Memory System Performance of Programs with Intensive Heap Allocation

AMER DIWAN, DAVID TARDITI, AND ELIOT MOSS
Carnegie Mellon University

Heap allocation with copying garbage collection is a general storage management technique for programming languages. It is believed to have poor memory system performance. To investigate this, we conducted an in-depth study of the memory system performance of heap allocation for memory systems found on many machines. We studied the performance of mostly functional Standard ML programs which made heavy use of heap allocation. We found that most machines support heap allocation poorly. However, with the appropriate memory system organization, heap allocation can have good performance. The memory system property crucial for achieving good performance was the ability to allocate and initialize a new object into the cache without a penalty. This can be achieved by having subblock placement with a subblock size of one word with a write-allocate policy, along with fast page-mode writes or a write buffer. For caches with subblock placement, the data cache overhead was under 9% for a 64K or larger data cache; without subblock placement the overhead was often higher than 50%.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*associative memories*; *cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*; C.4 [**Computer Systems Organization**]: Performance of Systems; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classification; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: Automatic storage reclamation, copying garbage collection, garbage collection, generational garbage collection, heap allocation, page mode, subblock placement, write-back, write-buffer, write-miss policy, write-policy, write-through

---

A paper containing some of the results presented in this article appeared in the 21st Annual Symposium on Principles of Programming Languages.
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. D. Tarditi is also supported by an AT & T Ph.D. Scholarship.
Views and conclusions in this article are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Project Agency, the United States Government, or AT&T.
Authors' addresses: A Diwan and E. Moss, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610; email: {Diwan; Moss}@cs umass.edu; D. Tarditi, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891; email: dtarditi@cs.cmu edu.

1. INTRODUCTION

Heap allocation with copying garbage collection is widely believed to have poor memory system performance [Koopman et al. 1992; Peng and Sohi 1989; Wilson et al. 1990; 1992; Zorn 1991]. To investigate this, we conducted an extensive study of memory system performance of heap-allocation-intensive programs on memory system organizations typical of many workstations. The programs, compiled with the SML/NJ compiler [Appel 1992], do tremendous amounts of heap allocation, allocating one word every 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 system performance was comparable to that of C and Fortran programs [Chen and Bershad 1993]: programs ran only 3 to 13% slower due to data cache misses than they would have with an infinitely fast memory. For other configurations, the slowdown due to data cache misses was often higher than 50%.

The memory system 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, page-mode writes, and cache sizes of 32K or larger. Heap allocation performs poorly on machines whose caches are smaller than the allocation area of the programs (256K or larger for the benchmarks studied here) and do not have one or more of the features mentioned above; this includes most of the current workstations.

Our work differs from previous reported work [Koopman et al. 1992; Peng and Sohi 1989; Wilson et al. 1990; 1992; Zorn 1991] on memory system performance of heap allocation in two ways. First, previous work used the overall miss ratio as the performance metric, which is a misleading indicator of performance. The overall miss ratio neglects the fact that read and write misses may have different costs. Also, the overall miss ratio does not reflect the rates of reads and writes, which may affect performance substantially. We use memory system contribution to cycles per instruction (CPI) as our performance metric, which accurately reflects the effect of the memory system on program running time. Second, previous work did not model the entire memory system: it concentrated solely on caches. Memory system 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 system behavior. We simulate the entire memory system.

We did the study by instrumenting programs to produce traces of all memory references. We fed the references into a memory system simulator which calculated a performance penalty due to the memory system. We fixed the architecture to be the MIPS R3000 [Kane and Heinrich 1992] and varied cache configurations to cover the design space typical of workstations such as DECStations, SPARCStations, and HP 9000 series 700. We studied eight substantial programs.

We varied the following memory system parameters: cache size (8K to 512K), cache block size (16 or 32 bytes), write-miss policy (write-allocate or write-no-allocate), subblock placement (with and without), associativity (one-

and two-way), TLB sizes (1 to 64 entries), write-buffer depth (1 to 6 deep), and page-mode writes (with and without). We simulated 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.

Section 2 gives background information. Section 3 describes related work. Section 4 describes the simulation methods, the benchmarks, and the memory system performance metrics used. Section 5 presents the simulation results, analyzes them, validates them, and gives an analytical model that extends them to programs with different allocation behavior. Section 6 suggests areas for future work. Section 7 concludes.

2. BACKGROUND

The following sections describe memory systems, garbage collection in SML/NJ, SML, and the SML/NJ compiler.

2.1 Memory Systems

This section describes cache organization for a single level of caching. A cache is divided into *blocks* which 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. 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 article, *subblock placement* implies a subblock 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 [1990] notes that this is a good choice.

A memory access to a location which is resident in the cache is called a *hit*. Otherwise, the memory access is a *miss*. 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 not being large enough to hold all the memory blocks used by a program. It is a *conflict miss* if it results from two memory blocks mapping to the same set [Hill 1988].

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, which differ 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 if write-through).
(3) *write-allocate, subblock placement*:
    If the tag matches but the valid bit is off:
    —Write the word to the cache (and to memory if write-through).

If the tag does not match:
—Allocate a block in the cache.
—Write the word to the cache (and to memory if write-through).
—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 [1993] for more information on write-miss policies.

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. The write buffer cannot always prevent stalls on writes to main memory. First, if the CPU writes to a full write buffer, the CPU must wait for an entry to become available in the write buffer. Second, if the CPU reads a location which is queued in the write buffer, the CPU may need to wait until the write buffer is empty. Third, if the CPU issues a read to main memory while a write is in progress, the CPU must wait for the write to finish.

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 [Patterson and Hennessy 1990]. For example, on a DECStation 5000/200, a non-page-mode write takes five cycles, while a page-mode write takes one cycle. Page-mode writes are especially effective at handling writes with high spatial locality, such as those seen when doing sequential allocation.

2.2 Memory System Performance

This section describes two metrics for measuring the performance of memory systems. One popular metric is the *cache miss ratio*. The cache miss ratio is the number of memory accesses which 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 system on overall system performance. A metric which better measures this is the contribution of the memory system to cycles per useful instruction (CPI); all instructions besides nops (software-controlled pipeline stalls) are considered useful. CPI is calculated for a program as the *number of CPU cycles to complete the program/total number of useful instructions executed*. It measures how efficiently the CPU is being utilized. The contribution of the memory system to CPI is calculated as the *number of CPU cycles spent waiting for the memory system/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 1.5, and the memory contribution to CPI is 0.3, 20% (0.3/1.5) of the CPU cycles are spent

```
cmp alloc + 12, top      , Check for heap overflow
branch-if-gt call-gc
store tag,(alloc)        ; Store tag
store ra,4(alloc)        , Store value
store rd,8(alloc)        ; Store pointer to next cell
move alloc + 4,result    , Save pointer to cell
add alloc,12             , Increment allocation pointer
```

Fig. 1. Pseudoassembly code for allocating a list cell.

waiting for the memory system (the rest may be due to other causes such as nops, multicycle instructions like integer division, etc.). CPI is machine dependent since it is calculated using actual penalties.

2.3 Copying Garbage Collection

A copying garbage collector [Cheney 1970; Fenichel and Yochelson 1969] reclaims an area of memory by copying all the live (nongarbage) data to another area of memory. All data in the garbage-collected area becomes garbage, and the area can be reused. Since memory is reclaimed in large contiguous areas, objects can be allocated sequentially from such areas in several instructions. Figure 1 gives an example of pseudoassembly code for allocating a list cell. ra contains the value to be stored in the list cell; rd contains the pointer to the next list cell; alloc is the address of the next free word in the allocation area; and top contains the end of the allocation area.

2.4 Garbage Collection in SML / NJ

The SML/NJ compiler uses a simple generational copying garbage collector [Appel 1989]. 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 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% [Appel 1992, 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 system behavior is that allocation sequentially 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 caches which cannot hold the allocation area will incur a large number of write misses.

For example consider the code in Figure 1 for allocating a list cell. Assume that a cache write miss costs 16 CPU cycles and that the block size is four words. On average, every fourth word allocated causes a write miss. Thus, the average memory system cost of allocating a word on the heap is four cycles.

The average cost for allocating a list cell is seven cycles (at one cycle per instruction) plus 12 cycles for the memory system overhead. Thus, while allocation is cheap in terms of instruction counts, it may be expensive in terms of machine cycle counts.

2.5 Standard ML

Standard ML (SML) [Milner et al. 1990] is a call-by-value, lexically scoped language with higher-order functions. SML encourages a nonimperative programming style. Variables cannot be altered once they are bound, and by default data structures cannot be altered once they are created. The only kinds of assignable data structures are ref cells and arrays, which must be explicitly declared. The implications of this nonimperative programming style for compilation are clear: SML programs tend to do more allocation and copying than programs written in imperative languages.

2.6 SML / NJ Compiler

The SML/NJ compiler [Appel 1992] 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 inline, except for the allocation of arrays. Optimizations done by the compiler include inlining, passing function arguments in registers, register targeting, constant folding, 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 [Appel 1987; Appel and Jim 1989]. 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 [Kranz et al. 1986]. However, using only a heap simplifies the compiler, the run-time system [Appel 1990], and the implementation of first-class continuations [Hieb et al. 1990]. The decision to use only a heap was controversial because it increases the amount of heap allocation greatly, which is believed to cause poor memory system 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 [1989] examined the data cache behavior of 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. [1990; 1992] argued that cache performance of programs with generational garbage collection will improve substantially when the youngest generation fits in the cache. Koopman et al. [1992] studied the effect of cache organization on combinator graph reduction, an implementation technique for lazy functional programming languages. They observed the importance of a write-allocate policy with subblock placement for improving heap allocation. Zorn [1991] studied the effect 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 when programs are run with mark-and-sweep they have substantially better cache locality than when run with stop-and-copy.

Our work differs from previous work in two ways. First, previous work used the overall miss ratio as the performance metric, which is a misleading indicator of performance. The overall miss ratio neglects the fact that read and write misses may have different costs. Also, the overall miss ratio does not reflect the rates of reads and writes, which may substantially affect performance. We use memory system contribution to CPI as our performance metric, which accurately reflects the effect of the memory system on program running time. Second, previous work did not model the entire memory system: it concentrated solely on caches. Memory system 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 system behavior. We simulate the entire memory system.

Appel [1992] estimated CPI for the SML/NJ system on a single machine using elapsed time and instruction counts. His CPI differs substantially from ours. However, Appel has confirmed our measurements by personal communication and later work [Appel 1993]. The reason for the difference is that instructions were undercounted in his measurements.

Jouppi [1993] 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 architectural 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 many of the write misses.

4. METHODOLOGY

We used trace-driven simulations to evaluate the memory system performance of programs. For 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 or unrepresentative benchmarks can be equally misleading. In this work, much effort has been devoted to addressing these issues.

Section 4.1 describes our trace generation and simulation tools. Section 4.2 states our assumptions and argues that they are reasonable. Section 4.3 describes and characterizes the benchmark programs used in this study. Section 4.4 describes the metrics used to present memory system performance.

4.1 Tools

We extended QPT (Quick Program Profiler and Tracer) [Ball and Larus 1992; Larus 1990; Larus and Ball 1992] to produce memory traces for SML/NJ

programs. QPT rewrites an executable program to produce compressed trace information; QPT also produces a program-specific regeneration program that expands the compressed trace into a full trace. Because QPT operates on the executable program, it can trace the SML code and the garbage collector (which is written in C).

We extended Tycho [Hill and Smith 1989] for the memory system simulations. Our extensions to Tycho include a write-buffer simulator.

We obtained allocation statistics by using an allocation profiler built into SML/NJ. The profiler instruments intermediate code to increment appropriate elements of a *count array* on every allocation. We extended this profiler to count the number of assignments.

4.2 Simplifications and Assumptions

We tried to minimize assumptions which might reduce the validity of our simulations. This section describes the important assumptions we made.

(1) *Simulating Write-Allocate/Subblock Placement with Write-Allocate/No Subblock Placement.* Tycho does not simulate subblock placement so we approximate it by simulating *write-allocate/no subblock* and ignoring the reads from memory that occur on a write miss. This can cause a small inaccuracy in the CPI numbers. The following example illustrates this.

Suppose that the cache block size is 2 words, that the subblock size is 1 word, that a program writes the first word in a memory block, and that the write misses. In subblock placement, the word will be written to the cache, and the second word in the cache block will be invalidated. However, the simplified model will mark both words as valid after the write. Subsequently, if the program reads the second word, the read will incorrectly hit. Thus the CPI reported for caches with subblock placement can be less than the actual CPI. These incorrect hits, however, occur rarely since SML programs tend to do few assignments (see Section 4.3) and since most writes are to sequential locations.

(2) *Ignoring the effects of context switches and system calls.*

(3) *The simulations are driven by virtual addresses.* Some machines such as the SPARCStation II have physically indexed caches, and will have different conflict misses than those reported here.

(4) *Placing code in the text segment instead of the heap.* This improves performance over the unmodified SML/NJ system. It reduces garbage collection costs, since code is never copied, and avoids instruction-cache flushes after garbage collections.

(5) *Used default compilation settings for SML/NJ.* Default compilation settings enable extensive optimization (Section 2.6). Evaluating the impact of these optimizations on cache behavior is beyond the scope of this article.

(6) *Used default garbage collection settings.* The preferred ratio of heap size to live data was set to 5 [Appel 1989]. The softmax, which is the desired upper limit on the heap size, was set to 20MB; the benchmark programs

Table I. Benchmark Programs

Program	Description
CW	The Concurrency Workbench [Cleaveland et al. 1993] is a tool for analyzing networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. The input is the sample session from Section 7.5 of Cleaveland et al. [1993].
Knuth-Bendix	An implementation of the Knuth-Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyzer generator, implemented by James S Mattson and David R. Tarditi [Appel et al. 1989], processing the lexical description of Standard ML.
Life	The game of Life, written by Chris Reade [Reade 1989], running 50 generations of a glider gun. It is implemented using lists.
PIA	The Perspective Inversion Algorithm [Waugh et al. 1990] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" Fortran benchmark [Crowley et al. 1978], translated into ID [Ekanadham and Arvind], and then translated into Standard ML by Lal George.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
YACC	A LALR(1) parser generator, implemented by David R. Tarditi [Tarditi and Appel 1990], processing the grammar of Standard ML.

never reached this limit. The initial heap size was 1MB. We did not investigate the interaction of the sizing strategy and cache size. Understanding these tradeoffs is beyond the scope of this article.

(7) *MIPS as a prototypical RISC machine.* All the traces are for the DECstation 5000/200, which uses a MIPS R3000 CPU.

(8) *All instructions take one cycle with a perfect memory system.* This affects only write-buffer costs, since multicycle instructions give the write buffer more time to retire writes. The inaccuracy introduced by this assumption is negligible, since Section 5.4 shows that write-buffer costs are small.

(9) *Assuming CPU cycle time does not vary with memory organization.* This may not be true, since the CPU cycle time depends on the cache access time, which may differ across cache organizations. For example, a 128K cache may take longer to access than an 8K cache.

4.3 Benchmarks

Table I describes the benchmark programs.[1] *Knuth-Bendix*, *Lexgen*, *Life*, *Simple*, *VLIW*, and *YACC* are identical to the benchmarks measured by Appel [1992]. The description of these benchmarks is copied from Appel [1992]. Table II gives the following for each benchmark: lines of SML code excluding comments and empty lines, maximum heap size, compiled-code size, and user-mode CPU time on a DECStation 5000/200. The code size includes 207KB for standard libraries, but does not include the garbage

[1]Available from the authors.

Table II. Sizes of Benchmark Programs

Program	Size			Run Time	
	Lines	Heap Size (KB)	Code Size (KB)	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 III. Characteristics of Benchmark Programs

Program	Inst Fetches	Read (%)	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

collector and other run-time support code, which is about 60KB. The run times are the minimum of five runs.

Table III characterizes the benchmark programs according to the number and kinds of memory references they do. All numbers are reported as a percentage of instructions. The *Reads*, *Writes*, and *Partial writes* columns list the reads, full-word writes, and partial-word writes done by the program *and* the garbage collector; the *assignments* column lists the noninitializing writes done by the program only. The *Nops* column lists the nops executed by the program and the garbage collector. All the benchmarks have long traces; most related works use traces that are an order of magnitude smaller. Also, the benchmark programs do few assignments; the majority of the writes are initializing writes.

Table IV 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,[2] records, and others (includes spill records, arrays, strings, vectors, ref cells, store list records, and

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

Table IV. Allocation Characteristics of Benchmark Programs

Program	Allocation (words)	Escaping %	Escaping Size	Known %	Known Size	Callee Saved %	Callee Saved Size	Records %	Records Size	Other %	Other Size
CW	56,467,440	4.0	4.12	3.3	15.39	67.2	6.20	19.5	3.01	6.0	4.00
Knuth-Bendix	67,733,930	37.6	6.60	0.1	15.22	49.5	4.90	12.7	3.00	0.1	15.05
Lexgen	33,046,349	3.4	6.20	5.4	12 96	72.7	6.40	15.1	3.00	3.7	6.97
Life	37,840,681	0.2	3.45	0.0	15.00	77.8	5.52	22.2	3 00	0.0	10.29
PIA	18,841,256	0.4	5.56	28.0	11.99	25.0	4.69	12.7	3.41	33 9	3.22
Simple	80,761,644	4.0	5.70	1.1	15.33	68.1	6.43	8.3	3.00	18 5	3.41
VLIW	59,497,132	9.9	5.22	6.0	26.62	61.8	7.67	20.3	3.01	2.1	2.60
YACC	17,015,250	2.3	4.83	15.3	15.35	54.8	7.44	23.7	3.04	4.0	10.22

Table V Timings of Memory Operations

Task	Timings (Cycles)
Non-page-mode write	5
Page-mode write	1
Partial-word write	11
Page-mode flush	4
Read 16 bytes from memory	15
Read 32 bytes from memory	19
Refresh period	195
Refresh time	5
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
TLB miss	28

floating-point numbers). For each allocation kind, the % column gives the total words allocated for objects of that kind as a percentage of total allocation, and the *Size* column gives the average size in words, including the one-word tag, of an object of that kind.

4.4 Metrics

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

Table V lists the timings 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. In addition to the times in Table V, all reads and writes may also incur write-buffer penalties. In an actual implementation, there may be a one-cycle penalty for write misses in caches with subblock placement. This is because a tag needs to be written to the cache after the miss is detected. This does not change our results, since it adds at most 0.02–0.05 to the CPI of caches with subblock placement.

We used a DRAM page size of 4K in the simulation of page-mode writes. Page-mode flush is the number of cycles needed to flush the write pipeline after a series of page-mode writes.

TLB data is reported as the TLB miss contribution to the CPI. This metric is used instead of just CPI to allow us to present the measurements for all the benchmarks in one chart. A virtual memory page size of 4K was used in the simulations.

5. RESULTS AND ANALYSIS

In Section 5.1 we present a qualitative analysis of the memory behavior of programs compiled with SML/NJ. In Section 5.2 we list the cache and TLB configurations simulated and explain why they were selected. In Sections 5.3, 5.4, and 5.5 we present data for memory system performance, write-buffer performance, and TLB performance. In Section 5.6 we validate the simulations. In Section 5.7 we present an analytical model that extends these results to programs with different allocation behavior.

5.1 Qualitative Analysis

Recall from Section 2 that SML/NJ uses a copying collector. The most-important property of a copying collector with respect to memory system behavior is that allocation initializes memory in an area that has not been touched since the last garbage collection. This means that for caches that are not large enough to contain the allocation area there will be many write misses. The slowdown that these write misses translate to depends on the memory system organization.

Recall from Section 4.3 that SML/NJ programs have the following important 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 system organizations where the CPU has to wait for a write to be written through (or back) to memory will perform poorly. Even memory systems where the CPU does not need to wait for writes if they are issued far apart (e.g., two cycles apart in the HP 9000 series 700) may perform poorly due to the bunching of writes. This means that the memory system needs two features. 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 system must avoid reading a cache block from memory if it is going to be written before being read. Of course, this requirement only holds for caches with a *write-allocate* policy. Subblock placement [Koopman et al. 1992], a block size of one word, and the ALLOCATE instruction [Peng and Sohi 1989] can achieve this. Since the effects on cache performance of these features are similar, we discuss only subblock placement. For large caches, when the allocation area fits in the cache and

Table VI. Memory System Organizations Studied

Write Policy	Write Miss Policy	Subblocks	Assoc	Block Size	Cache Sizes	Write Buffer	Page mode
through	allocate	yes	1, 2	16, 32 bytes	8K-512K	1–6 deep	yes
through	allocate	no	1, 2	16, 32 bytes	8K-512K	6 deep	no
through	no allocate	no	1, 2	16, 32 bytes	8K-512K	6 deep	no

Table VII. Memory System Organization of Some Popular Machines

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

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 16 bytes. This is stronger than subblock placement since it has a full tag on every "subblock."

thus there are few write misses, the benefit of subblock placement will be reduced.

5.2 Cache and TLB Configurations Simulated

The design space for memory systems is enormous. There are many variables involved, and the dependencies among them are complex. Therefore we could study only a subset of the memory system design space. In this study, we restrict ourselves to features found in *currently popular* RISC workstations [Cypress 1990; DEC 1990a; 1990b; Slater 1991]. Table VI summarizes the cache organizations simulated. Table VII lists the memory system organizations of 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 700 series), there are some exceptions (notably SPARCStations).

We report data only for write-through caches, but the CPI for write-back caches can be inferred from the data for write-through caches. While write-through and write-back caches have identical misses, their contribution to the CPI may differ for two reasons.

First, a write hit or miss in a write-back cache may take one cycle more than in a write-through cache. A write-back cache must probe the tag *before* writing to the cache [Jouppi 1993], unlike a write-through cache. It is easy to adjust the data for write-through caches for this to obtain the data for write-back caches. If the program has w writes and n useful instructions, then the CPI for a write-back cache can be obtained by adding w/n to the CPI of the write-through cache with the same size and configuration. For

VLIW w/n is 0.18. Second, write-through and write-back caches may have different write-buffer penalties because they do writes to main memory with different frequencies and at different points. We expect the write-buffer penalties for write-back caches to be smaller than those for write-through caches since writes to main memory are less frequent for write-back caches than for write-through caches. This difference between write-through and write-back caches is likely to be negligible since the write-buffer penalty is small even for write-through caches.

We simulated fully associative, unified TLBs from 1 to 64 entries with an LRU replacement policy. Some machines (such as the HP 9000 series) have separate instruction and data TLBs. From Section 5.5 it is clear that for the benchmarks even small unified TLBs perform well.

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.

5.3 Memory System Performance

We present memory system performance in summary graphs and breakdown graphs. Each summary graph summarizes the performance of one benchmark program for a range of cache sizes (8K to 512K), write-miss policies (write-allocate or write-no-allocate), subblock placement (with or without), and associativity (1 or 2). Each curve in a summary graph corresponds to a different memory system 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 system overhead into its components for one configuration in a summary graph. The write-buffer depth in these graphs is fixed at six entries.

In this section we present only the summary graphs for VLIW (Figure 2). The data for other programs are similar and are given in the Appendix. Figures 3, 4, and 5 are the breakdown graphs for VLIW for the 16-byte block size configurations; the remaining breakdown graphs for VLIW are similar and omitted for conciseness. The breakdown graphs for the other benchmarks are similar (and predictable from the summary graphs) and are thus omitted for the same reason. The full set of graphs is available from the authors.

In the summary graphs, the *nops* curve is the base CPI: the total number of instructions executed divided by the number of useful (not nop) instructions executed; this corresponds to the CPI for a perfect memory system. For the breakdown graphs, the *nop* area is the CPI contribution of nops; *read miss* is the CPI contribution of read misses; *write miss* is the CPI contribution of write misses (if any); *inst fetch 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.

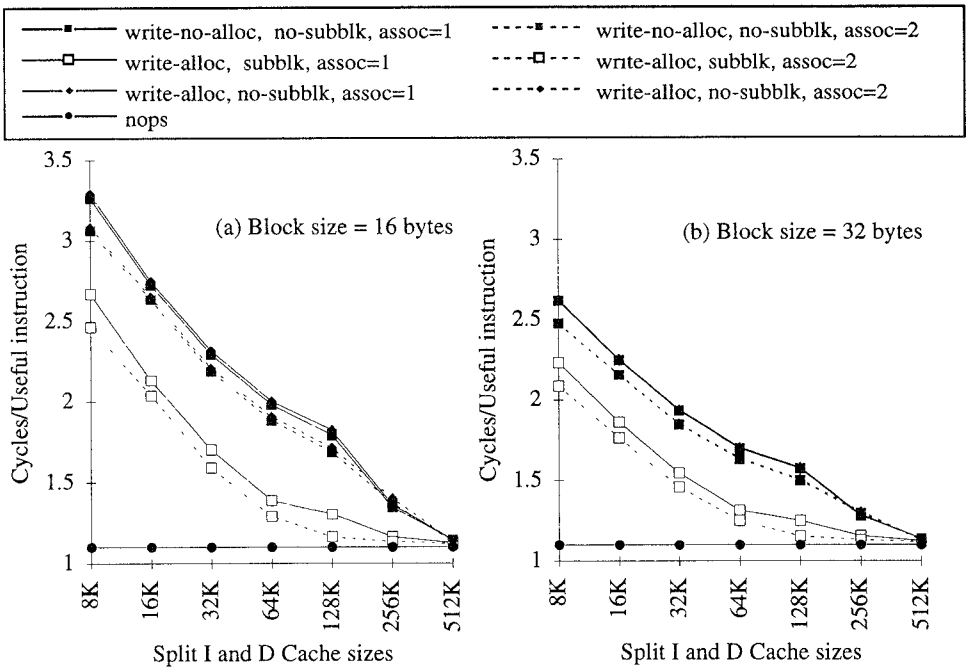

Fig. 2. VLIW summary.

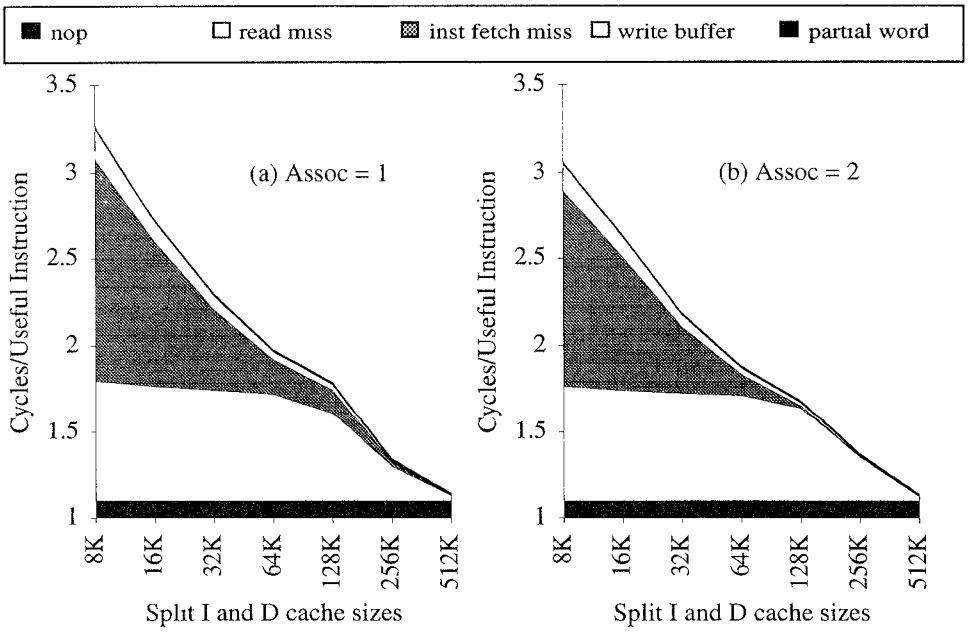

Fig. 3. VLIW breakdown, write-no-allocate, no subblock, block size = 16.

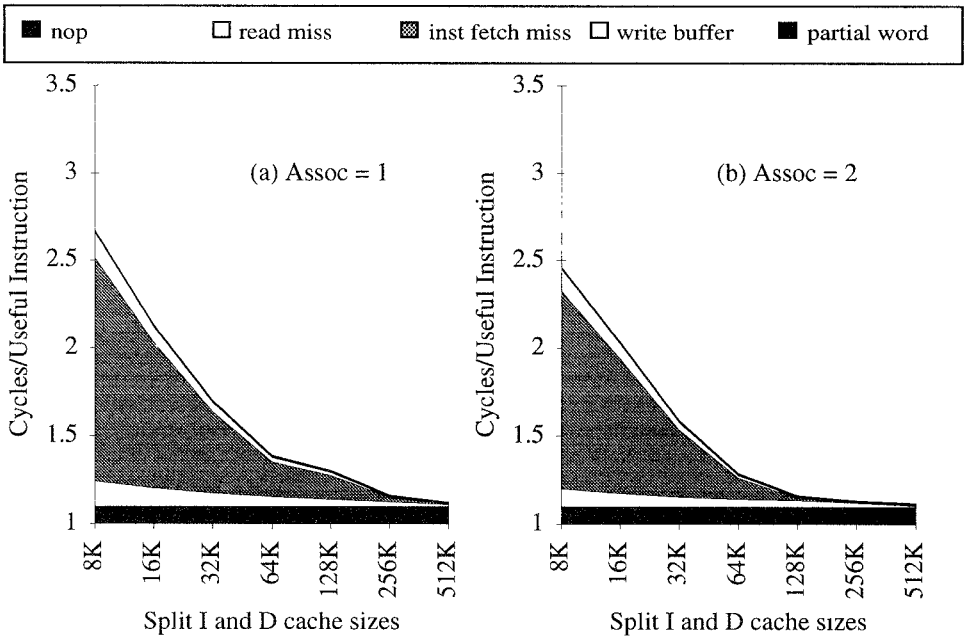

Fig. 4. VLIW breakdown, write-allocate, subblock, block size = 16.

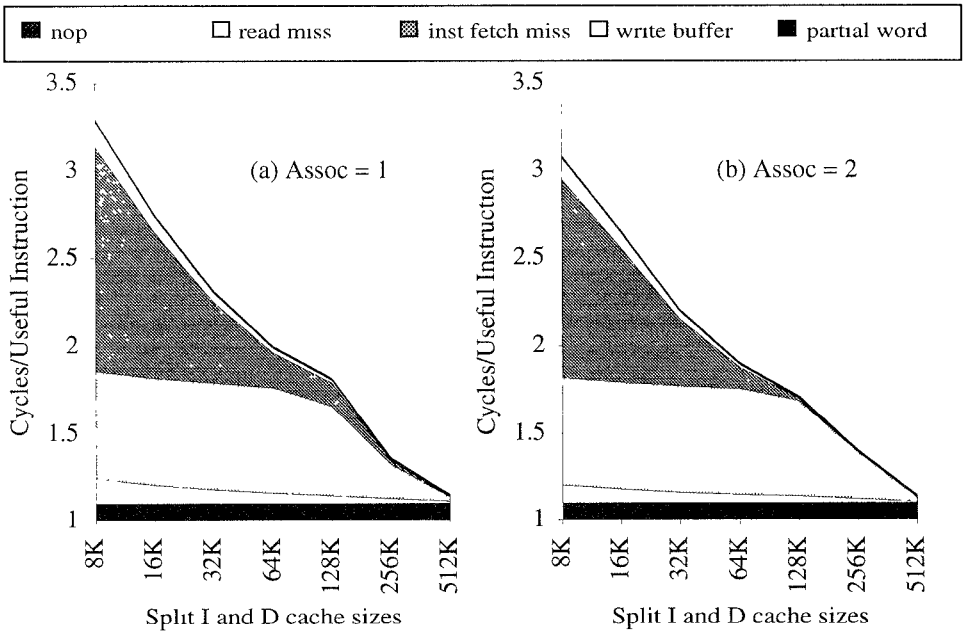

Fig. 5. VLIW breakdown, write-allocate, no subblock, block size = 16.

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

In the following subsections 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 system performance of the benchmark programs.

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*; it substantially outperforms all other configurations. For a memory system with 64K direct-mapped write-allocate caches and four-word blocks, subblock placement reduces the CPI by 0.35 to 0.88, with an arithmetic mean improvement of 0.55. Surprisingly, for sufficiently large caches with the *write-allocate/subblock placement* organization, the memory system performance of SML/NJ programs is acceptable; the overhead due to data cache misses ranges from 3 to 13% (arithmetic mean 9%) for 64K direct-mapped caches and 1 to 13% (arithmetic mean 9%) for 32K two-way associative caches. Recall that the 64K direct-mapped configuration corresponds to the DECStation 5000/200 memory system. The memory system overhead of SML/NJ programs on the DECStation 5000/200 is similar to that of C and Fortran programs [Chen and Bershad 1993]. It is worth emphasizing that the memory system performance of SML/NJ programs is *good* on some current machines *despite the very high miss rates*; for a 64K cache with a block size of 16 bytes, the write-miss and read-miss ratios for VLIW are 0.23 and 0.02 respectively.

Recall that in Section 5.1 we argued that the benefit of subblock placement would be substantial, but that the benefit would decrease for larger caches. The summary graphs indicate that the reduction in benefit is not substantial even for 128K cache sizes; however, the benefit of subblock placement decreases sharply for larger caches for six of the benchmark programs. This suggests that the allocation area size of six of the benchmark programs is between 256K and 512K.

The performance of *write-allocate/no subblock* is almost identical to that of *write-no-allocate/no subblock* (Knuth-Bendix is an exception). The difference between them is so small in most graphs that the two curves overlap. This suggests that an address is being read 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 the programs allocate 0.4–0.9 bytes per instruction, our results suggest that a read of a block occurs within 9K–20K instructions of its being written.

The benefit of subblock placement is not limited to strongly functional languages such as Standard ML. Jouppi [1993] reports that subblock placement combined with an 8K data cache and a 16-byte cache line eliminates 31% of the memory references for C programs. Reinhold [1993] finds that the memory performance of Scheme programs is good with subblock placement.

5.3.2 *Changing Associativity.* From Figure 2 we see that increasing associativity improves all organizations. The improvement in going from one-way to two-way set associativity is much smaller than the improvement obtained from subblock placement. For a memory system with 64K write-allocate caches and four-word blocks, increasing associativity reduces the CPI by 0.01 to 0.09, with an arithmetic mean improvement of 0.06. 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 [Hill 1988].

From Figures 3, 4, and 5 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), 4(a), and 5(a)) the instruction cache penalty is substantial for 128K or smaller caches. For caches with subblock placement, the instruction cache penalty can dominate the penalty for the memory system. 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 large enough to hold the working set. When the benchmark programs are examined, the performance of the instruction cache is not surprising: the code consists of small functions with frequent calls, which lowers 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 a memory system with 64K direct-mapped write-allocate caches, increasing block size reduces the CPI by 0.14 to 0.35, with an arithmetic mean improvement of 0.22. For the *write-allocate* organizations, 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 [Koopman et al. 1992]. 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 of the reads is comparable to that of the writes.

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

5.3.4 *Changing Cache Size.* Three distinct regions of performance can be identified for cache sizes. The first region corresponds to the range of cache

sizes when the allocation area does not fit in the cache (i.e., allocation happens in an area of memory which is not cache resident). For most of the benchmarks, this region corresponds to cache sizes of less than 256K (for Simple and Knuth-Bendix this region extends beyond 512K). In this region, increasing the cache size uniformly improves performance for all configurations. However, the performance improvement from doubling the cache size is small.

From the breakdown graphs we see that in the first region 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, small improvements due to increasing the cache size may not be achieved in practice.

The second region ranges from when the allocation area begins to fit in the cache until the allocation area fits in the cache. For most of the benchmarks (once again excepting Simple and Knuth-Bendix), this region corresponds to cache sizes in the range 256K to 512K.[3] In this region, increasing the cache size sharply improves the data cache performance for memory organizations without subblock placement. However, increasing the cache size in this region has little to offer for instruction cache performance because the instruction cache miss penalty is already low at this point.

The third region corresponds to cache sizes when the allocation area fits in the cache. For five of the benchmarks, this region corresponds to caches larger than 512K (for Lexgen, Knuth-Bendix, and Simple this region starts at larger cache sizes). In this range, increasing the cache size has little or no impact on memory system performance because everything remains cache resident, and thus there are no capacity misses to eliminate.

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

5.4 Write-Buffer Depth

In Section 5.3.5 we showed that a six-deep write buffer coupled with page-mode writes was able to absorb the bursty writes in SML/NJ programs. In this section we explore the impact of write-buffer depth on the write-buffer contribution to CPI. Since the speed at which the write buffer can retire writes depends on whether or not the memory system has page-mode writes, we conducted two sets of experiments: one with and the other without page-mode writes. We varied the write-buffer depth from 1 to 6. We conducted this study for two of the larger benchmarks: CW and VLIW. We fixed

[3] For Lexgen this region extends a little beyond 512K.

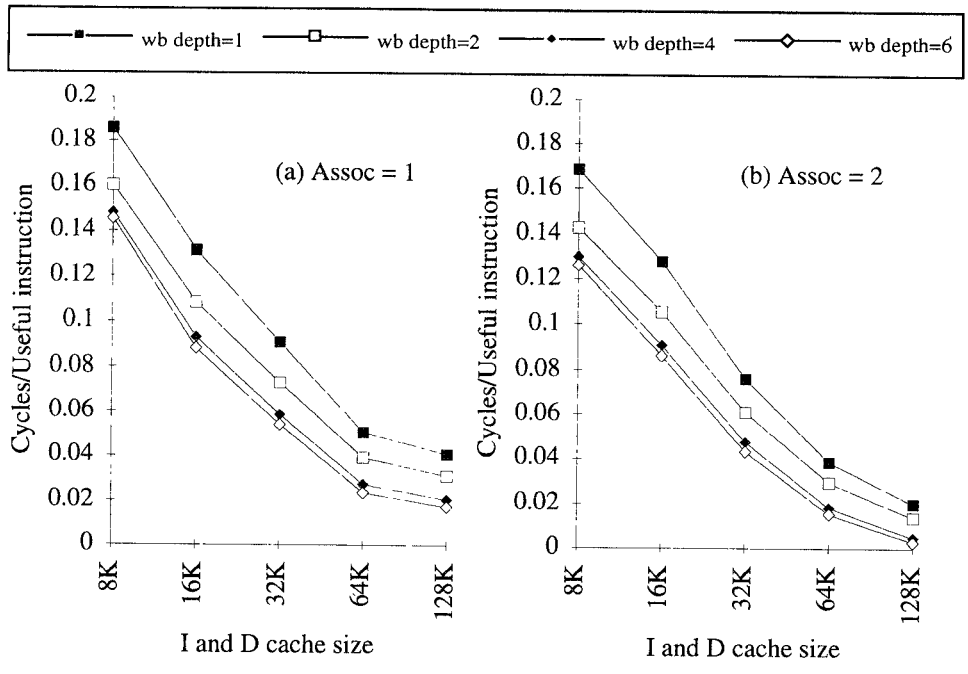

Fig. 6. Write-buffer CPI contribution for VLIW, with page-mode writes.

the block size at 16 bytes and the write miss policy at *write-allocate/subblock placement*.

Figure 6 gives the write-buffer costs for VLIW with caches of associativity one and two and in a memory system with page-mode writes; Figure 7 does the same in a memory system without page-mode writes. The graphs plot the CPI contribution of the write buffer against cache size; there is one curve for each write-buffer depth. Increasing the cache size or associativity reduces the number of read and instruction-fetch misses, and thus reduces the number of main-memory transactions. Reducing the number of main-memory transactions increases the effectiveness of the write buffer since the write buffer fills up less frequently and has more cycles in which to retire its writes (Section 2.1).

In memory systems with page-mode writes (Figure 6), the difference between the CPI contribution of a one-deep write buffer and a six-deep write buffer is less than 0.05. This is surprisingly small considering the burstiness of the writes. This is due to the effectiveness of page-mode writes. An example illustrates this.

Suppose that a program is allocating and initializing a four-word object and that the write buffer is one-deep. Further suppose that the write buffer is empty and that the instructions doing the allocation all hit in the instruction cache. The first write does not stall the CPU since the write buffer is empty. At the next write one cycle later the write buffer is full, and the CPU stalls. After four cycles (see Table V) the write is placed in the write buffer. This

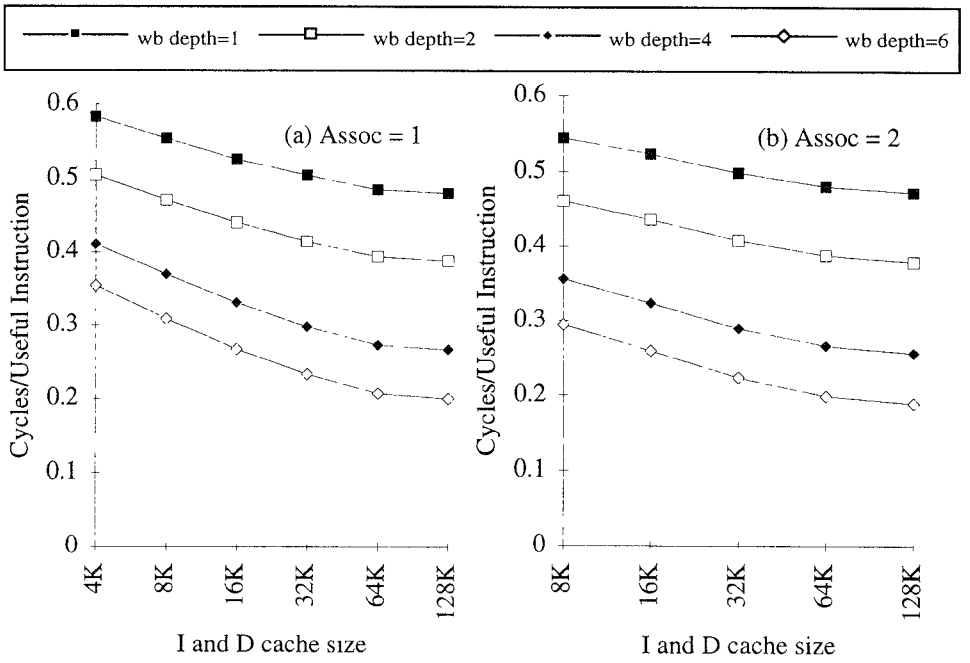

Fig. 7. Write-buffer CPI contribution for VLIW, without page-mode writes.

write, however, is likely to be on the same DRAM page as the previous write, since it is to the next address. It will therefore complete in one cycle. All subsequent writes to initialize this object find an empty write buffer since they also complete in one cycle due to page-mode writes.

Due to sequential allocation, it is likely that writes to initialize objects allocated one after another will also be on the same DRAM page. In the best case, with no read misses and refreshes, a *write-buffer-full* delay will happen only once per N words of allocation, where N is the size of the DRAM page. Thus, the write buffer depth has little effect on the performance of SML/NJ programs if the memory system has page-mode writes. To confirm this explanation, we measured the probability of two consecutive writes being on the same DRAM page. This probability averaged over the benchmarks was 96%.

The small impact of write-buffer depth on performance does not imply that a write buffer is useless if the memory system has page-mode writes. Instead, it says that a deep write buffer offers little performance improvement in a memory system with page-mode writes if the programs have *strong spatial locality* in the writes and if the majority of the reads and instruction fetches hit in the cache. *Strong spatial locality* means that the probability that two consecutive writes are to the same DRAM page is high.

Write-buffer depth is important, however, if the memory system does not have page-mode writes (Figure 7). In this case, a six-deep write buffer

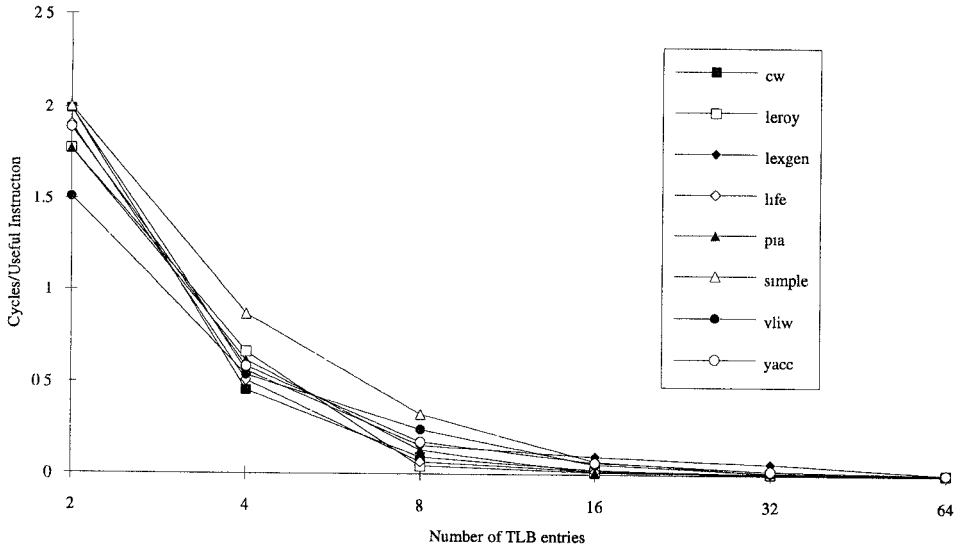

Fig. 8. TLB contribution to CPI.

performs much better than a one-deep write buffer. Note that Figures 6 and 7 have different scales.

5.5 TLB Performance

Figure 8 gives the TLB miss contribution to the CPI for each benchmark program. We see that the CPI contribution of TLB misses falls below 0.01 for all our programs for a 64-entry unified TLB; for half the benchmarks, it is below 0.01 even for a 32-entry TLB.

5.6 Validation

To validate our simulations, we ran each of the benchmarks five times on a DECStation 5000/200 (running Mach 2.6) and measured the elapsed *user time* for each run. The programs were run on a lightly loaded machine but not in single-user mode. The simulations with *write-allocate/subblock placement*, 64K direct-mapped caches, 16-byte blocks, and 64-entry TLB correspond closely to the DECStation 5000/200 with the following three important differences. First, the simulations ignored the effects of context switches and system calls. Second, the simulations assumed a *virtual address = physical address* mapping which can have many fewer conflict misses than the random mapping used in Mach 2.6 [Kessler and Hill 1992]. Third, the simulations assume that all instructions take exactly one cycle (plus memory system overhead).

In order to minimize the memory system effects of the virtual-to-physical mapping and context switches, we took the minimum CPI of the five runs for each program and compared it to the CPI obtained via simulations. We present our findings in Table VIII; *Measured (sec)* is the user time of the

Table VIII. Measured versus Simulated

Program	Measured (sec)	Measured CPI	Simulated CPI	Multicycle CPI	Discrepancy (%)
CW	25 83	1 42	1.39	0.00	2.48
Knuth-Bendix	14.95	1.27	1.21	0.00	5.22
Lexgen	16.13	1.40	1.31	0.00	6.29
Life	17.16	1.23	1.21	0.00	1.19
PIA	6.41	1.43	1.18	*	17.62
Simple	29.81	1.33	1.21	*	9.03
VLIW	25.61	1.76	1.39	0.20	9.66
YACC	6.58	1.39	1.36	0.00	2.20

*cannot be determined without simulating the CPU and/or FP unit pipelines.

program in seconds; *Measured CPI* is the CPI obtained from the measured time; *Simulated CPI* is the CPI obtained from the simulations; *Multicycle CPI* is the overhead of multicycle instructions when it could be accurately computed; *Discrepancy* is the discrepancy between the simulated CPI plus the multicycle CPI and the measured CPI as a percentage of measured CPI.

Table VIII shows that with the exception of PIA, the discrepancy is less than 10% and that the actual runs validate the simulations. The discrepancy in PIA is due to multicycle instructions which comprise 4.8% of the total instructions executed. Since multicycle instructions do not cause a stall until their result is used, their cost can usually be determined only by simulations. We were able to determine the overhead of multicycle instructions accurately for VLIW since the results of most multicycle instructions are used immediately afterward. In the case of PIA, the distance between multicycle instructions and their use varies considerably. However, even if each multicycle instruction stalls the CPU for half its maximum latency, the discrepancy falls well below 10%. Thus, multicycle instructions can explain the discrepancy for PIA.

5.7 Extending the Results

Section 5.3 demonstrated that heap allocation can have a significant memory system cost if new objects cannot be allocated directly into the cache. In this section, we present an analytic model which predicts the memory system cost due to heap allocation when this is the case. The model formalizes the intuition presented in Section 5.1 and predicts the memory system cost due to heap allocation when block sizes, miss penalties, or heap allocation rates change. We use the model to speculate about the memory system cost of heap allocation for caches without subblock placement if SML/NJ were to use a simple stack.

5.7.1 *An Analytic Model.* Recall that heap allocation with copying garbage collection allocates memory which typically has not been touched in a long time and is unlikely to be in the cache. This is especially true when the allocation area does not fit in the cache. When newly allocated memory is initialized, write misses occur. The rate of write misses depends on the

Table IX. Percentage Difference between Analytical Model and Simulations

Cache Size (Kilobytes)	Write-No-Alloc/No Subblock (%)	Write-Alloc/No Subblock (%)
8K	7.12	2.4
16K	6.84	2.2
32K	7.02	2.2
64K	10.8	5.7
128K	31.8	23.5
256K	128.8	111.4
512K	1847.7	1746.2

allocation rate (a words/instruction) and the block size (b words). Given the rate of write misses, we can calculate the memory system cost, C, due to heap allocation. The read and write miss penalties are r_p and w_p respectively.

Under the assumption that the allocation area does not fit in the cache, i.e., initializing writes miss, we have

$$C_{\text{write alloc}} = w_p * a/b.$$

Under the additional assumption that programs touch data soon after it is allocated, we have

$$C_{\text{write no alloc}} = r_p * a/b.$$

Since the benchmarks do few assignments, the cost of heap allocation should account for the difference in CPIs when the write-miss policy is varied. Hence,

$$C_{\text{write alloc/no subblock}} \approx CPI_{\text{write alloc/no subblock}} - CPI_{\text{write alloc/subblock}}$$

$$C_{\text{write no alloc/no subblock}} \approx CPI_{\text{write no alloc/no subblock}} - CPI_{\text{write alloc/subblock}}$$

Table IX shows the average (arithmetic mean) difference between the predicted cost, C, and the actual difference in CPIs, as a percentage of the actual difference in CPIs. The values used to calculate Table IX were: $b = 4$, $r_p = 15$, and $w_p = 15$.

The model is accurate for cache sizes of 128K or less, when the allocation area does not fit in the cache. As expected, the model is inaccurate when the allocation area fits in the cache. The percentage difference heads toward infinity as the benefit of subblock placement becomes negligible. Thus, this model can be used to predict the memory system cost of heap allocation for small cache sizes.

5.7.2 *SML/NJ with a Stack.* We can speculate about the memory system cost of heap allocation in SML/NJ when a stack is used using this model. In the absence of first-class continuations, which the benchmarks do not use, callee-save continuations can be stack-allocated easily. The callee-save continuations correspond to procedure activation records. The first two columns of Table X give the rate of heap allocation with and without heap allocation of callee-save continuations.

Table X. Expected Memory System Cost of Heap Allocation for Caches without Subblock Placement (assuming procedure activation records are stack-allocated in SML/NJ)

Program	Allocation Rate Including Callee-Save Conts. (Words/Useful Instruction)	Allocation Rate Excluding Callee-Save Conts. (Words/Useful Instruction)	C (Cycles/Instruction)
CW	0.12	0.04	0.15
Knuth-Bendix	0.23	0 12	0 44
Lexgen	0.11	0.03	0 12
Life	0.11	0.02	0.09
PIA	0.17	0 13	0.47
Simple	0.14	0.05	0.17
VLIW	0.16	0.06	0.23
YACC	0.14	0.07	0.24
Median	0.14	0.05	0.20

Assuming only continuations are stack-allocated, column 3 of Table X presents an estimate of the memory system cost of heap allocation for caches that do not have subblock placement and are too small to hold the allocation area. The block size is 16 bytes, the read-miss penalty 15 cycles, and the write-miss penalty for the no-subblock caches 15 cycles. Since the read and write miss penalties are the same, C is the same for write-allocate and write-no-allocate organizations.

This is an upper bound on the expected memory system cost of heap allocation with a stack because it may be possible to stack-allocate additional objects [Kranz et al. 1986]. We see that even with a simple stack, the memory system costs due to heap allocation for caches without subblock placement will probably be significant for SML/NJ programs.

6. FUTURE WORK

We suggest three directions in which this study can be extended:

—measuring the impact of other architectural features not explored in this work,

—measuring the impact of different compilation techniques, and

—measuring other aspects of programs.

Regarding architectural features, there is a need to explore memory system performance of heap allocation on newer machines. As CPUs get faster relative to main memory, memory system performance becomes even more crucial to good performance. To address the increasing discrepancy between CPU speeds and main-memory speeds, newer machines, such as Alpha workstations [DEC 1992], often have features such as secondary caches, stream buffers, and register scoreboarding. The interaction of these features with heap allocation needs to be explored.

Regarding different compilation techniques, the impact of stack allocation is worth measuring. A stack reduces heap allocation (which performs badly on most memory system organizations) in favor of stack allocation (which can

have good cache locality since it focuses most of the references to a small part of memory, namely the top of the stack). For SML/NJ programs, the majority of heap-allocated objects can be allocated on the stack (Table IV). Therefore stack allocation can substantially improve performance of SML/NJ programs on memory organizations without subblock placement or with small cache sizes. However, stack allocation can slow down exceptions, first-class continuations, and threads. A careful study is needed to evaluate the pros and cons of doing stack allocation.

Regarding measuring other aspects of programs, several areas seem promising for future work:

(1) Measuring the impact of different garbage collection algorithms on cache performance.
(2) Measuring the impact of changing various garbage collector parameters (such as allocation area size) on cache performance.
(3) Measuring the cost of various operations related to garbage collection: tagging, store checks, and garbage collection checks. A preliminary study of this is reported in Tarditi and Diwan [1993].
(4) Measuring the impact of optimizations on cache performance.

7. CONCLUSIONS

We have studied the memory system performance of heap allocation with copying garbage collection, a general automatic storage management technique for programming languages. Heap allocation is useful for implementing language features where objects may have indefinite extent, such as list-processing, higher-order functions, and first-class continuations. However, heap allocation is widely believed to lead to poor memory system performance [Peng and Sohi 1989; Wilson et al. 1990; 1992; Zorn 1991]. This belief is based on the high (write) miss ratios that occur when new objects are allocated and initialized.

We studied the memory system performance of mostly functional SML programs compiled with the SML/NJ compiler. These programs heap allocate at intensive rates. They use heap-only allocation: all allocation, including activation records, is done on the heap. We simulated a wide variety of memory systems typical of current workstations.

To our surprise, we found that heap allocation performed well on some memory systems. In particular, on an actual machine (the DECStation 5000/200), the memory system performance of heap allocation was good. However, heap allocation performed poorly on most memory system organizations. The memory system property crucial for achieving good performance was the ability to allocate and initialize a new object into the cache without a penalty. This can be achieved by having subblock placement or a cache large enough to hold the allocation area, along with fast page-mode writes or a sufficiently deep write buffer.

We found for caches with subblock placement, the arithmetic mean of the data cache penalty was under 9% for 64K or larger caches; for caches without

Table XI. CPI with a Perfect Memory System

Program	CPI
CW	1.15
Knuth-Bendix	1.06
Lexgen	1.14
Life	1.18
PIA	1.09
Simple	1.08
VLIW	1.10
YACC	1.13

subblock placement, the mean of the data cache penalty was often higher than 50%. We also found that a cache size of 512K allowed the allocation area for six of the benchmark programs to fit in the cache, which substantially improved the performance of cache organizations without subblock placement.

The implications of these results are clear. First, a stack is not needed to achieve good memory system performance. Given the right memory system, heap allocation of procedure activation records can also have good memory system performance. Heap allocation can be used without a performance penalty in place of stack allocation, even though it is a more-general storage management technique. Second, computer architects can better support languages which make heavy use of dynamic storage allocation on machines with small primary caches by using subblock placement with a subblock size of one word.

APPENDIX. SUMMARY TABLES

Table XI gives the CPI of the benchmark programs for a perfect memory system. Table XII gives the CPI for each of the benchmark programs for the different memory organizations. The following abbreviations are used in the CPI table:

wa/wn: write allocate/write no allocate
s/ns: subblock placement/no subblock placement
4/8: Block size = 4 words/Block size = 8 words.

ACKNOWLEDGMENTS

We thank Peter Lee for his encouragement and advice during this work. We thank E. Biagioni, B. Chen, O. Danvy, A. Forin, U. Hoelzle, K. McKinley, E. Nahum, D. Stefanović, and D. Stodolsky for comments on drafts of this article, B. Milnes and T. Dewey for their help, J. Larus for creating QPT, M. Hill for creating Tycho, and A. W. Appel, D. MacQueen, and many others for creating SML/NJ.

Table XII. Cycles per Useful Instructions

	Associativity = 1							Associativity = 2						
Config	8K	16K	32K	64K	128K	256K	512K	8K	16K	32K	64K	128K	256K	512K
							CW							
wn,ns,4	2.41	2.07	1.88	1.73	1.43	1.30	1.18	2.22	1.96	1.78	1.62	1.42	1.30	1.17
wa,ns,4	2.44	2.09	1.90	1.74	1.44	1.31	1.18	2.24	1.98	1.79	1.63	1.43	1.31	1.17
wa,s,4	1.96	1.62	1.44	1.39	1.24	1.20	1.17	1.77	1.50	1.33	1.25	1.21	1.18	1.16
wn,ns,8	2.18	1.89	1.72	1.60	1.37	1.27	1.18	1.98	1.76	1.62	1.50	1.35	1.26	1.17
wa,ns,8	2.19	1.89	1.72	1.60	1.37	1.27	1.18	1.98	1.76	1.61	1.49	1.35	1.26	1.17
wa,s,8	1.88	1.59	1.43	1.38	1.23	1.20	1.17	1.68	1.46	1.32	1.25	1.21	1.18	1.16
							Leroy							
wn,ns,4	2.32	2.18	1.96	1.87	1.79	1.65	1.37	2.17	2.03	1.89	1.82	1.76	1.65	1.40
wa,ns,4	2.53	2.40	2.18	2.08	1.98	1.83	1.50	2.38	2.25	2.09	2.00	1.95	1.83	1.57
wa,s,4	1.66	1.52	1.30	1.20	1.15	1.12	1.09	1.51	1.37	1.21	1.12	1.10	1.09	1.08
wn,ns,8	2.05	1.93	1.75	1.67	1.60	1.50	1.30	1.92	1.81	1.68	1.61	1.57	1.50	1.33
wa,ns,8	2.12	2.00	1.82	1.73	1.66	1.56	1.35	1.99	1.87	1.74	1.67	1.63	1.55	1.38
wa,s,8	1.56	1.44	1.26	1.18	1.13	1.11	1.09	1.43	1.32	1.19	1.11	1.09	1.08	1.07
							Lexgen							
wn,ns,4	3.59	1.94	1.84	1.72	1.65	1.50	1.31	2.04	1.85	1.70	1.64	1.60	1.50	1.34
wa,ns,4	3.61	1.96	1.85	1.74	1.67	1.51	1.31	2.06	1.87	1.71	1.66	1.61	1.51	1.35
wa,s,4	3.17	1.53	1.42	1.31	1.27	1.21	1.19	1.62	1.43	1.28	1.22	1.20	1.19	1.18
wn,ns,8	2.96	1.78	1.67	1.57	1.51	1.39	1.27	1.86	1.71	1.56	1.49	1.45	1.38	1.28
wa,ns,8	2.97	1.79	1.68	1.57	1.51	1.40	1.27	1.87	1.71	1.57	1.50	1.46	1.39	1.28
wa,s,8	2.70	1.51	1.41	1.30	1.26	1.21	1.19	1.60	1.44	1.30	1.22	1.20	1.18	1.18
							Life							
wn,ns,4	1.79	1.70	1.65	1.62	1.61	1.42	1.20	1.77	1.64	1.61	1.60	1.60	1.48	1.19
wa,ns,4	1.80	1.70	1.65	1.62	1.60	1.42	1.20	1.74	1.64	1.60	1.60	1.59	1.48	1.19
wa,s,4	1.39	1.29	1.24	1.21	1.20	1.19	1.19	1.33	1.23	1.20	1.19	1.19	1.19	1.18
wn,ns,8	1.65	1.55	1.49	1.47	1.46	1.34	1.19	1.57	1.54	1.46	1.45	1.44	1.37	1.19
wa,ns,8	1.65	1.55	1.50	1.47	1.45	1.34	1.19	1.57	1.51	1.45	1.45	1.44	1.37	1.19
wa,s,8	1.39	1.29	1.24	1.21	1.20	1.19	1.19	1.31	1.25	1.20	1.19	1.19	1.18	1.18
							Pia							
wn,ns,4	2.23	1.93	1.80	1.75	1.62	1.34	1.12	2.23	1.83	1.73	1.72	1.63	1.36	1.12
wa,ns,4	2.27	1.96	1.82	1.77	1.65	1.36	1.12	2.26	1.85	1.76	1.74	1.66	1.39	1.12
wa,s,4	1.66	1.36	1.22	1.18	1.15	1.13	1.10	1.66	1.25	1.15	1.14	1.12	1.11	1.10
wn,ns,8	1.92	1.70	1.59	1.55	1.46	1.26	1.11	1.87	1.60	1.53	1.51	1.45	1.28	1.11
wa,ns,8	1.95	1.72	1.60	1.56	1.47	1.27	1.11	1.89	1.61	1.54	1.52	1.47	1.30	1.11
wa,s,8	1.55	1.32	1.21	1.17	1.14	1.12	1.10	1.49	1.22	1.14	1.13	1.11	1.10	1.10
							Simple							
wn,ns,4	2.32	2.02	1.79	1.73	1.70	1.68	1.62	1.98	1.74	1.70	1.70	1.68	1.66	1.63
wa,ns,4	2.35	2.05	1.81	1.75	1.72	1.70	1.64	2.01	1.76	1.72	1.72	1.69	1.68	1.65
wa,s,4	1.80	1.50	1.26	1.21	1.19	1.18	1.15	1.46	1.21	1.18	1.17	1.16	1.15	1.14
wn,ns,8	2.03	1.79	1.57	1.51	1.49	1.47	1.43	1.72	1.52	1.49	1.48	1.47	1.46	1.44
wa,ns,8	2.05	1.80	1.58	1.53	1.50	1.48	1.44	1.74	1.54	1.50	1.49	1.48	1.47	1.44
wa,s,8	1.70	1.45	1.23	1.18	1.16	1.15	1.13	1.39	1.19	1.15	1.15	1.14	1.13	1.13
							VLIW							
wn,ns,4	3.26	2.73	2.30	1.98	1.79	1.35	1.15	3.06	2.64	2.19	1.88	1.69	1.37	1.14
wa,ns,4	3.29	2.75	2.32	2.00	1.82	1.36	1.15	3.08	2.66	2.21	1.91	1.72	1.40	1.14
wa,s,4	2.67	2.13	1.70	1.39	1.31	1.17	1.13	2.47	2.04	1.59	1.29	1.16	1.14	1.12
wn,ns,8	2.62	2.25	1.93	1.70	1.57	1.28	1.14	2.48	2.16	1.85	1.63	1.50	1.30	1.13
wa,ns,8	2.63	2.26	1.94	1.70	1.58	1.29	1.14	2.48	2.16	1.85	1.64	1.51	1.31	1.13
wa,s,8	2.23	1.86	1.55	1.31	1.25	1.16	1.12	2.09	1.76	1.46	1.25	1.16	1.13	1.12

Table XII—*Continued*

						Yacc								
wn,ns,4	2.38	2.16	1.99	1.90	1.83	1.64	1.33	2.13	1.99	1.89	1.84	1.79	1.65	1.33
wa,ns,4	2.42	2.20	2.02	1.92	1.86	1.67	1.34	2.16	2.02	1.92	1.86	1 82	1.68	1.35
wa,s,4	1 85	1.63	1.45	1.35	1 32	1.27	1.21	1.59	1.45	1.35	1.29	1.26	1.24	1.20
wn,ns,8	2.11	1.90	1.75	1.67	1.62	1.49	1.28	1.87	1.75	1.67	1.62	1.58	1.49	1.28
wa,ns,8	2.13	1.92	1.77	1.68	1.63	1.50	1.29	1.89	1.76	1.68	1.63	1.59	1.50	1.29
wa,s,8	1.76	1.55	1.40	1.32	1.29	1.25	1.20	1.52	1.40	1.32	1.27	1.24	1.22	1.19

REFERENCES

APPEL, A. W. 1987. Garbage collection can be faster than stack allocation. *Inf. Process. Lett. 25*, 4, 275–279.

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper. 19*, 2 (Feb.), 171–184.

APPEL, A. W. 1990. A runtime system. *Lisp Symb. Comput. 3*, 4 (Nov.), 343–380.

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, Mass.

APPEL, A. W. AND JIM, T. Y. 1989. Continuation-passing, closure-passing style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, Tex.). ACM, New York, 293–302.

APPEL, A. W., MATTSON, J. S., AND TARDITI, D. 1989. A lexical analyzer generator for Standard ML. User manual distributed with Standard ML of New Jersey.

BALL, T. AND LARUS, J. R. 1992. Optimally profiling the tracing programs. In the *19th Symposium on Principles of Programming Languages*. ACM, New York.

CHEN, J. B. AND BERSHAD, B. N. 1993. The impact of operating system structure on memory system performance. In the *14th Symposium on Operating Systems Principles* ACM, New York.

CHENEY, C. 1970. A nonrecursive list compacting algorithm *Commun ACM 13*, 11 (Nov.), 677–678.

CLEAVELAND, R., PARROW, J., AND STEFFEN, B. 1993. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst. 15*, 1 (Jan.), 36–72.

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

CYPRESS. 1990. *SPARC RISC User's Guide*, 2nd ed. Cypress Semiconductor, San Jose, Calif.

DEC. 1990a. *DS5000/200 KN02 System Module Functional Specification*. Digital Equipment Corp., Palo Alto, Calif.

DEC. 1990b. *DECStation* 3100 *Desktop Workstation Function Specification*, 1 3 ed. Digital Equipment Corp., Palo Alto, Calif.

DEC. 1992. *DECchip* 21064-*AA Microprocessor Hardware Reference Manual*, 1st ed. Digital Equipment Corp., Maynard, Mass.

EKANADHAM, K. AND ARVIND. 1987. SIMPLE: An exercise in future scientific programming. Tech. Rep. Computation Structures Group Memo 273, MIT, Cambridge, Mass. July. Also available as IBM/T. J. Watson Research Center Research Rep. 12686.

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

HIEB, R., DYBVIG, R. K., AND BRUGGEMAN, C. 1990. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, N Y.). ACM, New York, 66–77.

HILL, M. D. 1988. A case for direct mapped caches. *Computer 21*, 12 (Dec.), 25–40.

HILL, M. AND SMITH, A. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (Dec.), 1612–1630.

JOUPPI, N. P. 1993. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, Calif.). ACM Press, New York, 191–201.

KANE, G. AND HEINRICH, J. 1992. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, N.J.

KESSLER, R. E. AND HILL, M. D. 1992. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst. 10*, 4 (Nov.), 338–359.

KOOPMAN, P. J., JR., LEE, P., AND SIEWIOREK, D. P. 1992. Cache behavior of combinator graph reduction. *ACM Trans. Program. Lang. Syst. 14*, 2 (Apr.), 265–277.

KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction* (Palo Alto, Calif.). ACM, New York, 219–233.

LARUS, J. R. 1990. Abstract execution: A technique for efficiently tracing programs. *Softw. Pract. Exper. 20*, 12 (Dec.), 1241–1258.

LARUS, J. R. AND BALL, T. 1992. Rewriting executable files to measure program behavior. Tech. Rep. Wis 1083, Computer Sciences Dept., Univ. of Wisconsin-Madison, Madison, Wisc. Mar.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

PATTERSON, D. A. AND HENNESSY, J. L. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, Calif.

PENG, C.-J. AND SOHI, G. S. 1989. Cache memory design considerations to support languages with dynamic heap allocation. Tech. Rep. 860, Computer Sciences Dept., Univ. of Wisconsin-Madison, Madison, Wisc. July.

PRZYBYLSKI, S. A. 1990. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo, Calif.

READE, C. 1989. *Elements of Functional Programming*. Addison-Wesley, Reading, Mass.

REINHOLD, M. B. 1993. Cache performance of garbage-collected programming languages. Ph.D. thesis, Laboratory for Computer Science, MIT, Cambridge, Mass.

SLATER, M. 1991. PA workstations set price/performance records. *Microprocess. Rep. 5*, 6 (Apr.), 1.

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

TARDITI, D. AND DIWAN, A. 1993. The full cost of a generational copying garbage collection implementation. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*. ACM, New York.

WAUGH, K. G., MCANDREW, P., AND MICHAELSON, G. 1990. Parallel implementations from function prototypes: A case study. Tech. Rep. Computer Science 90/4, Heriot-Watt Univ., Edinburgh, U.K.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1990. Caching considerations for generational garbage collection: A case for large and set-associative caches. Tech. Rep. EECS-90-5, Univ. of Illinois at Chicago, Chicago, Ill. Dec.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1992. Caching considerations for generational garbage collection. In the *1992 ACM Conference on Lisp and Functional Programming* (San Francisco, Calif.). ACM, New York, 32–42.

ZORN, B. 1991. The effect of garbage collection on cache performance. Tech. Rep. CU-CS-528-91, Univ. of Colorado at Boulder, Boulder, Colo. May.

Received December 1993; revised January 1995; accepted May 1995