Memory Subsystem Performance of Programs Using Copying Garbage Collection

Amer Diwan  David Tarditi  Eliot Moss

December 10, 1993
CMU-CS-93-210

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper will appear in the Proceedings of the 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Portland, Oregon, January 16-19, 1994. It is also published as Fox Memorandum CMU-CS-FOX-93-06

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.

1The 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.
1 Introduction

Heap allocation with copying garbage collection is widely believed to have poor memory subsystem performance [30, 37, 38, 23, 39]. 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 SMLNJ compiler [31], 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 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 [30, 37, 38, 23, 39] 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 [22] 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 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 [31].

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 [31] 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:
• 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 [21] 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.19

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 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 instruction2). 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.

---

1 Recall subblock size is assumed to be 1 word.

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

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 [17, 11] recovers 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 [27]. 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% [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) \cite{29} 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 \texttt{rplaca} and \texttt{rplacd} do not exist for the default definition of lists in SML. The only kinds of assignable data structures are \texttt{ref} cells and arrays\footnote{Although the language definition omitted arrays, all implementations have arrays.}, which must be explicitly 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 \cite{33}. 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 \cite{3} is a publicly available compiler for SML. We used version 9.91. The compiler concentrates on making allocation cheap and function calls fast. Allocation is done in-line, except for the allocation of arrays. Aggressive 3-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 \cite{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 \cite{21}. However, using only a heap simplifies the compiler, the run-time system \cite{2}, and the implementation of first-class continuations \cite{18}. 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 \cite{30}, examined the data cache behavior of some small Lisp programs. They used trace-driven simulation, and proposed an \texttt{ALLOCATE} instruction for improving cache behavior, which allocates a block in the cache without fetching it from memory. Wilson \textit{et. al.} \cite{37,38} argued that cache performance of programs with generational garbage collection will improve substantially when the youngest generation fits in the cache. Koopman \textit{et. al.} \cite{23} 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. Zorn939 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.

Appel31 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 under-
counted in his measurements.4

Jouppi211 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 QPT7, 25, 26 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 Tycho20 for the memory subsystem simulations. Tycho uses a special case of all-
associativity simulation28 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 simplifica-
tions 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 $20000\text{K}^4$.

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 \(^6\). 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\(^9\). 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\(^10\), records, and others (includes spill records, arrays, strings, vectors, ref cells, store list records, and floating point numbers). 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.

---

\(^4\)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.

\(^9\)This table corrects one given in the POPL '94 paper, which did not include allocation data for floating point numbers. Our thanks to Darko Stefanovic for bringing this to our attention.

\(^10\)Closures for callee-save continuations can be trivially allocated on a stack in the absence of first class continuations.
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\(^{\ddagger}\) processing the lexical description of Standard ML.
Life The game of Life implemented using lists 32.
PIA The Perspective Inversion Algorithm 36\(^{\ddagger}\) decides the location of an object in a perspective 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 35 processing the grammar of Standard ML.

Table 1: Benchmark Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Heap size (K)</th>
<th>Code size (K)</th>
<th>Non-gc (sec)</th>
<th>Gc (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CW</td>
<td>5728</td>
<td>1107</td>
<td>894</td>
<td>22.74</td>
<td>1.69</td>
</tr>
<tr>
<td>Knuth-Bendix</td>
<td>491</td>
<td>2786</td>
<td>251</td>
<td>13.47</td>
<td>1.48</td>
</tr>
<tr>
<td>Lexgen</td>
<td>1224</td>
<td>2162</td>
<td>305</td>
<td>15.07</td>
<td>1.06</td>
</tr>
<tr>
<td>Life</td>
<td>111</td>
<td>1026</td>
<td>221</td>
<td>16.97</td>
<td>0.19</td>
</tr>
<tr>
<td>PIA</td>
<td>1454</td>
<td>1025</td>
<td>291</td>
<td>6.07</td>
<td>0.34</td>
</tr>
<tr>
<td>Simple</td>
<td>999</td>
<td>11571</td>
<td>314</td>
<td>25.58</td>
<td>1.23</td>
</tr>
<tr>
<td>VLIW</td>
<td>3207</td>
<td>1088</td>
<td>486</td>
<td>23.70</td>
<td>1.91</td>
</tr>
<tr>
<td>YACC</td>
<td>5751</td>
<td>1632</td>
<td>580</td>
<td>4.60</td>
<td>1.98</td>
</tr>
</tbody>
</table>

Table 2: Sizes of Benchmark Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Inst Fetches</th>
<th>Reads (%)</th>
<th>Writes (%)</th>
<th>Partial Writes (%)</th>
<th>Assignments (%)</th>
<th>Nops (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CW</td>
<td>523,245,987</td>
<td>17.61</td>
<td>11.61</td>
<td>0.01</td>
<td>0.41</td>
<td>13.21</td>
</tr>
<tr>
<td>Knuth-Bendix</td>
<td>312,086,438</td>
<td>19.66</td>
<td>22.31</td>
<td>0.00</td>
<td>0.00</td>
<td>5.92</td>
</tr>
<tr>
<td>Lexgen</td>
<td>328,422,283</td>
<td>16.08</td>
<td>10.44</td>
<td>0.20</td>
<td>0.21</td>
<td>12.33</td>
</tr>
<tr>
<td>Life</td>
<td>413,536,662</td>
<td>12.18</td>
<td>9.26</td>
<td>0.00</td>
<td>0.00</td>
<td>15.45</td>
</tr>
<tr>
<td>PIA</td>
<td>122,215,151</td>
<td>25.27</td>
<td>16.50</td>
<td>0.00</td>
<td>0.00</td>
<td>8.39</td>
</tr>
<tr>
<td>Simple</td>
<td>604,811,016</td>
<td>23.86</td>
<td>14.06</td>
<td>0.00</td>
<td>0.05</td>
<td>7.58</td>
</tr>
<tr>
<td>VLIW</td>
<td>399,812,033</td>
<td>17.89</td>
<td>15.99</td>
<td>0.10</td>
<td>0.77</td>
<td>9.04</td>
</tr>
<tr>
<td>YACC</td>
<td>133,043,324</td>
<td>18.49</td>
<td>14.66</td>
<td>0.32</td>
<td>0.38</td>
<td>11.14</td>
</tr>
</tbody>
</table>

Table 3: Characteristics of benchmark programs
Table 1: Allocation characteristics of benchmark programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Allocation (words)</th>
<th>Escaping %</th>
<th>Known %</th>
<th>Callee Saved %</th>
<th>Records %</th>
<th>Other %</th>
</tr>
</thead>
<tbody>
<tr>
<td>CW</td>
<td>56,467,440</td>
<td>4.0</td>
<td>3.3</td>
<td>15.39</td>
<td>67.2</td>
<td>6.20</td>
</tr>
<tr>
<td>Knuth-Bendix</td>
<td>67,733,930</td>
<td>37.6</td>
<td>0.1</td>
<td>15.22</td>
<td>49.5</td>
<td>4.90</td>
</tr>
<tr>
<td>Lexgen</td>
<td>33,046,349</td>
<td>3.4</td>
<td>0.2</td>
<td>12.96</td>
<td>72.7</td>
<td>6.40</td>
</tr>
<tr>
<td>Life</td>
<td>37,840,681</td>
<td>0.2</td>
<td>0.0</td>
<td>15.00</td>
<td>77.8</td>
<td>5.52</td>
</tr>
<tr>
<td>FIA</td>
<td>18,841,256</td>
<td>0.4</td>
<td>28.0</td>
<td>11.99</td>
<td>25.0</td>
<td>4.69</td>
</tr>
<tr>
<td>Simple</td>
<td>80,781,644</td>
<td>4.0</td>
<td>1.1</td>
<td>15.33</td>
<td>68.1</td>
<td>6.63</td>
</tr>
<tr>
<td>VLIW</td>
<td>59,497,132</td>
<td>9.9</td>
<td>6.0</td>
<td>26.62</td>
<td>61.8</td>
<td>7.67</td>
</tr>
<tr>
<td>YACC</td>
<td>17,015,250</td>
<td>2.3</td>
<td>15.3</td>
<td>15.35</td>
<td>54.8</td>
<td>7.44</td>
</tr>
</tbody>
</table>

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.

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 CPU does not have 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 [23], a block size of 1 word, and the ALLOCATE directive 30 can all achieve this. For large caches, when the allocation area fits in the cache and thus there

11In an actual implementation, the penalty of a miss may be one cycle since unlike hits, the tag and valid bits need 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.

12Since the effects on cache performance of these features are so similar, we talk just about subblock placement.
Table 5: Penalties of memory operations

<table>
<thead>
<tr>
<th>Task</th>
<th>Penalty (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non page mode write</td>
<td>3</td>
</tr>
<tr>
<td>Page mode write</td>
<td>1</td>
</tr>
<tr>
<td>Read 16 bytes from memory</td>
<td>15</td>
</tr>
<tr>
<td>Read 32 bytes from memory</td>
<td>19</td>
</tr>
<tr>
<td>Write hit or miss (subblocks)</td>
<td>0</td>
</tr>
<tr>
<td>Write hit (16 bytes, no subblocks)</td>
<td>0</td>
</tr>
<tr>
<td>Write hit (32 bytes, no subblocks)</td>
<td>0</td>
</tr>
<tr>
<td>Write miss (16 bytes, no subblocks)</td>
<td>15</td>
</tr>
<tr>
<td>Write miss (32 bytes, no subblocks)</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 6: Cache organizations studied

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., DEC Stations, HP 700 series), there are some exceptions (notably SPARC).

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 31 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 1.1).

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 21. 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 offer 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. 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.

Table 7: Memory subsystem organization of some popular machines

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Write Policy</th>
<th>Write Miss Policy</th>
<th>Write Buffer Subblocks</th>
<th>Assoc</th>
<th>Block Size</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS3100</td>
<td>through</td>
<td>allocate</td>
<td>4 deep</td>
<td>1</td>
<td>4 bytes</td>
<td>64K</td>
</tr>
<tr>
<td>DS5000 200</td>
<td>through</td>
<td>allocate</td>
<td>5 deep</td>
<td>yes</td>
<td>16 bytes</td>
<td>64K</td>
</tr>
<tr>
<td>HP 9000</td>
<td>back</td>
<td>allocate</td>
<td>none</td>
<td>no</td>
<td>32 bytes</td>
<td>64K 2M</td>
</tr>
<tr>
<td>SPARCStation II</td>
<td>through</td>
<td>no allocate</td>
<td>4 deep</td>
<td>no</td>
<td>32 bytes</td>
<td>64K</td>
</tr>
</tbody>
</table>

Note:
- SPARCStation 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 32K 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 containing an instruction and data cache. The hint can specify that a block will be overwritten before being read. This avoids the read time on a cache.

---

13For 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.
In this paper we present only the summary graphs for CW (Figure 2). The summary graphs for other programs are similar to those for CW and are thus omitted for space considerations. 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.

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. 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.

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

In Sections 5.3.1, 5.3.2, 5.3.3, and 5.3.4 we describe the impact of write-miss policy and subblock placement, associativity, block size, and cache size on the memory subsystem performance of the benchmark programs. In Section 5.3.5 we give the write buffer and partial-word write overheads.

### 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). 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 benefits would decrease for larger caches. Our data indicates that the reduction in benefits 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). 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

---

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.

17 For the penalties used, a 17% overhead translates roughly into one fetch from memory—instruction or data— every 100 useful instructions.
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 [19].

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. 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.

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 instruction cache penalty is reduced by the two-way associative cache organizations, suggesting a large number of conflict misses in the instruction cache.

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 [23]. 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.

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 (perhaps 256K or bigger) once the allocation area fits in the cache (this will not be nearly as significant for caches with subblock placement). 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.
Figure 2: CW summary, write buffer depth=6
Figure 3: CW, write no alloc, no subblk, block size=16, wb depth=6
Figure 4: CW,write alloc,subblk,block size = 16,wb depth = 6

(a) assoc = 1

(b) assoc = 2
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 [30, 37, 38, 39], 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%.

While associativity, cache sizes, and block sizes affected performance, their contribution to performance was usually small. Associativity and cache sizes had little impact on data cache performance, but were more important for instruction cache performance.

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, Brad Chen, Olivier Danvy, Alessandro Forni, Urs Hodelie, Kathryn McKinley, Erich Nahum, and Darko Stefanovic for comments on drafts of this paper. We thank Peter Lee for his encouragement and advice during this work. We thank Brian Minees 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, Tycio and Dineroll. 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


