in the cache, the value is returned quickly. Otherwise the CPU must wait for the value to be fetched from main memory.

Caches work by reducing the average memory access time. This is possible since memory accesses exhibit *temporal* and *spatial* locality. Temporal locality means that a memory location that was referenced recently will probably be referenced again soon and is thus worth storing in the cache. Spatial locality means that a memory location near one which was referenced recently will probably be referenced soon. Thus, it is worth moving the neighboring locations to the cache.

2.1.1 Cache organization

This section describes cache organization for a single level of caching. A cache is divided into *blocks*, each of which has an associated *tag*. A cache block represents a block of memory. Cache blocks are grouped into *sets*. A memory block may reside in the cache in exactly one set, but may reside in any block within the set. The tag for a cache block indicates what memory block it holds. A cache with sets of size n is said to be *n-way associative*. If n=1, the cache is called *direct-mapped*. Some caches have *valid bits*, to indicate what sections of a block hold valid data. A *subblock* is the smallest part of a cache with which a valid bit is associated. In this paper, *subblock placement* implies a subblock size of one word, *i.e.*, valid bits are associated with each word. Moreover, on a read miss, the whole block is brought into the cache not just the subblock that missed. Przybylski [32] notes that this is a good choice.

A memory access for which a block is resident in the cache is called a *hit*. Otherwise, the memory access is a *miss*.

A read request for memory location m causes m to be mapped to a set. All the tags and valid bits (if any) in the set are checked to see if any block contains the memory block for m. If a cache block contains the memory block for m, the word corresponding to m is selected from the cache block. A read miss is handled by copying the missing block from the main memory to the cache.

A write hit is always written to the cache. There are several policies for handling a write miss, differing in their performance penalties. For each of the policies, the actions taken on a write miss are:

1. write no allocate:
   - Do not allocate a block in the cache
   - Send the write to main memory, without putting the write in the cache.
2. write allocate, no subblock placement:
   - Allocate a block in the cache.
   - Fetch the corresponding memory block from main memory.
   - Write the word to the cache and to memory.
3. write allocate, subblock placement[1]:
   - Allocate a block in the cache.
   - Write the word to the cache and to memory.
   - Invalidate the remaining words in the block.

Write allocate/subblock placement will have a lower write miss penalty than *write allocate/no subblock placement* since it avoids fetching a memory block from main memory. In addition, it will have a lower penalty than *write no allocate* if the written word is read before being evicted from the cache. See Jouppi [22] for more information on write miss policies.

A miss is a *compulsory miss* if it is due to a memory block being accessed for the first time. A miss is a *capacity miss* if it results from the cache (size C) not being big enough to hold all the memory blocks used by a program. This corresponds to the misses in a fully associative cache of size C with LRU replacement policy (minus the compulsory misses). It is a *conflict miss* if it results from two memory blocks mapping to the same set. [20]

A *write buffer* may be used to reduce the cost of writes to main memory. A *write buffer* is a queue containing writes that are to be sent to main memory. When the CPU does a write, the write is placed in the write buffer and the CPU continues without waiting for the write to finish. The write buffer retires entries to main memory using free memory cycles. A *write buffer stall* occurs if the write buffer is full

[1]Recall subblock size is assumed to be 1 word.

when the CPU tries to do a write or tries to read a location queued up in the write buffer.

Main memory is divided into DRAM pages. *Page-mode writes* reduce the latency of writes to the same DRAM page when there are no intervening memory accesses to another DRAM page.

2.1.2 Memory subsystem performance

This section describes two metrics for measuring the performance of memory subsystems. One popular metric is the *cache miss ratio*. The cache miss ratio is the number of memory accesses that miss divided by the total number of memory accesses. Since different kinds of memory accesses usually have different miss costs, it is useful to have miss ratios for each kind of access.

Cache miss ratios alone do not measure the impact of the memory subsystem on overall system performance. A metric which better measures this is the contribution of the memory subsystem to CPI (cycles per useful instruction[2]). CPI is calculated for a program as *number of CPU cycles to complete a program / total number of useful instructions executed*. It measures how efficiently the CPU is being utilized. The contribution of the memory subsystem to CPI is calculated as *number of CPU cycles spent waiting for the memory subsystem / total number of useful instructions executed*. As an example, on a DECStation 5000/200, the lowest CPI possible is 1, completing one instruction per cycle. If the CPI for a program is 1.50, and the memory contribution to CPI is 0.3, 20% of the CPU cycles are spent waiting for the memory subsystem (the rest may be due to other causes such as nops, multi-cycle instructions like integer division, etc.). CPI is machine dependent since it is calculated using actual penalties.

2.2 Copying garbage collection

A copying garbage collector [18, 11] reclaims an area of memory by copying all the live (non-garbage) data to another area of memory. This means that all data in the garbage-collected area is now garbage, and the area can be re-used. Since memory is always reclaimed in large contiguous areas, objects can be sequentially allocated from such areas at the cost of only a few 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.

The SML/NJ compiler uses a simple generational copying garbage collector [28]. Memory is divided into an old generation and an allocation area. New objects are created in the allocation area; garbage collection copies the live objects

[2]All instructions besides nops are considered to be useful. A nop (null operation) instruction is a software-controlled pipeline stall

```
% 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

in the allocation area to the old generation, freeing up the allocation area. Generational garbage collection relies on the fact that most allocated objects die young; thus most objects (about 99% [3, p. 206]) are not copied from the allocation area. This makes the garbage collector efficient, since it works mostly on an area of memory where it is very effective at reclaiming space.

The most important property of a copying collector with respect to memory subsystem behavior is that allocation initializes memory which has not been touched in a long time and is thus unlikely to be in the cache. This is especially true if the allocation area is large relative to the size of the cache since allocation will knock everything out of the cache. This means that for small caches there will be a large number of (write) misses.

For example consider the code in Figure 1. Assume that a cache write miss costs 16 CPU cycles and that the block size is 4 words. On average, every fourth word allocated causes a write miss. Thus, the average memory subsystem cost of allocating a word on the heap is 4 cycles. The average cost for allocating a cons cell is seven cycles (at one cycle per instruction) plus 12 cycles for the memory subsystem overhead. Thus, while allocation is cheap in terms of instruction counts, it is expensive in terms of machine cycle counts.

2.3 Standard ML

Standard ML (SML) [30] is a call-by-value, lexically scoped language with higher-order functions, garbage collection, static typing, a polymorphic type system, provable safety properties, a sophisticated module system, and a dynamically scoped exception mechanism.

SML encourages a non-imperative programming style. Variables cannot be altered once they are bound, and by default data structures cannot be altered once they are created. Lisp's `rplaca` and `rplacd` do not exist for the default definition of lists in SML. The only kinds of assignable data structures are `ref` cells and arrays[3], which must be ex-

[3]Although the language definition omitted arrays, all implementations

plicitly declared. To emphasis the point, assignments are permitted but discouraged as a general programming style. The implications of this non-imperative programming style for compilation are clear: SML programs tend to do more allocation and copying than programs written in imperative languages.

SML is most closely related to Lisp and Scheme[34]. Implementation techniques for one of these languages are mostly applicable to the other languages, with the following caveats: SML programs tend to be less imperative than Lisp or Scheme programs and Scheme and SML programs use function calls more frequently than Lisp, since recursion is the usual way to achieve iteration in Scheme and SML.

2.4 SML/NJ compiler

The SML/NJ compiler [3] is a publicly available compiler for SML. We used version 0.91. The compiler concentrates on making allocation cheap and function calls fast. Allocation is done in-line, except for the allocation of arrays. Aggressive β-reduction (inlining) is used to eliminate functions calls and their associated overhead. Function arguments are passed in registers when possible, and register targeting is used to minimize register shuffling at function calls. A split caller/callee-save register convention is used to avoid excessive spilling of registers. The compiler also does constant-folding, elimination of functions which trivially call other functions, limited code hoisting, uncurrying, and instruction scheduling.

The most controversial design decision in the compiler was to allocate procedure activation records on the heap instead of the stack [1, 5]. In principle, the presence of higher-order functions means that procedure activation records must be allocated on the heap. With a suitable analysis, a stack can be used to store most activation records [25]. However, using only a heap simplifies the compiler, the run-time system [2], and the implementation of first-class continuations [19]. The decision to use only a heap was controversial because it greatly increases the amount of heap allocation, which is believed to cause poor memory subsystem performance.

3 Related Work

There have been many studies of the cache behavior of systems using heap allocation and some form of copying garbage collection. Peng and Sohi [31] examined the data cache behavior of some small Lisp programs. They used trace-driven simulation, and proposed an ALLOCATE instruction for improving cache behavior, which allocates a block in the cache without fetching it from memory. Wilson *et. al.* [38, 39] argued that cache performance of programs with generational garbage collection will improve substantially when the youngest generation fits in the cache. Koopman *et. al.* [24] studied the effect of cache organization on combinator graph reduction, an implementation technique for lazy functional programming languages. Combinator graph reduction does more heap allocation and assignments than SML/NJ programs. They observed the importance of a write-allocate policy with subblock placement for improving heap allocation. Zorn [40] studied the impact of cache behavior on the performance of a Common Lisp system, when stop-and-copy and mark-and-sweep garbage collection algorithms were used. He concluded that programs run with mark-and-sweep have substantially better cache locality than when run with stop-and-copy.

These works all used data cache miss ratios to evaluate cache performance. They did not separate read and write misses, despite the different costs of these misses. Also, they did not simulate the entire memory subsystem. Our work separates read misses from write misses and completely models the memory subsystem, including write buffers and page-mode writes.

Appel [3] estimated CPI for the SML/NJ system on a single machine using elapsed time and instruction counts. His CPI differs substantially from ours. Apparently instructions were undercounted in his measurements [4].

Jouppi [22] studied the effect of cache write policies on the performance of C and Fortran programs. Our class of programs is different from his, but his conclusions support ours: that a write-allocate policy with subblock placement is a desirable architecture feature. He found that the write miss ratio for the programs he studied was comparable to the read miss ratio, and that write-allocate with subblock placement eliminated the cost of write misses. For programs compiled with the SML/NJ compiler, this is even more important due to the high number of write misses caused by allocation.

4 Methodology

We used trace-driven simulations to evaluate the memory subsystem performance of programs. For trace-driven simulations to be useful, there must be an accurate simulation model and a good selection of benchmarks. Simulations that make simplifying assumptions about important aspects of the system being modeled can yield misleading results. Toy benchmarks, or unrepresentative benchmarks, can be equally misleading. We have devoted much effort to addressing these issues.

4.1 Tools

We have extended QPT [7, 26, 27] to produce memory traces for SML/NJ programs. QPT rewrites an executable program to produce a full instruction and data trace. Because QPT operates on the executable program, it can trace both the SML code and the garbage collector (written in C).

We used Tycho [21] for the memory subsystem simulations. Tycho uses a special case of *all-associativity simula-*

have arrays.

tion [29] to simulate multiple caches concurrently. We have added a write-buffer simulator to Tycho, which concurrently simulates a write buffer for each instruction and data cache pair being simulated. The write-buffer simulator also takes page-mode writes and memory refreshes into consideration.

4.2 Simplifications and Assumptions

We wanted to simulate the memory subsystems as completely as we could. Thus, we have tried to minimize simplifications which may reduce the validity of our data. The most important simplifications are:

1. We ignore the effects of context switches and system calls.
2. Our simulations are driven by virtual addresses even though many current machines have physically-addressed caches.
3. We use default compilation flags which enable extensive optimizations. We set the soft limit of the garbage collector to 20000K[4].
4. When comparing different cache organizations we assume that the CPU cycle time is the same.

4.3 Benchmarks

Table 1 describes the benchmark programs[5]. *Knuth-Bendix, Lexgen, Life, Simple, VLIW*, and *YACC* are identical to the benchmarks measured by Appel [3][6]. Table 2 gives 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)[7], and run time, in seconds, on a DECStation 5000/200. The run times are the minimum of five runs.

Table 3 characterizes the memory references of the benchmark programs. The *Writes* column lists the number of full word writes done by the program *and* the garbage collector; the *Assignments* column lists the non-initializing writes done by the program only. The *Partial Writes* column lists the number of partial word (bytes, half-word, etc.) writes done by the program and the garbage collector[8]. All the benchmarks have long traces; most other work on memory system performance uses traces that are an order of magnitude smaller. The benchmark programs do few assignments; the majority of the writes are initializing writes.

Table 4 gives the allocation statistics for each benchmark program. All allocation and sizes are reported in words. The *Allocation* column lists the total allocation done by the benchmark. The remaining columns break down the allocation by kind: closures for escaping functions, closures for known functions, closures for callee-save continuations[9], records, and others (includes spill records, arrays, strings, vectors, ref cells, and store list records). For each allocation kind, the % column is the percentage of total allocation allocated for that kind of object and *Size* is the average size (including the 1 word tag) for that kind of object.

4.4 Metrics

We state cache performance numbers in *cycles per useful instruction* (*CPI*). All instructions besides *nops* are considered useful.

Table 5 lists the penalties used in the simulations. These numbers are derived from the penalties for the DECStation 5000/200, but are similar to those in other machines of the same class. Note that write misses have no penalty (besides write buffer costs) for caches with subblock placement[10].

5 Results and Analysis

Section 5.1 qualitatively analyzes the memory behavior of programs. Section 5.2 lists the cache configurations simulated and explains why they were selected. Sections 5.3 presents and analyzes data for memory subsystem performance.

5.1 Qualitative Analysis

Recall from Section 2 that SML/NJ uses a copying collector which leads to a large number of write misses. The slowdown this translates into depends on the cache organization being used.

Recall from Section 4.3 that SML/NJ programs have the following properties. First, they do few assignments; the majority of the writes are initializing writes. Second, programs do heap allocation at a furious rate: 0.1 to 0.22 words per instruction. Third, writes come in bunches because they correspond to initialization of a newly allocated area.

The burstiness of writes combined with the property of copying collectors mentioned above suggests that an aggressive write policy is necessary. In particular, writes should not stall the CPU. Memory subsystem organizations where the CPU has to wait for a write to be written to memory will perform poorly. Even memory subsystems where the

[4] This is large enough to allow the garbage collector to resize the heap as needed.

[5] Available from the authors.

[6] The description of these benchmarks have been copied from [3].

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

[8] Partial-word writes are distinguished from full-word writes since they are often more expensive than full-word writes. We charge 11 cycles for each partial-word write.

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

[10] In an actual implementation, the penalty of a miss *may* be one cycle since unlike hits, the tag and valid bits needs to be written to the cache after the miss is detected. This will not change our results since it adds at most 0.02–0.05 to the CPI of caches with subblock placement.

Program	Description
CW	The Concurrency Workbench [12] is a tool for analyzing networks of finite state processes expressed in Milner's Calculus of Communicating Systems.
Leroy	An implementation of the Knuth-Bendix completion algorithm.
Lexgen	A lexical-analyzer generator [6], processing the lexical description of Standard ML.
Life	The game of Life implemented using lists [33].
PIA	The Perspective Inversion Algorithm [37] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program [13].
VLIW	A Very-Long-Instruction-Word instruction scheduler.
YACC	An implementation of an LALR(1) parser generator [36] processing the grammar of Standard ML.

Table 1: 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 2: 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 3: Characteristics of benchmark programs

		Escaping		Known		Callee Saved		Records		Other	
Program	Allocation	%	Size	%	Size	%	Size	%	Size	%	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	37,849,681	0.2	3.45	0.0	15.00	77.8	5.52	22.2	3.00	0.0	10.29
PIA	13,047,041	0.6	5.56	40.4	11.99	36.5	4.73	18.4	3.41	4.1	16.01
Simple	67,261,664	4.8	5.70	1.3	15.33	81.8	6.43	9.9	3.00	2.2	5.01
VLIW	59,496,919	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 4: Allocation characteristics of benchmark programs

Task	Penalty (in cycles)
Non page mode write	5
Page mode write	1
Read 16 bytes from memory	15
Read 32 bytes from memory	19
Write hit or miss (subblocks)	0
Write hit (16 bytes, no subblocks)	0
Write hit (32 bytes, no subblocks)	0
Write miss (16 bytes, no subblocks)	15
Write miss (32 bytes, no subblocks)	19

Table 5: Penalties of memory operations

CPU does not need to wait for writes if they are issued far apart (e.g., 2 cycles apart in the HP 9000 series 700) may perform poorly due to the bunching of writes. This leads to two requirements on the memory subsystem. First, a write buffer or fast page mode writes are essential to avoid waiting for writes to memory. Second, on a write miss, the memory subsystem must avoid reading a cache block from memory if it will be written before being read. Of course, this requirement only holds for caches with a *write-allocate* policy. Subblock placement [24], a block size of 1 word, and the ALLOCATE directive [31] can all achieve this[11]. For large caches, when the allocation area fits in the cache and thus there are few write misses, the benefit of subblock placement will be reduced.

5.2 Cache configurations simulated

Since the design space for memory subsystems is enormous we had to prune the design space that we could study. In this study, we restrict ourselves to features found in *currently popular* RISC workstations. Exploration of more exotic memory subsystem features is left to future work. Table 6 summarizes the cache organizations simulated. Table 7 lists the memory subsystem organization for some popular machines.

We simulated only separate instruction and data caches (*i.e.,* no unified caches). While many current machines have separate caches (e.g., DECStations, HP 9000 series), there are some exceptions (notably SPARCs).

We simulated cache sizes from 8K to 128K. This range includes the primary caches of most current machines (see Table 7). We consider only direct mapped and two-way set associative caches (with LRU replacement).

We simulated block sizes of 16 bytes and 32 bytes. Przybylski [32] notes that block sizes of 16 or 32 bytes optimize the read access time for the memory parameters used in the CPI calculations (see Section 4.4).

We report data only for write-through caches but the CPI for write-back caches can be inferred from our graphs. Write-through and write-back caches give identical misses, but the penalties for write hits and write misses differ. A write hit or miss in a write-back cache may take one cycle more than in a write-through cache [22]. This tells us at most how much the write-through graphs need to be shifted to obtain the CPI graphs for write-back caches. For instance, if the program has w writes and n useful instructions, then we must add w/n to the CPI. For `CW` this adds 0.13. Write-through and write-back caches may have different write buffer penalties. We expect the write buffer penalties for write-back caches to be smaller than that for write-through caches since writes to main memory are less frequent for write-back caches than for write-through caches. In any case, write buffer penalties are negligible even for write-through caches (Section 5.3).

Two of the most important cache parameters are *write allocate* versus *write no allocate* and *subblock placement* versus *no subblock placement*. Of these, the combination *write no allocate/subblock placement* offers no improvement over *write no allocate/no subblock placement* for cache performance. Thus, we did not collect data for the *write no allocate/subblock placement* configuration.

We restrict ourselves only to the first two levels of the memory hierarchy, which on most current machines corresponds to the primary cache and main memory. The results, however, are mostly applicable when the second level is a secondary cache and the cost of accessing the secondary cache is similar to the cost of accessing main memory on the DECStation 5000/200[12]. In such machines, there is a memory subsystem contribution to the CPI that we did not measure: a miss on the second level cache. Therefore the CPI obtained on these machines can be higher than that reported here.

We do not simulate the exotic features appearing on some newer machines, such as stream buffers, prefetching, and victim caches. These features can reduce the cache miss rates and miss costs. Further work is needed to understand the impact of these features on performance of heap allocation.

5.3 Memory Subsystem Performance

Memory subsystem performance is presented in summary graphs and breakdown graphs. Each summary graph summarizes the memory subsystem performance of one benchmark program for a range of write-miss policies (write allocate or no write allocate), subblock placement (with or without), cache sizes (8K to 128K), and associativity (1 or 2). Each curve in a summary graph corresponds to a different memory subsystem organization. There are two summary graphs for each program, one for a block size of 16 bytes and another for a block size of 32 bytes. Each breakdown graph breaks down the memory subsystem overhead into read misses, instruction-fetch misses, write-buffer overhead, and partial-word write overhead for one configuration in a summary graph. The write-buffer depth in these graphs is fixed at 6 entries.

In this paper we present only the summary graphs for `CW` (Figure 2). The summary graphs for other programs

[11] Since the effects on cache performance of these features are so similar, we talk just about subblock placement.

[12] For instance, Borg et al. [8] use 12 cycles as the latency for going to the second level cache and 200–250 cycles for going to memory.

Write Policy	Write Miss Policy	Write Buffer	Subblocks	Assoc	Block Size	Cache Sizes
through	allocate	6 deep	yes	1, 2	16,32 bytes	8K–128K
through	allocate	6 deep	no	1, 2	16,32 bytes	8K–128K
through	no allocate	6 deep	no	1, 2	16,32 bytes	8K–128K

Table 6: Cache organizations studied

Architecture	Write Policy	Write Miss Policy	Write Buffer	Subblocks	Assoc	Block Size	Cache Size
DS3100 [16]	through	allocate	4 deep	—	1	4 bytes	64K
DS5000/200 [15]	through	allocate	6 deep	yes	1	16 bytes	64K
HP 9000 [35]	back	allocate	none	no	1	32 bytes	64K–2M
SPARCStation II [14]	through	no allocate	4 deep	no	1	32 bytes	64K

Note:

- SPARCStations have unified caches.
- Most HP 9000 series 700 caches are much smaller than 2M: 128K instruction cache and 256K data cache for models 720 and 730, and 256K instruction cache and 256K data cache for model 750.
- The DS5000/200 actually has a block size of four bytes with a fetch size of sixteen bytes. This is actually stronger than subblock placement since it has a full tag on every "subblock".
- The higher end HP 9000 machines (model 735 and above) provide a cache-control hint in their store instructions [9]. The hint can specify that a block will be overwritten before being read; this avoids the read if the write misses.

Table 7: Memory subsystem organization of some popular machines

are similar to those for `CW` and are thus omitted for space considerations[13]. Any significant differences between `CW`'s graphs and the omitted graphs are noted in the text. Figures 3 and 4 are the breakdown graphs for `CW` for the 16 byte block size configurations; the remaining breakdown graphs for `CW` are omitted for space considerations. The breakdown graphs for the other benchmarks are similar and are thus also omitted for space considerations [14].

In the summary graphs, the *nops* curve is the base CPI: the number of useful (not nop) instructions executed divided by the total number of instruction executed; this corresponds to the CPI for a perfect memory subsystem[15]. For the breakdown graphs, the *nop* area is the CPI contribution of nops; *read miss* is the CPI contribution of read misses; *if miss* is the CPI contribution of instruction fetch misses; *write buffer* is the CPI contribution of the write buffer; *partial word* is the CPI contribution of partial-word writes[16].

The 64K point on the *write alloc, subblock, assoc=1* curves corresponds closely to the DECStation 5000/200 memory subsystem.

In the following sections we describe the impact of write-miss policy and subblock placement, associativity, block size, cache size, write buffer, and partial-word writes on the memory subsystem performance of the benchmark programs.

[13] The full set of graphs are available via anonymous ftp from ibis.cs.umass.edu in pub/memory-subsystem.

[14] `Lexgen`'s graphs are a little different in that there is a steep drop in the instruction cache contribution to the CPI in going from an 8K to 16K cache.

[15] *nops* constitute between 5.9% and 15.4% of all instructions executed for the benchmarks (see Section 4.3).

[16] This overhead is so small that it is not visible in most of the breakdown graphs.

5.3.1 Write Miss Policy and Subblock Placement

From the summary graphs, it is clear that the best cache organization we studied is *write allocate/subblock placement*; in every case, *write-allocate/subblock placement* substantially outperforms all other configurations. Surprisingly, for sufficiently large caches with the *write allocate/subblock placement* organization, the memory subsystem performance of SML/NJ programs is acceptable (around 17% or less overhead)[17]. For caches with *write allocate/subblock placement*, the average memory subsystem contribution to the CPI over all benchmarks is 16% for 64K direct mapped caches and 17% for 32K two-way associative caches. The DS5000/200 organization does well for most programs. It is worth emphasizing that the memory subsystem performance of SML/NJ programs is *good* on some current machines *despite the very high miss rates*; for a 64K *write allocate/no subblock placement* organization with a block size of 16 bytes, the write miss and read miss ratios for `CW` are 0.18 and 0.04 respectively.

Recall that in Section 5.1 we argued that subblock placement would be a big win, but its benefit would decrease for larger caches. Our data indicates that the reduction in benefit is not substantial even for 128K cache sizes although a slight tapering off is seen in `CW`. This indicates that 128K is not large enough to hold the allocation area of most of the benchmark programs.

The performance of *write allocate/no subblock* is almost identical to that of *write no allocate/no subblock* (`Leroy` is an exception)[18]. This suggests that an address is being read

[17] For the penalties used, a 17% overhead translates roughly into one fetch from memory—instruction or data—every 100 useful instructions.

[18] The difference between *write allocate/no subblock* and *write no allo-*

soon after being written; even in an 8K cache, an address is read after being written before it is evicted from the cache (if it was evicted from the cache before being read, then *write allocate/no subblock* would have inferior performance). The only difference between these two schemes is *when* a cache block is read from memory. In one case, it is brought in on a write miss; in the other, it is brought in on a read miss. Because SML/NJ programs allocate sequentially and do few assignments, a newly allocated object remains in the cache until the program has allocated another `C` bytes, where `C` is the size of the cache. Since our programs allocate 0.4–0.9 bytes per instruction, our results suggest that a read of a block occurs within 9K–20K instructions of it being written.

5.3.2 Changing Associativity

From Figure 2 we see that increasing associativity improves all organizations. However the improvement in going from one-way to two-way set associativity is much smaller than the improvement obtained from subblock placement: in most cases, it improves the CPI by less than 0.1. The maximum benefit from higher associativity is obtained for small cache sizes (less than 16K). However, increasing associativity may increase CPU cycle time and thus the improvements may not be realized in practice [20].

From Figures 3 and 4 we see that higher associativity improves the instruction cache performance but has little or no impact on data cache performance. Surprisingly, for direct mapped caches (Figures 3 (a) and 4 (a)) the instruction cache penalty is substantial for caches smaller than 128K. For caches with subblock placement, the instruction cache penalty dominates the penalty for the memory subsystem. The improvement observed in going to a two-way associative cache suggests that a lot of the penalty from the instruction cache is due to conflict misses and that from the data cache is due to capacity misses: the data cache is simply not big enough to hold the working set. When the code produced by SML/NJ is examined, the performance of the instruction cache is not surprising: the code consists of small functions with frequent calls, which lower the spatial locality. Thus, the chances of conflicts are greater than if the instructions had strong spatial locality.

5.3.3 Changing Block Size

From Figure 2 we see that increasing block size from 16 to 32 bytes also improves performance. For the *write allocate* organizations, an increased block size decreases the number of write misses caused by allocation. When the allocation area does not fit in the cache, doubling the block size can halve the write-miss rate. Thus, larger block sizes improve performance when there is a penalty for a write miss [24]. In particular, larger block sizes have little to offer to caches with *write allocate/subblock placement*. From Figure 2 we see that the *write no allocate* organizations benefit just as much from larger block size as *write allocate/no subblock placement*; this suggests that the spatial locality in the reads is comparable to that in the writes.

Note that subblock placement improves performance more than even two-way associativity and 32 byte blocks combined.

cate/no subblock is so small in most graphs that the two curves overlap.

5.3.4 Changing Cache Size

Increasing the cache size improves performance for all configurations. In most cases, the performance improvement from doubling the cache size is small. We expect to see a sharp improvement in performance for some larger cache size once the allocation area fits in the cache (this will not be nearly as significant for caches with subblock placement)[19]. From the breakdown graphs we see that the cache size has little effect on the data cache miss contribution to CPI. Most of the improvement in CPI that comes from increasing the cache size is due to improved performance of the instruction cache. As with associativity, cache sizes have interactions with the cycle time of the CPU: larger caches can take longer to access. Thus, improvement due to increasing the cache size may not be achieved in practice.

5.3.5 Write Buffer and Partial-Word Write Overheads

From the breakdown graphs we see that the write buffer and partial word write contribution to the CPI is negligible. A six deep write buffer coupled with page-mode writes is sufficient to absorb the bursty writes. As expected, memory subsystem features which reduce the number of misses (such as higher associativity and larger cache sizes) also reduce the write buffer overhead.

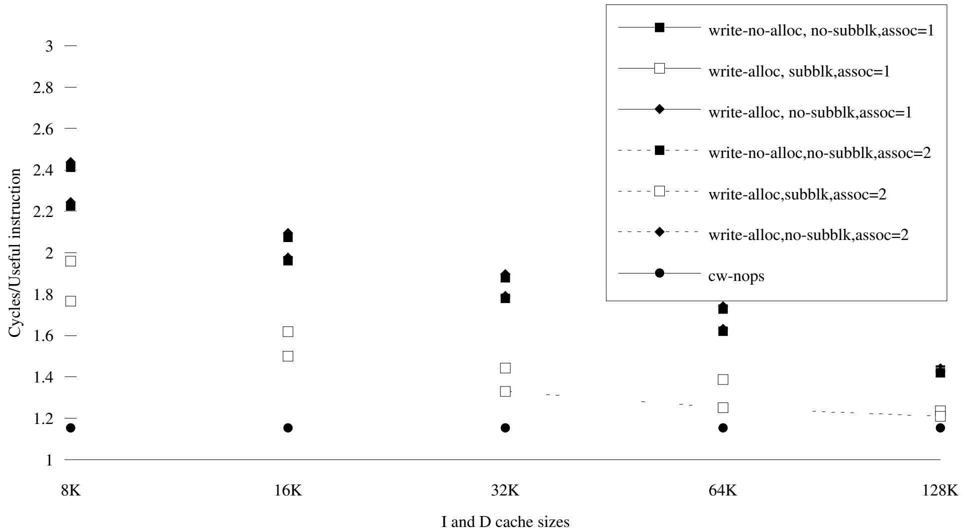
6 Conclusions

We described an in-depth study of the memory subsystem performance of programs compiled with SML/NJ. The important characteristics of these programs, with respect to memory subsystem performance, were intensive heap allocation and the use of copying garbage collection.

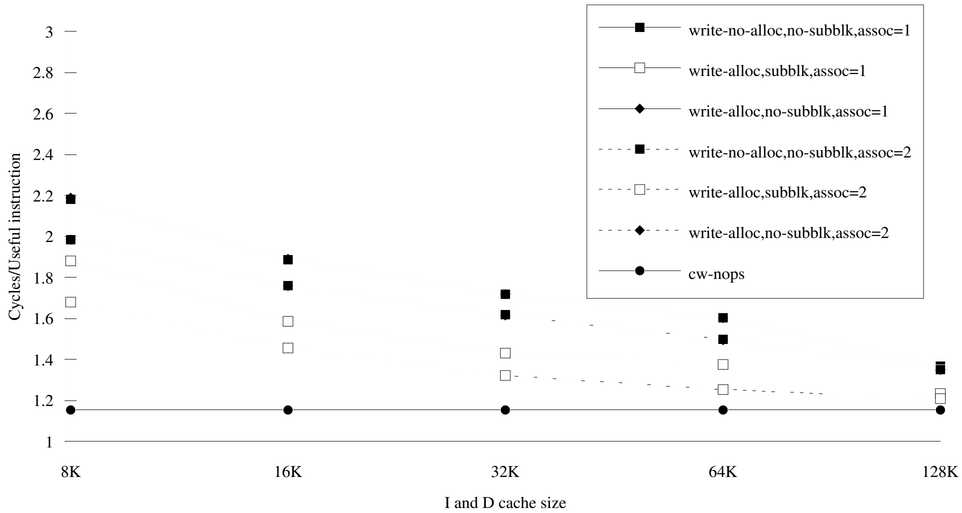
In agreement with [31, 38, 39, 40], programs with intensive heap allocation performed poorly on most memory subsystem organizations. However, on some current machines (in particular the DECStation 5000/200), the performance was good.

The memory organization parameter crucial for good performance was subblock placement. For caches with subblock placement, the memory subsystem overhead was under 17% for 64K or bigger caches; for caches without subblock placement, the overhead was often as high as 100%.

[19] Subsequent work has shown that 512K is large enough to hold the allocation area of most of the benchmark programs [17].

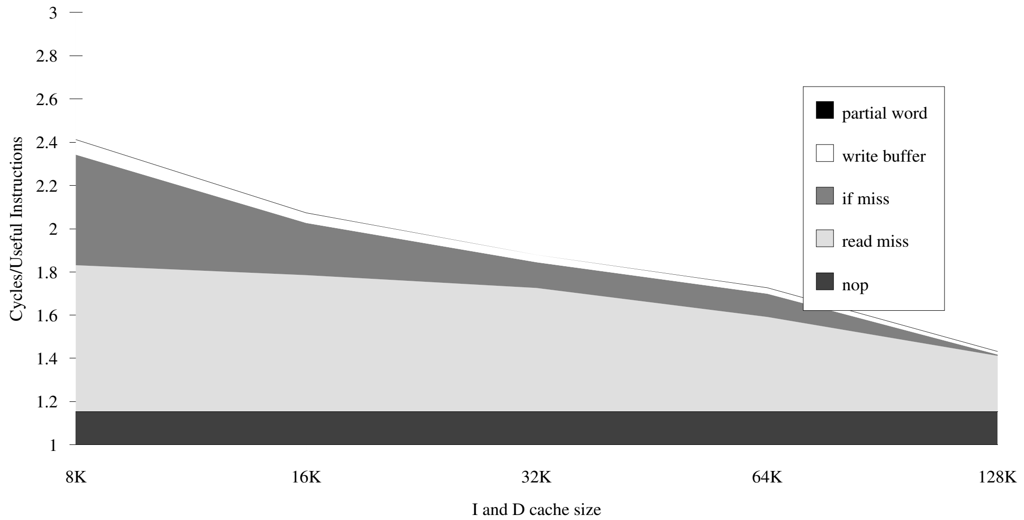


(a) block size=16 bytes

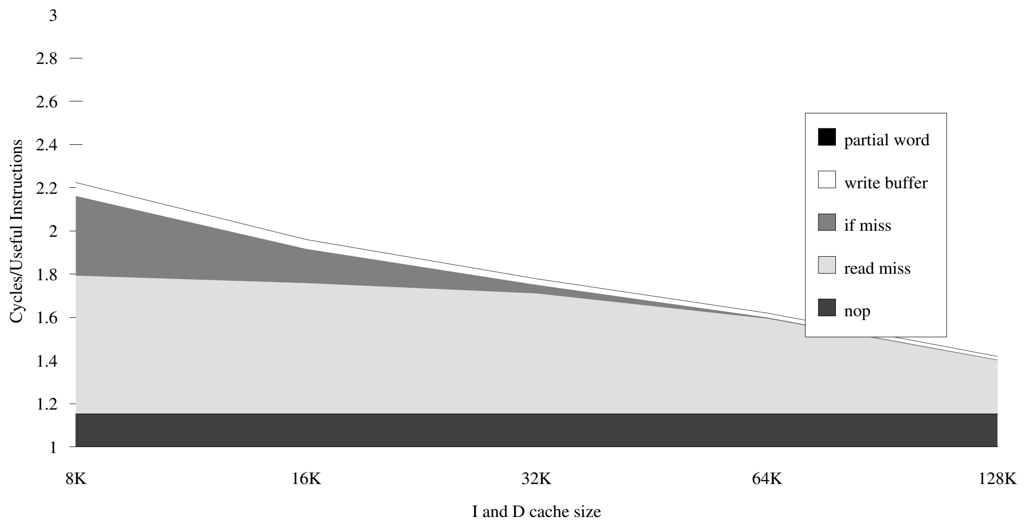


(b) block size=32 bytes

Figure 2: CW summary, write buffer depth=6

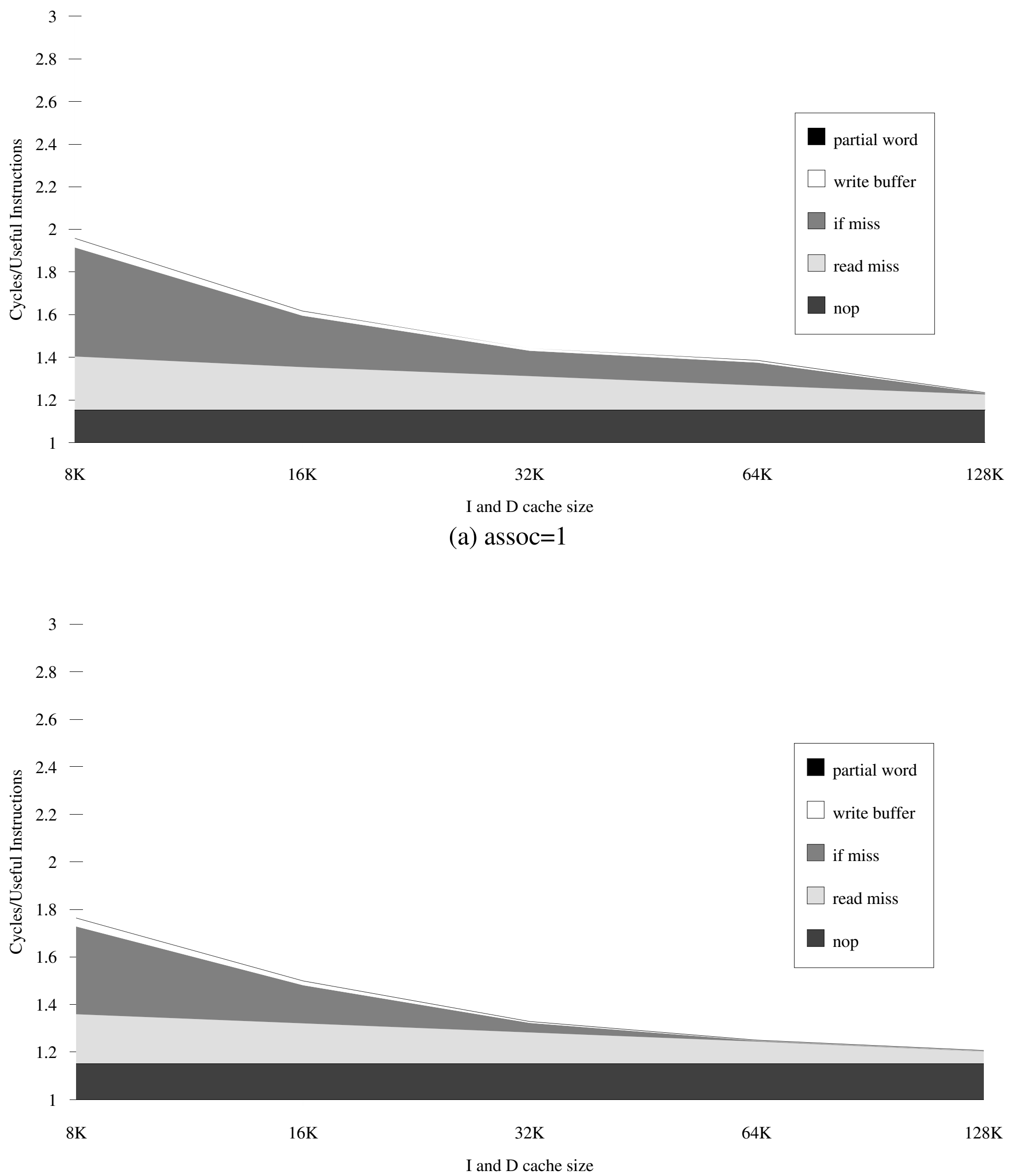


(a) assoc=1



(b) assoc=2

Figure 3: CW breakdown, write no alloc, no subblk, block size=16, wb depth=6



(a) assoc=1

(b) assoc=2

Figure 4: CW breakdown, write alloc, subblk, block size=16, wb depth=6

Associativity and cache sizes (up to 128K) had little impact on data cache performance, but were more important for instruction cache performance. While higher associativity, larger cache sizes, and larger block sizes improved performance, their contribution to performance was usually small.

To summarize, most current machines support heap allocation poorly. For these machines, compilers should avoid heap allocation as much as possible. However, with the appropriate memory subsystem organization, heap allocation can achieve good memory subsystem performance.

7 Acknowledgements

We would like to thank Edoardo Biagioni, Eric Brown, Brad Chen, Olivier Danvy, Alessandro Forin, Urs Hoelzle, Kathryn McKinley, Erich Nahum, and Darko Stefanović for comments on drafts of this paper. We thank Peter Lee for his encouragement and advice during this work. We thank Brian Milnes and the facilities at CMU for setting up the hardware according our every whim. We thank Tom Dewey for explaining the partial-word write mechanism in the DS5000/200 to us. We thank Andrew Appel, Dave MacQueen and many others for creating SML/NJ. We thank James Larus for creating qpt and for answering the questions which arose while we were extending his tool. We thank Mark Hill for creating his cache simulators, Tycho and DineroIII. Last but not least, we thank all the members of the Fox project for their interest in this work and for accommodating our demand for compute cycles.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

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

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, first edition, 1992.

[4] Andrew W. Appel. Personal Communication. March 22 1993.

[5] Andrew W. Appel and Trevor Y. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM.

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

[7] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *19th Symposium on Principles of Programming Languages*. ACM, January 1992.

[8] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from RISC machines: Generation and analysis. Technical Report 89/14, DEC Western Research Laboratory, September 1989.

[9] Brian Case. PA-RISC provides rich instruction set within RISC framework. *Microprocessor Report*, 5(6), April 1991.

[10] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system perf ormance. In *Fourteenth Symposium on Operating System Principles*. ACM, Dec 1993.

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

[12] 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.

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

[14] Cypress Semiconductor, Ross Technology Subsidiary. *SPARC RISC User's Guide*, second edition, February 1990.

[15] Digital Equipment Corporation. *DS5000/200 KN02 System Module Functional Specification*.

[16] Digital Equipment Corporation, Palo Alto, CA. *DECStation 3100 Desktop Workstation Function Specification*, 1.3 edition, August 1990.

[17] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Work in progress, oct 1993.

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

[19] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77. ACM, June 1990.

[20] Mark D. Hill. A case for direct mapped caches. *Computer*, 21(12):25–40, December 1988.

[21] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

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

[23] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[24] Philip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *Transactions on Programming Languages and Systems*, 14(2):265–277, April 1992.

[25] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[26] James R. Larus. Abstract Execution: A technique for efficiently tracing programs. *Software Practice and Experience*, 20(12):1241–1258, December 1990.

[27] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report Wis 1083, Computer Sciences Department, University of Wisconsin-Madison, March 1992.

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

[29] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

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

[31] 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.

[32] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann Publishers, first edition, 1990.

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

[34] Jonathan Rees and William Clinger. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[35] Michael Slater. PA workstations set price/performance records. *Microprocessor Report*, 5(6), April 1991.

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

[37] 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.

[38] 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.

[39] 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.

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