CACHE PERFORMANCE OF PROGRAMS WITH INTENSIVE HEAP ALLOCATION AND GENERATIONAL GARBAGE COLLECTION

Marcelo José de Rezende Gonçalves

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

May 1995

Abstract

This dissertation presents a study of cache performance of programs with intensive heap allocation and generational garbage collection, such as ML programs compiled by the *Standard ML of New Jersey* compiler (SML/NJ). SML/NJ programs allocate objects dynamically in the heap at a very fast rate, due to the allocation of function activation records (closures) in the heap.

The memory reference patterns of a set of SML/NJ programs are studied, showing that nearly all writes are for sequential allocation, and most reads are to recently allocated objects. The cache performance of the programs is studied with different cache parameters varied: cache and block size, associativity, and write-miss policy. The results confirm the findings of previous studies, that the write-miss policy has the most significant effect on the performance of these programs. Because of the way these programs reference memory it is important that the cache allocates a block on writes, without fetching the block from memory.

This dissertation also investigates the use of software alternatives to improve cache performance, that have been suggested but not fully analyzed in previous studies, such as fitting the allocation space in the cache. The trade-offs between cache and garbage collection overhead are quantitatively analyzed. The effects of other garbage collection parameters on performance, such as the frequency of major collections, and the algorithm to traverse the graph of reachable objects are also considered.

The results of this study show that heap allocation, even intensive heap allocation of closures, can have good cache performance, as long as the architecture uses a favorable write-miss policy. On machines with an unfavorable write-miss policy there are alternatives that can be applied, such as fitting the allocation space in the cache.

Acknowledgements

I would like to thank my advisor, Andrew Appel, not only for his help during this research, but also for his support during the years I spent in Princeton.

I am indebted to Lal George for accepting to be a reader at short notice, and for many suggestions to improve the quality of this text. Thanks also to Doug Clark for his comments on both the final and early drafts of this thesis. Anne Rogers also made many comments to an early draft of this dissertation.

E. Gun Sirer wrote the reliable simulator used in this research, and John H. Reppy provided me with a prototype of the new SML/NJ garbage collector. Zhong Shao gave me a lot of help with the SML/NJ system. Karin Petersen provided data on caches of various machines.

The staff of the Department of Computer Science at Princeton provided valuable help throughout my years in Princeton. Especial thanks to Sharon Rodgers.

I would like also to thank my family for all of their support and encouragement.

This research was supported by the National Council for Technological and Scientific Development of Brazil (CNPq) Grant 200579/90.8, and by the National Science Foundation Grant CCR-9200790.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation presents a study of cache performance of programs with intensive heap allocation and generational garbage collection, such as ML programs compiled by the *Standard ML of New Jersey* compiler (SML/NJ)[11].

ML is a strict, statically-type-checked language, with parametric polymorphism, that encourages the use of a *mostly functional* programming style[53, 56]. Most data structures are *immutable*, that is, they are written only once, when they are created. The language also provides reference variables that can be updated, but these variables must be distinguished from immutable data structures, and their use is discouraged. ML provides higher order functions, which can be treated like any other ordinary data structures, and can be stored, passed as arguments, or returned as the result of other function calls. Data objects are allocated dynamically, and there is no explicit deallocation. Implementations must provide automatic deallocation, usually by a *garbage collector*.

The SML/NJ implementation is based on *continuation-passing style* (CPS) [8]. The compiler uses heap-allocated *closures* to represent first class function values. A closure is a data structure that contains the machine-code address and bindings for the free variables of a function. Because of the use of heap-allocated closures, programs compiled by the SML/NJ compiler (*SML/NJ programs*) tend to allocate objects in the heap at a very fast rate: on a RISC machine, a typical program allocates about one word for every six machine instructions. To provide efficient allocation, SML/NJ allocates sequentially from a contiguous area, the *allocation space*, and for efficient deallocation, it uses a *generational copying garbage collector* [80].

1.1 Motivation

With the rapid increase in processor speed, which is not being matched by a similar increase in memory speed, *caches* are becoming increasingly important for higher performance in computer systems. Caches can hold only a small subset of the instructions and data in use by a program and they are effective in improving performance only if programs have good locality properties, that is, either the same locations are used many times (temporal locality) or adjacent locations are used within a short interval (spatial locality) [42].

Since good cache performance depends on the locality properties of programs, it is important that these properties be well understood both by hardware and software designers. A better understanding of a program's locality properties can lead to the design of better cache architectures that can make use of those properties to improve performance, or it can help software designers modify their programs to match features provided by a particular cache architecture.

Functional programs with dynamic heap allocation had been generally believed to have poor cache locality, because of their use of large memory areas [83]. To address this problem, Wilson, Lam, and Moher [83], and Zorn [87] proposed the use of generational garbage collection techniques to improve cache performance. They claim that cache performance can be improved by keeping the allocation space (or *youngest generation*) in the cache (that is, making the allocation space the size of the cache or smaller). But because making the allocation space small makes garbage collections more frequent, and thus can increase garbage collection overhead, this idea is practical only for "large" caches, otherwise the extra garbage collection overhead would offset the lower cache miss overhead. On a different path, Peng and Sohi [57] proposed new cache architecture features that could significantly improve the performance of programs with dynamic heap allocation. They observed that the memory access characteristics of these programs, with most writes being to the allocation of a new object on the heap, can cause many write misses (*allocation misses*), the number of which tend to be inversely proportional to block size, and independent of cache size. In their study, Peng and Sohi showed that a cache allocate instruction, which allocates a word directly in the cache without fetching it from memory, could eliminate most of these allocation misses. Koopman et al. [46], and Diwan et al. [28, 29] showed that programs with dynamic heap allocation can have good cache performance if the cache has features that achieve an effect similar to the allocate instruction described by Peng and Sohi. These features are related to the cache write miss policy [45], and the best policy for heap-allocating programs is one that allocates a block in the cache on a write miss, and does not fetch the block from memory. Unfortunately, because they can introduce problems in the cache design, these features are not common on current machines. Reinhold [60, 61] showed that, as long as allocation misses are not a problem, the use of large memory areas can actually be good for cache performance on direct-mapped caches, since the references tend to be spread evenly between cache blocks, minimizing conflict misses.

Besides the problems described in the previous paragraph, the SML/NJ designers' decision to allocate activation records on the heap has also been criticized because it may result in poor cache performance. Even though some closures must be heap-allocated because they can have an indefinite extent, the compiler could determine the extent of many closures at compile time, and some compilers use a more specialized mechanism to allocate closures [71, 50] such as a stack, which is generally believed to have better locality properties. In a recent study, Appel and Shao [3] compared the performance of SML/NJ programs with stack versus heap-allocated closures, and showed that the cache performance in both cases is similar if the right write-miss policy is used.

Despite extensive research on cache performance of programs with dynamic heap allocation and generational garbage collection, many issues remain open.

- Most previous studies have concentrated on performance, and have not analyzed in detail the memory reference patterns of those programs. Such an analysis can help understand the locality properties of the programs and explain why they have good or bad performance in a particular cache organization. Those studies that analyzed the memory reference patterns [57, 60, 61, 85], considered languages with stack-allocated activation records, that can have significantly different memory reference patterns from those programs with heap-allocated activation records. Stefanovic and Moss [73, 72] studied object behavior in SML/NJ programs, but the emphasis of their work is on garbage collection performance, and cache performance issues are only briefly addressed.

- Previous studies have either used simple garbage collectors, or did not vary many garbage collection parameters. Zorn's [85, 87] was the only study to consider the effects of many garbage collection parameters, but he did not consider the effects of the cache write miss policy.

- None of the previous studies have quantified the trade-offs between cache and garbage collection performance that arise if the allocation space is to be fitted in the cache.

The goal of this dissertation is to address these and other issues related to the cache performance of programs with intensive heap allocation and generational garbage collection, like SML/NJ programs.

Many of the results presented in this dissertation are also presented in Gonçalves and Appel [39].

1.2 Overview

This dissertation presents a study of cache performance and memory reference patterns of a benchmark suite of SML/NJ programs, using empirical and analytical methods. The study of the memory reference patterns of the programs shows a regular and cyclic pattern of mostly sequential writes and many reads to recently written memory locations. The cache performance on several different architectures is considered, and several cache design parameters are varied in the experiments: cache size, block size, associativity, and write miss policy. The experiments confirm that the write miss policy can have a large effect on performance. Other cache design parameters can also affect performance, but except for cache size, their effect is not as important as the write miss policy.

The effects of generational garbage collection on cache performance are also considered. Each program is run many times, with the size of the allocation space varying from 16K bytes to "infinite" (which is large enough for the programs to run without any garbage collections), and a quantitative analysis of the trade-offs between garbage collection and cache performance is made, to determine the feasibility of fitting the allocation space in different caches. Other garbage collection issues, such as the frequency of major collections and clustering techniques, are also considered.

The organization of this dissertation is the following. Chapter 2 makes an overview of cache architectures, generational garbage collection, and the SML/NJ compiler and run-time system, introducing terms and concepts that will be used on the remaining chapters.

Chapter 3 describes the methodology and tools used in this work. The benchmark programs are described in Section 3.1. Section 3.2 presents an analytical performance model used to predict the total cycles in the execution of a program, and compares the predictions of this model with measurements of run time on a real machine. Section 3.3

summarizes the simplifications and limitations of this work. And Section 3.4 describes the instruction level simulator and other tools used in this research.

Chapter 4 presents an analysis of the memory reference patterns and object characteristics of the benchmark programs. This study begins with a profile of object allocation, which is presented in Section 4.1. The main results in this section are that closures account for most of the allocation of the programs, but closures tend to have a short life, making user objects the most common objects in older generations. User objects are small, two to three words on average, while closures are about six words on average. Section 4.2 describes the write references of the programs, and shows that nearly all writes are for allocation, having a regular pattern. Section 4.3 analyses the read references, and shows that most reads are to recently allocated objects, and that most of these references are to closures. Finally, Section 4.4 analyses the memory references of minor and major collections, which can have a significant effect in the performance of some programs.

Chapter 5 studies the cache performance of the benchmark programs on various cache organizations. The effects of many design parameters are evaluated. Cache size is varied from 1K to 4M bytes, block size from 16 to 256 bytes, and set size from 1 to 16. Read and write miss ratios are studied independently, and the effects of three write miss policies are analyzed. The chapter begins with the miss ratios of fetch-on-write caches. Section 5.2 compares the miss ratios of write-validate and write-around caches to the miss ratios of fetch-on-write caches. Section 5.3 presents the miss ratios for set-associative caches. Section 5.4 presents the miss ratios of instruction caches, and Section 5.5 the miss ratios for unified caches. Section 5.6 shows the cache overhead for data, instruction, and unified caches. Finally, Section 5.7 compares the performance of the SML/NJ programs to that of the SPEC92 benchmarks, and to a benchmark of Scheme programs.

Chapter 6 considers the effects of garbage collection on cache performance. Two main issues are considered, the frequency of minor and major collections, and the algorithm to traverse the graphs of live objects. Section 6.1 presents a study on the effects of the size of the allocation space on cache performance. The benchmark programs are run with many different allocation space sizes, from 16K bytes to "infinite". Using the analytical model introduced in Section 3.2 and empirical measurements, a quantitative analyses of the tradeoffs between lower cache and higher garbage collection overhead is presented. The effects of the traversal algorithm in cache performance is analyzed in Section 6.2, where three different algorithms are compared, the standard breadth-first search used by the SML/NJ compiler, a depth-first search, and a modified breadth-first algorithm that linearizes lists. Chapter 6 also presents two cases studies in Section 6.3.2, where the effects of the size of the allocation space in the performance of SML/NJ programs in two machines with different cache organizations is analyzed.

Finally, Chapter 7 reviews related work, and summarizes the main results and conclusions of this research.

Chapter 2

Background

This chapter presents an overview of cache architectures, generational garbage collection, and the SML/NJ compiler and run-time system, introducing terminology and concepts that will be used in subsequent chapters.

2.1 Cache Architectures

This section briefly overviews the main issues related to cache architectures. More detailed information can be found in Smith [66], Przybylski [58], or in standard computer architecture textbooks [42, 74].

A *cache* is a small and fast memory that sits between the processor and the main memory. The cache contains a subset of the words in the address space of a program, which is the set of memory words that can be addressed by a program. In general, the memory of a computer system is organized as a hierarchy where the higher levels, which are closer to the processor, contain a subset of the words of the lower levels of the hierarchy. The cache is the first level of this hierarchy and the virtual memory the lowest. Multi-level caches are becoming common on current machines, with a first, second, and possibly a third level caches.

Caches are effective in improving performance only if there is a high probability that a word addressed by a program is found in the cache, avoiding a costly fetch from main memory. Caches usually work well because most programs tend to have good *locality* properties, that is, a few memory locations tend to be reused many times (*temporal locality*), and adjacent memory locations tend be be referenced within a short time (*spatial locality*).

A memory reference that is found in the cache is called a *cache hit*, while a memory reference that is not found in the cache and must be fetched from main memory is called a *cache miss*. The time to fetch a word from the cache is the *hit time*, and the time to fetch a word from memory is the *miss penalty*. The ratio of references that are misses to the total cache references is the cache *miss ratio* (or miss rate). Performance (measured by program execution time) depends both on hit time, miss penalty, and miss ratio. The lower the hit time, the miss penalty, and the miss ratio are, the better the performance is.

A cache consists of an array of fixed-size *blocks* (or lines) that can keep the contents of one memory block. An auxiliary array of *tags* contains a directory of which memory block is stored in each cache block. When the processor makes a memory reference (read or write) the the tag array is searched to find whether the block that contains the address is present in the cache. If the block is in the cache, then there is a cache hit, and the requested word is returned to the processor (or written to the cache). Otherwise there is a miss and the block that contains the requested word must be fetched from memory into the cache. Besides the tag, the directory contains control bits that keep status information on each cache block, such as if the block contains valid data (*valid bit*), or if its contents have been modified (*dirty bit*).

A cache is *fully associative* if a memory block can be placed in any block in the cache. In this case, all entries in the tag array must be searched in parallel to find where the desired block is. A cache is *direct-mapped* if a memory block can be found in exactly one cache block. In this case, some mathematical function of the memory block address (usually the modulus by the number of cache blocks) is used to compute the unique entry that can contain a block. In general, the cache is organized as an array of fixed-size sets of blocks. A memory block can be located anywhere within a set, but in exactly one set. A cache is called *n-way set associative* (or n-way associative, or of degree n) if each set contains n blocks. A direct-mapped cache is a particular case of an n-way associative cache where the set size is one (1-way associative), and a fully associative cache with s blocks has only one set of size s. A memory address is usually divided in three parts, the tag, the index, and the block offset, where the index is used to select the set and the block offset selects a word within a block.

Cache misses can be divided into three types. A *compulsory* miss occurs when an address is referenced for the first time. A *capacity* miss occurs when a program makes references to more blocks that can fit into the cache, and thus some blocks must be temporarily replaced.

A *conflict* miss occurs when more blocks that can fit into a set map to the same set, causing some blocks in the set to be replaced. By definition, fully associative caches do not have conflict misses. Compulsory misses depend on the number of distinct blocks referenced by a program. Capacity misses depend on the cache size, and tend to decrease as cache size increases, while conflict misses depend on the degree of associativity, and tend to decrease as the degree of associativity increases.

2.1.1 Cache design issues

The goal of cache design is to balance cost and performance [42]. In general, the designer seeks to obtain maximum performance while keeping cost at a reasonable level for a given machine. The design of a cache involves the selection of a number of different parameters, such as cache size, block size, and set-associativity, and there are usually trade-offs to be made in these selections. This section describes the most common of these paramenters and their relation to cost and performance.

Dual/unified caches

A cache is unified if the same cache is used for both instructions and data. Having separate caches for instructions and data allows instruction fetches and data reads and writes to be processed in parallel. Both dual (or split) and unified caches are commonly used. Dual caches are common for small first level on-chip caches, and unified caches are common for large second level or off-chip caches.

Cache size

Typical cache sizes range from a few kilobytes to a few megabytes. Large caches have lower miss ratios, but they may have higher access time, or they can require more area than what is available for an on-chip implementation. Small on-chip caches, from 8K to 32K bytes, are common on current microprocessors. Caches from 64K bytes to 4 megabytes are common for second-level or single-level off-chip caches.

Block (line) size

Large blocks may make better use of spatial locality and thus reduce miss ratios. But given a fixed cache size, increasing block size reduces the number of blocks in the cache, which can increase conflict misses, specially on small caches [58, 67]. Because there are more words to be transferred from memory to cache, large blocks can also have higher miss penalties, unless the width of the memory-cache data path is increased. Typical block sizes are 4 to 256 bytes, with 32 bytes being the most common on current machines.

Associativity

Usually, increasing associativity tends to decrease miss ratios, but the need to compare many tags in parallel can increase hit time and thus cause a negative impact on performance [43, 44, 58]. Direct-mapped caches and low degree (2 to 8-way) associative caches are the most common.

Replacement policy

On set-associative caches, the replacement policy determines which block in a set is replaced when a new block must be fetched into the cache. The optimal replacement policy (the one that gives the lowest miss ratio) is the one that replaces the block that is going to referenced furthest into the future, but using this policy is not possible in practice [74, Section 2.2.5]. The *least recently used* (LRU) policy usually gives a good approximation of the optimal replacement and is the most common replacement policy used in set-associative caches. Other possible replacement policies are *random* and FIFO.

Virtually/physically addressed caches

A cache is virtually addressed if the cache look-up is done before the translation of the virtual to physical address, otherwise it is physically addressed. Some machines do the virtual to physical translation in parallel with the cache tag look-up. This can be done only if the index and offset parts are the same for both the virtual and physical addresses, which limits the size of the index and offset bits, and thus may limit cache size unless the associativity is increased. For this reason many machines have caches with a high degree of associativity [42].

Fetch size/sub-block placement

The fetch size determines how many blocks to fetch on a miss. Usually only one block is fetched. Some machines may fetch more than one block or only part of a block, and have different fetch sizes for read and write misses. When less than one full block can be fetched, the fetch unit is called a sub-block, and there is usually one valid (and possibly a dirty) bit associated with each sub-block.

Blocking/non-blocking misses

On a blocking cache, the processor stalls on a cache miss until the block is fetched from memory. There are many improvements that are possible to this basic miss processing technique. First, it may be possible to fetch the word that causes the miss first, and restart the processor as soon as this word is available (critical word first/early restart). On processors with multiple independent functional units, it may be possible to execute other instructions that do not access memory and do not use the word that caused the miss while the block is being fetched. Writes that miss the cache can be sent to a write buffer and do not need to cause a stall.

In a more general scheme, there may be a number of pending misses at a time and the processor can continue to issue instructions as long as there are instructions available that do not need the results of a pending miss, and the maximum number of pending misses is not reached [68]. This scheme can be costly to implement but it should become common on high performance superscalar processors. Under this scheme, cache misses may have no performance cost at all in the best (but unlikely) case.

Write hit policies

The are two alternatives on a write: the word can be written only to the cache (*write-back*), or it may be written both to the cache and to memory (*write-through*). The write-back policy can reduce the traffic between memory and cache, but it requires that a dirty block is first written to memory when it is replaced, which may cause a delay in the fetch of a new block. Many machines allow the selection of the write hit policy on a per-page or per-block basis.

Write buffers

A *write buffer* can store a word or a dirty block temporarily, avoiding processor stalls while the word or block is being written to memory. The processor must stall if the buffer is full, and thus usually buffers have more than one entry. It can happen that the write buffer contains the most recent copy of a block, and a word in that block is read, causing a cache miss. There are two alternatives: the buffer must be flushed before the block can be read, which may increase the miss penalty, or the block can be fetched from the buffer, which can increase the complexity of the buffer hardware.

2.1.2 Write miss policies

The write miss policy determines the action to be taken on a write miss. Unlike read misses, where the usual action is to fetch the block containing the addressed word, there are many alternative actions that can be taken on write misses. For example, it is possible to fetch a block, just like on a read miss, or to bypass the cache and write the word directly to memory. Because of the sequential allocation writes that may cause many write misses, the write miss policy is of great importance for the cache performance of programs with intensive heap allocation. This section presents a discussion on write miss policies, following the terminology of Jouppi [45]. The discussion below assumes that the architecture allows only aligned word writes. Many architectures allow writes to bytes, half-words, or unaligned writes, and some special action may be required in these cases.

The three main decisions that must be made on write misses are described below.

1. *Write-allocate versus no-write-allocate:* on a write-allocate cache, a block is allocated for the word being written, while on a no-write-allocate cache, the word is written only to memory and no cache block is allocated.

2. *Fetch-on-write versus no-fetch-on-write:* this policy determines if the block containing the word being written is to be fetched or not. If the block size is one word, then there is no need to fetch, since the word is going to be overwritten. But if the block size is greater than one word and the decision is not to fetch, then the remaining words in the block must be invalidated, and therefore the cache must have one valid bit per word if this policy is used with a block size greater than one.

3. *Write-before-hit versus no-write-before-hit:* this policy determines if the word is written in parallel with the tag look-up (and the detection of hit or miss) or only after the result of the look-up is decided. A write-before-hit policy can allow writes to be done faster, but because the previous contents of the block are overwritten on a miss, this policy can only be used with write-through caches. Write-before-hit is used only with

direct-mapped caches, since on associative caches it is not possible to know which block in a set to write before directory look-up.

Many combinations of these policies are possible (but not all combinations are meaningful), giving different cache organizations. In this work, three of these organizations will be considered:

Fetch-on-write A cache is fetch-on-write if it uses a write-allocate and fetch-on-write policy.

Write-validate A cache is write-validate if it uses a write-allocate and no-fetch-on-write policy.

Write-around A cache is write-around if it uses a no-write-allocate, no-fetch-on-write, and no-write-before-hit policy.

Fetch-on-write and write-validate caches may use either write-before-hit or no-write-before-hit policies. One additional organization described by Jouppi, write-invalidate, is not considered in this work. A cache is write-invalidate if it uses no-write-allocate, no-fetch-on-write, and write-before-hit.

The write miss policy can affect the write miss penalty and the miss ratio. Fetch-on-write caches must fetch a block from memory and thus may cause a pipeline stall, unless write misses are non-blocking. Write-validate does not need to cause a stall, since there is nothing to fetch from memory. Write-around usually writes the word into a write buffer, and unless the buffer is full it should not cause any delay. Write-validate and write-around eliminate all write misses, since there is no penalty. However, they can add more read misses, even though the number of extra read misses is never greater than the number of write misses eliminated. On write-validate caches, an extra read miss may occur only if any invalidated word in the block that was not fetched on the write miss is read before being written (when there is a read miss the block is fetched and thus all the words become valid, and there is no extra miss). On write-around caches, an extra read miss occurs if any word in the block that was not fetched (possibly the written word itself) is read before being written. Write-around tends to add more read misses than write-validate, because the word written is more likely to be read again soon than the other words in the block.

In general, write-validate caches perform better than write-around caches, and write-around performs better than fetch-on-write. Write-validate caches should be particularly effective for programs with writes to sequential addresses, such as the SML/NJ programs studied in this work. Studies by Peng and Sohi [57], and Diwan, Tarditi, and Moss [29], which are confirmed in this work (Section 5.2), show that write-validate caches are the best type for programs with sequential heap allocation.

Unfortunately, there are many problems in the implementation of write-validate caches, which may cause cache designers to avoid this organization [45].

- Write-validate requires one valid bit per word. Moreover, if the architecture allows byte writes or unaligned writes, then either one valid bit per byte, or different policies for word and byte/unaligned writes must be used.

- Write-validate can make it more difficult to keep cache consistency on multiprocessor systems. Shared-bus multiprocessor systems have one cache per processor and use a write-back policy to reduce traffic on the shared bus. If the caches are write-validate, then there may be a block with part of its contents in memory and parts in several different caches, a problem that must be dealt with by the consistency protocol (when fetch-on-write or write-around are used only one cache or memory has a complete valid copy of a block). Solving this problem may require additional transactions that, beside the added complexity of the protocol, can increase traffic in the bus, possibly offsetting the benefits of write-validate.

The two important features of write-validate caches are: a cache block is allocated on a write miss, and the block that contains the missing word is not fetched from memory. These features are specially important for programs with sequential allocation and generational garbage collection because these programs are likely to soon read the word that is written (thus the importance of write-allocate) and they are likely to soon overwrite the other words in the block (thus the importance of no-fetch-on-write). There are other ways of providing these features, even if the cache is not write-validate, such as with instructions that explicitly tell the cache to allocate and not fetch on a write miss. Many architectures provide such instructions.

The HP Precision Architecture (HPPA) [24] uses a bit flag on store instructions to indicates to the processor whether the block containing the address written should be fetched on a cache miss. This feature should be easy to add to SML/NJ programs: all allocation writes can set the store flag bit to indicate that the block never needs to be fetched. The

PowerPC architecture has many cache control instructions that allow programmers to control the cache [25]. A *data-cache-allocate-with-zeroes* instruction can be used to simulate write-allocate with no-fetch-on-write. On machines with non-blocking reads it may also be possible to simulate write-allocate by prefetching the block in advance [9, 54]. This would probably cause a read miss, but since reads are non-blocking, there should be no extra stalls, unless there is a limit on pending cache misses. Both the PowerPC allocate instruction and the prefetch scheme have the disadvantage of requiring extra instructions to be added to a program.

Jouppi [45] lists many problems with these alternative schemes:

1. the entire cache line must be known to be allocated at compile time;

2. block size is an implementation, not an architectural feature, and these schemes would make programs implementation dependent;

3. a block that is only partially allocated may have to be written back to memory before it can be fully allocated (for example, on a context switch) and destroy the old contents of the block;

4. the extra allocate/prefetch instructions.

The first and third of these items are not real problems for SML/NJ programs, since the compiler knows which blocks are going to be allocated (all allocation-space blocks — see Section 2.3), and any non-allocated word in these blocks contains garbage and is never read before being written. The fourth item is not a serious problem, if the extra instructions can save many stall cycles due to misses. The only real problem is the dependency on the implementation, but there may be an alternative to this problem if the architecture provides an instruction to get the block size (the PowerPC provides such an instruction).

Some caches with sub-block placement, or with one-word blocks and fetch size greater than one, are similar to write-validate caches, since they have one valid bit per sub-block or word. They should behave like a write-validate cache, except when the old contents of a line are read before being written, which should not happen very often. Most of the results which will be presented later for write-validate caches, should also apply to these caches.

2.1.3 Cache architecture of current machines

Table 1 describes the basic features of the caches of some current machines. It is often not clear from the literature whether the caches are write-allocate or fetch-on-write. When there is no explicit description of what the cache does on a write miss, it is assumed that read and write misses are treated in a similar way, that is, the cache allocates and fetches a block on writes misses.

Table 1: Cache organization of some current machines and CPUs.

Machine/ CPU	Year	Cache size	Block size	Assoc.	Write-alloc.	Fetch-on-write	Comments/references
DEC5000/240	1990	64K-byte I 64K D	4 bytes	d-m	yes	no	Fetches 8 blocks (32 bytes) on a read miss. [27]
IBM RS6000	1990	8K I 64K D	64 128	2-way 4-way	yes	yes	Cache reload buffer should be able to handle reads to recently written data without stalls. Has cache allocate instructions. [13, 47]
Alpha 21064	1992	8K I 8K D	32	d-m	no	no	On-chip first level cache, with on-chip control for second level cache. [26]
DEC3000/500	1992	8K I(L1) 8K D(L1) 512K (L2)	32	d-m	no yes	no yes	Uses an Alpha 21064 CPU. [30, 31]
HPPA-RISC	1992	4K–1M I 4K-4M D	32	d-m	yes	no	On-chip support to dual off-chip caches. Cache size is system dependent. Fetch-on-write can be selected by setting a bit in the store instruction. [24, 35, 48]
SuperSPARC	1992	20K I 16K D	8 4	5-way 4-way	yes	no	On-chip first level cache. LRU replacement. [34]
Pentium	1993	8K I 8K D	32	2-way	yes	yes	On-chip first level cache. Pentium systems usually have a 256K second level cache. [2, 55]
PowerPC 601	1993	32K	64	8-way	yes	yes	On-chip first level cache. Uses LRU replacement policy. All PowerPCs have cache-line-allocate-with-zero instructions. [17]
PowerPC 603	1993	8K I 8K D	32	2-way	yes	yes	On-chip first level cache. Uses LRU replacement. [18]
PowerPC 604	1994	16K I 16K D	32	4-way	yes	yes	On-chip first level cache. Uses LRU replacement. [69]
Alpha 21164	1994	8K I (L1) 8K D (L1) 96K (L2)	32 32 32/64	d-m d-m 3-way	no	no	Random replacement for the L2 cache. On-chip control for a third level direct-mapped cache. [32]
Intel P6	1995	8K I (L1) 8K D (L1) 256K (L2)	32	2-way 4-way 4-way	?	?	L2 cache on a separate die, packaged with the CPU. [40]

Notes: I = instruction cache; D = data cache; d-m = direct-mapped cache; L1/L2 = first/second-level cache.

2.2 Generational Garbage Collection

This section briefly overviews garbage collection techniques, with an emphasis on generational collection. Wilson [81] makes a comprehensive survey of garbage collection techniques.

Garbage collection is the automatic reclamation of storage that is no longer in use by a program. Programs allocate objects in a heap, and the objects form a graph where each node is an object, and each link is a pointer. The garbage collector considers as *live* (that is, still being used by the program) all objects that can be reached by following a path of pointers from a set of standard *roots* (which are known to be live). Objects that can not be reached in this way are considered *dead* and their space is reclaimed by the collector. Objects that are reachable at the time of the collection are said to *survive* the collection.

Garbage collection techniques can be roughly divided into three groups:

Reference counting In a reference counting collector, each object is associated with a counter that keeps the number of references to that object. Each time a new reference is created/deleted the counter is incremented/decremented. When the value of the counter is zero, there are no references to the object, and it can be deallocated.

Mark-and-sweep Mark-and-sweep garbage collection has two distinct phases, marking, when the graph of live objects is traversed and all reachable objects are marked, and sweeping, when the entire heap is scanned (or swept) and the storage of objects that were not marked in the first phase is released (usually added to a free list).

Copying In a simple copying collection scheme, the heap is divided into two semi-spaces of equal size (Figure 1). The program (usually called the *mutator*) allocates objects sequentially in one semi-space, until there is no more space left in this semi-space (Figure 1a). At this point, a garbage collection is performed and live objects are traced and copied from the first semi-space (called *from-space*) to the other (*to-space*). After the garbage collection, all the live objects are compacted in the beginning of the second semi-space (Figure 1b). The role of the semi-spaces is then interchanged and the mutator resumes its allocation in the next available position in the semi-space where the live objects were copied to.

Each of these techniques has its advantages and disadvantages. Reference counting can detect garbage quickly, and it does not requires the program to be interrupted for a garbage

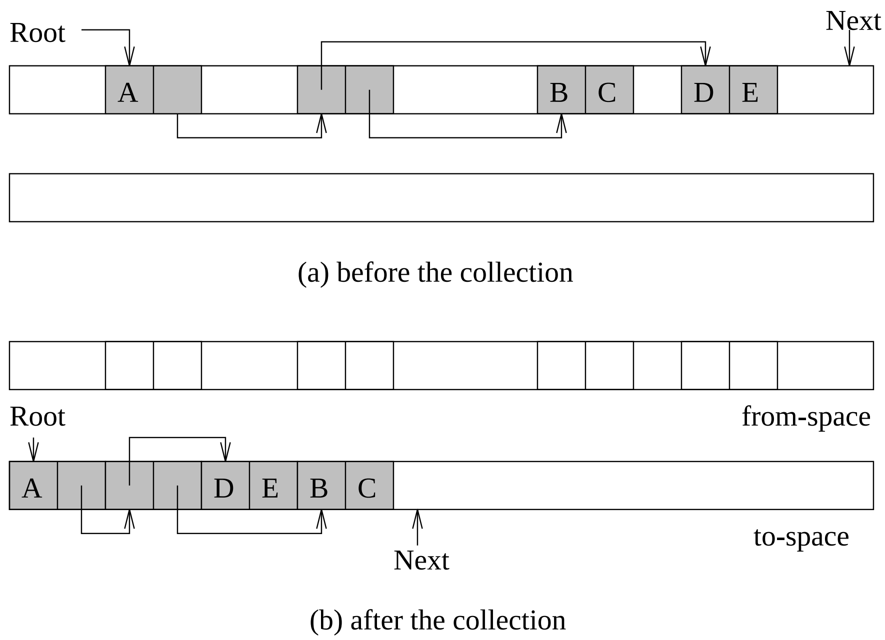


Figure 1: Copying garbage collection

collection, but updating the counters can be costly and the method is not able to handle cycles in the object graph. A copying collector compacts all live objects into one area, thus avoiding memory fragmentation, but it can use only half of the total memory available for allocation. Mark-and-sweep can use all the memory available, but it can lead to memory fragmentation. Copying collection needs to trace only live objects, while mark-and-sweep must scan the entire heap in the sweep phase. If few objects survive collections, then copying collection has an advantage over mark-and-sweep, but mark-and-sweep would be better if most objects survive the collection.

One problem with these garbage collection schemes is that they always trace (and copy) all live objects. Empirical measurements have shown that most objects usually tend to have a short *life* (that is, the period from the time they are allocated to the time they are no longer used by a program, and thus can be deallocated), and only few objects tend to have a long life. Furthermore, if an object survives one garbage collection, then it is likely that it survives many garbage collections. This observation is called the *generational principle* and it leads to an efficient garbage collection scheme called *generational garbage collection* [51, 80].

Generational garbage collected systems divide the heap into several areas, or *generations*, and each generation contains objects of a certain *age* range (the age of an object is the time elapsed since its allocation). The basic idea of generational collection is that not all generations need to be collected at all garbage collections. Generations that contain *older* objects, which are likely to survive many garbage collections, are collected only occasionally, while generations that contain *younger* objects are collected more frequently. Objects are allocated in one generation, usually called the *allocation space*, *youngest generation*, or *nursery*. The allocation space is the most frequently collected generation, and usually only a few objects in the allocation space survive a collection. Objects in a generation that survive a certain number of collections are promoted to the next older generation. Generational garbage collection is most commonly used with copying collection but it can also be used with mark-and-sweep or a combination of the two schemes.

A generational collector must be able to collect only some generations at a time, that is, it should be able to trace live objects in some generations without having to trace the entire heap. Usually objects in one generation can have pointers into objects in any generation, but an older object can only point to a younger object if the value of the older object has been updated since the allocation of the younger object. A *write barrier* can be used to track the (hopefully few) older objects that can point to younger objects, and thus only these need to be traced during a collection of the younger generations.

The design of a generational collector involves the choice of values for many parameters, such as the size of the allocation space, the number of generations, when to promote an object to an older generation, and how often to collect the older generations. Unfortunately, there are no general rules that can be applied, and these parameters are usually adjusted empirically [16, 72, 73, 78, 79, 84, 85].

Allocation-space size

The size of the allocation space determines the frequency of garbage collections. Increasing the allocation space decreases the frequency and cost of garbage collections, since it gives more time for objects to die between collections. Even though the frequency of collections decreases, the number of objects copied per collection usually increases. Interactive applications usually use a smaller allocation space to minimize pause times.

Promotion policy

The promotion policy decides when to promote an object to an older generation. In the simplest case, an object is promoted if it survives a garbage collection of the generation where it currently is. In general, an object can be promoted if it survives a certain number of collections, but if this number is greater than one, some form of keeping record of the "age" of an objects (that is, the number of collections that it survived) is necessary, which can add extra complexity to the collector.

The promotion policy can have a significant effect on garbage collection performance. If objects are promoted to soon, then they are likely to fill up the older generations, forcing frequent collections of those generations. If they are not promoted fast enough, they have to be copied many times, increasing the cost of collections.

Number of generations

The number of generations can vary from two, to an arbitrarily large number, but usually only few generations are used (two to eight). Having multiple generations should be advantageous only if the program tends to have many objects with different lifetimes.

The use of many generations introduces the problem of dividing the heap space among the generations, that is, how much space to assign to each generation. A bad division of the heap, for example, assigning to much space to one generation and too little to another, can not only affect the cost of collections, but also cause a program to run out of memory.

Collections of older generations

Collecting the older generations too frequently can increase garbage collection overhead, but not collecting them frequently enough can result in memory being filled with dead objects.

2.3 The SML/NJ System

This section describes the data representation, memory allocation, and garbage collector of the SML/NJ system. More comprehensive descriptions of the SML/NJ compiler and run-time system can be found in [8, 11, 62].

2.3.1 Heap-allocated closures

The SML/NJ implementation is based on *continuation-passing style* (CPS) [8]. The compiler uses heap-allocated *closures* to represent first class function values. A closure is a data structure that contains the machine-code address and bindings for the free variables of a function.

The usual function call mechanism in the implementation of programming languages involves the use of an activation record, which contains the values needed by the invocation of a function, such as space for the local variables of the function and pointers (*static links*) to the values of the free variables of the function. Activation records are usually allocated in a stack, with a stack frame being allocated when the function is invoked, and deallocated when it returns.

In ML, functions are first-class values, and, because an activation record can contain the free variables of a function that can be returned as the result of a function call and stored in a data structure for later use, activation records can have indefinite extent, and cannot always be stack allocated.

Because of the use of a continuation-passing style, where there is no need to explicitly save the context on a function call, the SML/NJ designers decided to avoid activation records completely, and use heap-allocated closures as the only mechanism for function calls [8]. There are several motivations to this decision [3]:

- the availability of efficient generational garbage collectors that can automatically deallocate closures at little cost;
- heap-allocated closures permit a more efficient data representation;
- heap-allocated closures are safer for space complexity;
- *call/cc* (call with current continuation) can be implemented more efficiently, which can enhance concurrent execution.

A major implication of the allocation of closures on the heap is that SML/NJ programs allocate objects at a very fast rate.

2.3.2 Data representation

The SML/NJ compiler uses a single memory word to represent all ML data types. This word can contain either an integer value (*unboxed* representation), or a pointer to another object that contains the full representation of the data type (*boxed* representation). The low-order bit of a word is used to distinguish between pointer and integer values. (This representation is used in version 0.93. More recent versions of the compiler use *representation analysis* and represent some data types unboxed [65].)

Data objects are allocated on the heap, and each object is preceded by a one word *descriptor* that contains the type and size of the object. ML is a statically typed language and ML programs usually do not need to know the type and size of an object at run-time, except for polymorphic equality testing. The object descriptor is used mainly by the garbage collector.

Many different types of objects are used to implement different ML data types. The use of different types allows a fast identification of objects that may contain pointers, or objects that may contain mutable reference values.

The types of objects that are allocated on the heap are described below.

Record Records are used to represent most immutable ML data types such as tuples, records, and also function closures (including activation records) that are generated by the compiler.

Pair A pair is a record of size two. Pairs are distinguished from other records because they are usually the most common objects used by ML programs, and it is possible to optimize their representation (for example, eliminating the descriptor).

Array Arrays of one or more mutable reference cells.

String Immutable byte string or code objects.

Byte-array Mutable byte strings.

Real Floating-point values.

Real-array Array of floating-point values.

2.3.3 Allocation and garbage collection

The first versions of SML/NJ used a simple two-generational copying garbage collector[5]. Future releases will use a multi-generational collector written by John Reppy[62]. This work uses an earlier prototype of the new run-time system and garbage collector, which is

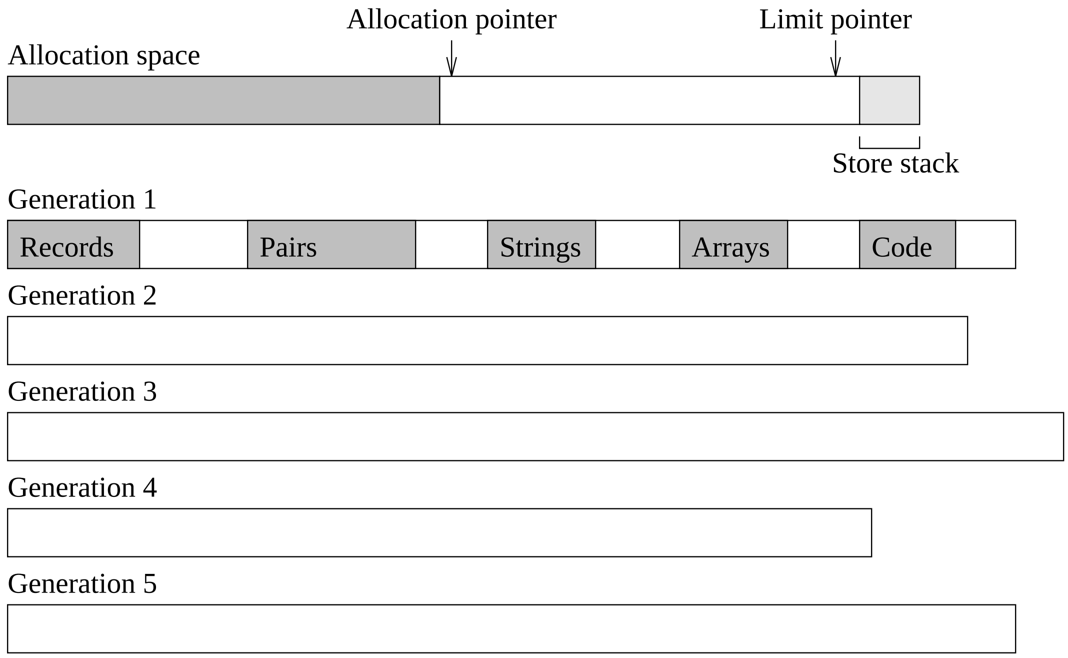


Figure 2: The SML/NJ heap, with five generations.

described in this section (since this work began, Reppy has made changes to the design of the collector, and there are some differences between the collector described here and the one that will be used on future SML/NJ releases).

The new SML/NJ run-time system divides the heap into two main areas, the *allocation space*, or youngest generation, and the older generations. The allocation space is a contiguous, fixed-size memory area. The size of the allocation space can be set by a command line argument. Allocation is controlled by two pointers, the *free pointer*, which points to the next free position, and the *limit pointer*, which points to the end of the allocation area. Figure 2 shows a picture of the SML/NJ heap organization.

The older generations can be divided in up to seven generations, named first through seventh. Objects in a higher numbered generation are always older than objects in a lower numbered generation. Objects in a generation are further divided into two age groups, called the younger and older objects of the generation. Each generation (but not the allocation space) is composed of five *arenas*, one for each of five classes of objects: *records*, *pairs*, *strings*, *arrays*, and *code objects*. Each arena contains objects of a certain type: the record, pair, and array arenas contain records, pairs, and arrays; the string arena contains non-code strings, byte-arrays, reals, and real-arrays; and the code strings are copied to the string arena. (Reppy has since changed this representation, eliminating the code arena, and adding a big-object area, where code strings and big objects are allocated. Big objects are collected using mark-and-sweep collection.)

Most objects are allocated in the allocation space, the only exception being big objects (greater than 2K bytes), which are allocated in the first generation. When the allocation space fills up, a *minor collection* is performed. A minor collection traces live objects in the allocation space, and copies them to the first generation. Since the collector does not know in advance how many objects in the allocation space are live, or the type of these objects, there must be at least enough space in each arena of the first generation to receive all objects in the allocation space. When the space left in any arena of the first generation is less than the allocation space size a *major collection* is performed. Major collections can collect from one to all of the older generations. On a major collection, older objects of generation i that survive the collection are promoted to generation $i + 1$, and younger objects of generation i that survive the collection are promoted to older objects of the same generation. If generation i is being collected, then generation $i+1$ must have at least enough free space to receive all older objects in generation i, otherwise generation $i + 1$ must also be collected.

Size of generations

The size of the allocation space can be set by a command line argument. (Because of the way the check for exhaustion of the allocation space is done on some implementations, there must be an unused area of 4096 bytes at the end of the allocation space, so its actual size is 4096 bytes less then the given size).

The sizes of the older generations changes dynamically during program execution. Each arena of the first generation is initialized to be two times the size of the allocation space. Every time generation i is collected, a new memory area is allocated for it, and its size is determined based on the size of the older area of generation $i - 1$ (or the allocation space if i is one), and the size of the younger area of generation i itself. In general, the size of each generation depends on the size of younger generations. In particular, the size of the first generational tends to be proportional to the size of the allocation space.

Write barrier

For each update to a reference cell that may contain a pointer value, the compiler generates code to record the updated address. This address is added to a store list that is implemented as a stack, on the higher area of the allocation space, having the limit pointer as the stack pointer. Thus, the limit pointer is decremented by one word every time an entry is added to the store stack. The store stack is used as part of the root set during minor collections. (Reppy has since abandoned this representation in favor of a linked list kept in the allocation space with other objects.)

The BIBOP table

The heap is divided into units of 64K bytes each, and the allocation space and older generations are made of a contiguous number of those units. The BIBOP table [41] contains a mapping of which heap area is assigned to each unit. Given an address, by looking up the appropriate entry in the table it is possible to find if that address is on the allocation space or a specific arena of one of the older generations. On major collections, the BIBOP table is also used to determine if objects are in *from-space* or *to-space*. Given a pointer to an object, the use of the BIBOP table allows the collector to quickly determine the type and location of the object.

Chapter 3

Methodology and Tools

This chapter describes the methodology and tools used in studying the cache performance of a set of ML programs. Studies of cache performance traditionally use computer simulation, analytical models, or a combination of both[58]. This study is based on simulation and a simple analytical model. An instruction level simulator for the MIPS architecture is used to obtain accurate measurements of instruction counts and cache references. This simulation technique is equivalent to *trace-driven* simulation, a commonly used method of analyzing cache performance. The results of the simulations are used in combination with the performance model to predict the overall performance of programs.

The chapter begins with a description of the benchmark suite of ML programs that were used on this study. Section 3.2 presents the performance model used to predict the overall performance of programs, and compares the predictions of this model with actual measurements of program run time. Section 3.3 discuss the limitations of the this study, and finally, Section 3.4 describe the tools used on the experiments.

3.1 Benchmarks

The benchmark suite of ten ML programs used in this study are described in Table 2. These programs are part of a benchmark suite that have been used in other research that evaluates the performance of SML/NJ programs[10, 64]. The programs Knuth-Bendix, Lexgen, Life, MLYACC, Simple, and VLIW have also been used in studies of cache performance and object behavior of SML/NJ[29, 73].

Four of these programs, Barnes-Hut, Mandelbrot, Ray, and Simple, are floating-point

Table 2: Description of the benchmark programs.

Program	Lines	Description
Barnes-Hut	1060	The Barnes-Hut N-body simulation program [15, 14], translated into ML by John H. Reppy.
Boyer	910	An ML implementation of the Boyer theorem prover benchmark [37].
Knuth-Bendix	580	An implementation by Gerard Huet of the Knuth-Bendix completion algorithm translated into ML by Xavier Leroy, processing geometry axioms.
Lexgen	1178	A lexical-analyzer generator, written by James S. Mattson and David R. Tarditi[12], processing the lexical description of Standard ML.
Life	140	Reade's implementation of the game of Life [59], running 50 generations of a glider gun.
Mandelbrot	60	A program to generate Mandelbrot sets, written by John H. Reppy.
MLYACC	7422	A LALR(1) parser generator, written by David Tarditi [76], processing the grammar of Standard ML.
Ray	423	A ray tracer, written by Don Mitchell, and translated into ML by John H. Reppy.
Simple	906	A spherical fluid-dynamics program [23, 33], translated into ML by Lal George.
VLIW	3571	A VLIW instruction scheduler written by John Danskin, processing a 97-instruction program.

Table 3: Total, user, garbage collection, and system times in seconds for the benchmark programs, measured on a DEC5000/240 work-station, with an allocation space of 512K bytes.

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

Table 4: Instruction counts, number of reads and writes for user code and garbage collection. The numbers inside parenthesis give the number of reads and writes as a percentage off all instructions. Garbage collection counts are for an allocation space of 512K bytes. All numbers are in millions.

Program	User					GC				
	instr.	reads		writes		instr.	reads		writes	
Barnes-Hut	617.5	179.8	(29.1%)	125.5	(20.3%)	39.7	6.1	(15.4%)	4.7	(11.8%)
Boyer	42.9	9.5	(22.2)	9.0	(20.9)	27.1	5.0	(18.5)	3.7	(13.7)
Knuth-Bend.	281.0	57.2	(20.4)	67.0	(23.8)	73.8	14.2	(19.2)	10.2	(13.8)
Lexgen	296.0	55.4	(18.7)	38.2	(12.9)	25.6	4.7	(18.4)	3.3	(12.9)
Life	422.9	50.0	(11.8)	38.3	(09.1)	4.9	0.9	(18.4)	0.6	(12.2)
Mandelbrot	367.6	82.7	(22.5)	53.1	(14.4)	1.2	0.2	(16.7)	0.1	(08.3)
MLYACC	97.8	21.3	(21.8)	18.7	(19.1)	32.1	5.8	(18.1)	4.3	(13.4)
Ray	679.2	192.4	(22.8)	141.3	(17.0)	2.7	0.5	(18.5)	0.3	(11.1)
Simple	764.3	174.3	(28.3)	130.3	(20.8)	232.0	38.0	(16.4)	27.5	(11.8)
VLIW	427.2	82.1	(19.2)	73.1	(17.1)	19.7	3.3	(16.8)	2.2	(11.2)
Arith. mean			(21.7%)		(17.5%)			(17.6%)		(12.0%)
Total	3664.6	877.0	(23.9%)	694.4	(19.0%)	458.8	78.7	(17.2%)	56.9	(12.4%)

intensive applications. The other programs use mostly integer instructions. Table 3 shows total run time of the programs when a 512K-byte allocation space is used, measured on a DEC5000/240 workstation [27], and the time breakdown by user, garbage collection, and operating system time. Because there may be a difference on the reported time on different runs of the same program, each program was run many times, and the run with minimum time taken (as recommended by the SPEC benchmark consortium [21]). Garbage-collection times may vary significantly when a different size for the allocation space is used. User time can also vary, as the size of the allocation space can affect the cache and virtual memory performance of the programs.

Garbage collection overhead (the fraction of total time spent on garbage collections) can vary significantly from one program to another. Boyer spends almost forty percent of its time on garbage collections. MLYACC and Simple also have high garbage collection overhead, more than twenty percent, but other programs, like Mandelbrot and Ray, have almost no overhead for garbage collection. System overhead, which includes the time spent on system calls, is usually small, the highest being eight and a half percent for Boyer.

Table 4 shows the number of instructions executed, and read and write references made by each program, broken down by user and garbage collection code. The garbage collection numbers are for a 512K-byte allocation space. The number of garbage collection instructions may vary significantly if a different allocation space size is used, but the number of user instructions remains constant. Table 5 shows garbage collection instructions executed when

Table 5: Garbage collection instruction counts when different allocation space sizes are used. In millions.

Program	GC instructions									
	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M
Barnes-Hut	193.7	116.6	90.1	55.8	44.8	39.8	36.4	35.3	32.6	28.4
Boyer	78.8	79.7	76.5	68.4	40.9	27.1	21.5	15.3	16.8	20.0
Knuth-Bend.	140.2	90.1	90.1	82.3	75.8	73.8	72.0	40.7	28.1	14.3
Lexgen	69.6	50.1	48.8	36.2	28.5	25.6	24.0	15.4	11.8	11.6
Life	29.7	17.0	10.3	7.3	5.7	4.9	4.0	2.9	1.8	1.3
Mandelbrot	18.8	9.1	4.9	2.9	1.7	1.2	0.8	0.6	0.6	0.6
MLYACC	78.1	56.0	49.9	43.5	36.1	32.1	25.2	15.1	13.0	9.5
Ray	81.1	35.9	17.5	8.6	4.6	2.7	1.7	1.1	0.8	0.7
Simple	576.2	544.3	419.6	330.1	267.3	232.4	128.1	91.6	105.4	64.0
VLIW	111.3	73.7	51.6	35.4	24.0	19.7	13.8	10.7	9.2	10.1

the programs are run with different allocation space sizes, varying from 16K to 8M bytes. The number of garbage collection instructions usually decreases as the allocation space increases, but because major collections are sensitive to the placement of collection points, sometimes the opposite happens (see, for example, the numbers for Simple for 2M and 4M allocation spaces). This problem will be discussed in more detail in section 4.4.

3.2 Performance Model

A simple analytical model is used to predict performance of the programs under different cache organizations. This model assumes a machine that can issue one instruction per cycle, executed in a strictly sequential order, and no pipeline stalls, except for memory accesses on cache misses. Loads and stores that hit the cache take one cycle, and cache misses take a fixed number of cycles. Caches are not lockup-free, that is, the CPU stalls for a number of cycles equal to the miss penalty. Different read and write miss penalties are allowed. The model does not consider the effects of writing dirty blocks, or words to memory (that is, the machine has an ideal write buffer), nor the effects of other features, such as TLB misses or virtual memory. The total number of cycles of a program is given by:

$$TotalCycles = I + I_{miss} \cdot R_{penalty} + R_{miss} \cdot R_{penalty} + W_{miss} \cdot W_{penalty} \quad (1)$$

where I is the instruction count, I_{miss} the number of instruction fetch cache misses, R_{miss} the number of read misses, W_{miss} the number of write misses, R_{penalty} the penalty for read

Table 6: Measured time on a DECstation 5000/240 compared to the time predicted by the simple machine model. All times are in seconds. The size of the allocation space is 512K bytes. The value on the column labeled *Difference* is given by $((Measured\ time/Predicted\ time) - 1) * 100$.

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

misses in cycles, and W_{penalty} the penalty for write misses in cycles.

This model assumes a machine with a fetch-on-write cache organization, in which the fetch of a block on a write miss causes the pipeline to stall. Many real machines do allow the pipeline to continue on write misses, while the block is fetched into the cache, but because there is usually a limit on the number of pending cache transactions, the pipeline may stall eventually on a subsequent instruction accesses that accesses the cache. This model can also be applied to write-validate and write-around machines (for which there are no write misses) by making the number of write misses to be zero.

Although this is a very simple model, it can still predict the performance of some programs on some real machines with reasonable accuracy. Table 6 compares the run time of each program, measured on a DEC5000/240 work-station[27], with the time predicted by the model. The DEC5000/240 has direct mapped, 64K-byte instruction and data caches, with 4-byte (one 32-bit word) blocks. On a read miss, the machine loads 8 consecutive blocks, aligned to an 8-block boundary, with a penalty of 29 cycles[1]. On writes, the word is written to the cache, with no fetches from memory, so there is no write miss penalty. The cache is write through and has an eight word write buffer that should handle most writes of

[1]There is an 8-cycle latency for page mode memory reads, 40ns per word to load each entry of a read buffer, and 8 cycles to transfer the words from the buffer to the cache [27]. The CPU cycle time is 25ns.

SML/NJ programs with negligible overhead [29]. For predicting the time on this machine with the model, the cache organization was approximated by 32-byte block write-validate cache, which should behaves similarly and give a close approximation of the actual miss ratios on the machine.

The numbers on Table 6 show that for some programs, the predicted run times are close to the measured run times. But there are some cases for which the difference can be high. Most of the difference should be caused by the programs performing instructions that take more than the one cycle assumed in the model. VLIW, the program with highest relative error, executes many integer division instructions, which take many cycles to complete. Barnes-Hut, Mandelbrot, and Simple do many floating point instructions, which also may take multiple cycles. Knuth-Bendix is the only case for which the high difference can not be explained by multiple cycle instructions.

Evaluating garbage collection overhead

A common metric used to evaluate performance of programs is cycles per instruction, or CPI. Using CPI, we can easily estimate the effects of cache misses on performance. In this study, we are in interested in the effects of garbage collection and cache misses on the performance of programs. But since changing garbage collection parameters, such as the size of the allocation space, may change the total of instructions executed by a program, we need a performance metric that takes this change into account. Since the number of user instructions is kept constant even if garbage collection parameters are changed, the total number of cycles can be divided by the number of user instructions giving a metric similar to CPI that can be used to estimate the effects of both garbage collection and cache misses.

We define CPUI, or cycles per user instructions, as:

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

where *TotalCycles* is the total number of cycles computed by the formula given above, and *number of user instructions* is the total number of instructions minus garbage collection instructions.

Summarizing results

Results for all the benchmark programs are usually summarized by assuming a workload where each benchmark has an equal weight. An unweighted arithmetic mean is always used to summarize the different metrics used to analyze the results, such as cache miss ratios, or CPUI. This use of arithmetic mean is consistent, since the quantities being averaged are not normalized numbers[36, Rule 3].

3.3 Limitations

This study makes many simplifications, some of which have already been described above.

The simulation results do not include the effects of operating system functions and multiprogramming context switching. As shown in Table 3, operating system functions can account for almost ten percent of total program run time. Operating system and context switching tend to have a small effect on small caches, but can significantly increase the miss ratios on large caches [1].

Only simple cache architectures are considered. Many features common on caches of current machines, such as write buffers, non-blocking misses, and multi-level cache hierarchies are not considered. (In the case of multi-level cache hierarchies, the miss ratios of a single level cache can be used as a good approximation of the miss ratios of the second level cache, if the first level cache is small and the second level cache is much larger than the first [58, Section 5.3].) The performance model also ignores a number of factors that may affect performance, multi-cycle instructions, memory bus traffic delays, TLB misses, out of order execution, and virtual memory. The effects of memory bus traffic delays should be negligible if a deep enough write buffer (four to six words) is used [29].

The study is restricted to a group of relatively small ML programs. Since different programs may have very different behavior, the results and conclusions presented here may not be valid for other programs. It is certainly not possible to study the behavior of all programs, but if at least it could be claimed that this benchmark suite is a representative sample of the whole class of ML programs, it would be reasonable to generalize the conclusions of this study to all ML programs. But since this benchmark is limited in size, it may not be truly representative of all ML programs.

Nevertheless, it is possible to identify features on these benchmark programs that should be common to most ML programs, like, for example, a cyclic allocation pattern and many references to recently allocated data. Therefore, it is reasonable to expect that many of the results of this study will be valid to most ML programs, and even to a broader class of functional programs with intensive heap allocation and generational garbage collection.

3.4 Tools

3.4.1 The MIPS instruction level and cache simulators

MIPSI, a MIPS instruction level simulator written by E. G. Sirer, was used to collect data on instruction counts, cache miss ratios, and memory reference patterns of the benchmark programs. MIPSI can run most MIPS binary files in UNIX *a.out* format. No modifications to the binary file are needed. System calls made by the simulator are intercepted by the simulator, which makes an actual system call and sends back the results to the program.

The simulator was ported to an OSF/1 environment, and the simulations were run on a DEC Alpha work-station. Because the binary files being simulated are for a MIPS/Ultrix environment and there is a small difference in the argument format of some system calls, it is necessary to translate arguments from Ultrix to OSF/1 format before making the actual call, and then map the results back to Ultrix format, so that the programs will get the correct results.

The simulator was extended with a cache simulator capable of simulating multiple cache architectures, and with an interface to the SML/NJ run-time system, that permits the collection of data specific to SML/NJ programs.

Cache simulations

Cache simulations are done "on the fly" without the explicit generation of address traces. The cache simulator can produce miss rates for several caches in one pass. In general, miss ratios for all cache sizes, for a fixed block size and write-allocate policy, are obtained in one pass. For direct-mapped fetch-on-write caches the technique of *forest simulation*[44] is used to speed up simulation (when simulating multiple direct-mapped caches, if a reference hits in a small cache, then it also hits in all bigger caches). For associative fetch-on-write caches a small direct-mapped cache is used to avoid looking up the references that hit on the small cache[44, 74] (if the reference hits in the direct-mapped cache, then it also hits in the associative caches).

Interface with the SML/NJ run-time system

The experiments that are described later in this thesis often require information that is specific to SML/NJ programs, such as distinction between user and garbage collection instructions, and the classification of memory references according to the heap area that contains the address (that is, allocation space or older generations). To collect these SML/NJ specific data, MIPSI was extended with an interface to the SML/NJ run-time system. The extended MIPSI can access the BIBOP table (see Section 2.3.3) to find the arena and generation of a memory reference, and it can identify the beginning and end of minor and major collections.

The simulator matches the instruction that begins a garbage collection to recognize the point where a collection begins. On the MIPS machine, this instruction is

SLT r21, r23, r19,

which compares the contents of register 23, the allocation pointer, with register 19, the limit pointer, and set register 21 to a non-zero value if the value of the allocation pointer is less than the value of the limit pointer. If the value left in register 21 is zero, the simulator assumes that a garbage collection begins with the next instruction. At the end of a garbage collection, another SLT instruction is executed, producing a non-zero value on register 21.

Some minor collections may not begin with the sequence describe above. For these collections, a different method is used, that is simply to wait for a JALR (jump and link register) instruction to the entry point for garbage collection code. Garbage collection terminates with a jump to the return address of the JALR instruction. The same technique is used to determine the begin and end of major collections.

There is a problem with the first method of detecting garbage collections. Since the run-time system, and library functions, are written in C, it is possible that some C function executes a SLT instruction that is identical to the one that begins garbage collections. But it is possible to combine the two methods describe above, and thus find if this problem happens in the benchmark programs. False garbage collection detection do not occur in any of the benchmark programs.

An additional way of interfacing the simulator with the SML/NJ run-time system is to use a special system call. This method is used, when necessary, to pass the starting address of the allocation space (which is only available at run-time, unlike the addresses of the BIBOP table and garbage collection entry points). The use of this method requires the addition of code to the run-time system to make the system call, but this extra code should have no effects on the results of the experiments.

3.4.2 Garbage collector instrumentation

The SML/NJ garbage collector was instrumented to collect data on allocation by the benchmark programs. Before each minor collection, and at the end of program execution, the allocation space is scanned, and the type and size of each object is recorded. The first generation is also checked for the allocation of large objects and code strings. During garbage collections, the size and type of each object copied are also recorded to produce statistics on object survival rates.

Besides instrumenting the collector, the compiler itself was also modified to generate a special descriptor for closures, so that it is possible to distinguish between closures and user data structures at run time. The modified compiler and garbage collectors were used only on experiments that require their use. Most experiments used the regular garbage collector and compiler.

3.4.3 Running the benchmark programs

To run SML/NJ with the new run-time system, the file that is executed by the operating system is the run-time system itself. The run-time system then either loads an existing heap image, or creates a new one, and then transfer control to the SML/NJ compiler. On the simulations, each program is run in the following way. First, the programs is compiled and a binary heap image of the entire heap is exported. Then, the simulator is used to run the run-time system, which loads the saved heap image with the compiled code and run the program. All default compiler parameters are used for compiling and running the programs.

The experimental results do not include the compilation of the benchmark itself, but to run each benchmark, a small expression that invokes the programs must be compiled and evaluated. The compilation of this expression should have a negligible effect on the performance of the programs.

Chapter 4

Memory Reference Patterns

This chapter describes the way the benchmark programs use memory. This study will be helpful to understand the results presented in later chapters. The memory references made by SML/NJ programs are divided in two groups, *garbage collection references* and *user references*, where user references are all references that are not garbage collection references. User references may be further divided into *ML references*, that are references to the allocation space or an older generation, and *run-time system references*, that are references to areas outside the ML heap, that are made by I/O or other library functions, or during system start-up and termination.

The chapter begins with a profile of allocation and an analysis of the lifetime of objects. Objects in the heap can be divided into closures and user data objects, with closures being the most common type. Objects tend to be small, but closures are about twice the size of user objects on average. Most objects have a short life, but the lifetime of closures tends to be shorter than user data objects, which results in user objects being predominant in the older generations. Section 4.2 describes the write patterns, showing that most of the writes are for allocation and follow a cyclic pattern. The patterns for reads are presented in Section 4.3, where it is shown that all programs do many reads to recently allocated objects. Finally, Section 4.4 describes the memory reference patterns for minor and major collections.

Table 7: General allocation data on the benchmark programs: total words allocated; allocation rate (words allocated divided by user instructions); average object size (including the descriptor); percentage of words allocated that are descriptors; and the fraction of allocation that is for big objects.

Program	Words allocated	Allocation rate	Average obj. size	Descriptors	Big objs.
Barnes-Hut	124,372,070	0.201	4.25 words	23.52%	-
Boyer	8,802,892	0.205	5.40	18.52	-
Knuth-Bendix	66,837,526	0.238	4.88	20.51	-
Lexgen	33,024,745	0.112	5.48	18.26	-
Life	38,104,988	0.090	4.66	21.46	-
Mandelbrot	48,712,866	0.133	4.59	21.77	-
MLYACC	17,635,415	0.180	5.82	17.18	0.14%
Ray	139,506,978	0.169	4.22	23.72	-
Simple	129,215,742	0.205	4.26	23.49	0.23
VLIW	67,076,487	0.157	5.01	19.98	0.07
Average		0.169	4.857	20.841	

4.1 Allocation and Object Behavior

Allocation Profile

Table 7 gives general allocation data for the benchmark programs: total number of words allocated, allocation rate, number of objects allocated, average object size, and percentage of allocated words that are descriptors. The *allocation rate* is the average number of words allocated per user instruction, and is computed by dividing the number of user instructions

Table 8: Allocation by type.

Program	Record	Pair	Array	String	Byte-array	Real
Barnes-Hut	52.04%	15.76%	-	0.15%	-	32.04%
Boyer	76.23	23.66	0.10%	0.01	-	-
Knuth-Bendix	80.34	19.65	0.01	-	-	-
Lexgen	80.92	18.01	0.15	0.93	-	-
Life	77.68	22.30	0.02	-	-	-
Mandelbrot	60.69	0.09	0.02	-	-	30.20
MLYACC	74.84	24.65	0.11	0.40	0.01%	-
Ray	57.68	7.93	0.01	-	-	34.39
Simple	57.69	32.56	0.01	-	-	9.75
VLIW	73.25	25.37	1.06	0.32	-	-

Table 9: Allocation break-down by closures and user data.

Program	Closures		User data	
	alloc. words	Avg. size	alloc. words	Avg. size
Barnes-Hut	44.81%	7.81 words	55.19 %	3.10 words
Boyer	81.98	6.53	18.02	3.02
Knuth-Bendix	86.97	5.37	13.03	3.02
Lexgen	80.51	6.52	19.49	3.30
Life	77.83	5.53	22.17	3.00
Mandelbrot	60.68	7.00	39.32	3.00
MLYACC	71.39	8.24	28.61	3.36
Ray	42.07	5.58	57.93	3.58
Simple	60.88	5.78	39.12	3.02
VLIW	76.91	6.33	23.09	2.95
Average	68.40	6.47	31.60	3.13

(from Table 4) by the number of words allocated by each program. All programs allocate at a very fast rate: the lowest rate, for Life, is one word per 11 instructions, and the highest, for Knuth-Bendix, is close to one word per four instructions.

Objects tend to be small, less than five words on average, including the one-word descriptor that precedes the objects. Descriptors are used only by the garbage collector and polymorphic equality testing, but since most objects do not survive a first garbage collection and the polymorphic equality test is not frequently used, most descriptors are never used. Descriptors account for a significant fraction of the words allocated, about 20% on average. Pairs that survive a garbage collection are copied to the pair arena without a descriptor.

Most objects are allocated in the allocation space. The only exceptions are code strings and big objects (of size equal to two kilobytes or more), which are allocated in the first generation. Code and big objects are a negligible fraction of the total allocation for the programs in the benchmark suite.

Table 8 show the allocation break-down by the type of objects. Records and pairs are by far the most common objects used by all programs, but floating point benchmarks also make extensive use of real values.

Objects can be divided into two main groups: closures and user data. Table 9 shows the distribution of allocation among those groups, and the average size of objects in each group. In most programs, closures account for more than half of the total allocation. On average, 68% of the allocation is for closures. Closure objects are typically much larger than user data objects. Pairs (and reals in the floating point benchmarks) are the most common

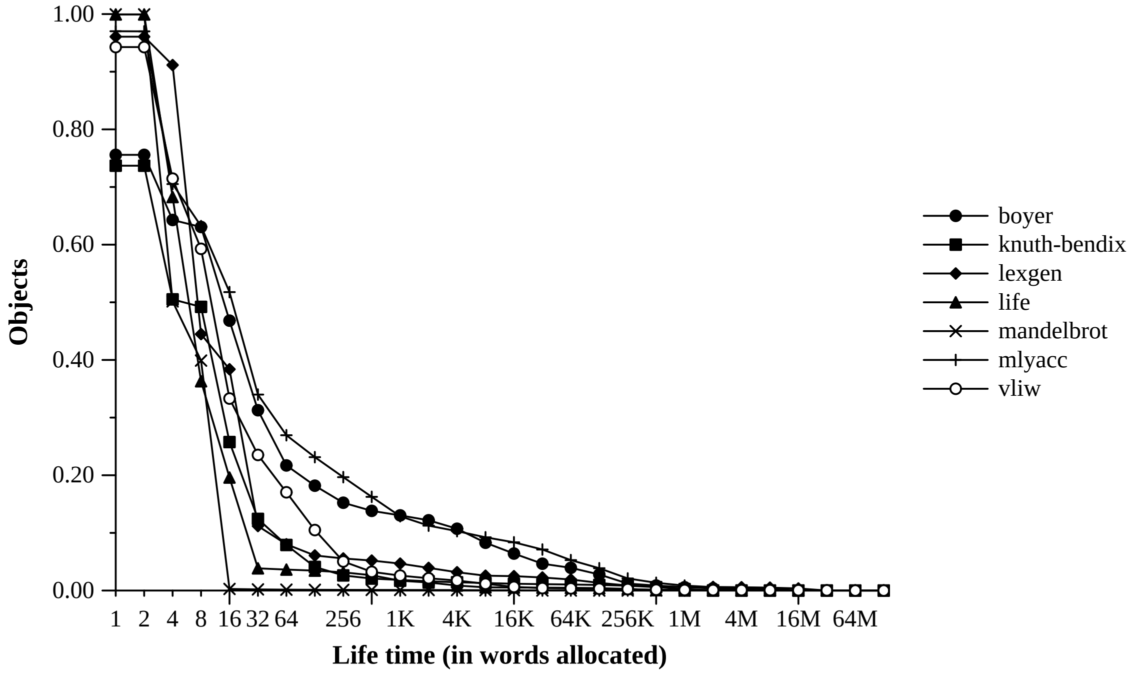


Figure 3: Life time in words allocated.

objects used to represent user data.

Object lifetime

Most objects have a very short life. Figure 3 shows the lifetime distribution of objects for some programs of the benchmark suite, with time measured in words allocated. An object is live from the time of its allocation, to the time of the last read or write to any word within the object (with an "infinite" heap, the lifetime of an object is given by the value of the allocation pointer at the time of the last reference to the object, minus the offset of the descriptor from the allocation base). Most objects live for at most a few hundred words of allocation. Some programs have many objects that are allocated but never used, as happens with Boyer and Knuth-Bendix (it is no surprise that Knuth-Bendix does not read many of the objects it allocates, since the numbers in Table 4 show that this program does more writes than reads).

An alternative definition for the lifetime of an object is to consider the object live if it survives a garbage collection, that is, the object can be reached by traversing pointers from a set of standard roots. Many objects may be reachable, even though they may never

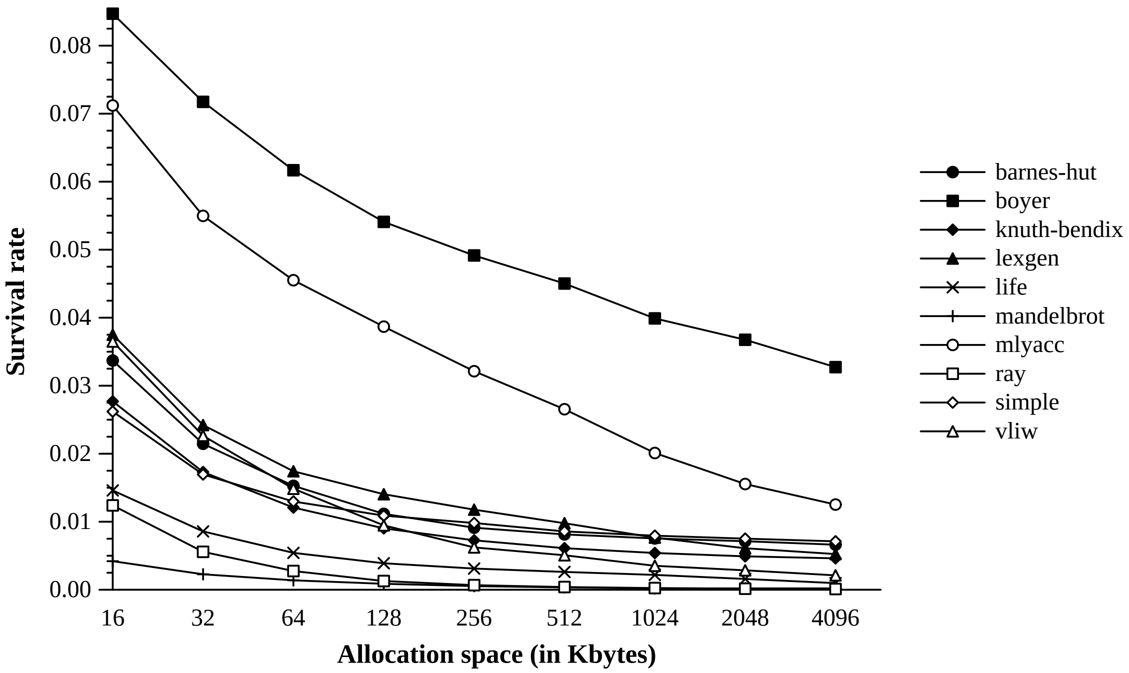


Figure 4: Survival rate (fraction of allocated words that survive minor garbage collections) as a function of the size of the allocation space.

be referenced again, so the set of reachable objects at a certain time is a superset of the live objects according to the first definition. As an example, consider a program that uses a large data structure, such as a dictionary. Depending on the inputs, the program may reference only a few entries in this dictionary, but it can not be known in advance which entries will be used.

Even if object lifetime is defined by reachability, it is still true that most objects have a very short lifetime. Figure 4 shows the *survival rate*, the fraction of allocated words that survive a minor garbage collection, for different sizes of the allocation space. For all programs the survival rate tends to decrease as the size of the allocation space increases, since there is more time between collections (this may not always be the case, since the survival rate may depend on the placement of collection points). But even when a small allocation space is used, only a small fraction of all words allocated survive a minor collection. The number of words that survive a minor collection can vary significantly from one program to another. The survival rate for Boyer and MLYACC, for example, can be several times higher than that of other programs. The survival rate is directly related to the garbage collection overhead of a program: the higher the survival rate is, the higher is garbage

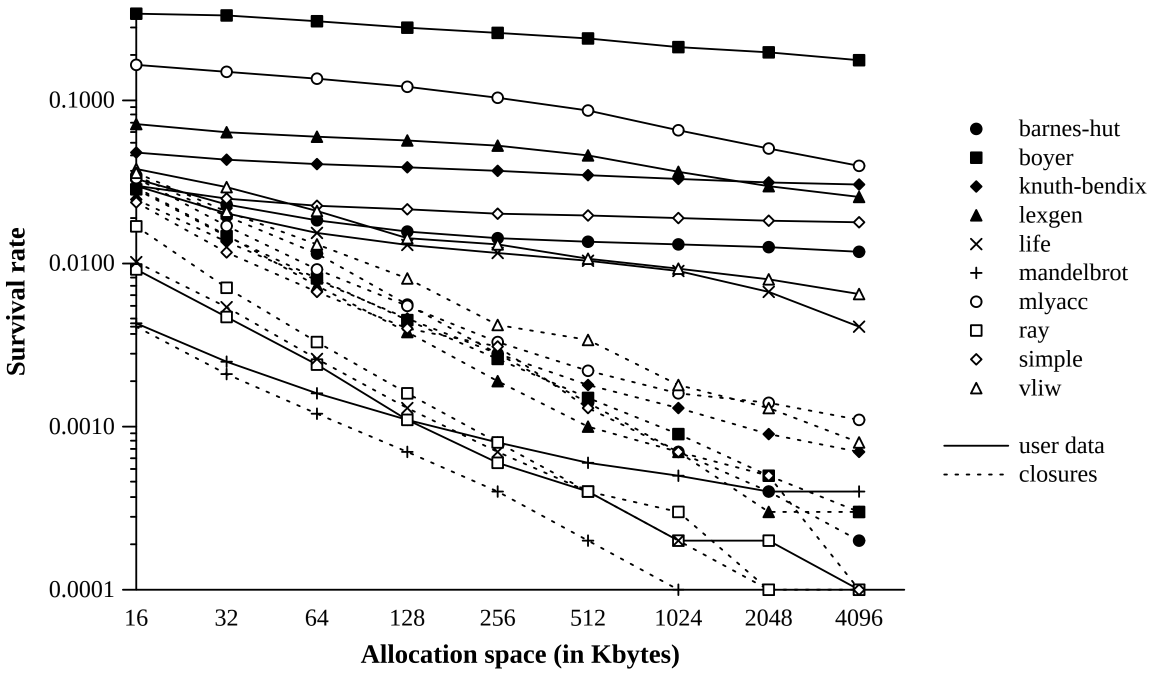


Figure 5: Survival rate of user data and closures. (Note the logarithmic scale.)

collection overhead.

Closures and user objects have different lifetimes. Figure 5 shows the survival rates for closures and user data. Except for one program, the survival rates for closures are much lower than the survival rates for user data, which means that the lifetime of closures tend to be shorter than the lifetime of user data objects. As a result, very few closures are ever promoted to the older generations. Table 10 shows the distribution of all objects that are promoted to the first generation. Although most allocated objects are closures, most objects copied on garbage collections are user objects. Since user data objects tend to be small, objects in the older generations tend to be smaller than objects in the allocation space. The fact that pairs in the older generations are not preceded by a descriptor also contributes to make older objects smaller than younger ones. The distribution of objects that are promoted to the first generation is likely to change if a different-size allocation space is used, but, in general, average object size tends to be small, with a predominance of user objects over closures. Table 11 show the average size of objects that are promoted to older generations, for different sizes of the allocation space. The size of objects tends to decrease as the size of the allocation space increases, since more closures die before being

Table 10: Break-down of data the survive minor collections (512K-byte allocation space) by closures and user data objects, and average size (in words) of all surviving objects. User data objects are further divided according to type. Depending on their size, closures can be either records or pairs (but most should be records).

Program	Closure	User data						Avg.
		Record	Pair	Array	String	Byte-ar	Reals	size
Barnes-Hut	7.74%	26.20%	3.63%	12.50%	0.00%	-	49.93%	3.47
Boyer	2.80	1.08	95.60	0.49	0.03	-	-	2.08
Knuth-Bendix	25.73	1.02	72.78	0.47	0.00	-	-	2.64
Lexgen	8.35	2.90	74.03	12.90	1.67	0.16%	-	2.54
Life	12.38	4.15	81.43	1.91	0.13	-	-	2.33
Mandelbrot	31.32	24.83	16.54	11.42	0.76	-	15.13	3.94
MLYACC	5.97	8.98	79.31	2.39	3.18	0.17	-	2.27
Ray	46.84	16.97	6.62	3.47	0.24	0.46	25.39	4.29
Simple	9.45	0.41	10.37	0.22	0.01	-	79.53	3.13
VLIW	51.29	11.59	31.31	0.76	4.81	0.23	0.00	3.55
Average	20.19	9.81	47.16	4.65	1.08	0.10	17.00	3.02

Table 11: The average size of allocated objects and average size (in words) of objects that are promoted to the older generations for different sizes of the allocation space.

Program	Obj.	Avg. size of objects surviving GC								
	size	16K	32K	64K	128K	256K	512K	1M	2M	4M
Barnes-Hut	4.25	4.45	4.25	4.01	3.69	3.47	3.34	3.26	3.22	3.19
Boyer	5.40	2.49	2.28	2.18	2.11	2.08	2.05	2.04	2.03	2.03
Knuth-Bendix	4.88	4.16	3.67	3.26	2.89	2.64	2.46	2.35	2.28	2.23
Lexgen	5.48	3.67	3.21	2.87	2.65	2.54	2.52	2.61	2.72	2.85
Life	4.66	3.16	2.92	2.68	2.45	2.33	2.22	2.19	2.17	2.23
Mandelbrot	4.59	4.69	4.57	4.39	4.14	3.94	3.59	3.35	3.19	3.09
MLYACC	5.82	2.81	2.54	2.39	2.31	2.27	2.27	2.28	2.32	2.37
Ray	4.22	4.45	4.24	4.39	4.36	4.29	4.19	4.18	3.92	3.63
Simple	4.26	4.01	3.59	3.32	3.10	3.13	2.98	2.95	2.96	2.94
VLIW	5.01	4.55	4.13	3.95	3.94	3.55	3.55	3.31	3.15	3.04

Table 12: Distribution of allocation and non-allocation writes.

Program	Allocation	Non-allocation	
		ML	run-time
Barnes-Hut	99.07%	0.71%	0.22%
Boyer	98.26	0.85	0.89
Knuth-Bendix	99.73	0.12	0.14
Lexgen	86.52	1.91	11.57
Life	99.59	0.19	0.22
Mandelbrot	91.81	8.04	0.15
MLYACC	94.35	3.12	2.54
Ray	98.71	1.17	0.12
Simple	99.41	0.53	0.07
VLIW	91.81	4.32	3.87
Average	95.93%	2.10%	1.98%

promoted to older generations.

4.2 User Writes

This section describe the user write patterns of the benchmark programs. Garbage collection writes will be considered in Section 4.4.

Since the ML language encourages programmers to use a mostly functional programming style, most objects in ML programs tend to be written only once, when they are first allocated. Writes that initialize an object in the allocation space are called *allocation writes* and all other writes are called *non-allocation writes*. Non-allocation writes are divided into ML writes, that are writes to the ML heap, to update a reference variable and the write barrier (writes to initialize a big object or code string allocated in the first generation are also classified as non-allocation but, as Table 9 shows, there are very few of those), and run-time system writes, that are all writes to areas outside the ML heap. Most writes in the benchmark programs are allocation writes, as shown in Table 12. Most programs have few non-allocation writes.

Since most writes are for allocation, and objects are allocated sequentially in the allocation space, the writes have a sequential pattern, that repeats itself with a frequency that depends only on the size of the allocation space (writes are not strictly sequential, because objects may be initialized in reverse order of fields, that is, the last field is initialized first.). The program begins allocating at the first address of the allocation space and continues

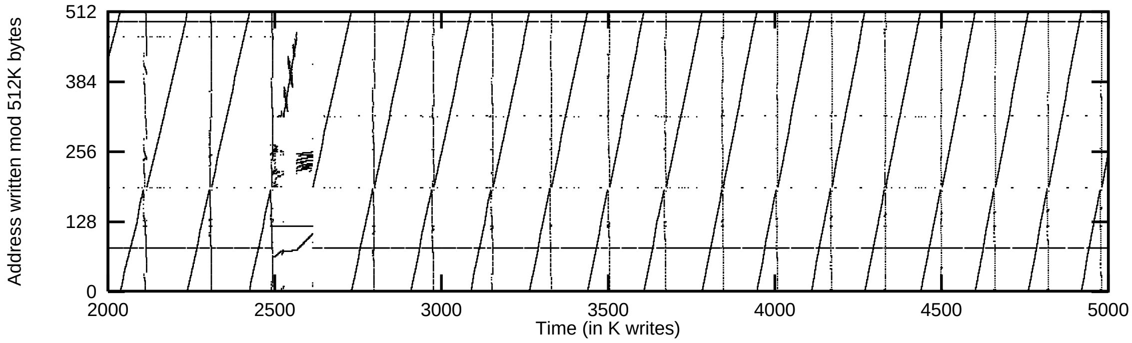


Figure 6: Write patterns for Lexgen.

advancing through the allocation space until the allocation limit is reached. At this point, a garbage collection is called to clear the allocation space, and after that another cycle begins.

This cyclic write pattern can be seen graphically in Figure 6, which shows the write references for the program Lexgen. The graph shows a plot of the address written (modulus 512K bytes) as a function of time, where time is measured in words written. This is the way the reference stream is seen by a 512K-byte cache. The unit of the time axis is 1024 writes, and the unit of the address axis is 1024 bytes (units of 1024 are used on both axes to reduce the number of points plotted; adding more points would not enhance the visual quality of the graphs in a noticeable way). The graph shows only a small interval during the execution of Lexgen, from time 2000K to 5000K. The program runs until time 42000K.

The cyclic allocation pattern can be clearly distinguished as the inclined lines. The vertical lines that appear at the end of each allocation cycle are minor garbage collections. The unusual pattern that occurs near time 2500K is a major collection. The horizontal lines correspond to non-allocation writes, and it can be seen that there are repeated writes to a few locations (since each dot corresponds to a 1K-byte area, or 256 addresses, the horizontal lines may be the result of several nearby locations being updated.) The different density of the horizontal lines means that some locations are updated more frequently than others. One horizontal line (the second from top) seems to move down after the major collection, which is likely to be caused by a reference variable being moved to a new location during the collection.

These patterns for user write reference give some hints on what to expect about cache performance for writes. If the allocation space is smaller than the cache, then most blocks of the allocation space should be in cache (except for those evicted because of conflicts with

Table 13: Read references to closures, user data objects, compile-time objects, and run-time system objects.

Program	Closure	User	Compile-time	Run-time sys.
Barnes-Hut	30.46%	52.43%	14.10%	3.01%
Boyer	44.42	20.06	33.95	1.57
Knuth-Bendix	59.50	36.13	4.09	0.28
Lexgen	35.34	45.47	7.55	11.64
Life	45.17	42.84	11.70	0.29
Mandelbrot	30.64	30.87	38.32	0.17
MLYACC	52.49	31.88	11.77	3.86
Ray	23.00	50.60	23.40	3.00
Simple	42.86	30.51	26.39	0.24
VLIW	51.15	27.94	15.86	5.05
Average	41.50	36.87	18.71	2.91

addresses in older generations), and most writes should be hits. If the allocation space is larger than the cache, then there are more active blocks than can fit in the cache, and every time the allocation cycle returns to a block, it would have already been evicted from the cache (at least for the more common replacement policies used in caches). Therefore, the expected write miss ratio should be close to $1/block\_size\_in\_words$.

4.3 User Reads

User reads are divided into four types: 1) reads to closures, 2) reads to user data objects, 3) reads to compile-time-allocated objects (which are non-integer constant values that appear in the source code and are usually embedded in the the code string [6]. Integer values are embedded in the machine code itself and do not generate data references.), and 4) reads to run-time system objects. Closures and user data objects may be in the allocation space or older generations. Compile-time-allocated objects are in older generations, and run-time system references are outside the ML heap. Table 13 shows the distribution of reads to each of these types. Closure and user data account for most references, but there are also many references to compile-time objects. Although closures account for a greater fraction of allocation, there are several cases in which there are many more references to user data than to closures. The number of compile-time and run-time system objects should be small, so whenever there are many references to these objects there should be repeated references to a few objects.

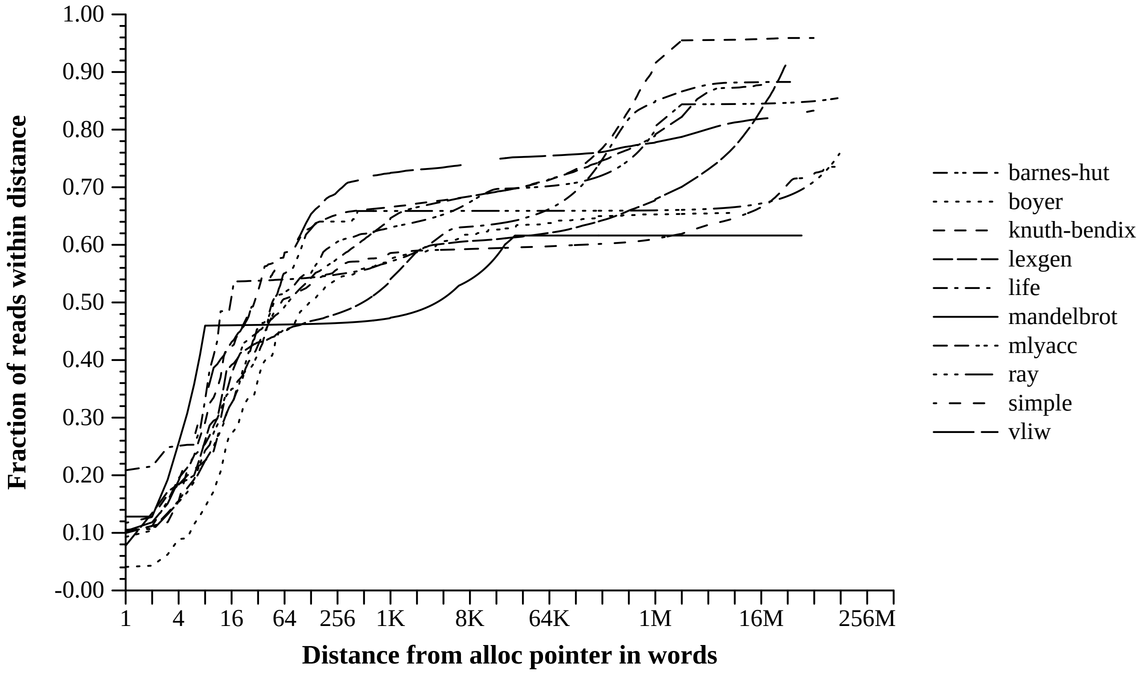


Figure 7: Cumulative distribution of reads as a function of the distance of the address referenced from the allocation pointer.

Table 14: Breakdown of references within close range from the allocation pointer. The first three columns gives the fraction of the references in the range that are to closures. The last three columns give the fraction of all closure references that are within the range. For example, in Boyer, 85.03% of the references within 1K bytes are to closures, and 84.55% of all closure references are within 1K byte from the allocation pointer.

Program	Closure references in range as a % of all references			Fraction of closure references as a % of all closure refs.		
	1K bytes	4K bytes	16K bytes	1K bytes	4K bytes	16K bytes
Barnes-Hut	43.24%	40.84%	41.00%	71.23%	78.64%	82.18%
Boyer	85.03	79.74	77.04	84.55	95.22	97.75
Knuth-Bendix	87.73	88.59	88.26	85.54	96.85	98.41
Lexgen	82.12	78.95	71.28	92.72	94.86	95.64
Life	80.47	79.92	78.15	95.88	96.77	98.33
Mandelbrot	66.36	66.05	64.86	99.79	99.87	99.90
MLYACC	77.20	77.88	77.70	69.45	82.11	91.59
Ray	34.87	34.15	35.98	80.94	92.25	99.97
Simple	69.81	70.21	70.47	82.24	91.04	96.02
VLIW	73.21	72.05	72.80	78.93	92.50	97.89
Average	70.00%	68.84%	67.75%	84.13%	92.01%	95.77%

Since most objects have a short lifetime, and most objects should be read at least once (some programs, like Knuth-Bendix, never read some objects, but these are not common cases), we may expect to have many references to objects that have been recently allocated. The data in Figure 7 shows that all programs do make many references to recently allocated objects.

Figure 7 gives the cumulative distribution of reads by age of the word read, where age is measured in number of words allocated. The data in Figure 7 was obtained by running the programs with an "infinite" allocation space, which means that there is no garbage collection. Thus, the age of a word is given by its distance to the allocation pointer. The allocation pointer always points one word ahead of the word with higher address written in the allocation space (this may be different from the value pointed to by the allocation pointer register, since the register may only be updated after an entire word is allocated). All objects allocated by the program are kept in the allocation space, but there are compile-time allocated objects in the older generations. References to run-time system objects are not included in the graphs, only references to closures, user data, and compile-time objects. Compile-time objects account for the difference from one to the highest value of each curve. Closure and user data references are not distinguished in the graphs.

The shape of the curves in Figure 7 can give some hints on the way each program references allocated objects. For example, a horizontal line in a curve shows that the program does not make any references to objects in a certain age range. The program Mandelbrot, for example, makes almost no references to objects which are more than 128K bytes from the allocation pointer. Most of the references of this program are either to recently allocated objects, or to objects in older generations. Most programs tend to do more references to either very young or very old objects.

The graphs show that many references are to objects close to the allocation pointer. For most programs, more than 50% of references are within 1K bytes from the allocation pointer. Closures tend have a shorter life than user objects, so we may expect that most closure references are within a short distance from the allocation pointer. Table 14 shows that this is indeed the case. For all programs, at least 70% of all closure references are within 1K bytes from the allocation pointer, and for all but one program, at least 91% of closure references are within 16K bytes from the allocation pointer. Most references to user objects, on the other hand, are to objects far from the allocation pointer.

In practice, the programs must be run with a finite allocation space, so references to

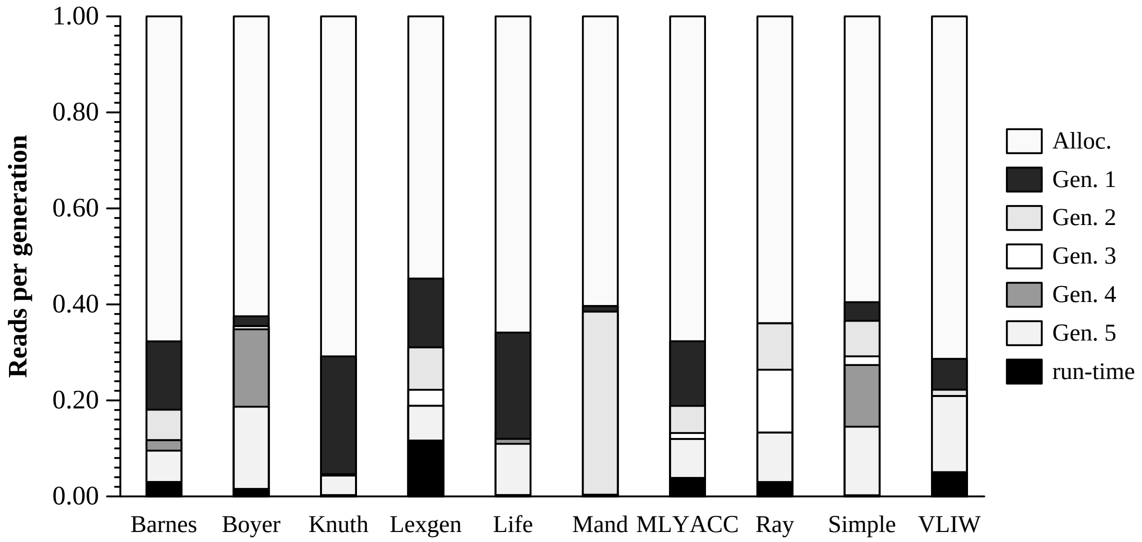


Figure 8: Distribution of data reads per generations. The size of the allocation space is 512K bytes.

closures and user data may be split between the allocation space and older generations. But since there are so many references to recently allocated objects, there should still be a significant fraction of references to objects in the allocation space. Figure 8 shows the distribution of reads to a 512K-byte allocation space, the five generations, and run-time system areas (outside the ML heap). On average, more than sixty percent of reads are to the allocation space.

Read references also show a cyclic pattern, following the cyclic allocation pattern. Figure 9 shows a plot of address read (modulo 512K) versus time (measured in words written)

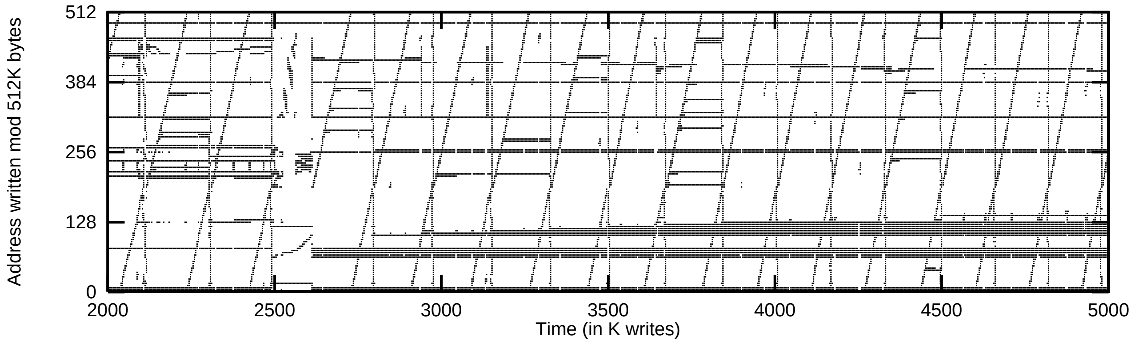


Figure 9: Read reference patterns for Lexgen.

for Lexgen, similar to that of writes (Figure 6). Inclined lines, which match the lines for writes, can be clearly distinguished. These lines are the reads to recently allocated objects, most of which are closures. For this program, most other reads tend to be concentrated within a relatively small number of locations, which explains the horizontal stripes in the graph. One of the stripes, near the bottom, becomes thicker as time goes by. These are probably addresses in the first generation, and the line gets thicker as more objects are added to the area on minor collections.

Assuming a write-allocate cache, we may expect that most references to recently allocated data should be cache hits, and most misses should be caused by references to older objects. But reads to older objects may account for up to forty percent of all reads, and all those references may cause misses. Some programs make many references to compile time objects, and since the number of those objects is small (they are constants that appear in the program source), they should have good temporal locality.

4.4 Garbage Collection References

During garbage collections we may distinguish two main groups of memory references: references to collector internal data (including the BIBOP), and references to the objects that are being collected. References to collector data structures account for the most references made by the collector. Except for the BIBOP table, all collector data structures are small.

The reference pattern to objects in the heap is simple: the collector reads an object and copies it to a new location. It then stores a forwarding pointer in the old copy. At a later time, the new copy is searched for pointers to new objects. Copying an object is similar to allocation (objects are added sequentially to an arena), but there are five different allocation arenas.

Garbage collection references may or may not be relevant to the overall performance of programs, since garbage collection overhead varies from program to program. Some programs, like Mandelbrot and Ray, have little overhead, and garbage collections references should have little effect on performance. But other programs, like MLYACC, Boyer, and Simple, can have high collection overhead, even with a large allocation space.

4.4.1 Minor collection references

We have seen that the ratio of live objects to allocated objects decreases as the size of the allocation space increases (Figure 4). The total number of objects copied per collection, however, may increase, since we have fewer collections. Although only a small fraction of allocated objects have to be copied, in general, the live objects tend to be spread across all the allocation space. We can expect a higher concentration of objects towards the limit pointer, since this area contains many young objects that did not have enough time to die before the collection. But for the rest of the heap, the distribution of live objects should to be more or less uniform.

The distribution of live objects across the allocation space can vary not only from one program to another, but also from one collection to another in the same program. Figures 10 and 11 shows the distribution of live objects over the allocation space for four programs of the benchmark suite. For each program, the distribution at ten randomly chosen collection points is shown. Most collections do encounter live objects all over the heap, but the stair-like shape of the curvers for Lexgen and MLYACC indicate the there is a tendency for objects being clustered on a number of areas. Simple, has a uniform distribution on most collections sampled.

4.4.2 Major collection references

Figure 12 shows the number of words copied on major collections for each program, when different sizes of the allocation space are used. The programs Life, Mandelbrot, and Ray have little major collection overhead. Because of the policy used to determine the size of older generations, the size of the first generation increases when the allocation space increases and, therefore, we would expect to see a decrease in words copied on major collections, just like on minor collections. However, sometimes there is an increase in the number of words copied. The reason is that, unlike minor collections, major collections are sensitive to the placement of collection points. This difference in behavior is probably caused by major collections being less frequent, and by a higher survival rate.

The survival rate for objects in the older generations is the ratio of words that are live before collections to the size in words of the entire area that is being collected. Table 15 shows the survival rate for young objects in the first generation, and Table 16 shows the survival rates for all major collections. The survival rates for the young objects of the first

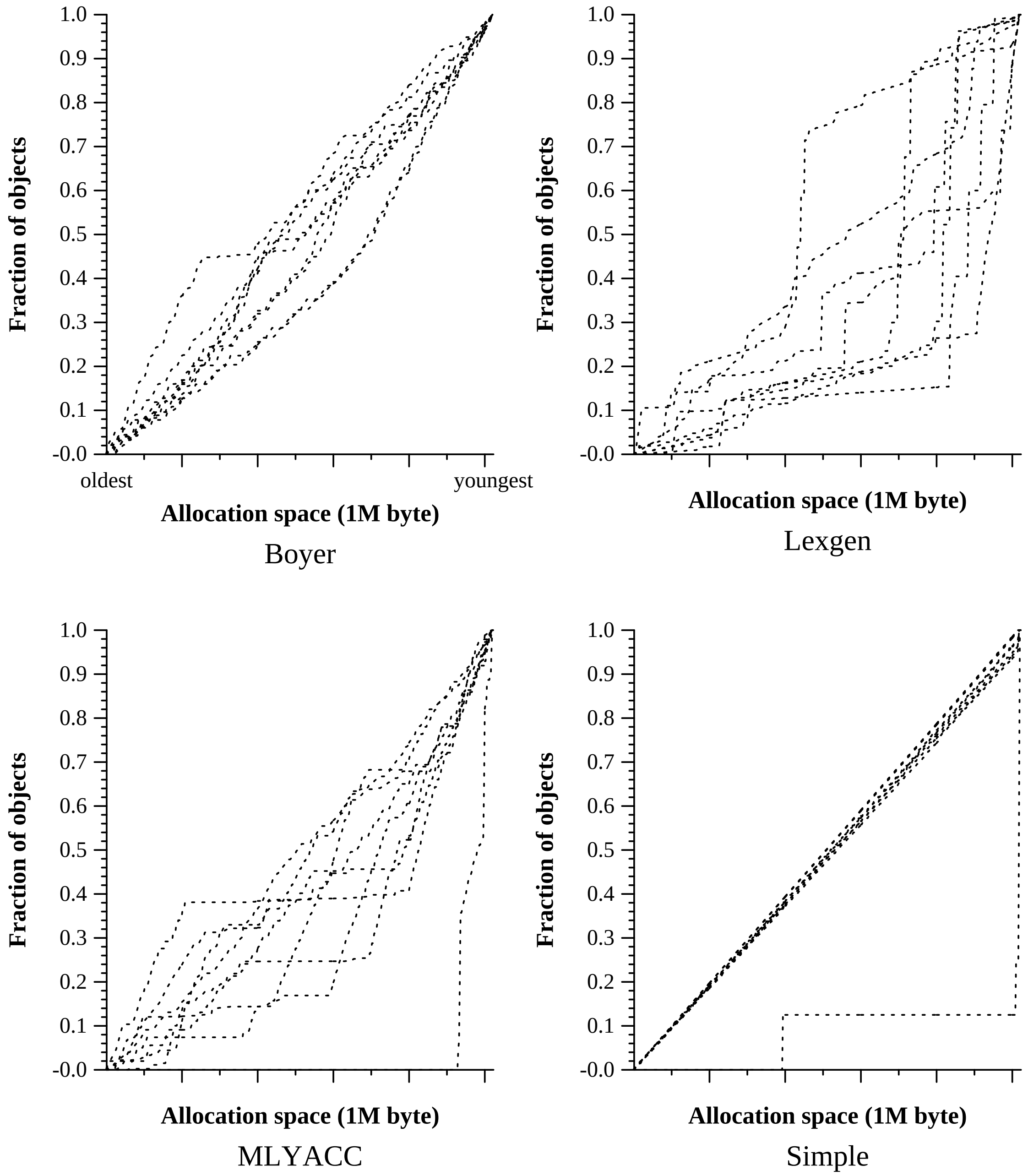


Figure 10: Cumulative distribution of live objects in the allocation space (size is 1 Megabytes) at minor collections. For each program, a random sample of ten collections was taken.

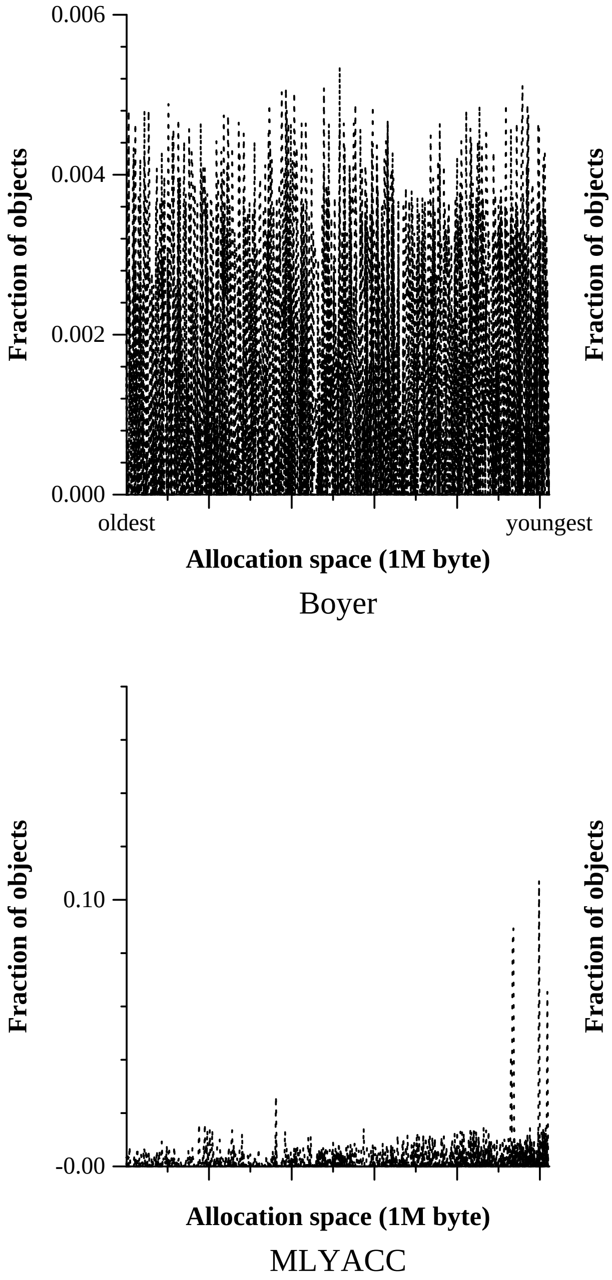


Figure 11: Same as Figure 10, but with a non-cumulative distribution.

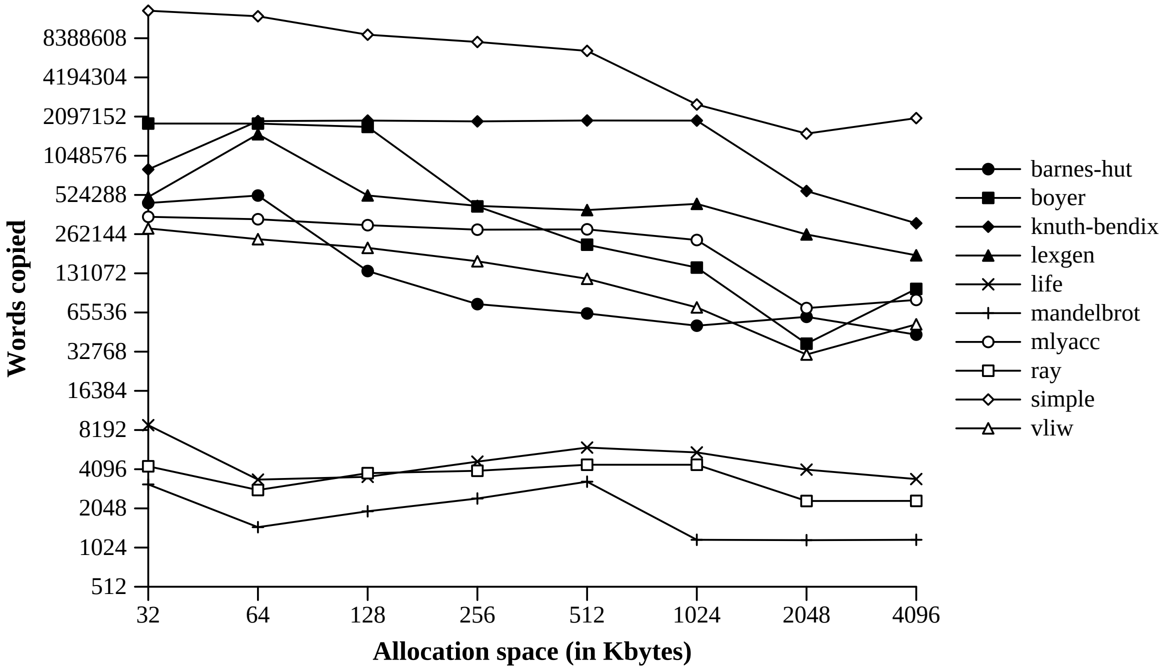


Figure 12: Words copied on major collections.

Table 15: Survival rates for the young objects in first generation.

Program	Survival rate to first major GC (%)							
	32K	64K	128K	256K	512K	1M	2M	4M
Barnes-Hut	16.99	27.42	10.04	7.05	6.43	6.17	7.74	7.28
Boyer	50.00	57.15	58.91	60.74	59.06	68.54	80.92	91.71
Knuth-Bendix	24.95	34.59	44.80	57.35	69.90	76.12	82.45	84.91
Lexgen	20.07	26.19	31.61	36.12	40.74	50.83	63.39	75.44
Life	2.85	1.40	2.03	3.48	8.27	5.65	7.83	10.81
Mandelbrot	2.81	2.48	3.67	17.39	28.44	8.44	8.44	8.42
MLYACC	17.72	24.07	28.23	25.11	26.27	33.97	39.09	33.86
Ray	0.50	0.56	1.52	3.05	9.39	4.24	14.58	14.60
Simple	54.55	71.79	83.02	82.02	88.57	76.93	71.48	76.56
VLIW	9.46	13.24	17.17	21.59	22.25	18.53	74.72	74.07

Table 16: Survival rates of all major collections.

Program	Survival rate to all major GCs (%)							
	32K	64K	128K	256K	512K	1M	2M	4M
Barnes-Hut	14.70	21.79	9.51	7.12	6.56	6.38	8.95	7.74
Boyer	81.43	78.63	86.43	67.45	56.78	66.50	70.83	85.41
Knuth-Bendix	48.64	79.06	84.81	90.06	93.69	95.67	91.37	89.82
Lexgen	43.10	77.87	58.08	62.56	62.48	72.09	68.23	79.95
Life	3.07	1.66	2.45	3.86	8.52	6.31	8.51	11.84
Mandelbrot	3.30	3.65	4.91	18.62	26.66	13.20	13.24	13.28
MLYACC	27.94	30.95	33.87	36.04	41.05	43.98	39.97	36.28
Ray	0.67	0.90	2.12	4.14	11.43	7.15	23.02	22.99
Simple	88.42	89.84	89.16	89.01	88.98	78.98	72.69	73.37
VLIW	16.82	20.77	25.01	30.01	28.20	21.60	74.76	49.18

generation are usually high, which means that the minor collections are doing a good job in eliminating most short-lived objects and the objects that remain tend to have a long life. For these programs, doing frequent major collections is not likely to be of much benefit, since there is not much free space to be reclaimed from the collection (for example, Simple have survival rates close to 90%).

Major collections may touch a potentially large heap area, since the older generations can be tens of megabytes in size. But most collections would collect only the first generation, which typically have a few megabytes in size. Most programs in the benchmark suite rarely collect many generations at once, unless a small allocation space is used.

4.5 Conclusion

SML/NJ programs do mostly allocation writes and have a regular cyclic write pattern, and therefore, we can expect the write miss ratio to be close to $1/block\_size\_in\_words$. Most reads are to recently allocated objects (and most of those reads are to closures) and they should cause few cache read misses. Most read misses should be caused by references to older objects.

The results presented here can also be useful to the design of garbage collectors. The high survival rates (near 90%) shown for some programs on major collections, shows that there is not much free space to be reclaimed, and thus there are few benefits from doing the collections. These programs probably need larger generations, so that they can run with less frequent major collections.

The shorter lifetime of closures suggests that closures and user objects might be treated differently. It might be possible, for example, to have two separate allocation arenas, one for closures, and other for user objects, with the two arenas being collected independently. The closure arena would be collected more frequently, but since most closures die quickly, these collections would have little overhead. The user arena could be collected less frequently, leaving more time for objects to die before the collection. However, this scheme would only be practical if the two arenas can be collected independently (or at least if the closure arena can be collected independently), which should be difficult unless objects in one arena are restricted to not point to objects in the other.

Chapter 5

Cache Performance

This chapter presents a study of the cache performance of SML/NJ programs on several cache architectures. Cache sizes range from 1K bytes to 4M bytes, block size from 16 bytes to 256 bytes, and set-associativity from one to sixteen-way. Three write allocation policies are considered, fetch-on-write, write-validate, and write-around. Unified and separate data and instruction caches are also considered, but except where noted, there are separate instruction and data caches, and the cache size refers to the size of a single component, that is, either the instruction or data cache.

Two metrics are used to analyze cache performance: miss ratios and cache overhead (cycles spent on cache divided by the instruction count). Most results are reported as the unweighted arithmetic mean of miss ratios or overheads, which assumes a workload where each benchmark is given an equal weight. The results in this chapter do not consider the effects of varying garbage collection parameters. Except where noted, results include both user and garbage collection code, and an allocation space of 512K bytes is used.

The chapter begins with the miss ratios on fetch-on-write caches. Section 5.2 compares the miss ratios of write-validate and write-around caches to the miss ratios of fetch-on-write caches. Section 5.3 presents the miss ratios for set-associative caches. Section 5.4 presents the miss ratios of instruction caches, and Section 5.5 the miss ratios for unified caches. Section 5.6 shows the cache overhead for data, instruction, and unified caches. Finally, Section 5.7 compares the performance of the SML/NJ programs to that of the SPEC92 benchmarks, and to a benchmark of Scheme programs, and presents the conclusions for this chapter.

5.1 Fetch-on-write Caches

5.1.1 Direct-mapped write misses

Section 4.2 showed that all benchmark programs do mostly allocation writes, and therefore, when the allocation space is larger than the cache, we should expect the write miss ratios to be mostly independent of cache size and close to $1/block\_size\_in\_words$. Figure 13a shows the average miss ratios for the benchmark programs with block size varying from 16 to 256 bytes, and compares them to the predicted values. The approximation is reasonably close in most cases, but sometimes the actual miss ratios are either lower or higher than the predicted value. When the cache is smaller than the allocation space, the miss ratios show a small decrease as cache size increases, but they become significantly lower when the cache size is larger than the allocation space (512K bytes).

One factor that can explain the small difference between the actual miss ratios and the predicted value is that garbage collection writes were not considered when making the prediction, but are included on the calculation of the miss ratios. Another factor that may decrease the miss ratios, is that some blocks may be "prefetched", either by the garbage collector, or by a read to the block after they are evicted from the cache by the allocation cycle. On the other hand, because of conflicts, some blocks may have two allocation misses on the same allocation cycle. This is particularly likely for large blocks on small caches, since less blocks in the cache increase the chances of conflicts. This is probably the reason miss ratios for small 128 and 256-byte block caches are higher than the predicted value.

Garbage collection may increase or decrease overall miss ratios, depending on whether garbage collection miss rates are higher or lower than user miss ratios and the weight garbage collection references have on total references for each program. Figure 13b shows the write miss ratios for 32-byte block caches when the allocation space size varies from 16K bytes (in which case there is a high garbage collection overhead) to "infinite" (in which case there are no garbage collections). For 32-byte block caches, garbage collection tends to have lower write miss ratios than user code, so the overall miss ratios tend to become lower. As the graphs show, when garbage collections becomes less frequent, the miss ratios become higher, and closer to the predicted value, which is 0.125 for a 32-byte block.

Another factor that makes write miss ratios diverge from the expected value are non-allocation writes. In general, the same locations are likely to be updated repeatedly (as the horizontal lines on the graph of Figure 6 in Chapter 4 show), and therefore those references

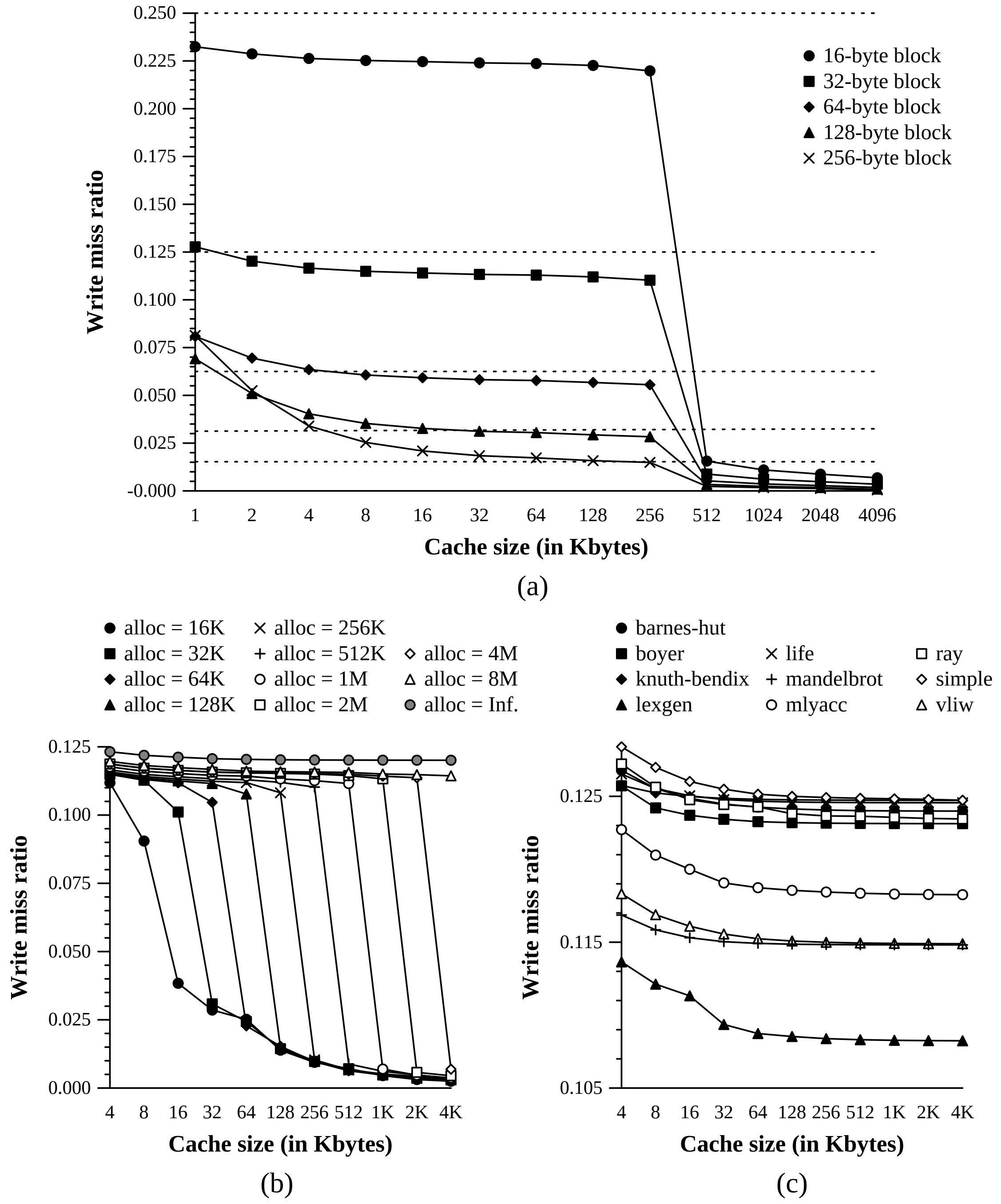


Figure 13: Write miss ratios on direct-mapped fetch-on-write data caches. (a) Write miss ratios for different block sizes, for a 512K-bytes allocation space. (b) Write miss ratios for different sizes of the allocation space, 32-byte block. (c) Write miss ratios for each program, with an infinite allocation space, 32-byte block.

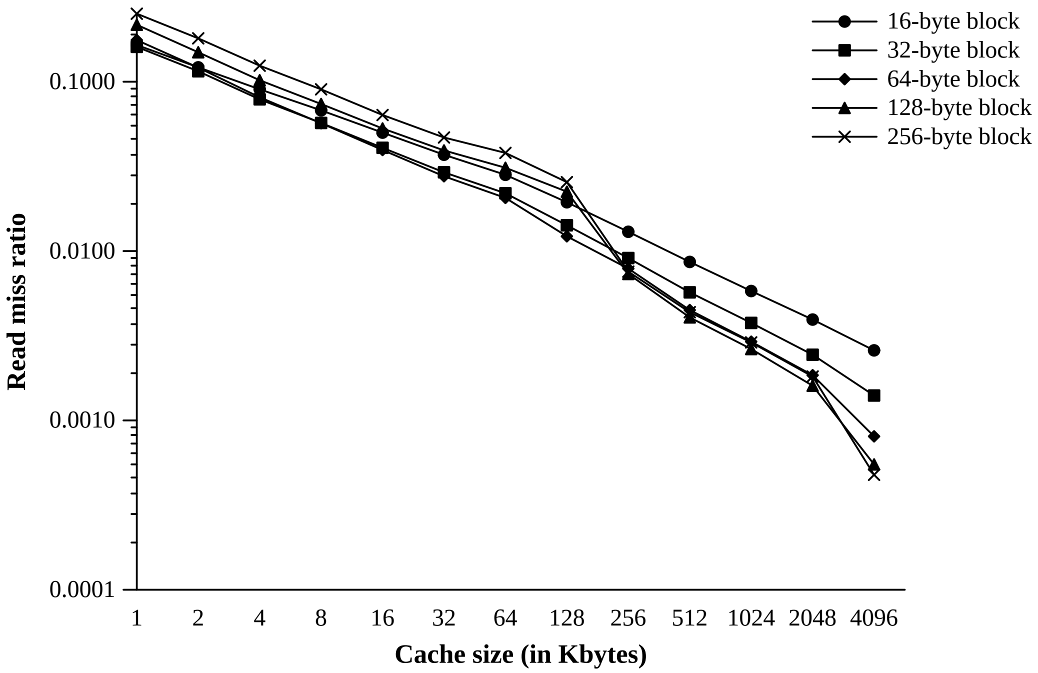


Figure 14: Read miss ratios on direct-mapped fetch-on-write data caches, for different block sizes, from 16 to 256 bytes. The allocation space is 512K bytes.

should have good temporal locality. Figure 13c gives write miss ratios for each program, with no garbage collection, on 32-byte block caches, and shows that the programs with more non-allocation writes (Lexgen, Mandelbrot, VLIW, and MLYACC) have lower write miss ratios than the other programs, and the more non-allocation writes the program makes, the lower the miss ratio is. But if block sizes were different the results could also be different, since there is a trade-off between the temporal locality of updates versus the spatial locality of sequential allocation. It might be possible that for large blocks, the inverse would occur, that is, programs with more non-allocation writes might have higher miss ratios.

5.1.2 Direct-mapped read misses

Figure 14 shows the average data read miss ratios for the benchmark suite on direct mapped fetch-on-write cache, with block size varying from 16 to 256 bytes. Unlike the case of writes, which depend mostly on block size (except if the allocation space does not fit in the cache), the read miss ratios decrease as cache size increases. The block size that has the lowest miss rates depend on cache size, but blocks of 32 and 64 bytes usually have the lowest or

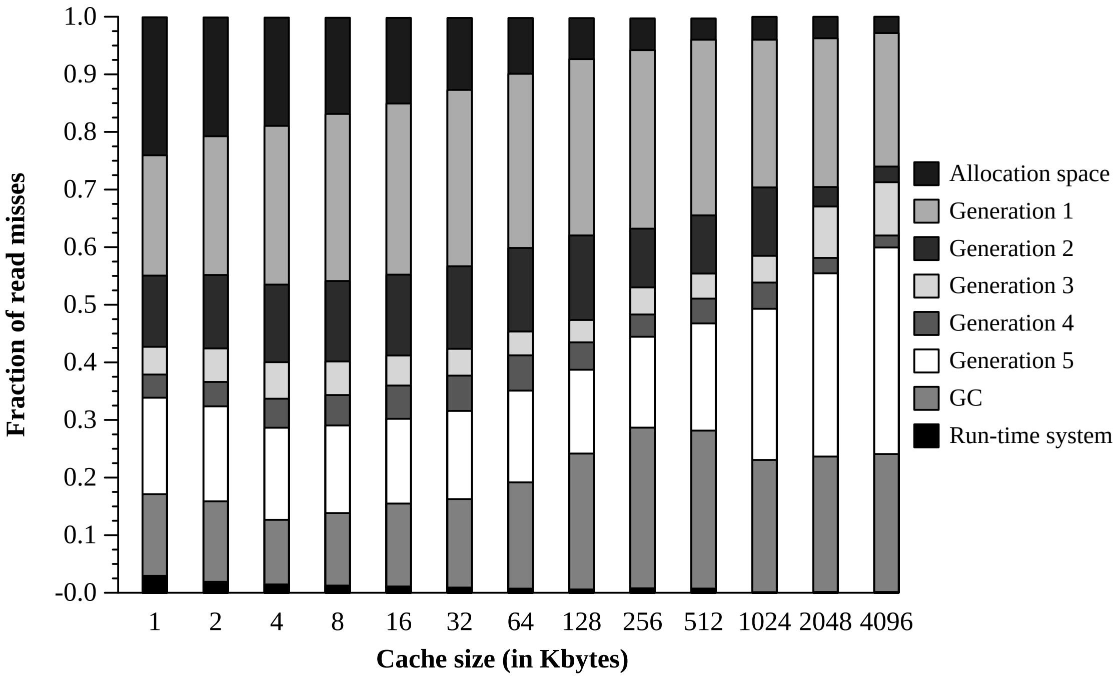


Figure 15: Distribution of read misses per heap area for direct-mapped fetch-on-write data caches. The allocation space is 512K bytes.

at least close to lowest miss ratios.

Large blocks can usually take advantage of spatial locality, but, on the other hand, for a fixed cache size, increasing block size decreases the number of blocks on the cache, which may cause an increase in conflict misses[67]. In general, a small cache with a small block may have lower miss ratios than a cache of the same size and large blocks, but as cache size increases, large blocks may give lower miss ratios. For the SML/NJ programs, even for small caches, 16-byte blocks have higher miss rates than 32 or 64-byte blocks, but for 256K and larger caches, the lowest miss ratios are those of 128-byte blocks, except for the 4M-byte caches, for which the lowest miss ratio is given by a 256-byte block.

There are two forms of spatial locality that may be found on SML/NJ programs. First, although most objects are small, they are at least two words long, and any references to a word within one object may be soon followed by a reference to another word within the same object. And second, because objects are small, there may be many objects on a single cache block, and two or more of these objects might be referenced within a short interval from each other. Pairs that are on the older generations, for example, are two words long

(8 bytes) and a single 32-byte block may hold up to 4 pairs at once.

Packing more objects into a single block is not the only advantage of increasing block size. Although most objects are small enough to fit into a single block, many objects may not be aligned to block boundaries, and therefore two adjacent words on the same object may reside on two different blocks. Therefore, two references to the same object within a short time from each other may cause two cache misses, instead of the single miss that would occur if the entire object were contained in a single block. Increasing block size decreases the number of boundaries, and the probability that objects cross block boundaries, reducing that chance that an object causes a double miss.

The degree to which spatial locality can be exploited depends on the time between references to objects or words within the same block. The longer the time, the higher the chance that the block is evicted from the cache (because of a conflict) before the second reference. For the SML/NJ programs, the fast allocation cycle is continuously evicting blocks from the cache, which may limit the use of spatial locality. For 4K-byte caches, for example, at an allocation rate of one word per six instructions, any block will be evicted within a time of at most 6144 instructions (it takes $6 * 1024 = 6144$ instructions to revisit the same cache block).

Based on the discussion on Section 4.3, we would expect most cache misses to be to older objects. Figure 15 shows the distribution of read misses for the allocation space, each of the older generations, garbage collection and run-time system. As expected, there are few misses on reads to the allocation space. Most misses are caused by references to objects on the older generations, but there is also a significant number of misses on garbage collections. Among the older generations, most misses are divided between the first and fifth generations. However, the distribution of misses for the older generations may differ considerably from one benchmark to the other, just like the distribution of read references does (Figure 8).

5.1.3 Individual miss ratios

Figures 16 and 17 show the write and read miss rates for each program in the benchmark suite. All programs have write miss ratios close to average when the allocation space is larger than cache size, and have a sharp decrease in write miss ratios at the point where the allocation space is the same size as the cache. Mandelbrot has very low read miss ratios, even on small caches. Most programs have close to average miss ratios on small caches,

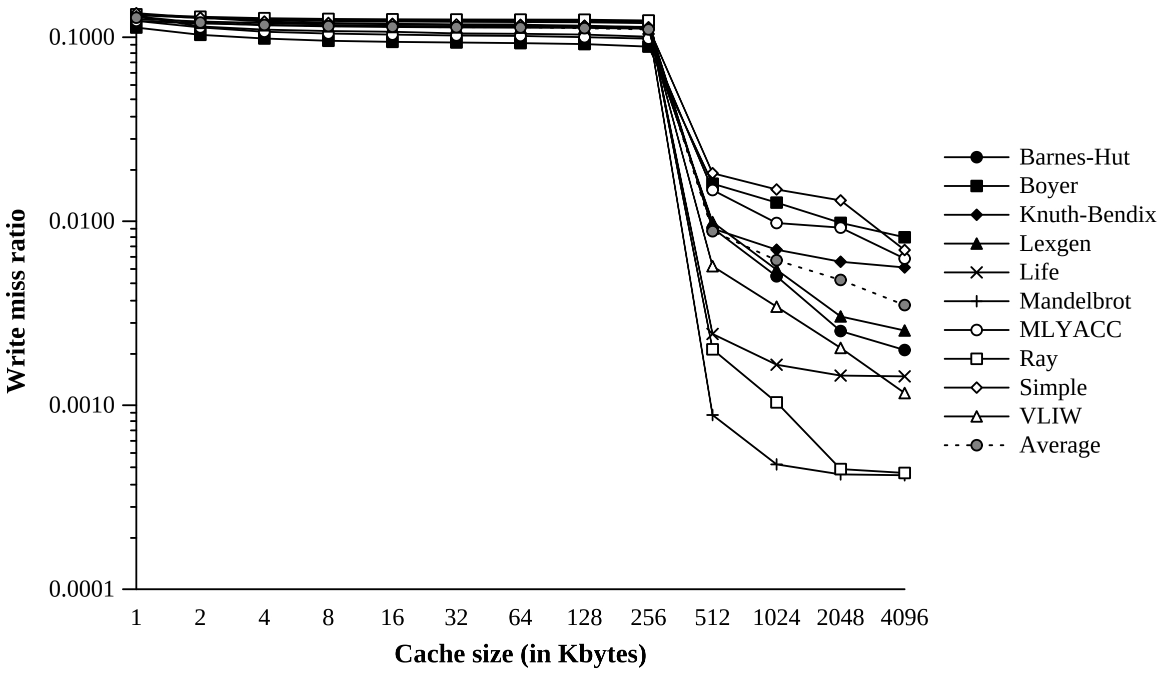


Figure 16: Write miss ratios on direct-mapped, fetch-on-write, 32-byte block data caches, for each program in the benchmark suite. The allocation space is 512K bytes.

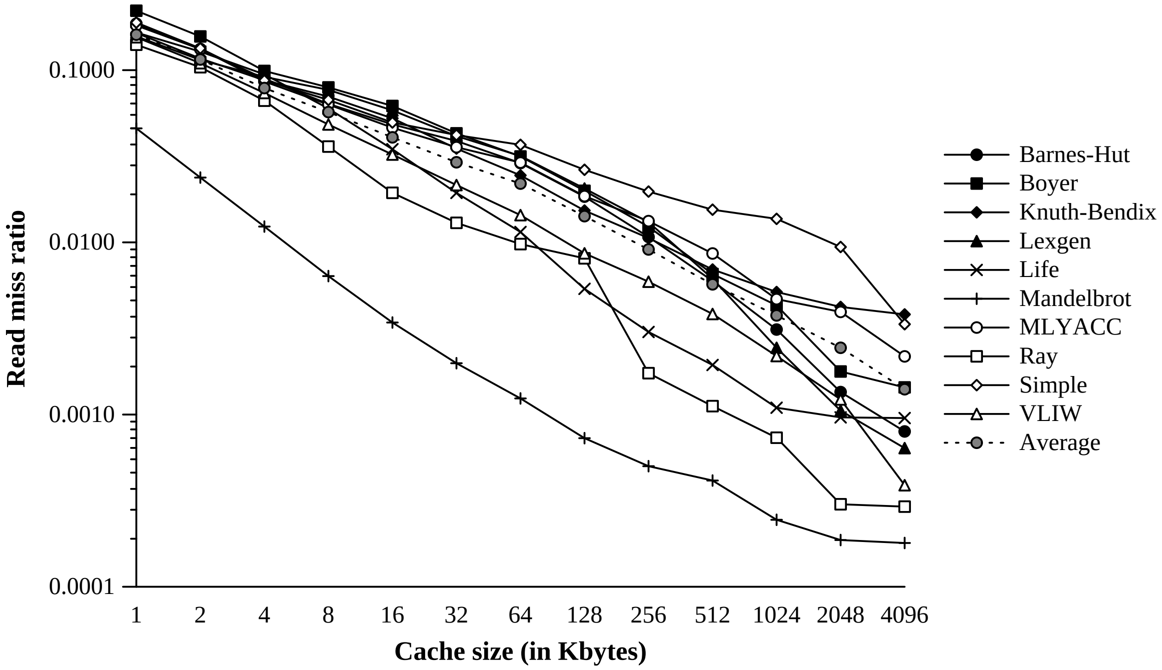


Figure 17: Read miss ratios on direct mapped, fetch-on-write, 32-byte block data caches, for each program in the benchmark suite. The allocation space is 512K bytes.

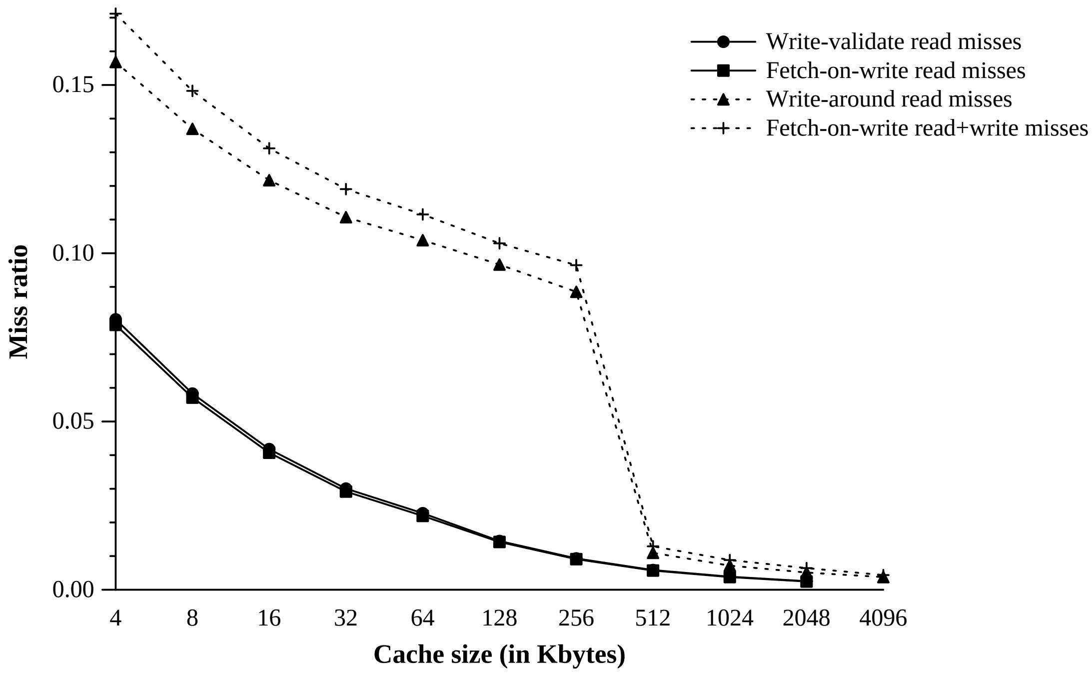


Figure 18: Miss ratios for write-validate and write-around caches compared to fetch-on-write data caches. All caches are direct-mapped with 32-byte block. The allocation space is 512K bytes.

but there is a significant difference in behavior from program to program on large caches. This is caused by the fact that each program has a different size for its working set. Many programs have a working set that is small enough to fit even on a mid-sized cache: Ray, for example, has a sharp decrease in miss ratios between cache sizes of 128K and 256K bytes, which suggests that its working set should be between 128K and 256K bytes. But programs like Simple and Knuth-Bendix, on the other hand, have large working sets, and have high miss ratios even on the large caches.

5.2 Write-validate and Write-around Caches

Write-validate and write-around caches do not have write miss penalties, and they usually outperform a fetch-on-write cache [45]. Although they do not have write misses, write-validate and write-around caches may have more read misses than a fetch-on-write cache, but in any case, with all other cache parameters being constant, they never have more read misses than the sum of read and write misses of a fetch-on-write cache.

Figure 18 compares the misses of write-validate and write-around caches to those of fetch-on-write caches. This comparison is made by means of ratios of misses per read reference. For the write-validate and write-around caches there are only read misses, but the fetch-on-write may have both read and write misses. Therefore, two sets of ratios are shown for the fetch-on-write caches, the first dividing the number of read misses by reads, and the second dividing the sum of read and write misses by reads.

The read miss ratio for the write-validate cache is almost identical to the read miss ratio for the fetch-on-write cache. This means that the write-validate cache eliminates the write misses of the fetch-on-write cache, without adding any read misses. On a write-validate cache, a write to a block that is not in the cache will cause a block to be allocated on the cache, with all words marked as invalid, except the one that is being written. Therefore, there would be an extra read miss only if one of the invalidated words where read, before being written. But most writes of the benchmark programs are for allocation, the programs never read the invalidated words in a block, since they contain non-initialized data. The invalidated words can be read only on non-allocation writes, but there are few of those writes.

Write-around caches, on the other hand, have a read miss on most of the blocks for which the fetch-on-write cache would have write misses. This happens because most objects are read shortly after they are allocated. The write-around cache saves a few misses over fetch-on-write, since there are still some objects that may not be read soon after being allocated. Garbage collection and non-allocation writes also may not be followed by a read to the same block within a short time.

5.3 Associative Caches

Figure 19 shows the read miss rates for direct mapped and set-associative 32-byte block caches, with a least recently used (LRU) replacement policy. Miss rates for associative caches are significant lower than miss rates for direct mapped caches. Four sets gives lower miss rates than two sets, but the difference between four, eight, and sixteen sets is negligible. Associativity usually reduces cache misses, by eliminating conflicts. In the case of SML/NJ programs, the allocation cycle is constantly removing blocks from the cache, but associativity may allow some blocks containing older objects to stay longer in the cache.

Figure 20 shows the write miss rates for direct-mapped and associative 32-byte block

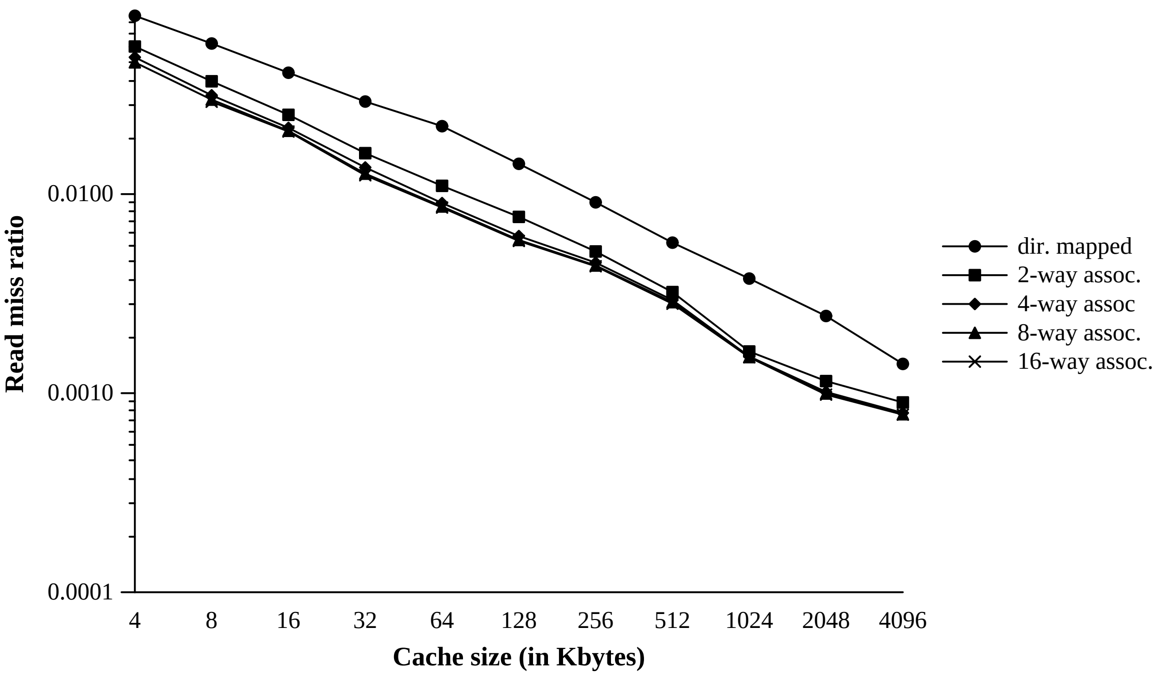


Figure 19: Read miss ratios for direct mapped and set-associative 32-byte block fetch-on-write data caches.

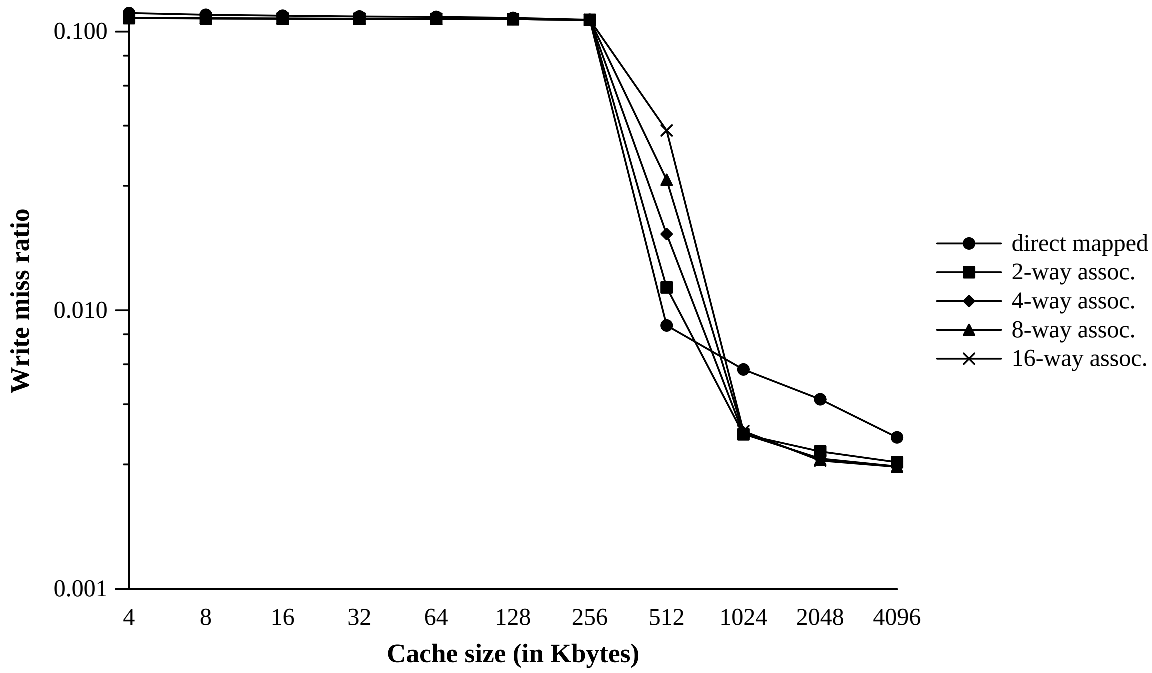


Figure 20: Write miss ratios for direct mapped and set-associative 32-byte block fetch-on-write data caches.

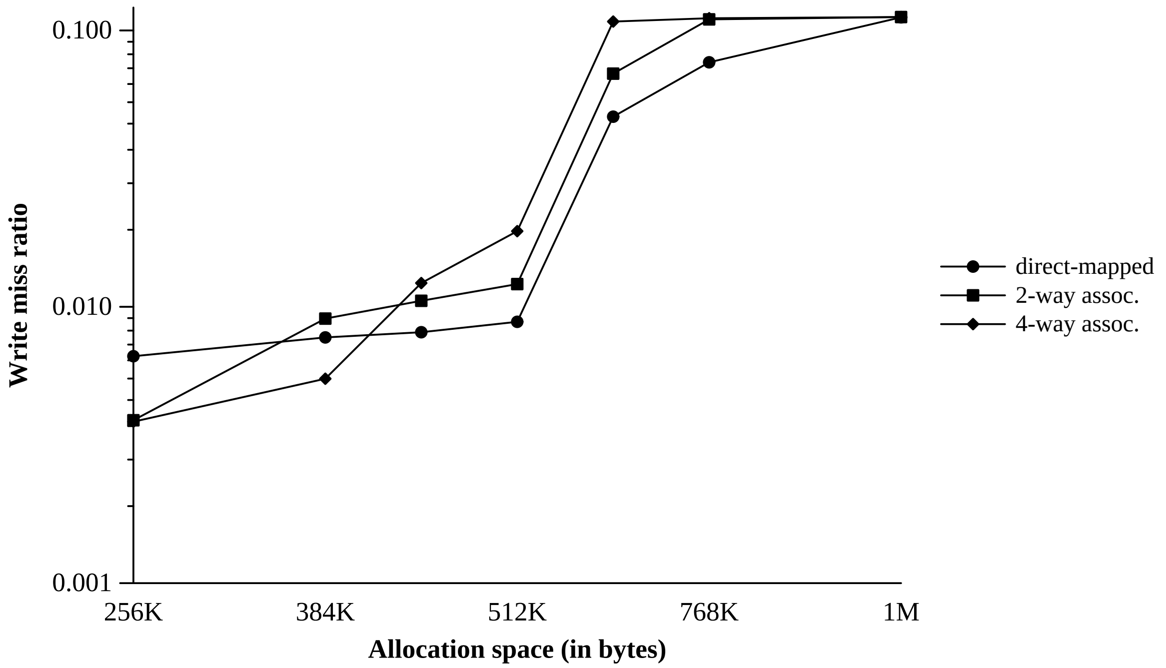


Figure 21: Write miss ratios as a function of allocation space size, for 32-byte block, 512K-byte direct-mapped, 2, and 4-way associative data caches.

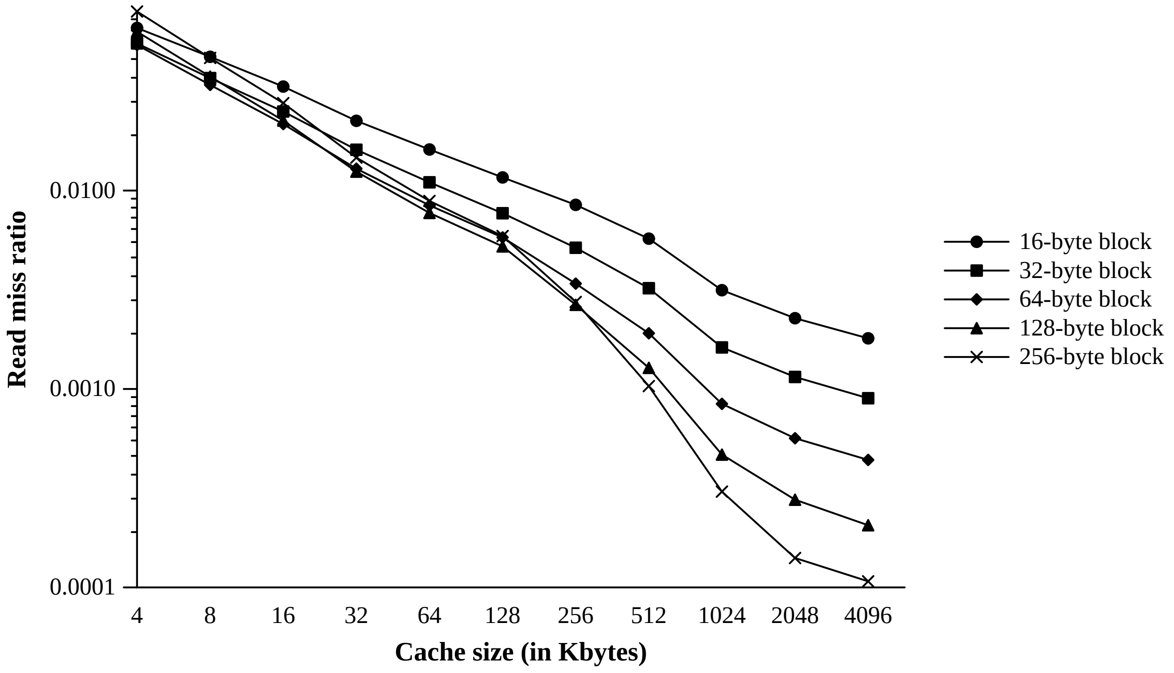


Figure 22: Read miss ratios for 2-way associative fetch-on-write data caches, with block size varying from 16 to 256 bytes.

caches. Associative caches usually have lower miss rates, but there is an unusual behavior for the 512K-byte caches, which are of the same size as the allocation space. For this size, a direct-mapped cache has the lowest miss ratios, and in general, caches with fewer sets have lower miss ratios than caches with more sets. (Wilson et. al[83] found a similar behavior on a set of Lisp programs.)

A comparison of Figures 19 and 20 shows that increasing associativity tends to lower read miss ratios, but the opposite may happen with write miss ratios. This difference happens because most read misses are caused by references to older objects, while most write misses are caused by allocation. The LRU replacement policy of associative caches is good for reads because it tends to keep blocks containing older objects in the cache, that are likely to be read again in the future. But, on the other hand, the block that is evicted is most likely a block from the allocation space that was least recently touched by the allocation cycle, a block that should soon be revisited by the allocation cycle.

The problem is that the LRU replacement of associative caches is not particularly effective for sequential or cyclic reference streams[74, Section 2.2.5], but direct-mapped caches may give a reasonable replacement policy in this case. Consider, for example, a reference stream that makes repeated references to sequential blocks B_0, B_1, B_2, that is, $B_0B_1B_2B_0B_1B_2\ldots$, and a cache with two blocks C_0 and C_1. In the direct mapped case, blocks B_0 and B_2 map to C_0, and B_1 maps to C_1. Thus, all references to B_1 (except for the first) are hits, while all references to B_0 and B_2 are misses. If the cache is associative (fully associative in this case), with an LRU replacement policy, then all references are misses. Initially, B_0 maps to C_0 and B_1 to C_1. B_2 then maps to C_0, since C_0 is the least recently used cache block. Next, B_0 maps to C_1, since C_1 is now the least recently used block. B_1 now misses, and so do all of the following references.

As an example, consider the case where the allocation space is of the same size as the cache. Let the allocation space blocks be A_0, A_1,..., A_{2n}. If there are only references to the allocation space, then there would be only compulsory misses, and miss ratios of direct-mapped and associative caches would be the same. But if there is one reference to an object outside the allocation space, then this reference would conflict causing some allocation space block to be removed from the cache. On an direct-mapped cache, there is only one block that can be removed, but on associative caches, there would be as many blocks as the degree of associativity. Suppose that on a direct-mapped cache, the block removed is A_i, where i is greater than $n/2$, which would cause one cache miss when A_i is allocated again.

On a 2-way associative cache, blocks A_j and A_i can be removed, where j is less than $n/2$. Because allocation is sequential, A_j is probably the block that will be replaced by the LRU policy. A_j will then cause a miss when it is allocated and A_i is likely the LRU block and it is removed from the cache, which will cause another miss when A_j is allocated again.

Therefore, given that most of the write references of the benchmark programs follow a sequential pattern, it is not unreasonable that direct-mapped caches have better performance than associative caches with LRU replacement. What may be most surprising, is the fact that the problem appears to happen only for the 512K-byte caches. (In fact, for the 256K caches, the lowest average miss ratios are for the two-way associative cache, although the difference between the miss ratios of the other associative caches is negligible.)

To explain why only the 512K-byte cache has lower write miss ratios for lower associativity caches, three cases must be distinguished. First, when the allocation space is smaller than the cache, there are few misses caused by allocation. For other references, the LRU replacement policy is better, so highly associative caches should have lower miss ratios. When the allocation space is the same size as the cache, there would still be few allocation misses if there were only references to the allocation space, but even though most references are to the allocation space, there are still many references (mostly reads) to objects in the older generations or outside the ML heap. When one of these references misses, the least recently used block from the allocation space is likely to be chosen to be evicted, which should be precisely the next block to be touched again by the allocation cycle. And finally, there is the case for which the allocation space is larger than the cache. The largest cache that is smaller than the allocation space is the 256K-byte cache, and therefore, even for the direct-mapped cache, every new block touched by the allocation cycle should be a miss. The difference in miss ratios would then be determined by the non-allocation references, and for those, LRU replacement should be the best option.

In general, the problem happens around the point where the allocation space matches the cache size. Figure 21 plots the miss ratios as a function of allocation space size for a 512K bytes, with allocation space size varying from half the cache size to twice the cache size, for direct-mapped, 2, and 4-way associative caches.

Associativity and block size

Figure 22 shows the miss ratios on 2-way associative caches for 16 to 256-byte blocks. Block size has a huge effect on the miss ratios of large caches: for 4M-byte caches, for

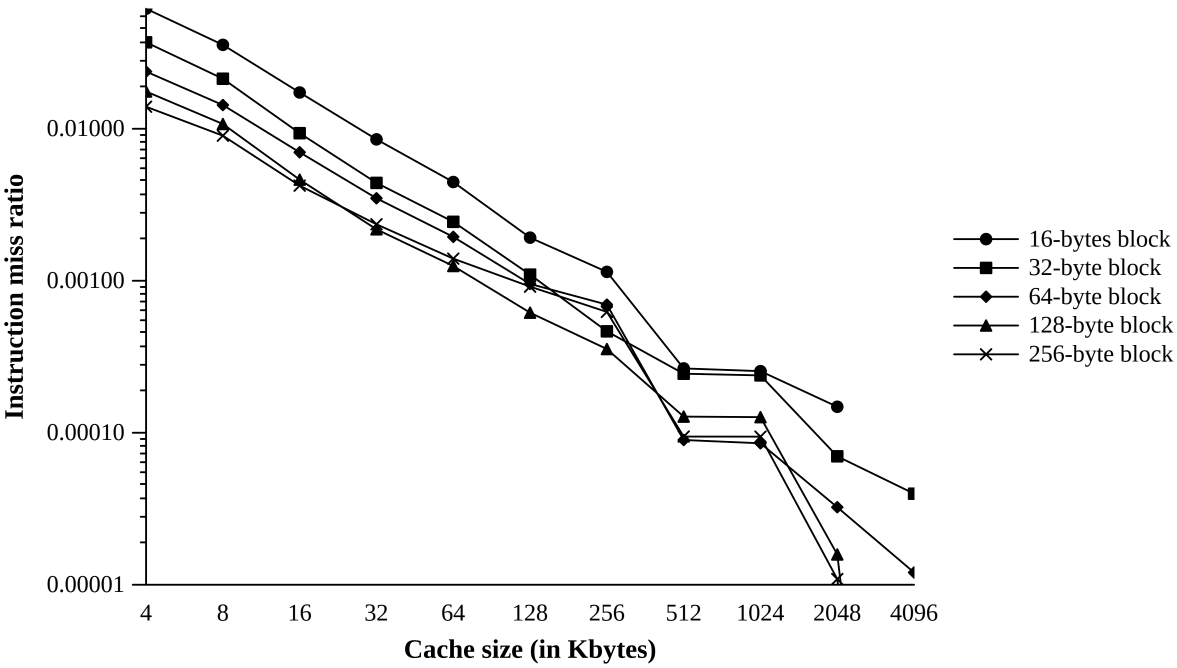


Figure 23: Miss ratios for direct-mapped instruction caches, with block size varying from 16 to 256 bytes.

example, decreasing block size from 256 to 16 bytes, increases the average miss ratios by more than 16 times. Moreover, the miss ratios for large blocks on associative caches are several time lower than the miss rates for direct-mapped caches. These results shows that the benchmark programs have a lot of spatial locality on the older generations (which as shown earlier account for most cache misses), that is, there is a large number of references to objects compacted into small areas. Because of the fast allocation cycle, however, which is continuously removing blocks from the cache, this locality can not be fully exploited on direct-mapped caches. On associative caches, on the other hand, the LRU replacement policy is likely to keep the blocks that are frequently used on cache, and thus a large number of references to older objects may be processed without any misses.

5.4 Instruction Caches

Figure 23 shows instruction miss rates for a separate instruction cache. Block size has a significant impact on instruction miss ratios, with large blocks having lower miss ratios, even for small caches. Miss ratios decrease sharply as cache size increases and become negligible

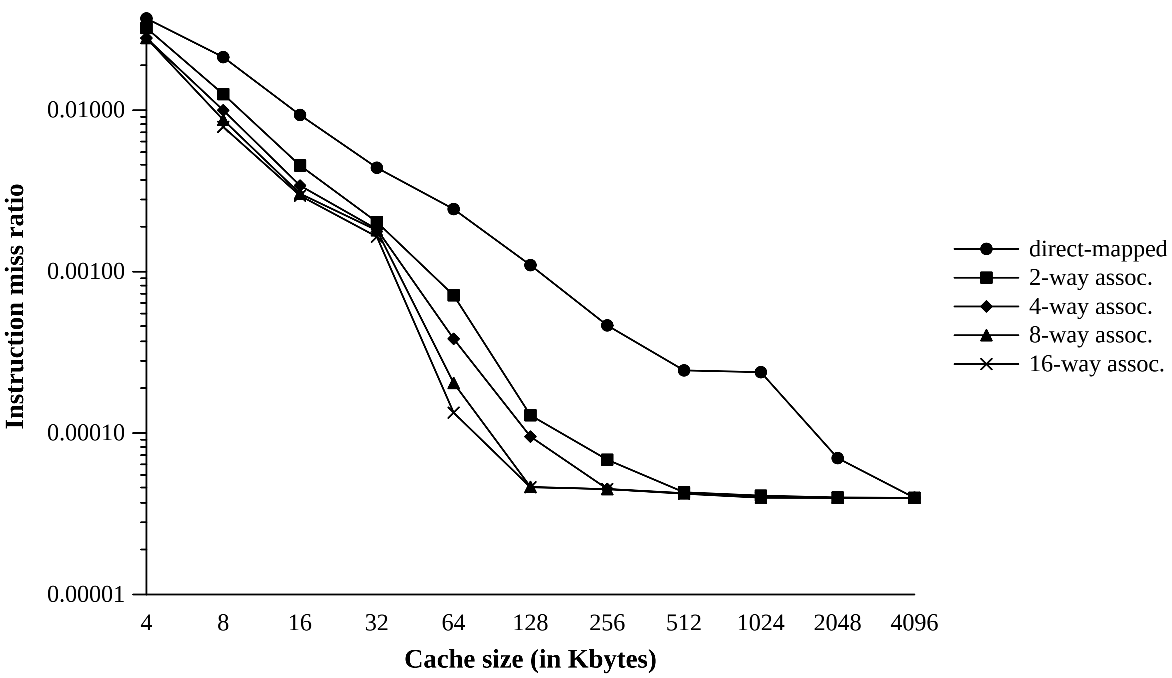


Figure 24: Miss ratios for direct mapped and set-associative, 32-byte block, instruction caches.

for large caches. Figure 24 shows instruction miss rates on direct-mapped and associative caches. Associative caches have significantly lower miss ratios, by a factor of two to ten times. The curves become flat for caches larger than a certain size (128K bytes for 8 and 16-way associative caches), which is probably caused by the elimination of most conflict and capacity misses, with only compulsory misses remaining. In all cases, miss ratios are negligible for large caches.

5.5 Unified Caches

Figure 25 shows the average miss ratios (for instruction and data references combined) on unified caches. Except for small caches, large blocks (128 and 256 bytes) have lower miss ratios. Miss ratios fall when the allocation space fits in the cache, as they do for a separate data cache.

5.6 Cache Overhead

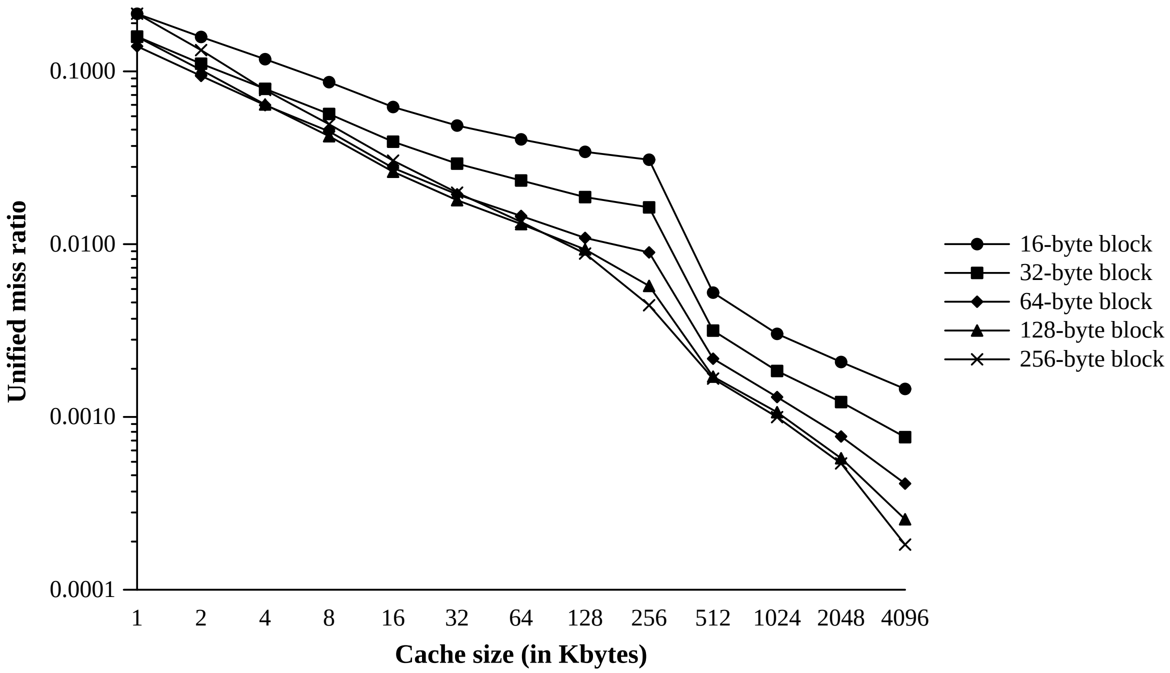


Figure 25: Miss ratios for unified direct-mapped fetch-on-write caches, with block size varying from 16 to 256 bytes.

Miss rates alone do not quantify precisely the effect of cache misses on performance, since this effect also depends on the miss penalty and the number of memory references relative to instruction count. This section presents an analysis of the effects of cache misses on performance based on cache overhead per instruction, which is the number of extra cycles per instruction that is caused by cache misses of a particular kind (data read, data write, instructions, or a combination of those). More precisely,

$$\text{Cache overhead} = \frac{\text{Misses} * \text{MissPenalty}}{\text{Instruction count}}.$$

As usual, the results are reported as the arithmetic mean of the cache overhead for each program. (The average overhead is *not* computed with average miss ratios. The overhead is computed for each program, and the average of overheads taken.)

A number of different miss penalties are considered, ranging from 5 to 160 cycles. This range includes some penalties that are typical to current machines (5 to 40 cycles), and penalties that may occur on future machines, if the gap between processor and memory speeds continues to increase. In practice, not all penalties would apply to all cache sizes. For example, it is unlikely that any machine would have an 8K-byte cache with an 160-cycle

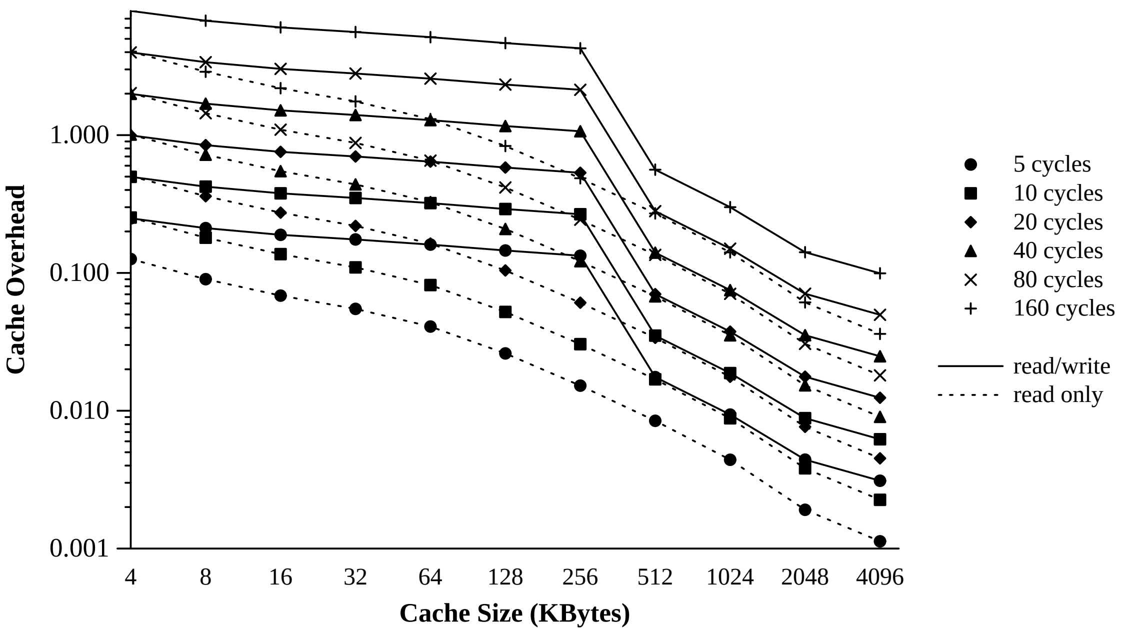


Figure 26: Data read and write cache overhead for direct mapped 32-byte block cache. The solid lines give overhead for read and write misses, which would be the overhead on a fetch-on-write cache. The dotted lines give overhead for read misses only, which would be the overhead on a write-validate cache.

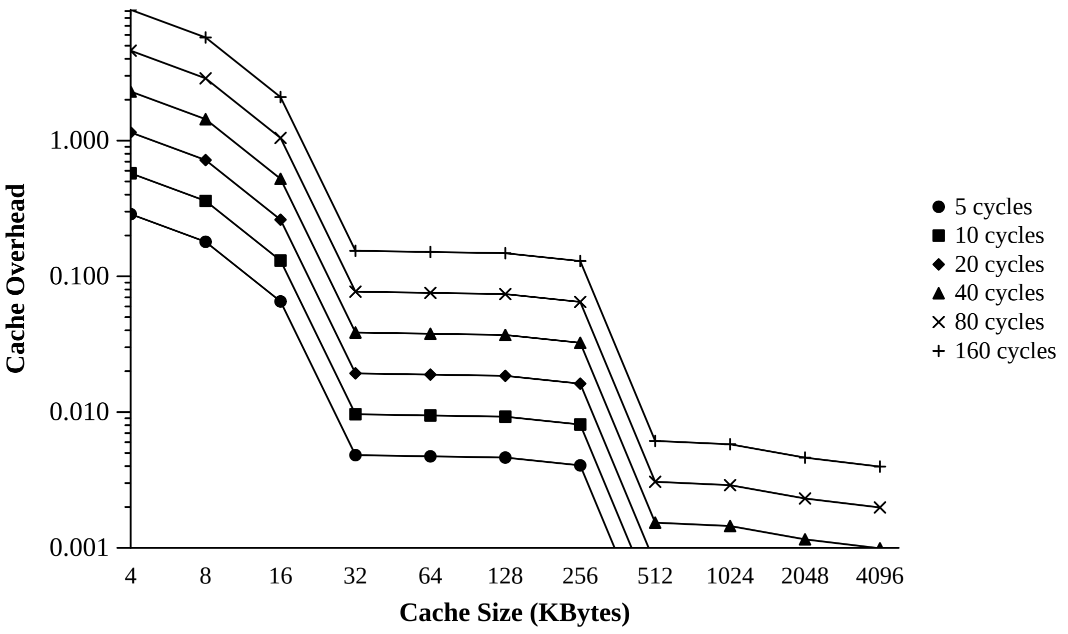


Figure 27: Instruction cache overhead for direct mapped fetch-on-write cache, 32 bytes block.

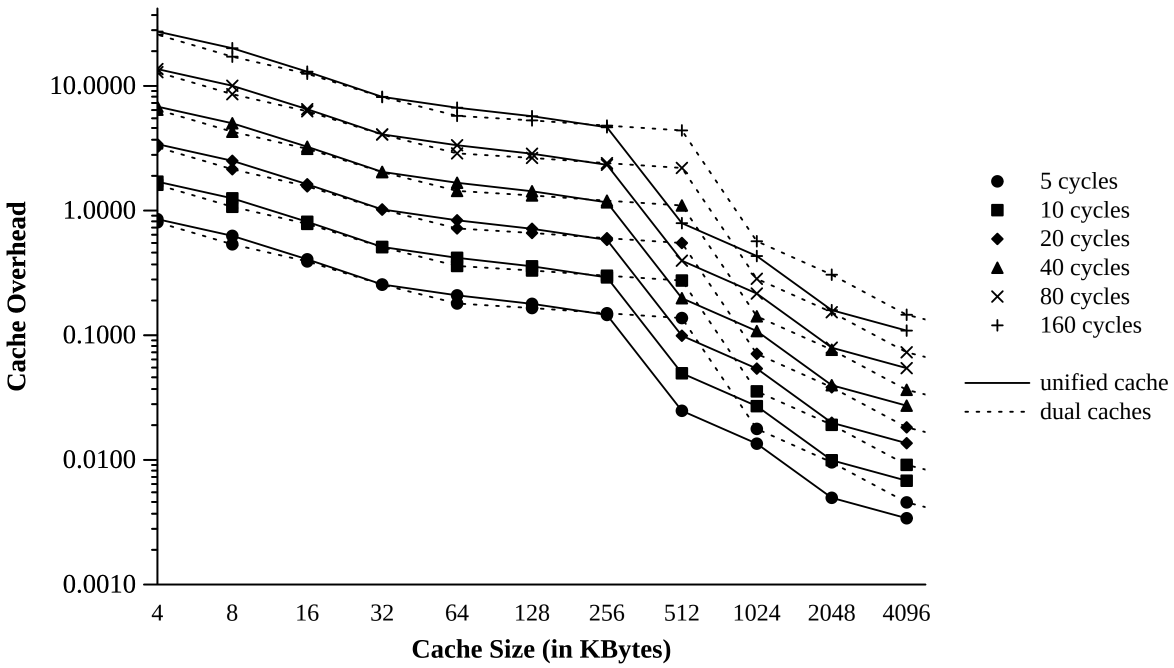


Figure 28: Total (instruction and data) cache overhead for direct-mapped 32-byte block fetch-on-write caches. The solid lines give the overhead for unified caches, and the dotted lines for separate instruction and data caches.

penalty; the designers of such a machine would certainly add a large second level cache to reduce the average penalty to a more reasonable level.

Figure 26 shows the average cache overhead for reads only, and read and write misses combined, for 32-byte-block direct-mapped data caches. The read overhead would be equivalent to that of a write-validate cache, and the read/write to that of a fetch-on-write cache.

Figure 27 shows the overhead for instruction cache misses, for a separate instruction cache. The overheads are somewhat high for small caches, but decreases sharply for caches between 32 and 256K bytes, becoming negligible for caches larger than 256K bytes.

Figure 28 shows the average overhead for unified fetch-on-write caches, compared to the total overhead for separate instruction and data caches. Unlike on the other graphs, the separate caches are labeled by their total size, to match the size of the unified cache. So at the point labeled 8K bytes, the unified cache is 8K bytes, and the separate instruction and data caches are 4K bytes each. For small caches, up to 512K bytes, which is the size of the allocation space, separate caches have lower overhead. For large caches, however, the unified cache is *much* better than the separate caches. The allocation space fits on the 512K byte cache, but does not fit on the 256K cache, which explains the huge difference in overhead for the 512K bytes. For caches bigger than 512K the difference is not so dramatic, but unified caches are still significantly better.

It is likely that the allocation cycle is continuously recycling the blocks of the small caches, which causes the instruction miss rate to go higher. Moreover, the interference between instructions and data is likely to introduce many conflicts, and this may offset the decreases the conflicts between data with data blocks (and instructions with instructions) that arise from increasing cache size. On the other hand, for large caches, the instructions do not need a very large cache, which leaves much of the bigger cache free to data, resulting in a significant decrease in data miss overhead, at the expense of at most a slight increase on instruction miss overhead.

5.6.1 Block size

Sections 5.1 through 5.5 showed the effects of block size on miss ratios, but since block size may affect miss penalties, lower miss ratios may not result in better performance. The cache miss penalty depends on memory latency and transfer rate, and is usually given by

$$Penalty = Latency + \frac{Words\ transferred}{Transfer\ rate},$$

where *Words transferred* is the number of words in a block. Because large blocks have more words to be transferred, they tend to have a higher penalty. It may be possible to increase the transfer rate by using wider memory and buses but, because of cost or other reasons, this may not always be a feasible or desirable solution. Another alternative that is more commonly used is to fetch the words out of order, getting the critical word first, and restarting the pipeline as soon as it is available. The machine can continue executing instructions while the remaining words in the block are being transferred. In this case, the penalty would be mostly independent on block size (though block size can still be a factor when there are many misses in sequence).

This section analyzes the trade-offs between lower miss ratios and lower miss penalties for different block sizes, assuming that changing block size does can affect the miss penalty, but does not affect the machine cycle time.

If the cycle time is not affected, then the cache overhead can be used to find the best block size. For block sizes $b1$ and $b2$, with miss penalties $p1$ and $p2$, the cache with block size $b1$ (C_{b1}) has better performance than the cache with block size $b2$ (C_{b2}) if the cache overhead of C_{b1} is less than the overhead for C_{b2}, or

$$\frac{\text{Misses}_{b1} * \text{Penalty}_{b1}}{\text{Instruction count}} < \frac{\text{Misses}_{b2} * \text{Penalty}_{b2}}{\text{Instruction count}}$$

which is equivalent to

$$\text{Penalty}_{b1} < \frac{\text{Misses}_{b2}}{\text{Misses}_{b1}} * \text{Penalty}_{b2}.$$

Figure 29 plots the ratio $\frac{\text{Misses}_{b2}}{\text{Misses}_{b1}}$ for direct-mapped caches and different b1, b2 values. The curves plot the break-even points, for which both blocks have equal performance. The large block size has better performance on the areas below or to the right of each curve. The graphs show the effect of read misses only (a write-validate cache) and both read and write misses (a fetch-on-write cache). Figure 30 plots the same ratios for 2-way associative caches.

Consider, for example, a 512K bytes write-validate cache, for a memory system with a 100ns latency, a one word (4 bytes) per 10ns transfer rate, and a machine with a 10ns cycle. The penalties are 14, 18, 22, 26, and 30 cycles for block sizes of 16, 32, 64, 128, and 256 bytes. The penalty ratio for 32 to 16-byte block is 1.286, and for a 512K-byte cache, this value is below the solid-line curve labeled "32/16" on Figure 29, and therefore the 32-byte block is better than the 16-byte block. Likewise, given the respective ratios of 1.22 and 1.18, we can conclude that a 64-byte block is better than a 32-byte one, and that a 128-byte

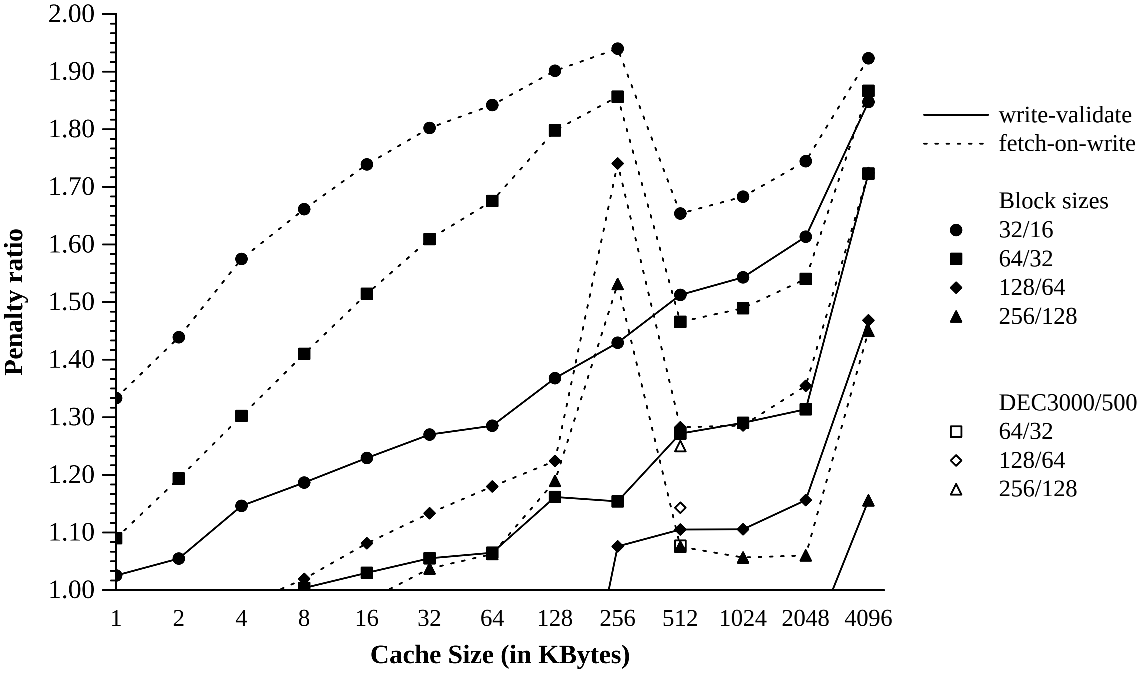


Figure 29: Block size versus penalty trade-off, for direct-mapped data caches.

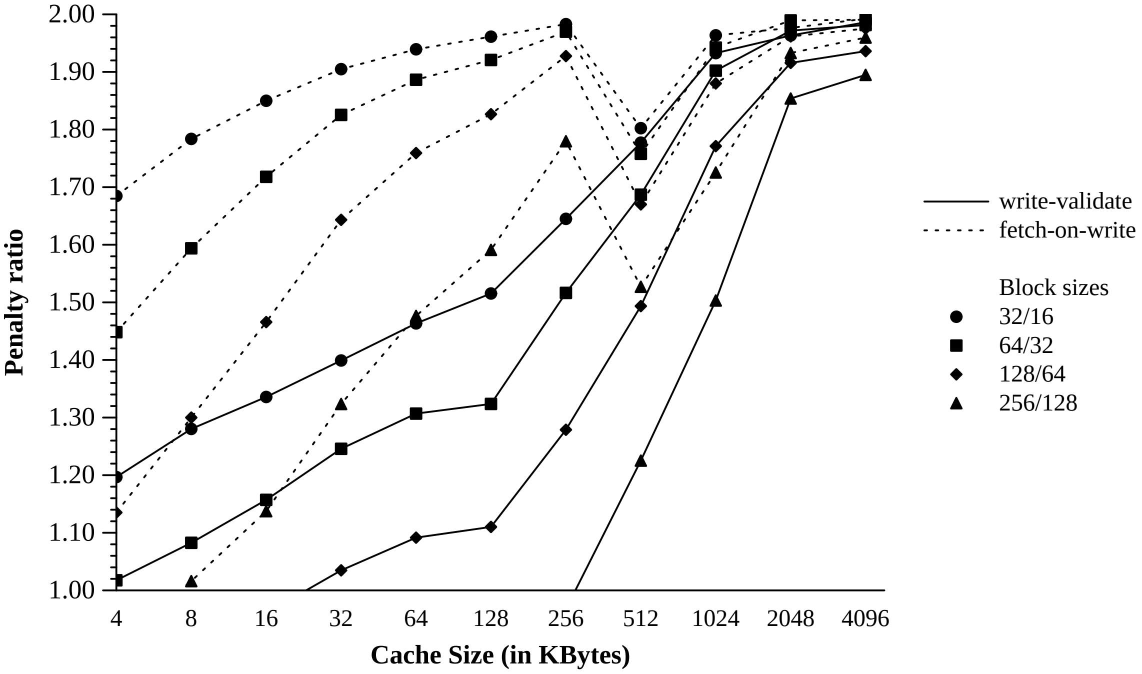


Figure 30: Block size versus penalty trade-off, for set-associative data caches.

block is better than a 64-byte block. Since a 256-byte block cache is never better than a 128-byte block cache (because 26-byte has higher miss ratios), we can conclude that a 128-byte block is the best choice for a 512K-byte cache. In a similar way, we can conclude that the best block size for a 512K-byte, 2-way associative cache is 256 bytes.

As another example, consider the 256K-byte cache. The 32-byte block is better than the 16-byte block, but 64-byte is not better than 32-byte block. The 128-byte block may or may not be better than the 32-byte block, we would have to plot the curve for "128/32" to decide that. But the "128/32" ratio is the product of the "64/32" by the "128/64" ratios, which is approximately $1.08 * 1.17 = 1.26$. The ratio penalty ratio for 128/32 is 1.44, and therefore the 128-byte block is better for a 256K-byte cache.

The DEC3000/500 [30] has a 512K-byte, 32-byte blocks, fetch-on-write direct-mapped second-level cache, with a 160ns (24 CPU cycles) latency and transfer rate of 16-byte per cycle, which give a miss penalty of 26 cycles. If the block size were increased to 64 bytes, then the penalty would be 28 cycles, and the ratio 1.077. This is well below the curve labeled "64/32" on Figure 29, and thus for this machine, a 64-byte block would give better performance than a 32-byte block. A 128-byte block would be the best choice for this machine. (But note that this is only an approximation, since we are ignoring the effect of the first level cache.)

For fetch-on-write caches, the best choice of the better block size would depend on whether or not the allocation space fits in the cache. The curves in Figures 29 and 30 drop significantly at the point where the allocation space begins to fit in the cache (512K bytes). But, in general, we may expect that large blocks should be the best choice for fetch-on-write caches, since they minimize the number of write misses.

5.7 Conclusion

Figure 31 compares the average miss ratios for the SML/NJ benchmark with those for the SPEC92 benchmark, as reported by Gee et al. [38]. In many cases, the miss ratios for the SML/NJ programs are lower, specially for large caches and small caches for which the block size is not too small. For example, for caches of up to 16K bytes with block larger than 32 bytes, the average miss ratios for the SML/NJ programs are significantly lower than those of the SPEC92 program. As cache size increases, because the write miss ratios of SML/NJ programs depend mostly on block size, they become higher than the SPEC

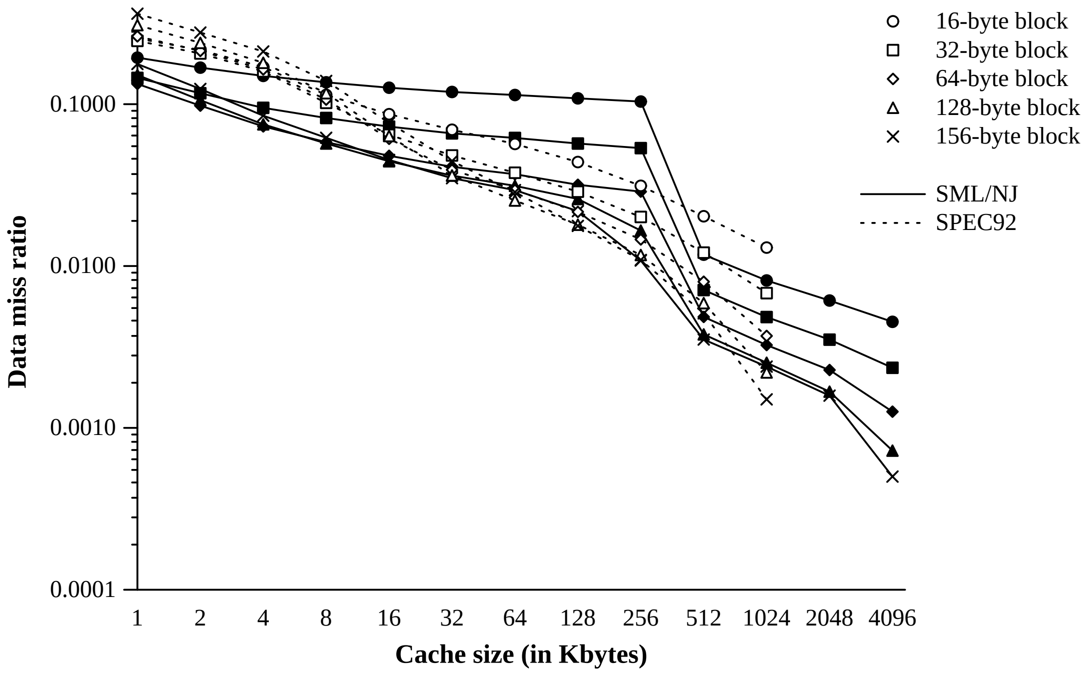


Figure 31: Average miss ratios for the SML/NJ programs and the SPEC92 benchmark, on direct mapped fetch-on-write caches.

programs. For large caches, for which the allocation space fits in the cache, the SML/NJ miss ratios become lower again. (But the SPEC92 programs are significantly bigger than the SML/NJ benchmark programs, so they may have a much larger working set, which would cause them to have higher miss rates on large caches.)

The fact that the SML/NJ programs miss ratios are lower in some cases, does not necessarily mean that they have better cache performance then the SPEC92 programs, since the programs in the two benchmarks have a different ratio of data references to instruction. The SML/NJ programs tend to have more memory references per instructions than the SPEC92 programs, and therefore they may have a higher cache overhead even if they have lower miss ratios. On average, 34.3 percent of the SPEC92 instructions are memory references (assuming a workload where each program has equal weight), compared to 39.2 percent of memory reference instructions for the SML/NJ benchmark. For an 8-Kbyte cache, with 32 byte blocks and a 10 cycle miss penalty, and assuming one instruction per cycle, the SPEC programs spend an average of 35% of their time on cache misses, compared with 32% for the ML programs. For 64-byte block caches, the cache overheads are 36% for SPEC and 23% for ML. (The overhead for the SPEC programs was computed using average instruction counts and miss ratios from Gee et al. [38].

Instruction miss rates for the SML/NJ programs on direct-mapped caches are higher than those of the SPEC92 benchmarks. This is likely to be the result of an aggressive in-line expansion policy by the SML/NJ compiler. The SPEC programs also include some long-running floating point programs that should spend a lot of time executing small inner loops, and thus they should have a low instruction rates.

Reinhold [60] gives the cache overhead for a set of 5 Scheme programs. Compared to those of the SML/NJ programs the overhead of the Scheme programs is significantly lower. For example, for a 32K bytes, 32 byte block, write-validate cache, a 9-cycle penalty, and run without garbage collections, the Scheme programs have a data cache overhead between 7 and 8 percent, compared to 11 percent for the SML/NJ programs on a similar cache. But the Scheme programs have a much lower ratio of memory references to instructions, 27.3 on average.

The cache overhead for the SML/NJ programs can be high on some cache architectures, particularly on fetch-on-write caches with small blocks. Clearly, write-validate caches, or other cache architectures that do not have a penalty for sequential allocation writes, are the best architectures for functional programs with sequential heap allocation. Write miss policy and cache size are the cache design parameters that have the largest effect on the cache performance of SML/NJ programs. Other parameters, such as block size and associativity can also affect performance, that they have a smaller effect. Usually a block size of 32 or 64 bytes, and direct-mapped or low associativity (2 or 4-way) should give the best or near best performance. Block size can have a large impact on performance if an unfavorable write-miss policy is used, in which case large blocks should give better performance.

The results presented on this chapter confirm previous results on the cache performance of programs with dynamic heap allocation [57, 46, 83, 60] and SML/NJ programs [29].

Finally, the results presented here are for one level cache hierarchies. For multi-level hierarchies they are certainly valid for the first level of the hierarchy, but they cannot be applied directly to the second level since the reference stream that reaches the second level cache is different from that seen by the first level cache. However, the solo miss ratios (that is, the miss ratio for a single level cache) can be used as an approximation of the global miss ratio of a second level cache, and this approximation is reasonably accurate if the second level cache is much larger than the first level cache [58, Section 5.3], which is usually the case. Most results presented here may also be valid for second level caches.

Chapter 6

Garbage Collection Effects on Performance

Chapter 5 showed that ML programs can have many write misses, because of the sequential allocation writes, and that most read misses are caused by references to old objects, that are located in the older generations. This chapter investigates the effects of garbage collection on cache performance and the possibility of using garbage collection to reduce both write and read misses of ML programs.

Write misses can be eliminated by fitting the allocation space in the cache (instead of using a large allocation space), but this makes garbage collections more frequent and can increase garbage collection overhead. Therefore, there is a trade-off between reduced cache overhead versus extra garbage collection time. If the penalty for write misses is high, then the time a machine spends waiting for cache misses can offset the extra garbage collection time, but this may not be the case if the penalty is low, or if the machine has no penalty for write misses. Intuitively, we can expect that for machines with large caches it may be better to fit the allocation space on cache, because garbage collection overhead should be reasonably low, but fitting the allocation space on a small cache may result in unacceptably high garbage collection overhead.

Read misses might also be affected by garbage collections. Since most objects have a short life, when a block containing an old object is fetched, this block may contain many dead objects, which are never going to be referenced again. Frequent collections can improve locality by compacting the few live objects in the heap into fewer memory blocks, increasing the chance that a fetched block contains many live objects, which may be referenced in the future without causing an additional cache miss. For SML/NJ programs specifically, because most old objects tend to be pairs, and pairs on the older generations are smaller (they do not have a descriptor field), there may be one extra advantage of doing frequent collections, since more pairs can fit into a single block. Another way by which garbage collection can improve spatial locality is by copying objects in such a way as to put related objects, that are likely to be referenced within a short time of each other (for example, adjacent cells of a list), in the same block. This can be easily achieved by using different algorithms to traverse and copy the graph of live objects during the collection, such as depth-first search.

This chapter presents a study of the effects of garbage collection on cache performance. Two main issues are considered, the size of the allocation space (in other words, the frequency of minor collections) and the algorithm used to traverse the graph of live objects. Section 6.1 presents a study on the effects of the size of the allocation space on cache performance. The benchmark programs are run with many different allocation space sizes, from 16M bytes to "infinite", where an infinite allocation space is big enough to hold all objects allocated by a program, and therefore the programs run without any garbage collections. The "infinite" allocation space is used only to help understanding the effects of garbage collection on cache performance. Usually, it is not possible, or practical, to use a very large allocation space (unless the program is small). Section 6.2 compares the miss ratios of three traversal algorithms, the standard breadth-first search used by the SML/NJ system, a depth-first collector, and a variation of the breadth-first collector that tries to put all adjacent list cells on contiguous addresses (*list linearization*). Section 6.3.2 presents an analysis the performance of the benchmark programs on two different machines, the DEC5000/240 MIPS workstations and the DEC3000/500 Alpha workstation.

6.1 Allocation Space Size

6.1.1 Write misses

Figures 32 and 33 shows the effects of the size of the allocation space on write misses, for direct-mapped, 2-way, and 4-way associative caches. The graphs plot average write miss ratios as a function of allocation space size, for caches from 4K bytes to 4M bytes. The write miss ratios are low when the allocation space fits in the cache (that is, the allocation space is of the same size or smaller than the cache size), but jump to a value close to 1/block size in words when the allocation space becomes larger than the cache.

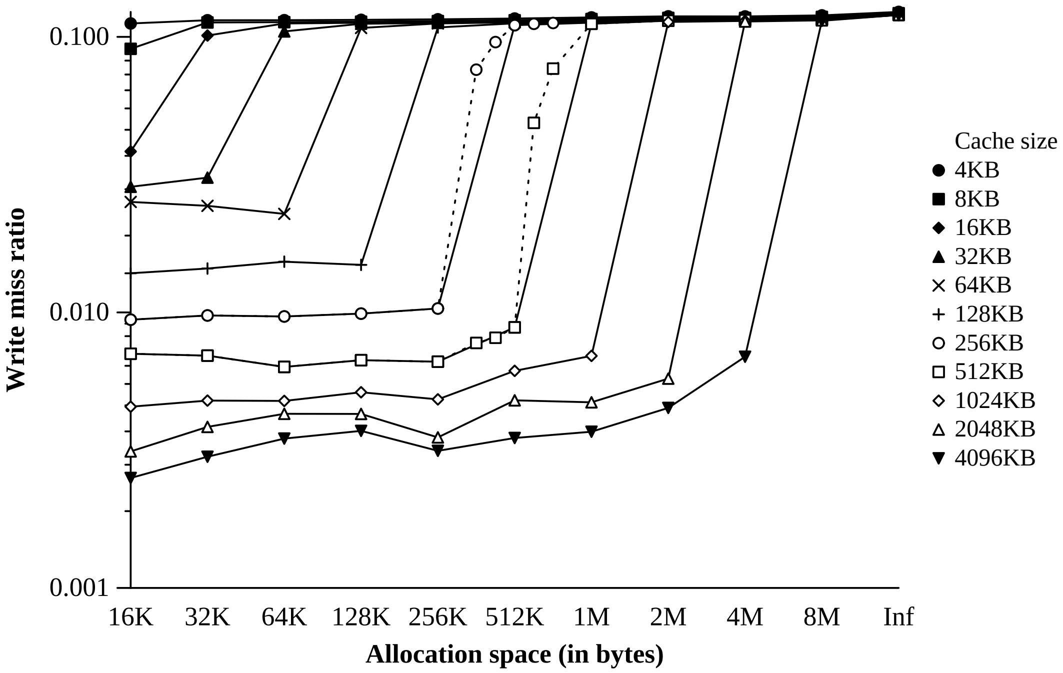


Figure 32: Average write miss ratios as a function of allocation space size, for direct-mapped, 32-byte block, fetch-on-write data caches.

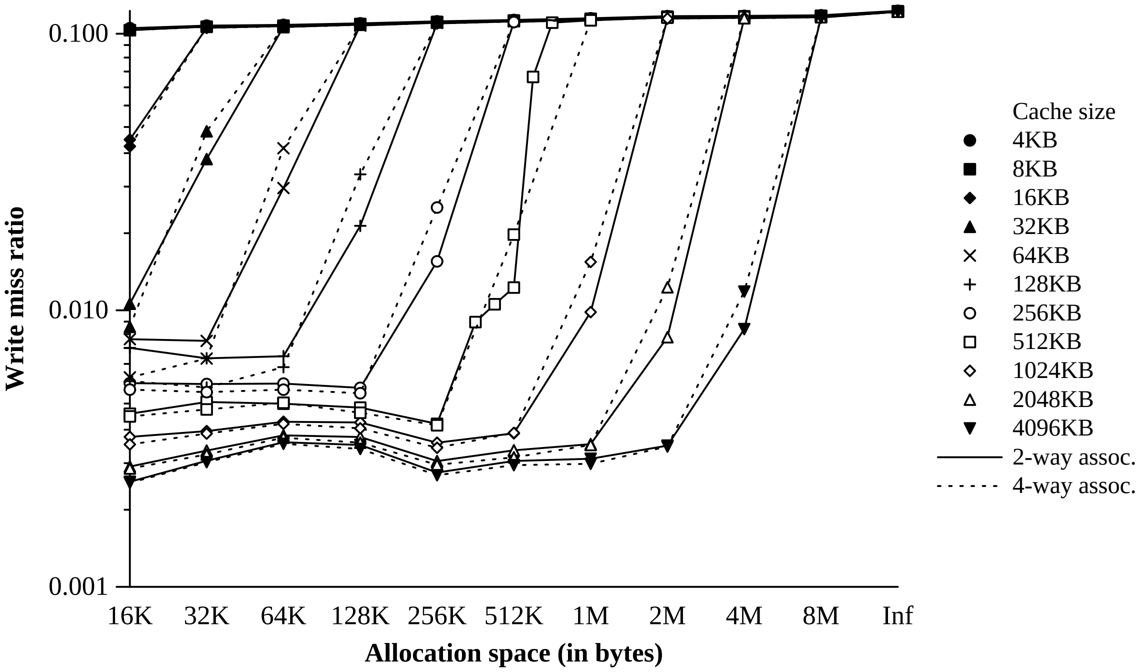


Figure 33: Average write miss ratios as a function of allocation space size, for 2-way associative, 32-byte block, fetch-on-write data caches.

When the allocation space is larger than the cache, the miss ratios do not depend on cache size, but when the allocation space fits in the cache, the miss ratios decrease as cache size increases, since most writes misses should be caused by conflicts between older generation and allocation space blocks, which should decrease when cache size increases.

On direct-mapped caches, the miss ratios are low until the point where the cache is of the same size as the allocation space, and then jumps to the higher value. But on associative caches, the miss ratios begin to rise even before the allocation space size becomes as large as the cache, as we would expect from the discussion in Section 5.3. When the allocation space is the same size as the cache the low-associativity caches have lower miss ratios than the high-associativity caches.

The graphs plot a limited number of allocation space sizes, and the remaining points are approximated by linear interpolation. The approximation should be reasonably accurate for points in either the low or the high area for each curve, but there may be a noticeable difference for the transition region. Figures 32 and 33 plot some intermediate points around the allocation space size points 256K and 512K bytes. The allocation space sizes at these intermediate points are 384K, 448K, 640K, and 768K bytes. On the direct-mapped caches, the miss ratios begin to rise very fast when the allocation space becomes larger than cache size. When the allocation space is one and a half times the cache size, the miss ratios are close to the maximum value, even though half of the allocation space at this point would not be removed from the cache by the allocation cycle. For the associative caches, the miss ratios begin to rise sharply after the allocation space becomes more than half of the cache size, and when the allocation space is one and a half times the cache size, the miss ratios already reach the maximum value. In general, direct-mapped caches should have lower miss ratios around the point where the allocation space matches the cache size.

6.1.2 Read misses

Figures 34 and 35 shows average read miss ratios as a function of allocation space size, for direct-mapped and 4-way associative caches. The size of the allocation space has no significant effect on the miss ratios of small direct-mapped caches, and there is only a slight increase in miss ratios for 4-way associative caches. For large caches, however, the curves tend to oscillate, with a significant difference between the minimum and maximum values. However, it is not clear from the graphs whether a small or large allocation space is better.

The different behavior of small and large caches can be explained intuitively by the

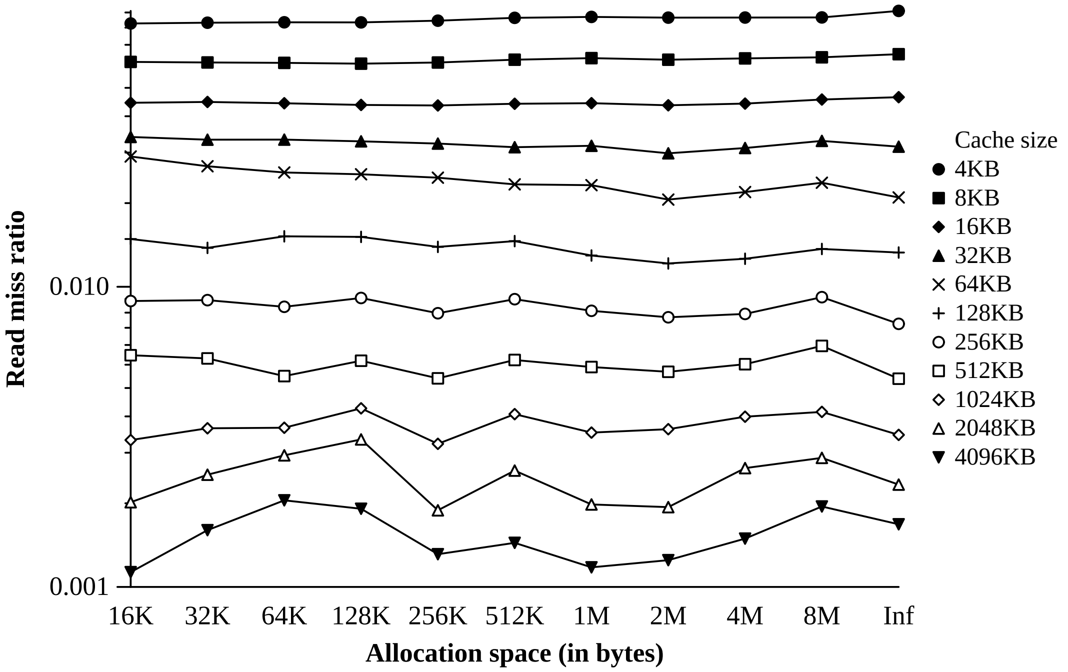


Figure 34: Average read miss ratios as a function of allocation space size, for direct-mapped, 32-byte block, fetch-on-write data caches.

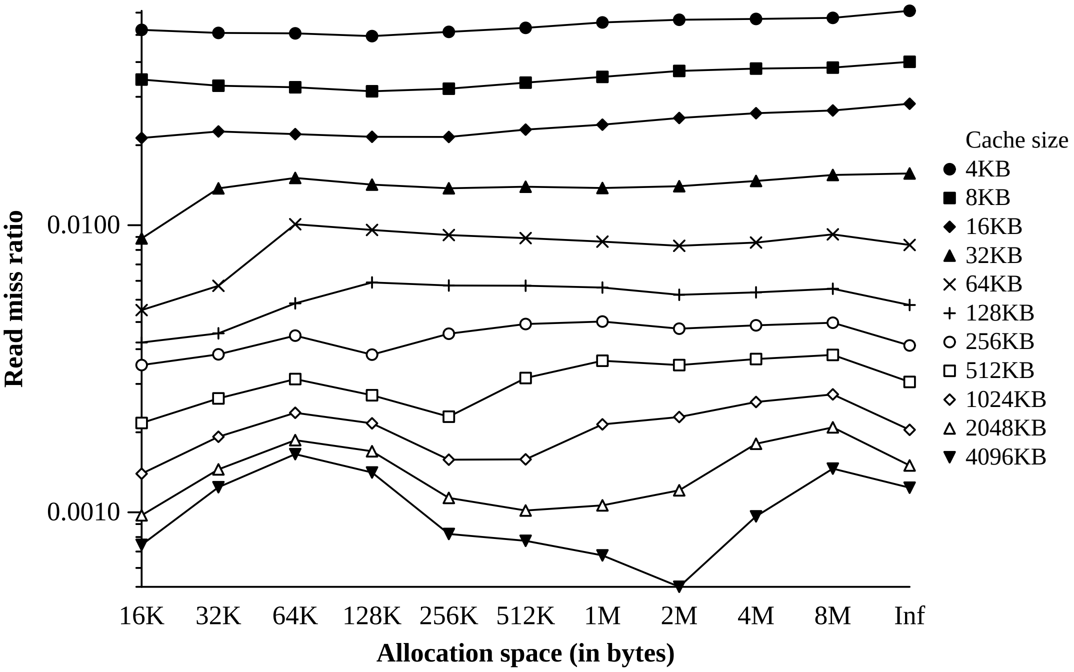


Figure 35: Average read miss ratios as a function of allocation space size, for 4-way associative, 32-byte block, fetch-on-write data caches.

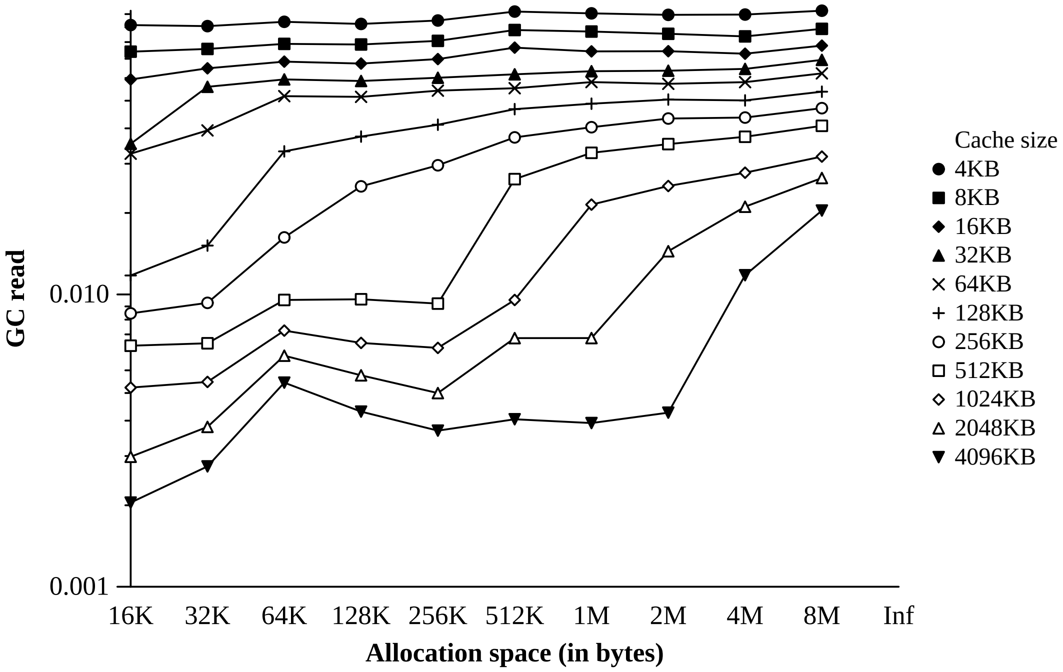


Figure 36: Average garbage collection read miss ratios as a function of allocation space size, fo direct-mapped, 32-byte block, fetch-on-write data caches.

different sources of cache misses, that is compulsory misses, capacity misses, and conflict misses. Most misses in a small cache should be capacity misses, but most misses on a large cache should be compulsory or conflict misses. Therefore, a large cache should be more affected by the different ways the collector may place objects in the heap, since the placement of two or more objects on addresses that map to the same cache block could result in more conflict misses.

The cyclic allocation pattern can also be a factor in the different behavior of small and large caches. Because the programs allocate on a very fast rate, a small cache should be filled mostly with recently allocated data, and blocks containing older objects should be replaced a short time after they are fetched. In this case, the frequency of collections should not make much difference, since a block would not be kept in the cache for a time long enough to take advantage of the presence of additional live objects in the block.

In general, we may expect that the effects of garbage collection on read misses would be more significant for large associative caches with large blocks, than for small direct-mapped caches with smaller blocks, since on a large associative cache, a block that is fetched into the cache may be kept in the cache for a much longer time.

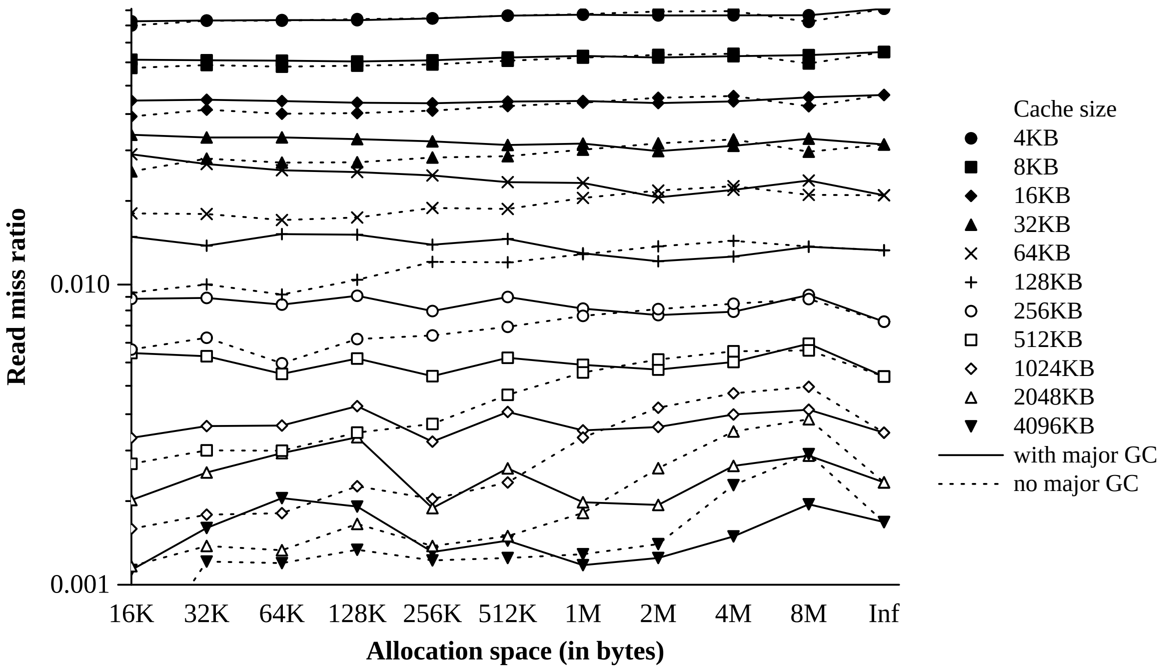
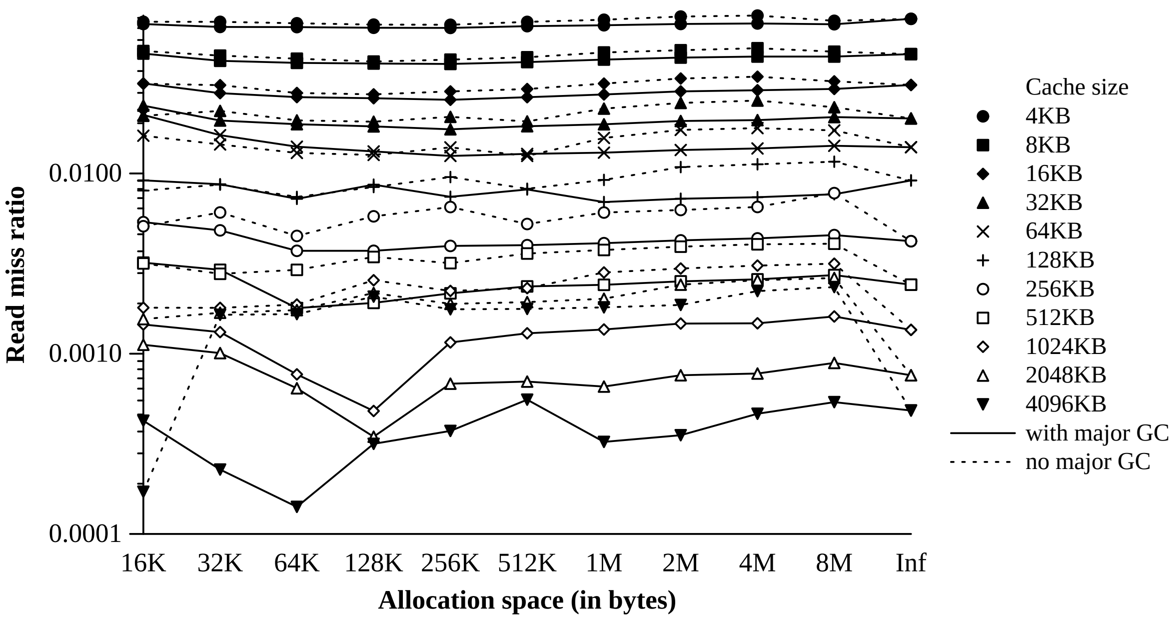


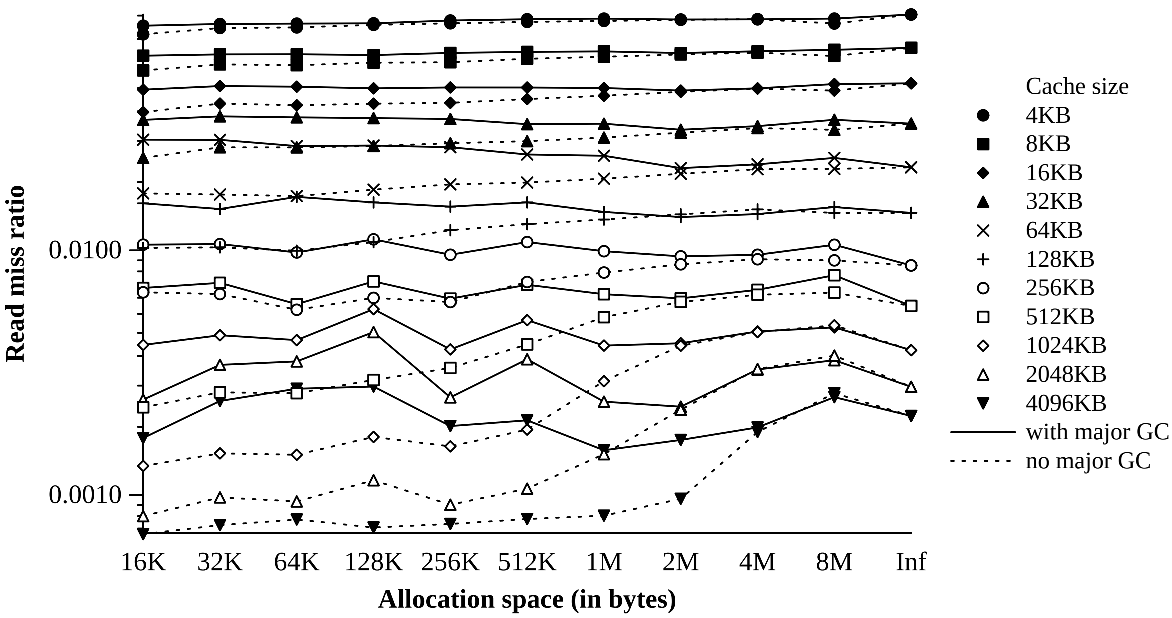
Figure 37: Average read miss ratios for the programs run with and without major collections. Direct-mapped data cache, 32-byte block.

Another fact that can be observed in Figures 34 and 35 is that the miss ratios for large caches begin to increase for large allocation spaces, but then decrease again when the allocation space is infinite, although they do not get to the minimum value. This fact can be explained by the effect of garbage collection misses: as the allocation space size increases, even though the total number of words copied decreases, the number of words copied per collection increases and therefore the miss ratios can also increase. The increase in the number of words copied per collection when the allocation space size increases should happen until a point where the allocation space size becomes close enough to infinite, at which point the number of words copied per collection should also decrease. Figure 36 shows garbage collection read miss ratios as a function of allocation space size, for direct-mapped data caches. As expected, garbage collection read miss ratios increase as the size of the allocation space increases, especially on large caches.

6.1.3 Major collections



(a) Barnes-Hut, Mandelbrot, Life, Ray



(b) Boyer, Knuth-Bendix, Lexgen, MLYACC, Simple, VLIW

Figure 38: Comparison of miss ratios with and without major GC, for (a) programs with low survival rate, and (b) high survival rates. Direct-mapped data caches, 32-byte block.

Section 6.1.2 considered the effects of the size of the allocation space on miss ratios, but when the allocation space size changes, the size of older generations, and therefore major collection frequency, may also change. This section analyses the effects of major collections on performance. The programs are run with varying allocation space sizes, but the size of the first generation is "infinite", and therefore there are no major collections. Although it is usually not possible to use an "infinite" first generation, it may be possible (on future versions on the SML/NJ system) to make the size of the first generation be independent on the size of the allocation space (although the size of the allocation space may put a lower bound on the size of the first generation), and thus keep the first generation large even if the size of the allocation space is small.

On the current implementation, each arena of the first generation must be at least as large as the allocation space. The first generation is usually small when the allocation space is small, and thus there are frequent major collections. On the other hand, if the allocation space is large, so is the first generation, and there are few major collections.

Figure 37 shows read miss ratios as a function of allocation space size, when the programs are run with an infinite first generation, compared with the miss ratios when the programs are run with regular major collections. Major collections have a significant effect on miss ratios of large caches. In general, miss ratios without major collections are lower for small allocation spaces, which suggests that doing frequent major collections may be bad for performance. On the other hand, for large allocation spaces (for example, 2M bytes or more), the miss ratios are lower when there are major collections, which suggests that not doing any major collections may also be a bad idea.

Major collections usually collect from a large area, and thus the collection itself may be a significant source of cache misses. Another important factor is that objects can be spread across many different areas, five different arenas for each generation, and five different generations. For many programs, there are few live older objects, and spreading these objects over many different areas may cause many conflict misses. For example, if a program has 1M-byte of older pairs, and they are all in contiguous addresses in the first generation, then there should be no conflicts on a 1M-byte cache. But if these pairs are divided between four generations, if the generations are aligned within cache boundaries, then many conflicts misses may occur on the same 1M-byte cache.

On the other hand, some programs have many objects that survive to minor collections but die soon after that. For these programs, there may be some benefit from doing more

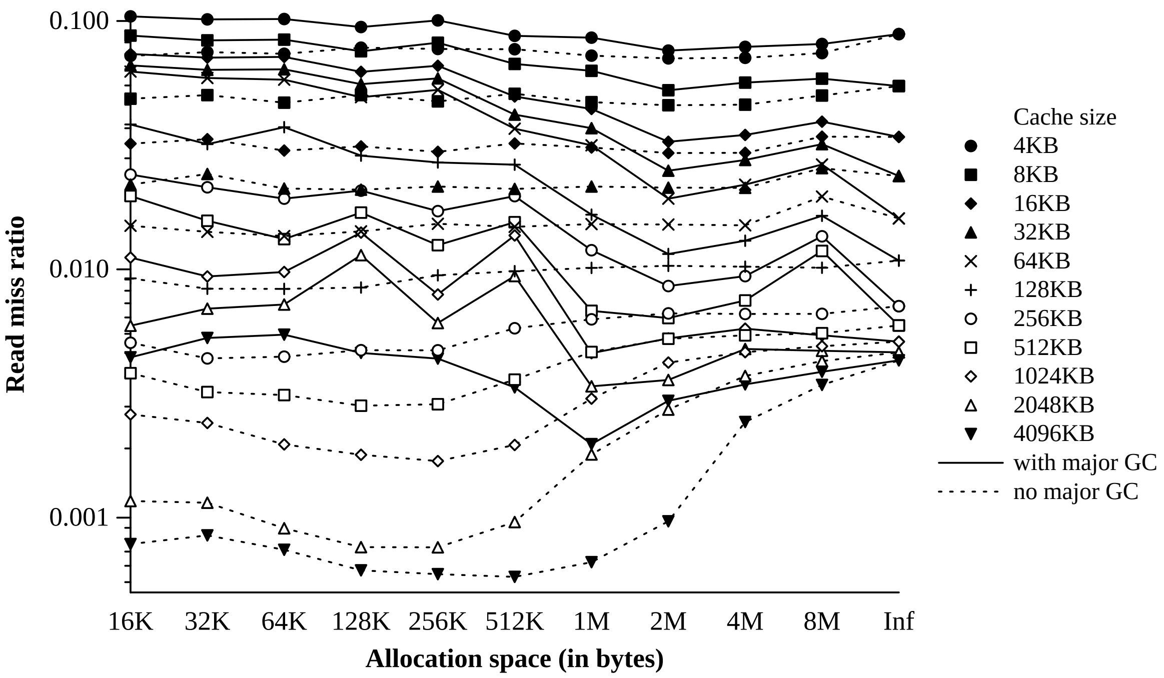


Figure 39: Read miss ratios as a function of allocation space size, for the program Simple.

frequent major collections, since the overhead for the collections should be small and the collection compacts the live objects on a fewer memory blocks.

Therefore, for programs that have a high major collection survival rate, it should be better to do few major collections, or even no garbage collection at all (if that is possible). On the other hand, programs with a low survival rate may benefit from a moderate number of major collections. Using the results from Tables 15 and 16 (Chapter 4) the programs can be divided into two groups, those with low major collection survival rates (Barnes-Hut, Life, Mandelbrot, and Ray), and those with high major collection survival rate (Boyer, Knuth-Bendix, Lexgen, MLYACC, Simple, and VLIW). The programs in the first group should have lower miss ratios when major collection are frequent, while the programs in the second group should have lower miss ratios when there are no major collections. Figure 38 shows that this is indeed the case.

Many programs in the benchmark suite, however, have little major collection overhead, even if the first generation is small, and for these programs there is not much difference between running them with or without major collections. Therefore, to have a better idea of the effects of major collections on cache performance, we must look at programs for which major collection policy can have a significant effect on performance. Simple is one of these programs.

Figure 39 shows Simple miss ratios as a function of allocation space size, when the program is run with regular major collections and without any major collections. The difference is significant even for small caches, and it is huge for large ones. The major collection survival rates for Simple are very high, near 90% for all major collections.

6.1.4 Descriptors

As shown in Section 4.1, descriptors account for about 20% of allocation on average, even though most of them would never be used. Descriptors are used only by the garbage collector and for polymorphic equality testing. Since ML is a statically typed language, it is possible to do garbage collection (and polymorphic equality testing) without using descriptors [4, 77], although this could increase the complexity and overhead of garbage collection.

The use of descriptors can add a significant overhead to a program, not only because of the 20% of extra writes, but also because of extra garbage collection and possibly cache overhead. Given a fixed allocation space, a program that allocates objects with descriptors would fill the allocation space faster than a program that does not use them, and thus would do more garbage collections collections that the descriptor-free program. This could increase garbage collection overhead, but on the other hand, a descriptor-free collector would have to determine the types of the objects in the heap, which can be costly.

Descriptors can also have an effect on cache performance since they can take space in the cache, even though they may never be used, possibly decreasing the cache capacity by as much as 20%.

One way to analyze the effects of descriptors is to consider the programs that use mostly pairs as user data structures. Pairs on the older generations do not have descriptors, and thus we would expect that if descriptors have a significant impact on miss ratios, then those programs that use many pairs should have lower miss ratios when minor collections are frequent. The programs Boyer, Knuth-Bendix, Lexgen, Life, and MLYACC all use many pairs. The average miss ratios for these programs show that miss ratios on small direct-mapped caches increase as the allocation space size increases, with a difference of up to 30% between the miss ratios from a 16K-byte to an infinite allocation space. But this is probably not only the result of putting more pairs into a block, but can also be due to all pairs on older generations being aligned to block boundaries (and thus there are not double misses when the head and tail of the pairs are read at the same time).

However, the read miss ratios are usually low, and even if miss ratios become lower, the effect on performance should be small. The effect of descriptors should be significant only for fetch-on-write and write-around caches, where the lower allocation rate can save 20% of the allocation misses.

6.1.5 Total performance

This section presents a quantitative analyses of the trade-offs between garbage collection and cache overhead. This analysis uses the *Cycles Per User Instruction* instruction metric that was defined on Section 3.2. Two types of cache architectures are considered, fetch-on-write and write-validate. As Section 5.2 shows, the results for fetch-on-write caches may be used as an approximation for write-around caches.

Garbage collection overhead

Figure 40 shows how garbage collection overhead (garbage collection cycles per user instructions) changes with allocation space size. Minor collection overhead always decreases when the allocation space increases. Usually, this is also the case for major collections, but some times major collections may increase when the allocation space size increases.

Garbage collection overhead varies considerably from one program to another. It is negligible for programs such as Ray and Mandelbrot, but it can be more than 100% for Boyer if the allocation space is small. Some programs do more minor than major collections, while others have more major than minor collections.

Fetch-on-write caches

Write misses are the major concern for fetch-on-write caches. In general, we may expect that fitting the allocation space on large caches would give better performance, but the extra garbage collection overhead caused by fitting the allocation space on a small cache should offset the lower cache overhead, and thus using a large allocation space would give better performance for small caches. Cache overhead depends not only on miss ratios, but also on the miss penalty. The higher the penalty, the higher the cache overhead is, and therefore, if the penalty is high, then it may be better to fit the allocation space even on a

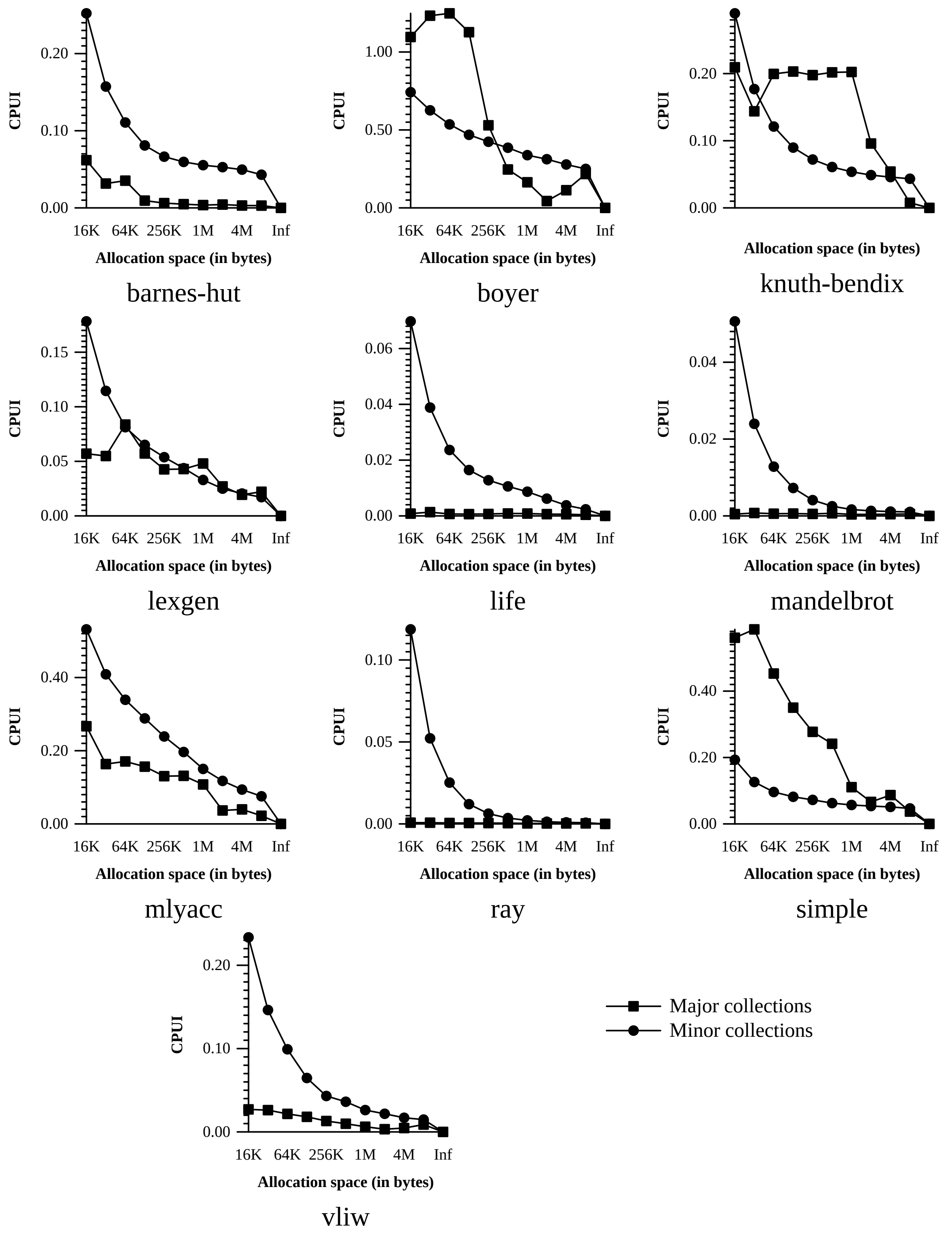


Figure 40: Minor and major garbage collection overhead for each program, assuming an ideal cache (no read or write misses).

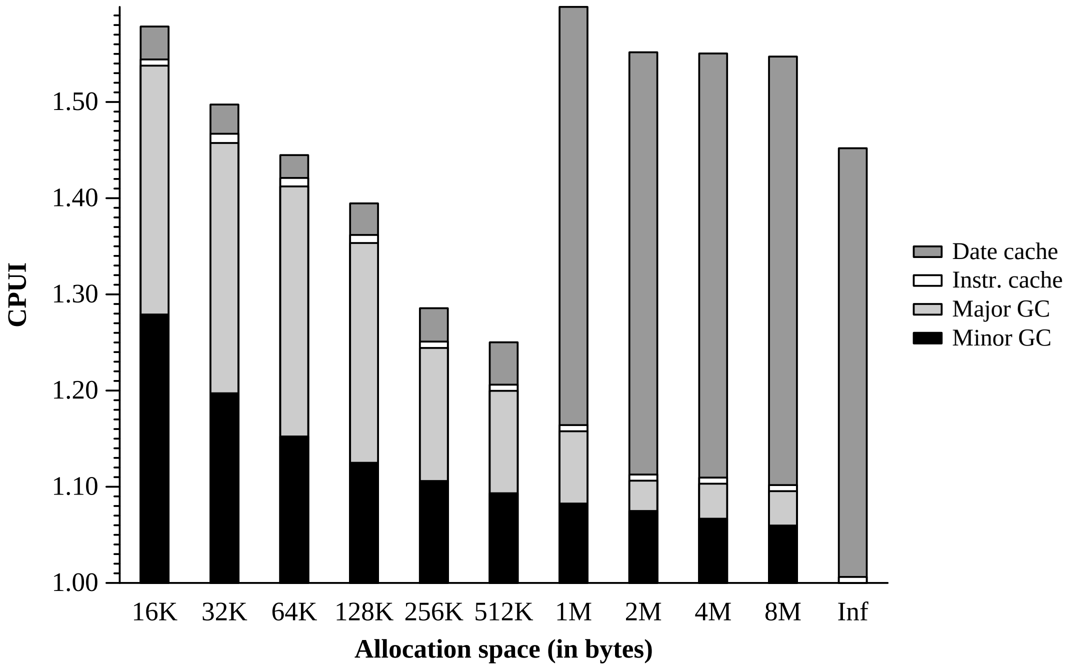


Figure 41: Cycles per user instruction for a 512K-byte dual fetch-on-write cache, with 32-byte block and 20-cycle miss penalty.

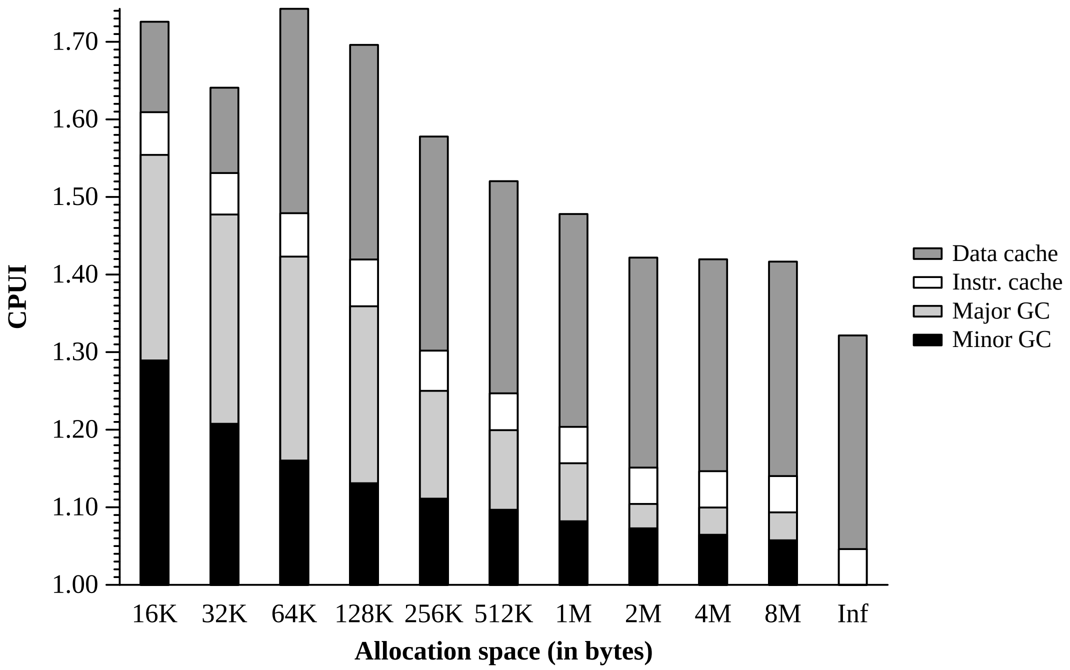


Figure 42: Cycles per user instruction for a 32K-byte dual fetch-on-write cache, with 32-byte block and 10-cycle miss penalty.

small cache.

Figure 41 shows average CPUI as a function of allocation space size, for a 512K-byte, 32-byte block, dual fetch-on-write caches, with 20 cycles miss penalty. The average CPUI is the arithmetic mean of the CPUI of each program. The graph shows CPUI broken down by its components: minor collections, major collections, user instruction cache misses, and user data cache misses. Garbage collection instruction and data cache misses are shown as part of garbage collection overhead. The average garbage collection overhead can be close to 50% for small caches, and is still high, close to 20%, for the allocation space that matches cache size. Data cache overhead is low when the allocation space fits in the cache, but becomes very high when the allocation space is larger than the cache. For this particular cache architecture, fitting the allocation space in the cache gives significantly better performance, in the average case, than using a large, or even an infinite, allocation space.

But things can be completely different if the cache size or the penalty change. For example, Figure 42 shows the average CPUI for 32K-byte fetch-on-write dual caches, with 10 cycles miss penalty. For this particular cache, even though the data cache overhead is high when the the allocation space is larger than the cache, the extra cache overhead of a large allocation space does not offset the lower garbage collection overhead of large allocation spaces.

In general, for a given cache size, if the miss penalty is high, then we may expect that fitting the allocation space on cache would have better performance, while if the penalty is low, then it may be better to use a large allocation space. There should be a certain value for the miss penalty for which the performance will be the same for an in-cache or large allocation space. Using the performance model of Section 3.2 it is possible to quantitatively analyze the trade-offs between garbage collection and cache overhead, and determine the break-even value for the miss penalty.

Given a cache of size c bytes, we can find the break even point by equating the total number of cycles for an allocation space of size c bytes and for an allocation space of size 8M bytes, and solving the resulting equation for p, the value of the miss penalty. Equating the cycle counts we have:

$$i + mp + rp + wp = i' + m'p + r'p + w'p$$

where i, m, r, and w are the instruction count, the number of instruction misses, data read misses, and write misses for the allocation space that fits in the cache, and i', m', r', and m

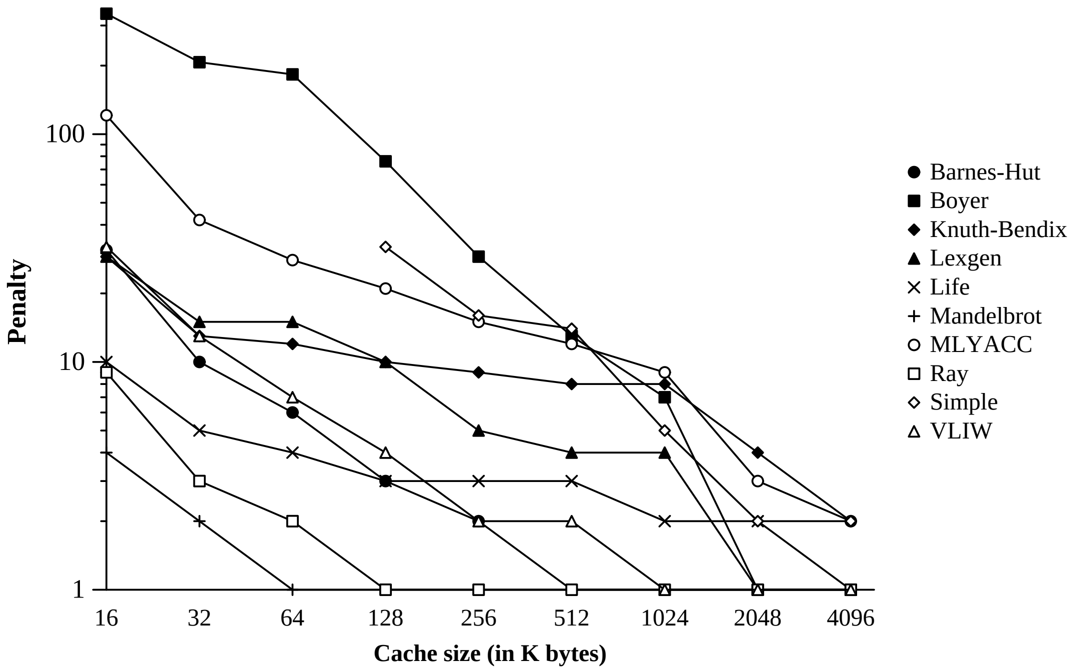


Figure 43: Minimum miss penalty for which it is better to have the allocation space be the same size of a fetch-on-write cache, for split instruction and data caches.

are the instruction count, number of instruction misses, data read misses, and write misses for the allocation space of 8M bytes. Solving the equation for p, we obtain the break-even penalty:

$$p = \frac{i' - i}{m + r + w - m' - r' - w'} \tag{2}$$

Figure 43 shows the break-even miss penalty for which fitting the allocation space in the cache (the allocation space is of the same size or smaller than the cache) has the same performance as when an 8M-byte allocation space is used, where the break-even penalty is computed by equation 2, and rounded up to an integer value. For a given cache size, fitting the allocation space in the cache gives better performance when penalties are higher than the break-even penalty, while using the 8M-byte allocation space gives better performance when penalties are lower than the break-even point. If a point is missing in a curve, then, for that cache size, fitting the allocation space in the cache always gives worse performance, regardless of the penalty. For these points, the break-even penalty computed by equation 2 is negative.

In general, for large caches it is usually better to fit all programs in the cache, even for

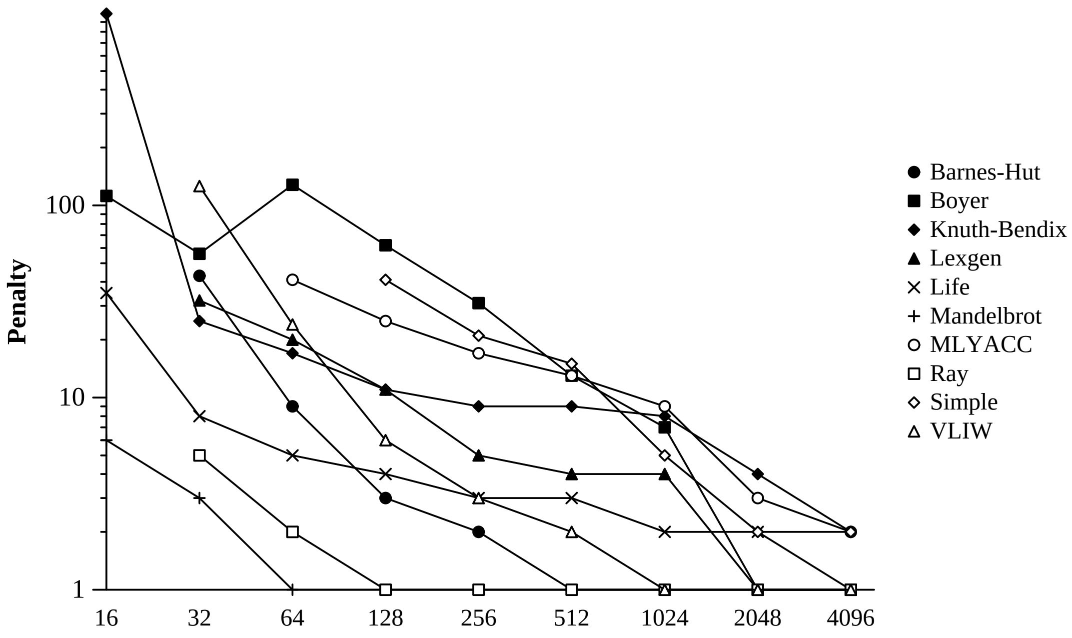


Figure 44: Minimum miss penalty for which it is better to have the allocation space be the same size of a fetch-on-write cache, for unified caches.

small penalties. For example, for a 1M-byte cache, all programs have better performance when the allocation space fits in the cache for penalties of 9 cycles or more. For a 512K-byte cache the allocation space should fit in the cache if the penalty is at least 14 cycles. Miss penalties of these magnitudes, or even higher, are common for large caches, and therefore it is reasonably safe to fit the allocation space in caches of 512K bytes or more.

For small caches, fitting the allocation space in the cache gives better performance only if penalties are high. Since most small caches tend to have small penalties (because if the memory is too slow, the designers of the machine would either choose a larger cache or add a large second level cache to reduce the average miss penalty), it may better to use a large allocation space with small caches. But here, there is a considerable difference from one program to another. Some programs (those with low garbage collection overhead, as Ray, Mandelbrot, and Life) would have better performance if the allocation space fits even on small caches. But for programs with high garbage collection overhead, such as Boyer and Simple, fitting the allocation space in the cache would never give better performance, or in the case of Boyer, only if the penalties are about hundreds of cycles.

In general, there may be a significant difference in garbage collection overhead between

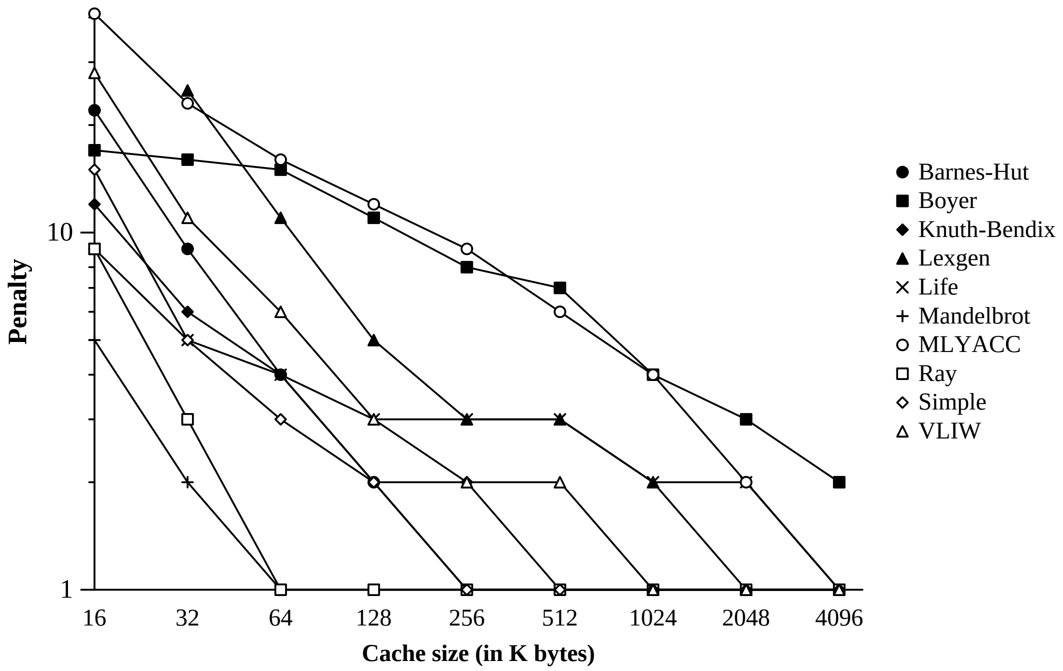


Figure 45: Same as Figure 43, but with no major collections (an infinite first generation.)

programs, so it may difficult to tell if for a specific program and a specific machine, the allocation space should be fitted in the cache or not. Either the worse case, or the average case from Figure 43 can be used to help making a decision, but since the model used to compute this information may not reflect exactly the performance of the machine, there may be a considerable margin of error. Another problem is that for some machines (such as non-blocking machines), it may be difficult to determine the exact penalty. The best approach is to run a few programs on the machine varying the size of the allocation space and then use the results of these runs to make a decision.

Because the size of older generations depend on the size of the allocation space, reducing the size of the allocation space to fit it in the cache increases not only minor collection but also major collection overhead. But it may be possible (in future implementations of the SML/NJ system) to allow the first generation to increase to compensate for the extra minor collection overhead. Figure 45 shows the break-even penalty, when an "infinite" first generation is used, in which case the programs run with no major collections. The break-even penalties in this case are much lower, showing that keeping the first generation can be a good alternative to compensate for the extra overhead of minor collections.

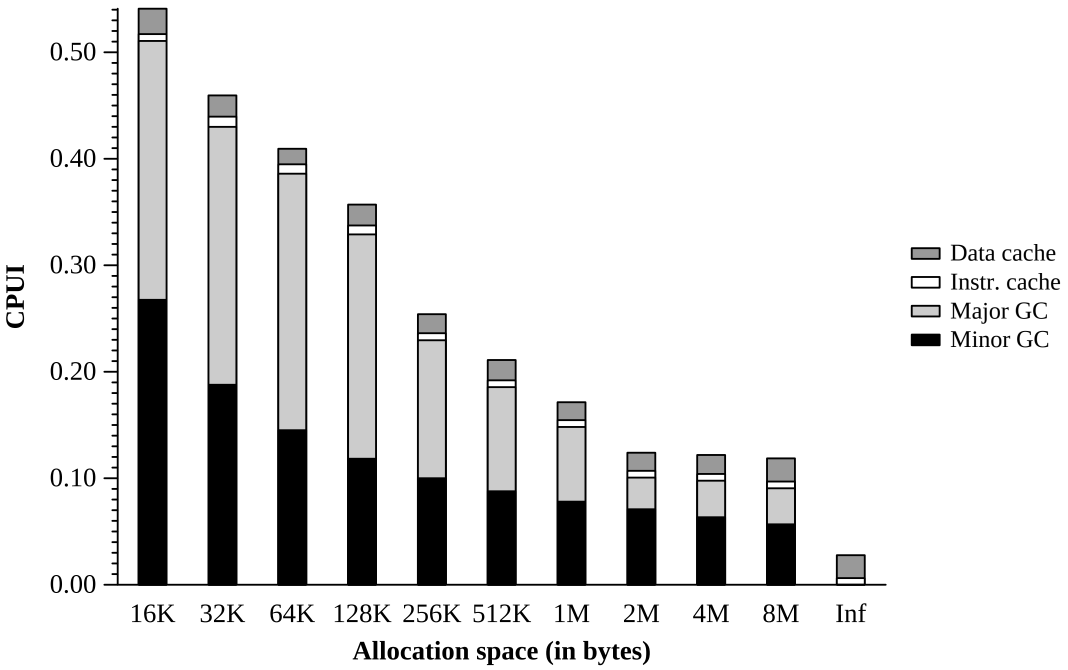


Figure 46: Average CPUI for 32-byte, 512K-byte, direct-mapped, write-validate data and instruction cache, with a 20 cycle read miss penalty.

Write-validate caches

The size of the allocation space does not have any significant effect on the miss ratios of small write-validate caches, and therefore the allocation space should be as large as possible, to reduce garbage collection overhead. For large caches, although there may be a large relative difference in miss ratios, the miss ratios are usually low and cache overhead should be small, unless the miss penalties are high, on the order of hundreds of cycles. Therefore, making the allocation space large should be the best policy for write-validate caches, or any caches for which there is no penalty for sequential allocation write misses.

Figure 46 shows the CPUI for 32-byte block, 512K-byte, direct-mapped write-validate data and instruction caches, with a 20-cycle read miss penalty. An “infinite” allocation space is clearly the best choice, but since it is not a practical one, the largest size possible should be the best choice. However, there is not much difference between the sizes of 2M, 4M, and 8M bytes, in part because of a small increase in data cache overhead that nearly matches the small decrease in garbage collection time.

In general, after a certain point, increasing the allocation space should result in only a

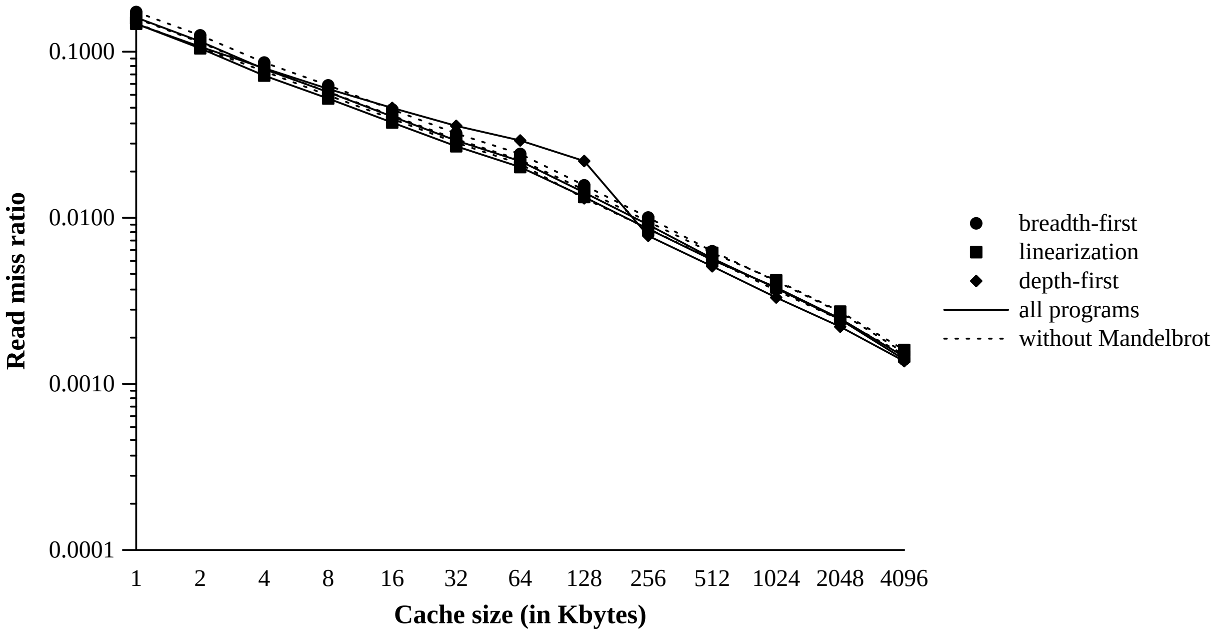


Figure 47: Read miss ratios for the standard breadth-first algorithm, depth-first and breadth-first with linearization, for 32 bytes block, direct-mapped caches and 1M-byte allocation space.

small decrease in garbage collection overhead, and therefore there is not much to gain by increasing the allocation space beyond a certain size, unless the allocation space gets close enough to “infinite”, which may also be impractical for most programs. The allocation space could be made as large as possible, limited only by the memory available on the machine, and by virtual memory performance. However, an allocation space of a few megabytes should give adequate performance for most programs.

6.2 Traversal Algorithms

Given that a cache block can contain many objects and the garbage collector copies objects to different locations, the order used to search and copy live objects (that determines which objects are stored near each other) can have an effect on the locality of reference, and thus on the cache performance of programs. Spatial locality may be improved if objects on the same block are likely to be referenced within a short time from each other. But since the garbage collector does not know how objects will be referenced in the future, this idea would only be effective if there is some heuristic that can predict the probable reference patterns

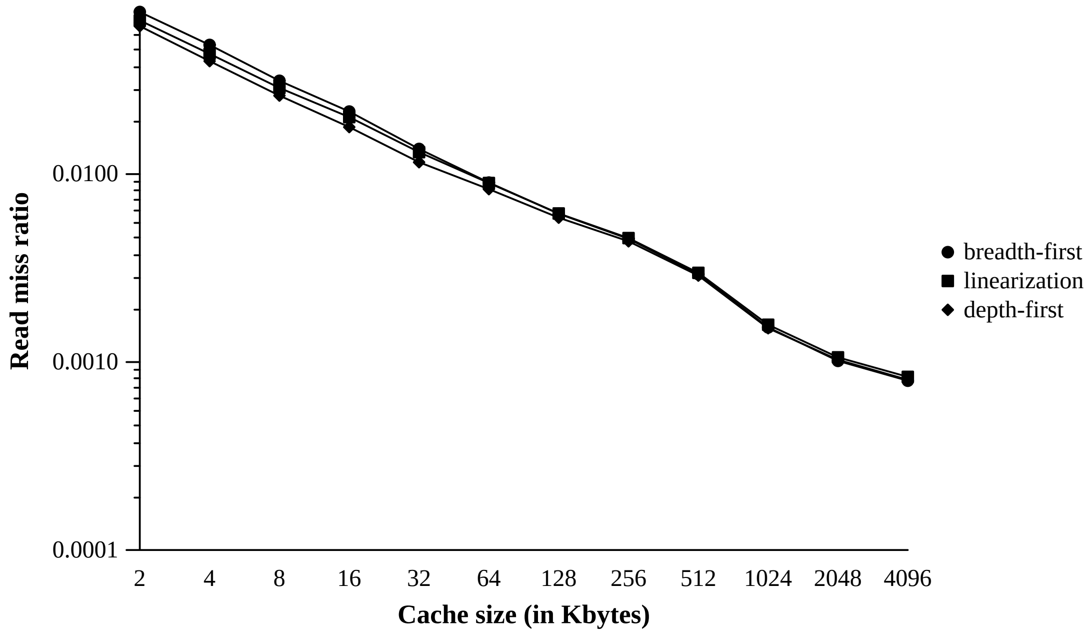


Figure 48: Miss ratios for the standard breadth-first algorithm, depth-first and breadth-first with linearization, for 32-byte block, 4-way associative caches and 1M-byte allocation space.

based on static properties of objects, such as the adjacency relations on the graph on live objects. For example, the distance (in links traversed) between objects might be used as a predictor of the time between references to those objects. Thus, two objects that are within a short distance from each other are more likely to be referenced within a short interval than two objects that are distant. The adjacency relations on the graph of live objects have often be used in an attempt to cluster objects in pages to improve virtual memory performance [20, 70, 82].

Most garbage collection algorithms use a particular order to traverse the graph of live objects, and this order determines the way objects are placed in to-space. The most common orders are depth-first and breadth-first [49], but many other variations are possible. Depth-first order tends to place adjacent objects close to each other in to-space, but breadth-first is more likely to put non-adjacent objects together.

Breadth-first traversal is the most commonly used copying collection algorithm because it can be efficiently implemented using Cheney's non-recursive algorithm [7]. This algorithm can be implemented without the use of any explicit auxiliary data-structures: it uses to-space as a queue for the breadth-first search, needing only two pointers to keep track of the head and tail of the queue. A depth-first search algorithm, on the other hand, requires an explicit stack to keep track of the path from one object to its descendents (but it is possible to use a pointer reversal scheme, and implement this stack without the requirement of any additional storage, even though there may be some additional overhead to reverse pointers [49]).

As an example of the differences between breadth-first and depth-first orders, consider a program that uses two long lists of integers, of about the same size, and suppose that a block can hold two list cells (the head of each cell is an integer, and the tail a pointer to another cell). A depth-first collector would put all the cells of each list in sequence, with two cells of the same list per block. The breadth-first traversal would put one cell from each list in a block. In this example, the depth-first collector would improve locality if the programs traverses one list first, and then the other, but the breadth-first traversal would be better if the program traverses the two lists simultaneously. In general, for a program with many different data structures, which may be used at different times, the breadth-first collector tends to put unrelated objects on a block, while the depth-first tends to put adjacent objects close on a block, and thus, depth-first order is usually expected to give better locality.

This section compares the miss ratios of the benchmark programs with three traversal algorithms: the standard breadth-first algorithm used by the SML/NJ compiler, a depth-first collector, and the original Cheney algorithm [19], which does a modified breadth-first traversal that does *list linearization,* that is, it puts adjacent cells of a list on contiguous memory locations. The depth-first collector always linearizes lists if the first word of a list cell is not a pointer, but it may arrange the objects in a different order if the first word is a pointer. The depth-first collector implements the stack using pointer reversal [49], and it does not use any extra storage. The list-linearization collector adds only a simple modification to the SML/NJ breadth-first collector: when an object is copied, if that object is a pair and its second field is a pointer to another pair that has not yet been copied, then that pair is copied immediately, and if its second field is a pointer, then the next object is also copied, and so on. The list-linearization collector does not guarantee that all lists are fully linearized (for example, if there is a pointer into the middle of a list, then the algorithm would copy the list as two linearized pieces, instead of a single one), but it should linearize most lists.

Figure 47 shows the average read miss ratios for the benchmark programs, on direct-mapped 32 byte block data cache, for the three traversal algorithms, the standard breadth-first collector, a depth-first collector, and breadth-first with list linearization. Surprisingly, the depth-first collector has higher miss ratios in many cases. This result from a problem with the program Mandelbrot, which has much higher miss ratios with depth-first. Probably the depth-first collector caused objects that are frequently accessed to be mapped to the same cache block, causing many conflict misses on 128K-byte or smaller caches. For most other programs, depth-first tends to have lower miss ratios, specially on small caches. List linearization has lower miss ratios than breadth-first, and only slightly higher than depth-first (except for Mandelbrot). The difference in miss ratios can be of up to 30% on the smallest cache, but it tends to get small as cache size increases.

Figure 48 shows the average read miss ratios on 32-byte-block, 4-way-associative data caches. Depth-first has lower miss ratios on small caches, with linearization ratios being slightly higher. But the difference between the miss ratios of the three algorithms becomes negligible as cache size increases.

The results presented above suggests that depth-first gives lower miss ratios (although is may give worse results sometimes, as happens with Mandelbrot), but the difference is small. Linearization gives only slightly higher miss ratios than depth-first, and since its implementation requires only a small change to the standard Cheney algorithm, it should be the best alternative.

6.3 Case Studies

This section presents an analysis of the performance of SML/NJ programs, with varying allocation space sizes, on two machines, with different cache architectures, the DEC5000/240 MIPS workstation, and the DEC3000/500 Alpha workstation.

6.3.1 DEC5000/240 (MIPS)

The DEC5000/240 [27] uses a 40MHz MIPS3000 processor, and has separate 64K-byte direct-mapped instruction and data caches. The block size is one word (4 bytes) but eight consecutive words, aligned to an eight-word boundary, are fetched on a cache read miss. The data cache is write-through, and there is an eight-entry write buffer, with 32 bits per entry, that should be able to handle most writes without stalls[29]. The write miss policy is write-allocate, and since the block size is only one word, there is no need to fetch any extra words (and no dirty blocks to write either). Therefore, this machine should have no penalty for write misses, and we would expect that a large allocation space gives best performance.

Table 17: Total run time (user, garbage collection, an system) on an DEC5000/240 MIPS workstation, running Ultrix version 4.3, for different sizes of the allocation space.

Program	Run time in seconds for allocation space size								
	16K	32K	64K	128K	256K	512K	1M	2M	4M
barnes-hut	32.39	28.74	29.27	28.00	27.60	27.48	27.50	28.20	27.20
boyer	3.56	3.13	3.86	3.56	2.86	2.18	2.00	2.12	1.89
knuth-bendix	13.84	12.04	12.18	11.94	11.32	12.64	12.10	10.84	10.91
lexgen	12.84	12.94	11.76	11.77	12.46	10.82	10.95	10.58	10.47
life	12.53	12.04	11.83	11.69	11.72	11.66	11.83	11.96	12.05
mandelbrot	11.89	11.51	11.39	11.23	11.20	11.20	11.16	11.15	11.15
mlyacc	5.76	5.15	4.81	4.63	4.55	4.39	4.24	3.92	3.90
ray	25.08	24.38	22.90	22.80	22.57	22.68	22.35	23.38	22.53
simple	47.59	50.17	42.60	43.38	38.49	37.72	32.03	28.18	27.26
vliw	25.31	24.03	23.18	22.14	21.56	21.51	21.41	21.48	21.89

Table 17 shows the total run time of the benchmark programs on a DEC5000/240, for varying allocation space sizes. Because of virtual memory limitations, most programs can not run with allocation spaces much larger than 4M bytes. As expected, the best performance is given by large allocation space sizes. For the programs with highest garbage collection overhead, the best performance is given by the 4M bytes allocation space. For the other programs the difference in time is small, and best performance is given by a large, but not necessarily the largest allocation space.

6.3.2 DEC3000/500 (Alpha)

The DEC3000/500 [31, 30] uses an Alpha 21064 processor, that has dual on-chip 8K-byte, 32-byte-block instruction and data caches, and on-chip support for a second level cache [26]. Cache misses are non-blocking, but there may be only one read miss pending at any time. The data cache is write-around, and there is a four entry write buffer, with 32 bytes (one block) per entry, that groups writes to the same block in a single entry.

The DEC3000/500 has a 512K-byte, 32-byte block, fetch-on-write, unified second level cache, called the Bcache. Although the Bcache is fetch-on-write, if a full block is to be overwritten, which can happen if an entry in the write buffer contains a full block (that is, all words in the entry have been written), then there is no fetch from memory.

Because the first level cache is write-around and the cache is too small to fit the allocation space in, the SML/NJ programs should have many misses to the first level cache. Based on the results of Chapter 5, we can estimate the miss ratio to be close to 1/8, since there are eight words per block (although the Alpha is a 64-bit machine, SML/NJ uses only 32-bit

Table 18: Total run time (user, garbage collection, an system) on an DEC3000/500 Alpha work-station, running OSF-1 version 3.0, for different sizes of the allocation space. The version of the compiler used is 1.05a.

Program	Run time in seconds for allocation space size										
	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M
Barnes-Hut	19.78	17.42	16.52	15.73	15.28	15.31	18.12	18.06	18.14	18.14	18.14
Boyer	1.11	0.97	0.88	0.83	0.81	0.78	0.90	0.89	0.88	0.89	1.00
Knuth-Bendix	4.68	4.02	3.73	3.58	3.50	3.53	4.36	4.44	4.55	4.60	4.75
Lexgen	4.43	4.04	3.79	3.72	3.67	3.64	3.91	3.84	3.87	3.96	4.21
Life	0.59	0.58	0.55	0.55	0.53	0.53	0.57	0.58	0.60	0.64	0.63
Mandelbrot	6.12	5.79	5.44	5.28	5.16	5.11	7.10	7.11	7.11	7.16	7.28
MLYACC	2.31	2.01	1.87	1.77	1.69	1.65	1.76	1.71	1.73	1.76	1.84
Ray	36.68	34.02	32.84	32.21	32.19	32.31	36.00	36.36	36.69	35.99	36.13
Simple	9.47	8.29	7.67	7.28	7.15	7.21	8.79	8.73	8.74	8.76	8.92
VLIW	7.83	6.96	6.46	6.28	6.33	6.27	6.93	6.87	6.87	6.90	7.04

loads and stores).

Although the Bcache is fetch-on-write, we may expect that, because writes are non-blocking and because of the merge operation of the write buffer, the SML/NJ programs should not have to pay a miss penalty on allocation writes. Since most writes are sequential, and the allocation rate is high, it is reasonable to expect that most write buffer entries will be full when they are flushed, and thus only rarely should a block be fetched from memory because of allocation write misses.

Table 18 gives the run time of the benchmark programs on a DEC3000/500, with different allocation space sizes. Unlike the other experiments, version 1.05 of the compiler was used here, because version 0.93 was not ported to the Alpha architecture (version 1.05 has some differences from version 0.93, in particular it does fewer closure allocations [64]).

The results show that fitting the allocation space in the 512K-byte Bcache gives the best performance. This is surprising because the results presented on Section 6.1 suggests that we should fit the allocation space on a cache only if there is a penalty for allocation misses. But because of non-blocking writes and the write buffer, we do not expect to have an allocation miss penalty on the Alpha. Nevertheless, there must be some fetches from main memory that are causing the machine to stall when the allocation space does not fit in the Bcache.

The following conjecture can explain the source of these fetches. As Section 4.3 shows, most reads are to recently written data. Because the first level data cache is write-around, many of these reads will be misses to the first level cache (the first read to a recently written block should be a miss), and because the miss policy for this cache is to allocate and fetch a block on read misses, the block that contains the missing word must be fetched — either from the Bcache, or from memory if the reference misses the Bcache[1]. If the allocation space does not fit in the Bcache, and the read is to a recently written word that is still in the write buffer, then this should be also a miss to the Bcache. There are two possibilities: the write buffer entry containing this word is full, or it is only partially filled. If the write buffer entry is full, then the entry can be flushed to the Bcache without causing a fetch from memory, and then the block is fetched into the first-level data cache. The problem is when the write buffer entry is only partially filled. In this case, the whole block must be fetched from memory, incurring a main-memory access penalty, to obtain the remaining words.

Using the data from Figure 7, in Chapter 4, we can estimate the probability of a write-buffer entry being flushed because of a read to a partially filled block, which can give an idea of the number of reads to recently written words that cause a block to be fetched from memory. Given the the fraction of reads (r) and writes (w) of a program, and the cumulative probability of reads to a word i words or less from the allocation pointer ($D[i]$), which can be obtained from Figure 7, the probability of a partially filled block being flushed from the write buffer, for an 8-word block, is $1 - S(8)$, where $S(n)$ is the probability of a block surviving to n writes without being flushed.

$$S(1) = 1.0$$

$$S(n) = S(n-1) \cdot \frac{w}{w + r \cdot D(n-1)}$$

To survive to n writes without being flushed, a block must survive to $n-1$ writes, and there should be not be a read to this block (that is, to a distance $n-1$ or less from the allocation pointer) after $n-1$ writes.

Table 19 shows the probabilities of a partially filled block being flushed from the write buffer for each program. On average 75% of the blocks allocated are probably flushed from the write buffer before the full block is allocated. This agrees with the observed data shown on Table 18. The write buffer is being flushed with a partially filled block most of the times, and thus there should be many fetches from memory, which explains why fitting the allocation space in the cache gives best performance.

In summary, although the Alpha machine has non-blocking misses, and a deep write-buffer that merges writes to the same block, because there are many reads to recently

[1]It is not clear from the Alpha manuals if the block can be fetched directly from the write buffer, but that does not matter in the following argument.

Table 19: Probability of a partially filled block being flushed because of a read.

Program	
Barnes-Hut	0.79
Boyer	0.70
Knuth-Bendix	0.63
Lexgen	0.80
Life	0.76
Mandelbrot	0.82
MLYACC	0.72
Ray	0.77
Simple	0.78
VLIW	0.72
Arith. mean	0.75

written data, this is not sufficient to avoid paying a penalty for allocation. Therefore, it is better to fit the allocation space in the second level cache, to avoid costly fetches from memory. One solution to this problem, other than having a write-allocate first level cache, would be to satisfy read misses from the write buffer when the word is present in the buffer. This would probably add some complexity to the load unit and write buffer, but would certainly decrease the number of fetches from memory.

Pre-fetching

Even if the allocation space fits in the second level cache, the Alpha machines should still have bad cache performance, due to the 8K-byte write-around first level data cache, which is too small for fitting the allocation space in. Since read misses are non-blocking it should be possible to use pre-fetch to simulate write-allocate. Appel [9] describes a technique to improve performance of SML/NJ programs on machines with write-around caches and non-blocking read misses, which consists of reading (and missing) a block several cycles in advance of writing to the same block. The value of the read is discarded, and thus there should be no pipeline stall, unless the limit on the number of pending misses is small. On an DEC3000/400, with an Alpha 21064 CPU, this scheme can improve performance by a factor of 1.2. But the 21064 can only handle one outstanding miss at at time, and therefore the pre-fetching instruction could cause a pipeline stall. This scheme should be even more effective on machines that can handle many outstanding misses simultaneously, such as the Alpha 21164, and it should be able to eliminate nearly all allocation misses.

6.4 Conclusion

Fitting the allocation space in the cache can be an effective way of improving cache performance of SML/NJ program on some architectures for which there is a penalty for write misses, in particular those with large caches or high miss penalties. For small caches with a penalty for write misses, even though the programs should have poor cache performance, fitting the allocation space in the cache is not a good alternative because the extra garbage collection overhead offsets the lower cache overhead, and therefore a large allocation space should give better performance. On caches that do not have a write miss penalty, the allocation space should be large, to reduce reduce garbage collection overhead.

Some traversal algorithms, like depth-first and list-linearization, may reduce miss ratios slightly, but this should not have a great effect on overall performance.

Chapter 7

Conclusion

This dissertation has presented a study of cache performance of programs with intensive heap allocation and generational garbage collection.

The study of the memory reference patterns in Chapter 4 shows some properties that are shared by all of the programs studied.

- Nearly all writes are for allocation;
- Most reads are to recently written words.
- Most objects have a short life.
- Most data objects are small.

Chapter 5 studied the performance of the programs when many cache design parameters are varied. Confirming the results of previous studies, the experiments presented showed that the write-miss policy can have a dramatic effect on the cache performance of these programs, with a write-validate policy being giving the best performance. Other cache design parameters, cache size, block size, associativity, and split/unified caches, were also studied.

Chapter 6 presented a study on the interaction of garbage-collection policies and cache performance. It was shown that the size of the allocation space can have a significant effect on the cache performance of fetch-on-write and write-around caches. Other garbage collection design parameters studied, frequency of major collections, and traversal algorithm, can also have an effect on cache performance, but this effect is usually small.

7.1 Related Work

Since garbage-collected functional languages appeared in the 1960's, there have been many studies on the locality of references of programs written in these languages. Earlier studies in the 70's and earlier 80's, such as Clark[20], Stamos[70], and Courts[22], concentrated on virtual memory performance, but because of the many differences between virtual memory and caches, many conclusions of these studies cannot be generalized to cache performance. These earlier studies also used simple mark-and-sweep and copying-collector schemes, and usually studied Lisp programs with stack-allocated activation records.

In recent years, there have been many studies of the cache performance of garbage-collected functional programs. Most of these studies have concentrated on Lisp or Scheme programs with stack-allocated activation records, but there have also been studies on the performance of SML/NJ programs and other programs with more intensive heap allocation.

Peng and Sohi [57] was one of the first studies to address the problem of cache performance of programs with dynamic allocation. They studied a set of Lisp programs, with stack-allocated activation records, and characterized the memory reference patterns of those programs. They identified the problem of allocation misses, and that those misses depend on block size. Their study uses only small caches, from 512 to 64K bytes, and vary block size and associativity. Based on their analyses of the memory references of their benchmark programs, they proposed three new cache architecture features:

1. a cache allocate instruction, that allocates a word directly on cache, without fetching the block from memory;
2. a biased-LRU replacement policy, that choses an arbitrary garbage block for replacement. This requires the presence of a garbage bit in the block, that is set by the garbage collector.
3. an optimized write-back scheme where garbage blocks are not written back to memory.

The last two optimizations depend on the availability of a garbage bit and the ability of the collector to detect garbage soon enough, which requires either frequent collections, or a reference-counting collector that can detect garbage immediately. They do not, however, analyze the trade-offs between garbage collection and cache performance.

Zorn [85, 86] studied the garbage collection performance of a benchmark suite of Lisp programs with stack-allocated activation records. His study addresses many garbage collection design issues, including locality of reference of virtual memory and caches. He compares the performance of mark-and-sweep and copy collectors, varying the frequency of collections. He claims that a mark-and-sweep collector can have better locality than a copying collector, and his major conclusion is that garbage collection can be used to improve performance, but he only considers fetch-on-write caches, and does not quantify the tradeoffs between cache and garbage collection overhead.

Wilson, Lam, and Moher [83] studied the performance of Scheme programs with stack-allocated closures. They considered direct-mapped and set-associative caches, and they only analyze miss ratio, without distinguishing between read and write misses. Their main conclusion is that programs with heap allocation and generational collection have poor locality, and that locality should be even worse on programs with higher allocation rate. They suggest that fitting the allocation in the cache could improve performance.

Koopman, Lee, and Siewiorek [46] studied the cache performance of combinator graph reduction, a technique to implement lazy functional languages, that needs intensive heap allocation and garbage collection. They varied many cache parameters, cache size, block size, associativity, and write miss policy. They conclude that good cache performance for their programs depends essentially on the availability of write-validate-like caches. They claim that in the absence of these features in hardware, the use of a software prefetch scheme (a memory read that discards its results) can improve performance by 20%.

Diwan, Tarditi, and Moss [29, 28] studied the cache performance of SML/NJ programs using a simple two-generational collector. They consider the effects of many cache parameters, varying cache size, block size, associativity, and write-miss policy. They show that these programs can have good performance on machines with write-validate-like caches, but there is a high cache overhead on unfavorable cache architectures.

Reinhold [60] studies the cache performance of Scheme programs with stack-allocated closures. He considers only direct-mapped write-validate caches, and the main conclusion of his research is that programs with heap allocation "are naturally well-suited to the direct-mapped caches typically found on high-performance computer systems", precisely because they tend to spread objects over a large area, thus minimizing conflicts. He argued that cache overhead is low and that any attempt to use garbage collection strategies to improve cache performance should be unsuccessful.

Stefanovic and Moss [72, 73] studied object behavior of an SML/NJ benchmark that contains many programs that are part of the benchmark used in this study. Their study is oriented towards garbage collection performance, and they make a detailed analysis of the object lifetimes.

Tarditi and Diwan [75] studied the cost of memory management for SML/NJ programs, on a machine with a write-validate-like cache, using a simple two-generational collector. They do not vary any cache or garbage collection parameters, but they break down the various costs of garbage collection and memory system (including cache). They conclude that on machines with this kind of cache organization, cache overhead is low, and there are not many opportunities to improve cache performance. They also identify the various costs involved in the garbage collector (like tracing, copying, tagging, etc.).

This work differ from previous studies in many ways. First, this is the first study to quantify the trade-offs between cache and garbage collection overhead. All of the previous studies that suggested the use of frequent collections to improve cache performance, considered only cache miss ratios, and completely ignored garbage collection overhead.

This is also the first work to make a detailed analysis of the memory reference patterns of programs with heap-allocated activation records, with a separate analysis of write and read references. Diwan, Tarditi, and Moss suggested (based on the fact that write-around and fetch-on-write caches have nearly the same number of cache misses), that SML/NJ programs tend to make frequent reference to recently allocated objects, a fact that is confirmed here by direct measurements.

This is also the first work to use a multi-generational collector, and to consider the effects and cost of varying multiple garbage collection parameters for programs with heap-allocated closures.

7.2 Implications

This study is based on the analyses of the memory reference patterns, cache, and garbage collection performance of a set of ten ML programs compiled with the SML/NJ compiler. A natural question is how the results presented here can be extended not only to other ML programs, but to other programs with intensive heap allocation and generational collection. The results presented in Chapters 5 and 6 should apply to most programs that have memory reference patterns similar to the programs in the benchmark suite (mostly allocation writes and many reads to recently written data). Many results should also apply to programs with stack-allocated activation records, in fact, most results are in agreement with studies of programs with heap allocation and stack-allocated activation records.

Consider ML programs with heap-allocated closures (heap-only programs), and the same programs with stack-allocated closures (stack programs). The only significant difference in performance would be for small fetch-on-write and write-around caches, where the allocation space does not fit in the cache. The stack programs should have lower cache overhead, because few stack allocations should be misses, but because they have heap allocation for user data structures, they should still benefit from write-allocate and no-fetch-on-write write miss policies. The cache overhead should be similar in both cases on write-validate caches or if the allocation space fits in the cache. There should also be some difference in garbage collection overhead, since the heap-only programs do more frequent garbage collections because of the higher allocation rate, but this difference should only be significant if the allocation space is small, otherwise garbage collection overhead should be low in both cases.

This study used version 0.93 of the SML/NJ compiler. Newer versions of the SML/NJ compiler uses a new closure representation method [64, 63], that significantly reduces closure allocation (by 40%). The results presented here should be still valid if the new closure representations are used. The lower allocation rate should decrease cache overhead on fetch-on-write and write-around caches, and decrease garbage collection overhead slightly, but should not have a significant impact on the results presented for write-validate caches.

Many of the results should also be valid if different garbage collection techniques are used. Many of the results presented in Chapter 5 agree with the results of previous studies that used different collectors, such as simple two-generational collectors or even mark-and-sweep collectors. The major difference would be related to fitting the allocation space in fetch-on-write or write-around caches, since the garbage collection overhead may change significantly if the size of the allocation space is changed.

The results of this study show that heap allocation, even intensive heap allocation of closures, can have good cache performance, as long as the architecture uses the right write-allocation policy. We hope that most future architectures would use the write miss policy that is needed for these programs to have good cache performance, that is, write-allocate and no-fetch-on-write. But even if this is not the case, there are some alternatives that can be applied, such as fitting the allocation space on large, or even mid-sized caches, or the use of special cache control instructions that are provided by some architectures.

7.3 Recommendations for Cache Designers

Write-validate caches are without any doubt the best organization for programs with heap allocation. Similar results can be achieved by an organization with a one-word block size, and a large fetch size on read misses. These organizations has the key features that are needed to provide these programs with efficient cache performance.

- All writes are allocated in the cache, which is important because most reads are to recently written data.
- There are no fetches from memory on write misses, which is important because nearly all objects are written only once, and no locations are read before being initialized.

Cache organizations that attempt to improve performance on write misses by writing the word to a write buffer, bypassing the cache, should be ineffective for programs with heap allocation, giving only a marginal improvement over a basic fetch-and-write scheme that treats read and write misses uniformly.

Other cache design parameters, such as block size, and associativity, are important, but should not have as large an effect on performance as the write miss policy has. Block size becomes more important if an unfavorable write-miss policy is used, in which case large blocks should be preferred. Usually a block size of 32 or 64 bytes, and direct-mapped or low associativity (2 or 4-way) should give best or at least close to best performance.

Cache size can also have a large effect on performance. If an unfavorable write-miss policy is used, then large caches (that can keep a moderately large allocation space) should be preferred.

As long as the right write-miss policy is used, the general guideliness for cache design given by Przybylski[58, Section 6.2.2] should provide a good basis for the design of good caches for programs with dynamic heap allocation.

If the cache does not have write-validate-like features, then at least some alternative form of simulating these features should be provided, such as cache allocate instructions or non-blocking prefetch instructions.

7.4 Recommendations for Garbage-collector Designers

On machines with write-validate caches (or with prefetch or cache control instructions that permit the simulation of write-allocate and no-fetch-on-write), the size of the allocation space should be as large as necessary to reduce garbage collection overhead, but small enough to avoid bad virtual memory performance. Garbage collection overhead and virtual memory performance should be the major concerns when tunning garbage-collection parameters on write-validate caches. The size of the allocation space does have an impact on miss ratios of large caches, but since miss ratios on these caches tend to be low, this would only have a significant impact on performance if the miss ratios are high, on the order of hundreds on cycles. Usually, an allocation space of a few megabytes should be large enough to reduce garbage collection overhead to a small level.

On machines with large (256K bytes or more) fetch-on-write or write-around caches, the allocation space should fit in the cache to eliminate allocation misses and improve cache performance, since for these cache sizes, garbage collection overhead should be lower than the cache overhead of the allocation misses. Machines with small (less than 32K bytes) fetch-on-write or write-around caches should have poor cache performance, but since fitting the allocation space in the cache should not be of any help in this case because garbage collection overhead would be too high, the best strategy should be to make the allocation space large. For 32 to 128K-byte caches, the decision to fit the allocation space on cache depends on the program. Only programs with low garbage collection overhead should have better performance if the allocation space fits in the cache, and even in this case, the performance improvement should be small. On multi-level cache hierarchies the allocation space should fit on the (usually large) second level cache.

The size of the older generations should not depend on the size of the allocation space (although the size of the allocation space may impose a lower limit to the size of the first generation), to allow for independent control of minor and major collection overhead. When the allocation space has to fit in the cache to improve cache performance, minor collections are more frequent and have higher overhead, but this should not cause a large increase in major collection overhead. If possible, the first generation should be made larger to reduce major collection overhead and compensate for the extra minor collection overhead.

The frequency of major collections can have an effect on cache performance of large caches, but since the miss ratios are usually low, this effect is significant only if miss penalties are high. This effect depends on the lifetime of older objects. If many objects that survive to a minor collection die after some time, then the program may benefit from some occasional collections. If many objects remain live through the entire run time of the program, then the number of major collections should be kept to the minimum possible.

If there any many active heap areas (generations or arenas within each generation) the location of these areas in memory can be important to avoid conflict misses, specially on large direct-mapped or low-associativity caches. Ideally, the generations should be placed randomly in memory, to avoid many objects mapping to only a few locations.

The traversal algorithm can have an effect on cache performance, but this effect should be usually small. Any collection algorithm should give a good performance on large caches. Depth-first and linearization may give a small performance improvement on small caches.

7.5 Future Work

The results presented in this work suggest that the distribution of older objects on the older generations can have a large effect on the miss ratios of large caches. If the gap between processor and memory speed continues to increase, then it is likely that references to older objects may have a large effect on performance, and major collection should become increasingly important, as it can rearrange objects to reduce conflicts or improve spatial locality.

Further analysis is needed to better understand the effects of major collection on cache performance. The programs studied here are relatively small, and most of them can easily be run without any major collections, needing only a few tens of megabytes of memory. In order to better understand the effects of major collections on performance, and to tune major collection parameters (like the number of generations and frequency of major collections) it is necessary to study long running programs, much longer than the programs studied in this research. Studying these large programs may require more efficient simulation techniques than the simple simulator used in this study, such as more efficient simulators or trace sampling techniques[52].

Bibliography

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 4(6):393–431, November 1988.

[2] Donald Alpert and Dror Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[3] Andrew Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton Univesity, Department of Computer Science, 1994.

[4] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, (2):153–162, 1989.

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

[6] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation: An International Journal*, 3:343–380, 1990.

[7] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*, pages 89–100. MIT Press, Cambridge, MA, 1991.

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

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

[10] Andrew W. Appel. Loop headers in lambda-calculus or CPS. *Lisp and Symbolic Computation*, (7):337–343, 1994.

[11] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming (LNCS 528)*, pages 1–13. Springer-Verlag, New York, August 1991.

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

[13] H. B. Bakoglu and T. Whiteside. RISC system/6000 hardware overview. In *IBM Risc System 6000 Technology*, pages 8–15. IBM Corporation, 1990.

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

[15] Joshua Barnes and Piet Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 446(324), December 1986.

[16] David A. Barret and Benjamin G. Zorn. Garbage collection using a dynamic threatening barrier. Technical Report CU-CS-659-93, University of Colorado ar Boulder, July 1993.

[17] M. C. Becker, M. S. Allen, C. R. Moore, J. S. Muhich, and D. P. Tuttle. The PowerPC 601 microprocessor. *IEEE Micro*, 13(5):54–68, October 1993.

[18] Brad Burgess, Nasr Ullah, Peter van Overen, and Deene ogden. The PowerPC 603 microprocessor. *Communications of the ACM*, 37(6):34–42, June 1994.

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

[20] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[21] Standards Performance Evaluation Corp. SPEC benchmark suite release 1.0, October 1989.

[22] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

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

[24] Eric DeLano, Will Walker, Jeff Yetter, and Mark Forsyth. A high speed superscalar PA-RISC processor. In *Proceedings of COMPCON*, pages 116–121, San Francisco, California, February 1992. IEEE Computer Society Press.

[25] Keith Diefendorff, Rich Dehler, and Ron Hochsprung. Evolution of the PowerPC architecture. *IEEE Micro*, 14(2):34–49, April 1994.

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

[27] Digital Equipment Corporation, Palo Alto, California. *DECstation and DECsystem 5000 Model 240 Technical Overview*, version 2 edition, February 1992. Order number EC-N0194-51.

[28] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Technical Report 93-227, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1993.

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

[30] Todd A. Dutton. The design of the DEC3000 model 500 AXP workstation. In *Proceedings of COMPCON*, pages 449–455, San Francisco, California, February 1993. IEEE Computer Society Press.

[31] Todd A. Dutton, Daniel Eiref, Hugh R. Kurth, James J. Reisert, and Robin L. Stewart. The design of the DEC3000 AXP systems, two high-performance workstations. *Digital Technical Journal*, 4(4), 1992.

[32] John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[33] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA,

July 1987. Simultaneously published as IBM/T.J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[34] Jean-Marc Frailong et al. The next generation SPARC multiprocessing system architecture. In *Proceedings of COMPCON*, pages 475–480, San Francisco, California, February 1993. IEEE Computer Society Press.

[35] Tom Asprey et el. Performance features of the PA7100 microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[36] Philip J. Flemming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

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

[38] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[39] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture.* ACM Press, June 1995.

[40] Tom R. Halfhill. Intel's P6. *Byte*, 20(4):42–58, April 1995.

[41] David R. Hanson. A portable storage management system for the Icon programming language. *Software—Practice and Experience*, 10:489–500, 1980.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

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

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

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

[46] Philip J. Koopman Jr., Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, april 1992.

[47] Willian R. Hardell Jr., Dwain A. Hicks, Lawrence C. Howell Jr., Warren E. Maule, Robert Montoye, and David P. Tutle. Data cache and storage control units. In *IBM Risc System 6000 Technology*, pages 44–50. IBM Corporation, 1990.

[48] Ed Keane and Pat McGuire. HP9000/735 second generation PA-RISC Snakes workstation. In *Proceedings of COMPCON*, pages 426–432, San Francisco, California, February 1993. IEEE Computer Society Press.

[49] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[50] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, New Haven, CT, 1987.

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

[52] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, April 1995.

[53] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[54] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. 5rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (Sigplan Notices v. 27 n. 9)*, pages 62–73. ACM Press, 1992.

[55] John Novitsky, Mani Azimi, and Raheel Ghaznavi. Optimizing systems performance based on Pentium processors. In *Proceedings of COMPCON*, pages 63–72, San Francisco, California, February 1993. IEEE Computer Society Press.

[56] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

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

[58] Steven Przybylski. *Cache and Memory Hierarchy Design: a Performance Directed Approach*. Morgan Kaufmann, 1990.

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

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

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

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

[63] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, Department of Computer Science, 1994. Technical Report CS-TR-475-94.

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

[65] Zhong Shao and Andrew Appel. A type-based compiler for standard ML. In *Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation*, June 1995.

[66] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[67] Alan J. Smith. Line (block) size choice for CPU cache memories. *IEEE Transactions on Computers*, 36(9):1063–1075, September 1987.

[68] Gurindar Sohi and Manoj Franklin. High-bandwith data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, 1991.

[69] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 risc microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[70] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, 1984.

[71] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[72] Darko Stefanovic. Generational copying garbage collection for Standard ML: a quantitative study. Master's thesis, Object Systems Laboratory, Department of Computer Science, University of Massachusetts, Amherst, 1993.

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

[74] Harold S. Stone. *High Performance Computer Architecture.* Addison-Wesley, third edition, 1993.

[75] David Tarditi and Amer Diwan. Measuring the cost of memory management. Submitted to the 1994 Conference on Lisp and Functional Programming, 1994.

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

[77] Andrew P. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceeding of the 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11. ACM Press, 1994.

[78] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the 1988 Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 1–17. ACM Press, November 1988.

[79] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[80] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[81] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing Surveys (available for annonymous FTP from ftp.cs.utexas.edu in the file `/pub/garbage/bigsurv.ps`), 1994.

[82] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, Canada, June 1991.

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

[84] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *Proceedings of the 1988 Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 23–35. ACM Press, November 1989.

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

[86] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of th 1990 ACM Conference on Lisp and functional programming*, June 1990.

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