Abstract

SAWYER, RICHARD K. Page Pinning Improves Performance of Generational Garbage Collection. (Under the direction of Dr. Edward F. Gehringer).

Garbage collection became widely used with the growing popularity of the Java programming language. For garbage-collected programs, memory latency is an important performance factor. Thus, a reduction in the cache miss rate will boost performance. In most programs, the majority of references are to newly allocated objects (the nursery). This work evaluates a page-mapping strategy that pins the nursery in a portion of the L2 cache. Pinning maps nursery pages in a way that prevents conflict misses for them, but increases the number of conflict misses for other objects. Cache performance is measured by the miss-rate improvement and speedup obtained by pinning on the SPECjvm98 and the DaCapo benchmarks.

Pinning is shown to produce a lower global miss rate than competing virtual-memory mapping strategies, such as page coloring and bin hopping. This improvement in miss rate shortens overall execution time for practically every benchmark and every configuration. Pinning greatly reduces average pause time and variability of pause times for nursery collections.

Page Pinning Improves Performance of Generational Garbage Collection

by

Richard K. Sawyer

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2006

Approved By:

_____	_____
Dr. Suleyman Sair	Dr. Gregory T. Byrd

_____
Dr. Edward F. Gehringer
Chair of Advisory Committee

Dedication

To my wife, my parents, and my sister

Biography

Richard Sawyer was born on April 24th 1979 in Fontana, California. In 2001, he graduated from North Carolina State University with Bachelor of Science degrees in Computer and Electrical Engineering with a minor in mathematics. Currently, he lives happily with his wife and their pets in Durham, NC. With the completion of this thesis, he is receiving his master's in computer engineering.

Acknowledgments

I thank Dr. Gehringer for being my advisor, and I also thank the rest of my advisory committee, Dr. Byrd and Dr. Sair.

Thanks to Dr. Mueller for allowing me to use the OS cluster to run my simulations. Thanks to Eliot Moss and Chris Hoffmann for their help in creating new system calls in DSS. Thanks to Eric Sills for his weekend help with the NC State HPC grid. Many thanks to Balaji Iyengar for helping cross-compile Jikes and helping in the setup of my simulation environment.

I thank my loving wife, Keegan, for supporting me during this project. I could not have done it without her. And thanks to all my family and friends for their love and encouragement along the way. A special thanks to my friends, Kera and Maurice, for all the times we studied together, and to Steven for encouraging me to finish this semester.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Memory management has long been a challenge for programmers. Programming languages like FORTRAN, C, and C++ require the programmer to allocate and de-allocate memory. Garbage collection was introduced in Lisp (1959), in order to automate reclamation of memory that was no longer in use. But it was not until Java exploded on the programming scene in 1995 that garbage collection hit the programming mainstream.

In a garbage-collected program, program execution is broken into two phases, the mutator phase and the garbage collection phase. In the mutator phase, the program is modifying the data to accomplish the overall program goal. The garbage-collection phase accomplishes the task specifically related to automatic memory management: to free memory that is no longer in use.

Although garbage collection simplifies memory management, there are overheads. One overhead is pause time, which occurs when the mutator phase is halted for garbage collection. This takes time away from the meaningful work of the program, increases the

program's instruction count, and potentially causes a program to miss a real-time deadline. Real-time programs typically do not use garbage collection because of this pause time.

Pause time can be tackled by reducing cache misses. When a cache miss occurs the processor is stalled to wait for main memory. The amount of time the processor waits to access main memory is known as memory latency. Given the growing gap between processor speed and memory speed, many strategies have been developed to hide memory latency and improve cache performance.

Two frequently proposed strategies, page coloring and bin hopping, deal with memory latency by modifying the virtual page mapping. Page coloring takes advantage of spatial locality of virtual addresses by mapping consecutive virtual pages to consecutive physical page frames, so that they do not conflict in the cache. Bin hopping takes advantage of temporal locality by mapping the most recently mapped virtual pages to consecutive physical page frames, so that they do not conflict in the cache.

A new page-mapping strategy to improve cache performance is *pinning*. Pinning can be used in generational garbage collectors where new objects are allocated to the *nursery* space (also called the youngest generation) in the heap. Generational-copying garbage collectors do more frequent collections on the small nursery space and a full heap collection only when necessary. Pinning maps nursery pages in a way that prevents conflict misses in the L2 cache, but increases the number of conflict misses for other objects. With pinning, all accesses to nursery objects hit in the L2 cache, except for misses due to replacements that occur because of context switches (cold misses can be avoided in caches that have the ability to allocate memory in cache, without fetching from main memory). The cost of such a

strategy is low; pinning only requires a software change to the virtual-page mapping functions of the operating system.

The goal of this thesis is to evaluate whether pinning boosts cache performance for generational garbage-collected programs. These experiments study the miss-rate improvement and speedup obtained by pinning on a set of Java programs represented by the SPECjvm98 and the DaCapo benchmarks.

Chapter 2

Background

This chapter introduces concepts and terminology used in this research. Memory allocation is discussed in section 2.1. Section 2.2 discusses basic garbage collection terminology. Section 2.3 discusses virtual page mapping and explains the concepts of page coloring and bin hopping.

2.1 Memory Allocation

Memory can be allocated according to three strategies: static, stack, and heap. Static memory allocation occurs at compile time; this memory remains accessible from the beginning to the end of program execution. Static memory contains such items as global variables and constants. The other two kinds of memory allocation are performed during run time. Stack memory typically contains local variables, such as function parameters and return values that are allocated on a procedure call and deallocated on a return. The third type of memory allocation, the one studied here, is the heap. The heap consists of dynamically allocated objects, whose size and number is unknown at compile time.

2.2 Garbage Collectors

This section describes the basics of garbage collection. There are three classic garbage-collection strategies, as classified by Jones and Lins [12]; reference counting, mark-sweep, and copying.

2.2.1 Reference Counting

Reference counting tracks the number of pointers referencing a heap object. For example, in a doubly-linked list an item in the middle of the list would typically have a reference count of two. When an object is allocated on the heap, the reference count is set to one. The reference count is incremented and decremented as a program is executed, so that garbage collection is distributed throughout the mutator phase. If the reference count is zero, the object is free in the heap. An advantage of this strategy is that an object can be immediately reclaimed when it becomes garbage. In other words, the heap space used by that object can be re-used without delay once the reference count reaches zero. The disadvantage is that every time a pointer is created or overwritten the reference count must be updated, increasing the overhead of every pointer manipulation. Pause time is dispersed throughout the mutator phase, but cyclic data requires special attention.

2.2.2 Mark-sweep

Mark-sweep garbage collection is split into two phases called ☺ “mark” and “sweep”. A bit is associated with each object to indicate that the object is alive. During the mark phase, all objects reachable from the roots (live data) are marked. Once an object is marked, it is not traversed again. After all live data is marked, the sweep phase begins. During the sweep phase, all unmarked objects are placed back on the free list. Allocation of objects

continues until no more space is available on the heap. This garbage-collection strategy handles cyclic data without special code. The cost of this strategy is a long pause time every time heap space is exhausted.

2.2.3 Copying

Copying garbage collection strategies divide the heap into two semispaces called *tospace* and *fromspace*. As the semispace names suggest, objects come from the fromspace and go to the tospace during collection. Specifically, objects are allocated at the top of tospace. Similarly to mark-sweep, garbage collection begins when tospace is exhausted.

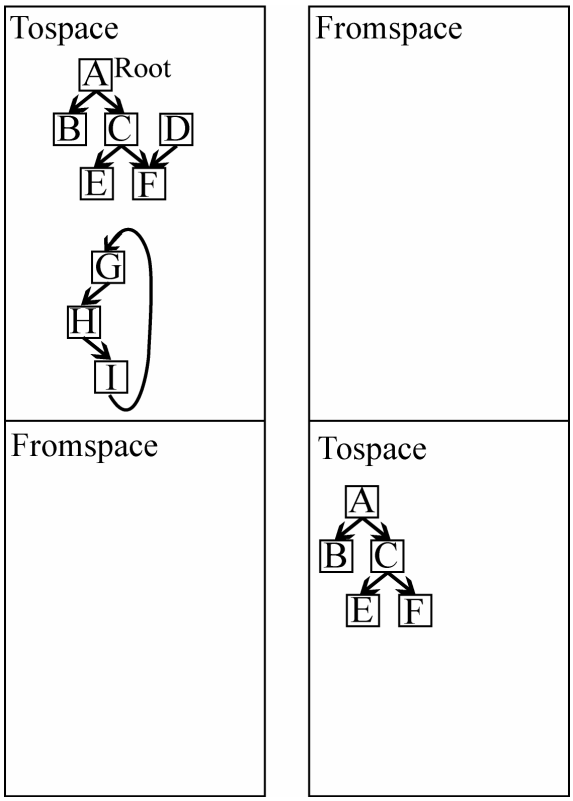

Figure 1: In a copying garbage collector, the live data is copied from the tospace to the fromspace, and then the roles are swapped until the next collection.

During collection, the roles of tospace and fromspace are reversed. The collector searches for live data in the fromspace and then copies it to the tospace. Once all live data is copied, the mutator phase resumes. The advantage of copying is that it allows object allocation to be faster than other garbage collection methods [12]. All live data is compacted at the bottom of the tospace, preventing the heap from becoming fragmented. New objects are allocated by incrementing the free-space pointer within tospace. The free-space pointer will be pointing to the end of the tospace when the heap is exhausted. The disadvantage of copying, however, is that the amount of address space required is doubled. This causes increased page faults because both semispaces must be touched during any garbage collection.

2.2.4 Generational-Copy

Generational-copy collectors are based on the assumption that objects die young [20, 15]. Kim and Hsu found that 65% to 99% of objects were short-lived in Java programs [15]. Kim and Hsu defined the lifetime of an object as the total number of data references between the first and last reference to an object. Short-lived objects have a lifetime less than 1 million references.

In the generational-copy strategy, the heap is divided into two or more generations. New objects are allocated in the young generation and are promoted to an older generation the longer they survive. The advantage of generational-copy collection is that pause times are decreased because shorter collections are performed on the small young generation, and full heap collections are done much less frequently — only when necessary. A benefit of generational copying over (non-generational) copying is that the long-lived objects are not copied repeatedly between semispaces; once the old object is promoted to old space it will not be touched during nursery collections. When an object in old space points to an object in

the nursery, that pointer becomes part of the root set when calculating live data during a nursery collection.

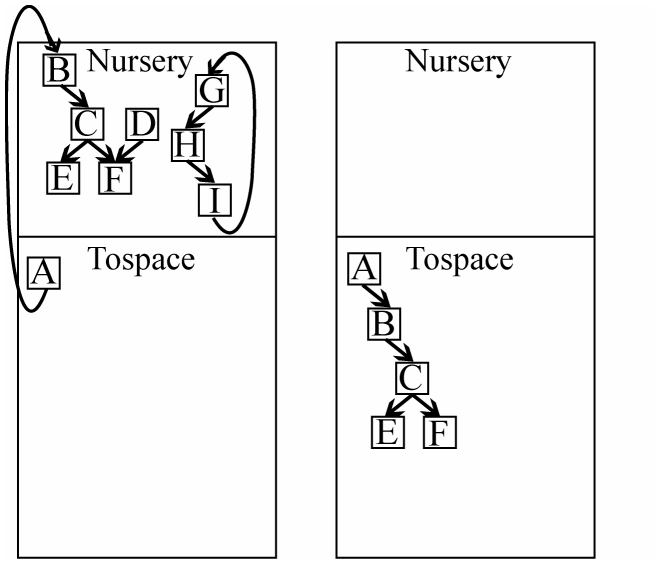

Figure 2: In a generational-copying garbage collector, allocation occurs in the nursery. During a nursery collection, live data is copied to an older generation.

2.3 Virtual Page Mapping

Virtual memory allows each process to have a full range of addresses, even though the processes share only one physical address space in main memory, e.g. 2^{32}–1. Most virtual memories are paged. Paged virtual memory divides the address space into page-sized blocks. Each virtual page is then mapped to a page frame in physical memory. Virtual memory simplifies the task of the programmer in two ways: (1) the programmer does not have to worry about collisions with other processes' objects in main memory, and (2) it automates the process of swapping information between main memory and secondary storage.

2.3.1 Page Coloring

Page coloring improves cache performance by mapping consecutive virtual pages to consecutive physical page frames. Mapping in this manner takes advantage of the *spatial* locality of memory accesses and thus can reduce conflict misses in the cache. Page frames are grouped by *color*. The number of colors is determined by dividing the cache size by the page size. For example, if a direct-mapped cache is 512KB and the page size is 4KB, there are 128 colors. Page coloring uses bit selection to choose the color of a particular virtual memory page. A page frame is then selected from the pool of available page frames with the same color.

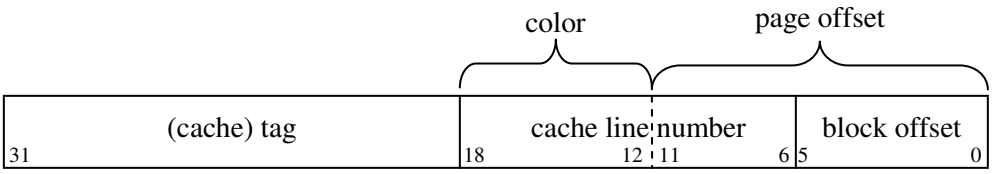

Figure 3: This shows the bit selection for a 512KB direct-mapped cache with a block size of 64 bytes. It also shows the bit selection for page coloring given a page size of 4KB.

2.3.2 Bin Hopping

Bin hopping takes advantage of *temporal* locality by mapping the most recently mapped virtual pages to consecutive physical page frames, so that they do not conflict in the cache. Page frames are divided into groups or *bins*. Each bin is a collection of page frames that map to a page-sized chunk of the cache. The number of bins is the cache size divided by the page size. On a page fault, the selection of a page frame is determined by incrementing the bin count. The bin count wraps around to zero when the final bin is reached.

2.4 Summary

The three basic memory allocation strategies are static, stack, and heap. The memory allocation strategy discussed in this thesis is the heap. Garbage collection is a method to automate heap management. The three primary garbage collection methods are reference counting, mark-sweep, and copying. This thesis investigates a variation of copying known as generational-copying collection. This thesis also studies the interaction of virtual memory and the cache during garbage collection. The two main virtual page mapping strategies, page coloring and bin hopping, are compared to our pinning strategy.

Chapter 3

Related Work

The pinning strategy discussed in this thesis was first studied by Reddy [17] and Krishnakumar [16]. These first studies demonstrated that pinning can provide cache miss-rate improvement for IBM Smalltalk programs. This thesis extends their work by studying the effects of the pinning strategy with Java, a more widely used language, using the SPECjvm98 suite, and the more memory-intensive DaCapo benchmarks. Also measured in this thesis, is speedup using a cycle-accurate simulator to observe the performance improvement provided by pinning.

Wilson, Lam, and Moher [21] investigated the effects of generational copying garbage collection in Scheme 48, a modified version of Scheme. They found that direct-mapped caches outperformed set-associative caches when the cache size was near the size of the nursery. For caches sizes larger than the nursery, set-associative outperformed direct-mapped. They found that the majority of misses in a cache larger than the young generation were conflict misses. This thesis expands upon concepts in Wilson, Lam, and Moher by pinning the youngest generation in the L2 cache.

Boehm [4] also had the goal of reducing cache misses of garbage collected programs. Boehm reduced cache misses for a non-copying collector using two strategies "prefetch-on-grey" and "lazy sweeping". The former strategy prefetches data during the mark phase to improve cache performance. The latter strategy postpones the sweep phase until allocation time, so that when reallocation occurs the allocated block is already in the cache because it was just accessed during the sweep.

Similar to Boehm, Reinhold [18] used a non-copying collector in his research. Reinhold found that caches perform best with infrequent garbage collection. However, his studies only included direct-mapped caches, which now are almost extinct.

Chilimbi and Larus [5] studied a real-time profiling technique to improve locality of heap objects in order to reduce the cache miss rate and improve execution time. Their researched focused on longer-lived objects. Although this thesis concentrates on the short-lived young generation objects, it is possible to use their strategy in conjunction with pinning.

Diwan, Tarditi, and Moss [8] examined the effect of generational garbage collection on various cache structures. Their studies used the Standard ML of New Jersey compiler. They found that the best performance was achieved when the cache had a subblock placement (write-validate [13]) coupled with a write-allocate policy.

The Standard ML of New Jersey compiler was also used by Cooper, Nettles, and Subramanian [6]. Their goal was to reduce page faults by modifying the page-fault handler. Ideally, pages that no longer contained useful information during garbage collection were not written back to secondary storage. The goal of this thesis, however, is to reduce cache misses.

The main goal of the research discussed above and the goal of this thesis is the same: to reduce cache misses during garbage collection and improve overall program execution time. There are many published strategies [4, 5, 8, 18, 21] to reach this goal. This thesis applies a strategy (pinning), first studied by Reddy and Krishnakumar, to Java programs.

Chapter 4

Experimental Methodology

The goal of pinning is to decrease garbage collection pause time and program execution time. This is accomplished by pinning the youngest generation of the heap in the L2 cache to decrease the number of cache misses during garbage collection. A portion of the cache is reserved, and only the youngest generation is allowed to map to the page frames that map to the reserved portion of the cache. Reddy [17] found that it was best to reserve 25% of the cache. Therefore, all of our experiments reserve 25% of the L2 cache for the nursery. The identity of the reserved page frames depends upon the size of the cache and the cache line size. Consider that the first page frame maps to the lowest-numbered sets in the cache, beginning with set 0. The second page frame maps to sets adjacent to the first page frame, and so forth, until the last set is reached. After that, the next page frame again maps to the lowest-numbered sets in the cache. The page frames that map to the first quarter of the cache are the reserved frames (in these experiments).

Let us define a bucket as a collection of contiguous page frames that map to the reserved portion of the cache. Page frames in a particular bucket share a certain bit pattern.

For example in figure 2, page frames in bucket 15 have all ones in bits 8 to 11. Our strategy reserves the page frames in a particular bucket; and only maps virtual pages containing the youngest generation to these reserved page frames. The pinning strategy requires just one bucket; therefore the page fault handler can choose which bucket to pin for a particular process. For example, in figure 2 bucket 0 or bucket 15 could be pinned.

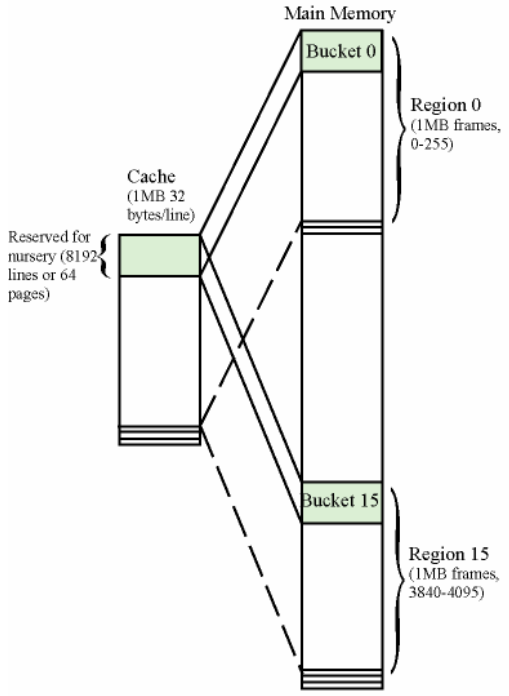

Figure 4: Pinning for a 1MB direct-mapped cache with 32-byte lines, to reserve a 256KB nursery in the cache. Here the first 8192 lines are reserved for the nursery. Each bucket is formed by taking the first 64 page frames (8192 lines) of each 1 MB region (= size of the cache). Only one bucket is chosen to contain new space. The choice of bucket is left to the memory allocator and page-fault handler.

In the Jikes research virtual machine (RVM), the address range of each heap space is determined at compile time. Using the GenCopy collector, the nursery space occupied the virtual address range of 0x74C00000 to 0x7FFFFFFF. In our experiments, the nursery size was restricted using two MMTk command-line parameters, X:gc:fixedNursery and X:gc:boundedNursery, so that the nursery was not allowed to exceed a size equal to ¼ of the L2 cache size. In order to simulate pinning, we extended Dynamic SimpleScalar (DSS) to

include virtual-to-physical address translation. When the executable (Jikes) invokes the `mmap` system call, a virtual page is added to the page table. If that virtual page lies within the nursery address range, then that page is mapped within a reserved page bucket. Otherwise, the page is mapped to a page frame outside the bucket.

4.1 Simulator

Dynamic SimpleScalar version 1.0.1 [9] was used in this experiment, primarily because it supports just-in-time compilers such as Jikes RVM. The functional cache simulator (sim-cache) was used to calculate miss-rate improvements, and the cycle-based simulator (sim-outorder) was used to calculate speedup. We made several modifications to the simulator: (1) A virtual-to-physical page mapping was added, (2) unsupported system calls were implemented (`rmdir`, `mkdir`, `nanosleep`, and `dirents64`), (3) in sim-cache the L2 cache was replaced with a single-pass cache simulator [10], and (4) several additional counters were added to measure cache statistics for each phase and for each heap space. By default, DSS passes the virtual addresses to the cache without translating them to physical addresses; this is sufficient for most experiments. Our experiments were dependent on the virtual page mapping, which required physical address translation. Virtual addresses are translated to physical addresses and the physical address is passed to the cache simulator.

Four additional system calls were implemented because they were needed for the DaCapo benchmarks to finish successfully. And, LRU stack depths were measured to simulate the miss rate of direct-mapped through 16-way L2 cache configurations in a single run using the all-associativity algorithm from Hill and Smith [10].

4.2 Virtual Machine

The Jikes RVM version 2.4.0 [11] was chosen for this study because it is an widely used open-source research virtual machine whose memory management toolkit (MMTk) [3] provides a generational copying garbage collector. It was configured to use the fast adaptive optimizing compiler and the generational copy collector. It was instrumented to signal the beginning and end of each garbage-collection phase using a virtual page that had a unique `madvise` class to pass signals to the simulator. Jikes timing is non-deterministic because the optimizing compiler is timer-based. Therefore, the compiler replay option was used to limit the variability among executions of the same benchmark. For compiler replay, the optimizing compiler decisions are not made at run time, instead the decisions are based on profile information. The profile information consists of the dynamic call graph, the frequency of each edge for inlining decisions, and the adaptive compilation level for each method. Each benchmark was run 7 times in a native PowerPC Linux environment, and the profile information from the run with the median execution time was used for our analysis.

4.3 Benchmarks

The DaCapo benchmarks [7] were chosen because they are readily available and stress the memory system. Two of the DaCapo benchmarks, batik and chart, were not used because they were incompatible with Jikes at the time this research was performed.

The SPECjvm98 benchmarks are commercially available benchmarks and add breadth to this study, showing the effect of pinning on a set of less memory-intensive benchmarks.

Table 1: DaCapo benchmark description [7]

Benchmark	Description
antlr	parses one or more grammar files and generates a parser and lexical analyzer for each.
batik	renders a number of SVG files
bloat	performs a number of optimizations and analysis on Java bytecode files
chart	uses JFreeChart to plot a number of complex line graphs and renders them as PDF
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
hsqldb	executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	interprets a series of Python programs
pmd	analyzes a set of Java classes for a range of source code problems
ps	reads and interprets a PostScript file
xalan	transforms XML documents into HTML

Table 2: SPECjvm98 benchmark descriptions [19]

Benchmark	Description
_201_compress	Modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built.
_202_jess	JESS is the Java Expert Shell System is based on NASA's CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase run time the benchmark problem iteratively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus the inference engine must search through progressively larger rule sets as execution proceeds.
_205_raytrace	A raytracer that works on a scene depicting a dinosaur, where two threads each renders the scene in the input file time-test model, which is 340KB in size.
_209_db	Performs multiple database functions on memory resident database. Reads in a 1 MB file which contains records with names, addresses and phone numbers of entities and a 19KB file called scr6 which contains a stream of operations to perform on the records in the file. The program loops and reads commands till it hits the `q' command.
_213_javac	This is the Java compiler from the JDK 1.0.2. As this is a commercial application, no source code is provided.
_222_mpegaudio	This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. As this is a commercial application only obfuscated class files are available. The workload consists of about 4MB of audio data.
_227_mtrt	This is a variant of _205_raytrace.
_228_jack	A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. See Looking for lex and yacc for Java? You don't know Jack - By Chuck McManis, Javaworld Magazine. The workload consists of a file named jack, which contains instructions for the generation of jack itself. This is fed to jack so that the parser generates itself multiple times. Because this is a commercial application, no source code is provided.

Chapter 5

Results

In the first analysis, the success of pinning can be measured by the improved cache miss rate. Improvements in cache miss rate translate into faster run times, which is an obvious benefit. Between miss rate and run time, miss rate is easier to measure. Miss rate depends only on cache organization and the behavior of the application and not on a host of architectural factors, such as memory characteristics, bus contention, or instruction-level parallelism artifacts such as size of the reorder buffer. Also, a single run of the simulator can be used to gather results for several different cache associativities [10], but run-times must be measured by separate runs for each cache organization. Section 5.1 shows several results related to miss-rate improvement. Pinning is compared to no pinning in section 5.1.1. Then we show the improvement over competing page-mapping strategies, page coloring and bin hopping, in sections 5.1.2 and 5.1.3, respectively.

5.1 Miss-rate Improvement

5.1.1 Pinning vs. no pinning

Pinning improves the global miss rate relative to the base case which is no page-placement strategy. Our base case assumes no virtual address translation; i.e., the virtual address generated by DSS is used as the physical address. This yields a conservative estimate of the improvement of pinning for the following reason: Two consecutive virtual pages are always mapped to two consecutive physical page frames, which means they will map to adjacent sets within the cache. There is no possibility that adjacent virtual pages will map to the same cache sets, unless there are very few sets in the cache (i.e., a very small or highly associative L2 cache). Otherwise, two adjacent virtual pages will never compete for the same cache set. In real systems, however, it is possible that two adjacent virtual pages are mapped to physical memory in such a way that they compete for the same cache lines. The assumption that physical address = virtual address tends to underestimate the contention for L2 cache lines and thus show better performance for the base case than would be seen in a real system.

The miss-rate improvement of pinning over no pinning is displayed in figure 5 on the next page. In each graph, there is a separate column for each benchmark. The last column in each graph shows the average global miss-rate improvement across all benchmarks. In general, the average improvement decreases as the associativity of the L2 cache increases, the exception is the 512KB L2 cache.

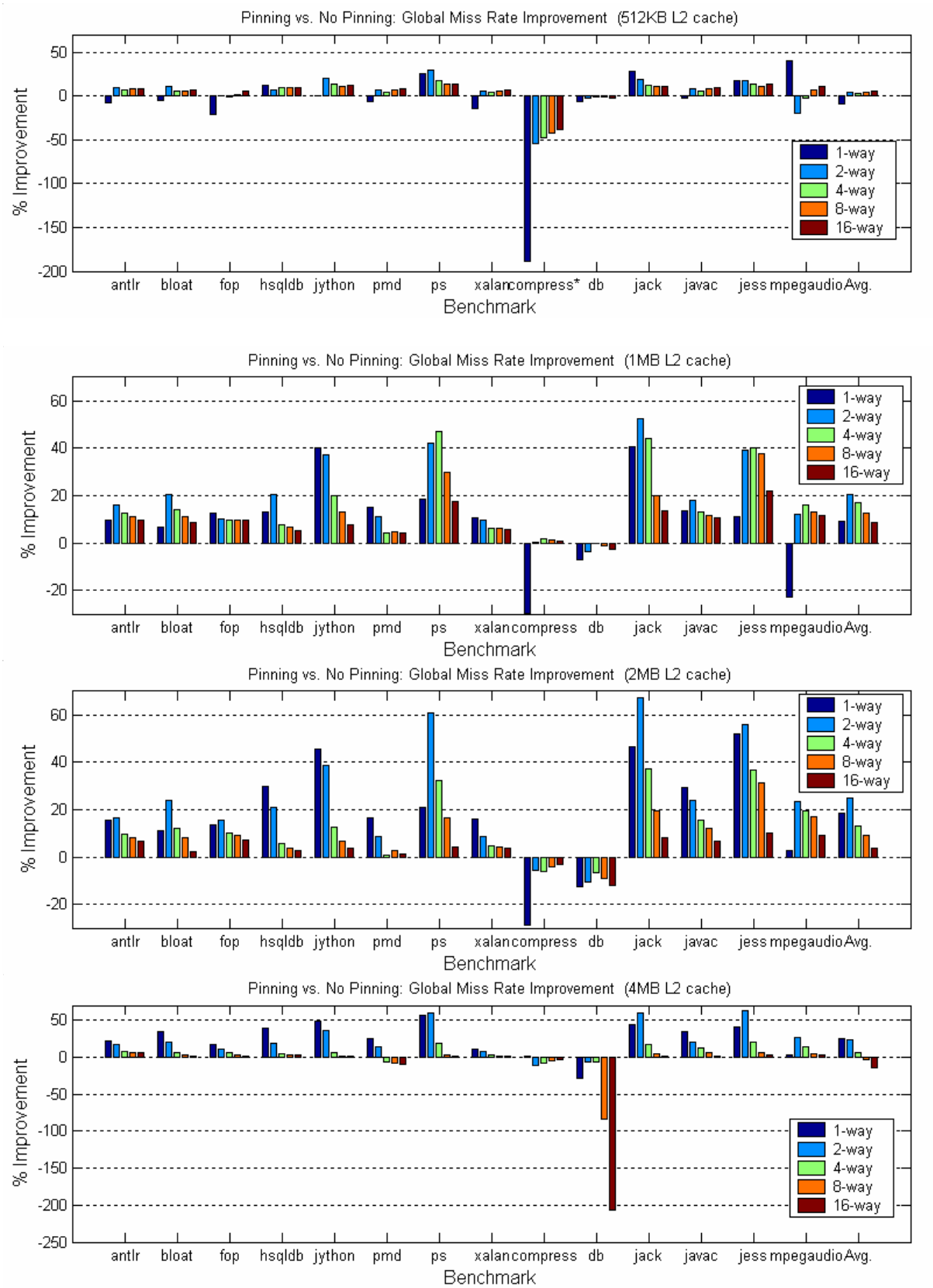

Figure 5: The pinning strategy improves average global miss rate compared to no pinning.
* denotes an impartial run

5.1.2 Pinning vs. page coloring

Pinning almost always improves global miss rate compared to page coloring (averaged over all benchmarks). Page coloring outperforms pinning only in a highly associative (8-way or 16-way) 4MB L2 cache. The results are shown in figure 6 on the following page. The average global miss-rate improvement is greater than 20% in all 2-way set-associative L2 caches 1MB or larger. Excluding the highly associative cases, pinning improves the global miss rate 7% to 26% for 4MB caches. The global miss-rate improvement ranges from 3%–9% for 512KB L2 caches, 9%–21% for 1MB L2 caches, and 4%–25% for 2MB L2 caches. The miss-rate improvements translate to shorter program execution times, which are discussed further in section 5.2.

5.1.3 Pinning vs. bin hopping

Pinning also improves global miss rate compared to bin hopping in all but the 512KB direct-mapped L2 cache (figure 7). The benefit of pinning generally decreases as the associativity of the L2 cache increases. Typically, there are fewer total misses in more highly associative caches, so there is less room for improvement. Excluding the direct-mapped case, pinning improves the global miss rate 3.1% to 5.3% for 512KB caches. The global miss-rate improvement ranges from 8.2%–20.0% for 1MB L2 caches, 2.8% to 14.9% for 2MB L2 caches, and 2.8% to 22.9% for 4MB caches.

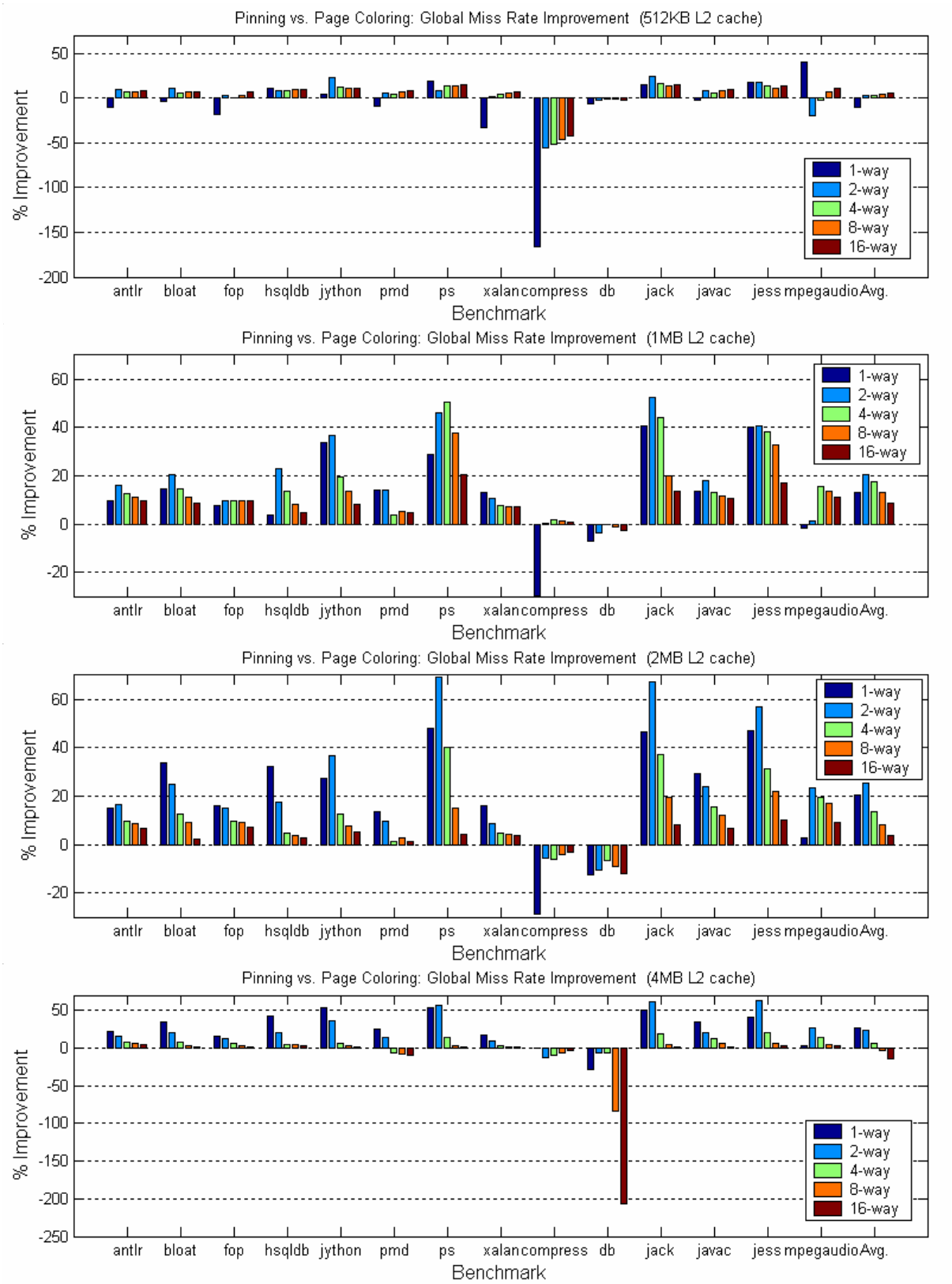

Figure 6: The pinning strategy improves average global miss rate compared to page coloring in all but the 512KB direct-mapped L2 cache.

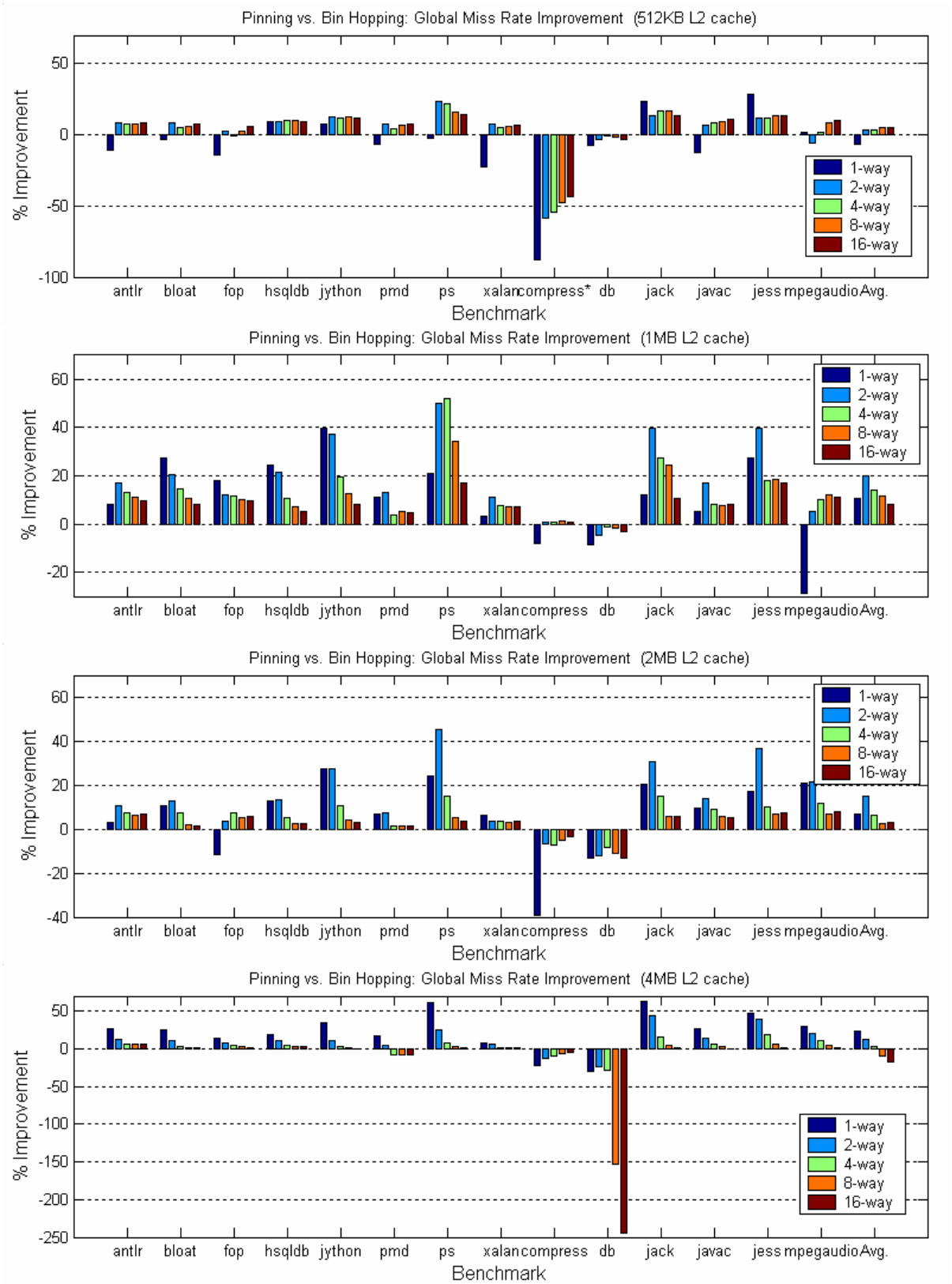

Figure 7: The pinning strategy improves global miss rate compared to bin-hopping in all but the 512KB direct-mapped L2 cache.
* denotes an impartial run

5.2 Speedup

5.2.1 Pinning vs. page coloring

Today's processor architectures allow instructions to complete out of order. This means that a long cache miss will not delay the execution of all instructions in a program, only those instructions that depend on the data being fetched. Therefore, it is not clear that an improvement in miss rate necessarily translates to a perceptible decrease in execution time. We measured speedup using DSS's sim-outorder simulator with an issue width of 4 and a memory latency of 200 cycles. This gives a realistic picture of the speedup yielded by the global miss-rate improvement due to pinning. Pinning provides positive speedup across all cache configurations studied, as seen in figure 8. Pinning provides the most speedup in 2-way cache configurations where the average is 5%–7%, and even in the higher associativities the average is 2%–3%.

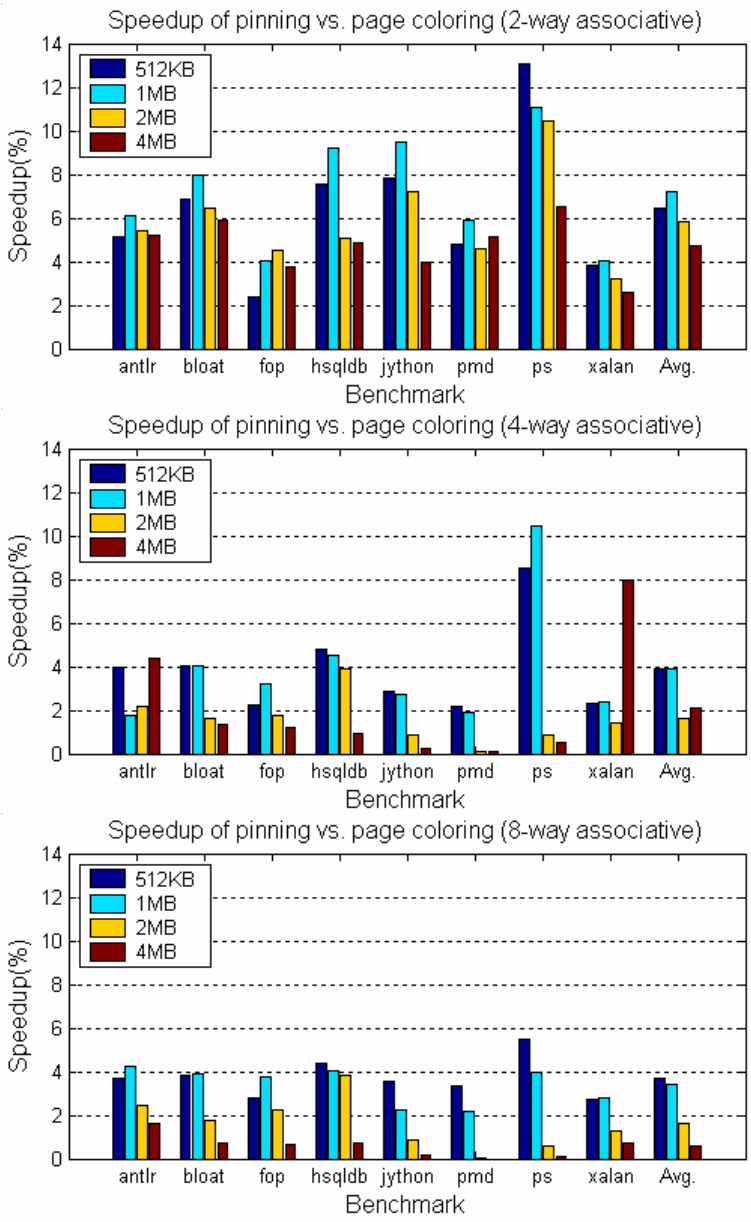

Figure 8: The pinning strategy improves execution time for all cache configurations studied.

5.2.2 Speedup in a highly associative cache with long memory latency

In the future, memory latencies, measured in terms of processor cycles, will be greater. Furthermore, the trend toward greater associativity in L2 caches can be expected to continue. The processor of a few years will have memory latencies of 400 cycles, and quite possibly L2 associativities of 32. We simulate a 4MB 32-way set associative cache using a FIFO replacement policy instead of LRU. FIFO is chosen because it is not practical to use LRU placement in a more highly associative cache. Figure 9 shows that pinning continues to provide a modest speedup even as caches grow in size and become more associative.

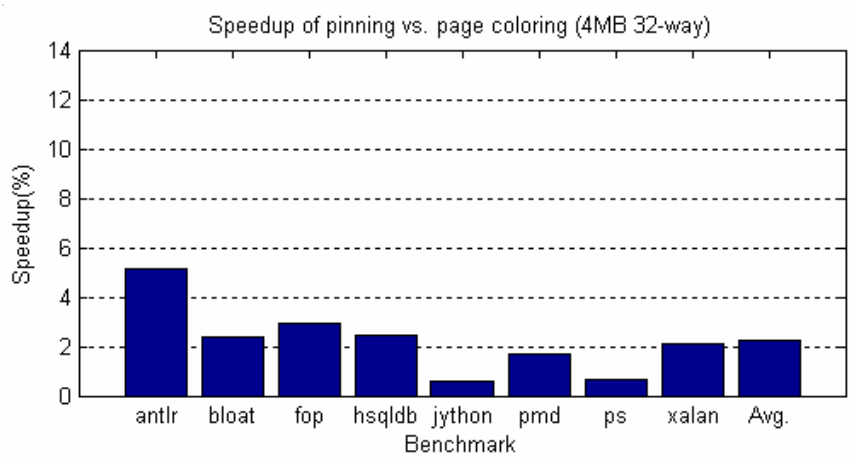

Figure 9: If the trend towards large highly associative caches and high miss penalty continues, pinning could still provide a performance improvement.

5.3 Heap Size

Pinning requires the nursery size to be restricted to a fraction of the L2 cache size. By default the garbage collector allows the size of the nursery to grow and shrink using the technique described by Appel [1]. Restricting nursery size causes the number of garbage collections to increase. As this happens, the number of objects promoted to old space also increases because new objects have less time to die before being promoted. The increase in the heap high-water mark is nominal (figure 10). The maximum heap size is unaffected for the benchmarks bloat and ps with a 1MB or 2MB L2 cache. The maximum heap size is *reduced* for the fop benchmark in a 1MB or 2MB cache and the antlr benchmark in a 2MB cache.

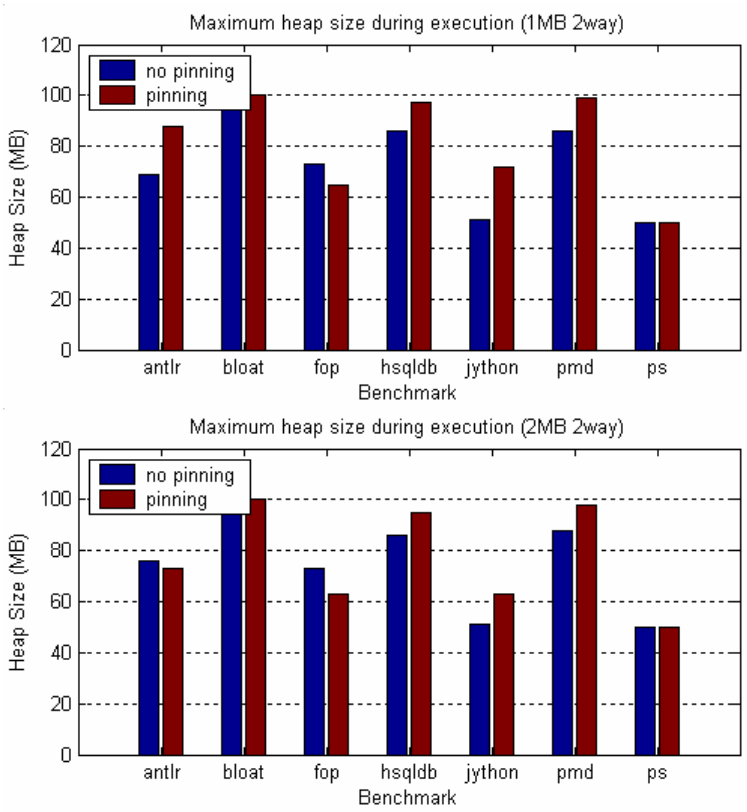

Figure 10: Restricting the nursery size for pinning can increase the number of promoted objects and potentially increase the maximum size of the heap during execution, but these graphs show that this change is nominal.

5.4 Garbage-Collection Pause Times

An important benefit of pinning is that it reduces the pause time in nursery collections. This reduction is a combination of two factors: restricting the nursery size and eliminating L2 cache misses within the nursery space. Restricting the nursery size limits the amount of data traversed at each nursery collection. Pinning the nursery in the L2 cache eliminates cache misses to the nursery objects accessed. The pause time is measured in cycles directly proportional to time using the out-of-order simulator. Figure 11 below shows that pause times are greatly reduced by 56% to 92% when using the pinning strategy. Histograms of the pause times for each benchmark are in Appendix A.

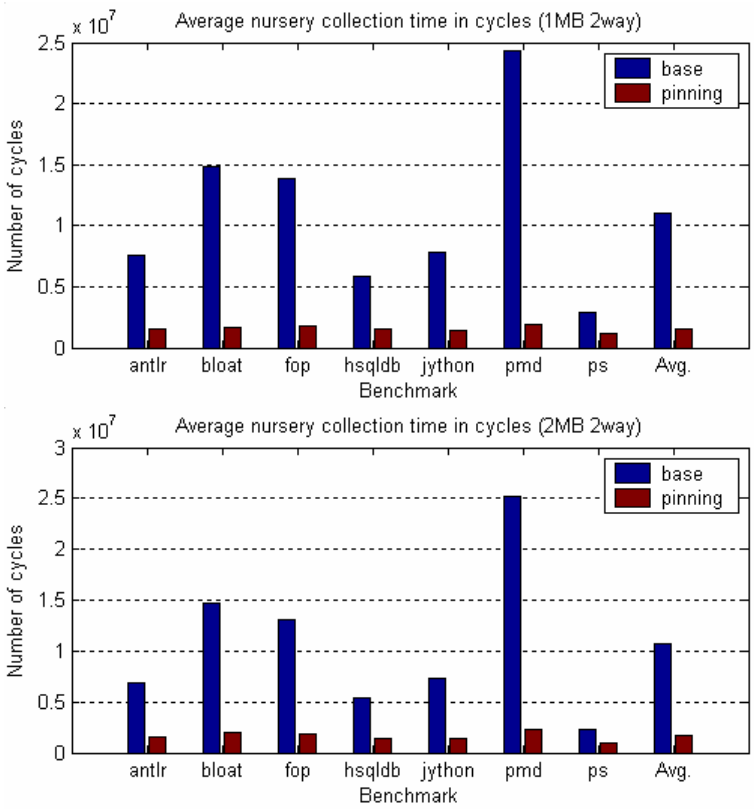

Figure 11: An important benefit of pinning is that it reduces the pause time for nursery collections.

As previously stated, eliminating misses in the L2 cache is one factor reducing pause times. The other factor is that the nursery is smaller. In figure 12, the nursery size is restricted for both the base case (page coloring) and the pinning case. This allows us to isolate the effect of eliminating misses in the L2 cache. The pause time is reduced 4% to 17% due to the elimination of misses. Thus, reducing nursery size is the biggest factor in reducing pause time. If it were not for pinning, then the overall runtime would be increased by limiting nursery size.

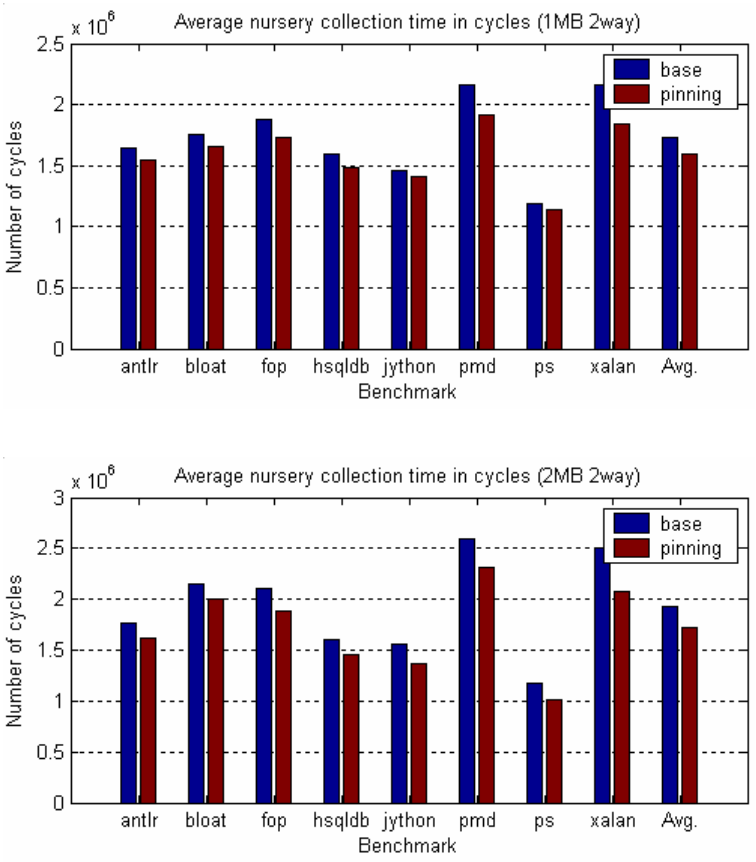

Figure 12: The decrease in pause time contributable to pinning the nursery in L2 cache.

Real-time programs typically do not use garbage collection because pause times are unpredictable. A benefit of pinning is that it reduces the variance of pause times. Thus, pinning aids programs with real-time deadlines because the pause time is less variable. Figure 13 shows that pinning reduces the variance in pause time by 95% to 99%. The greater variability, however, is in the old-generation collection. This is the first step in reducing variance, but it does not solve the entire problem.

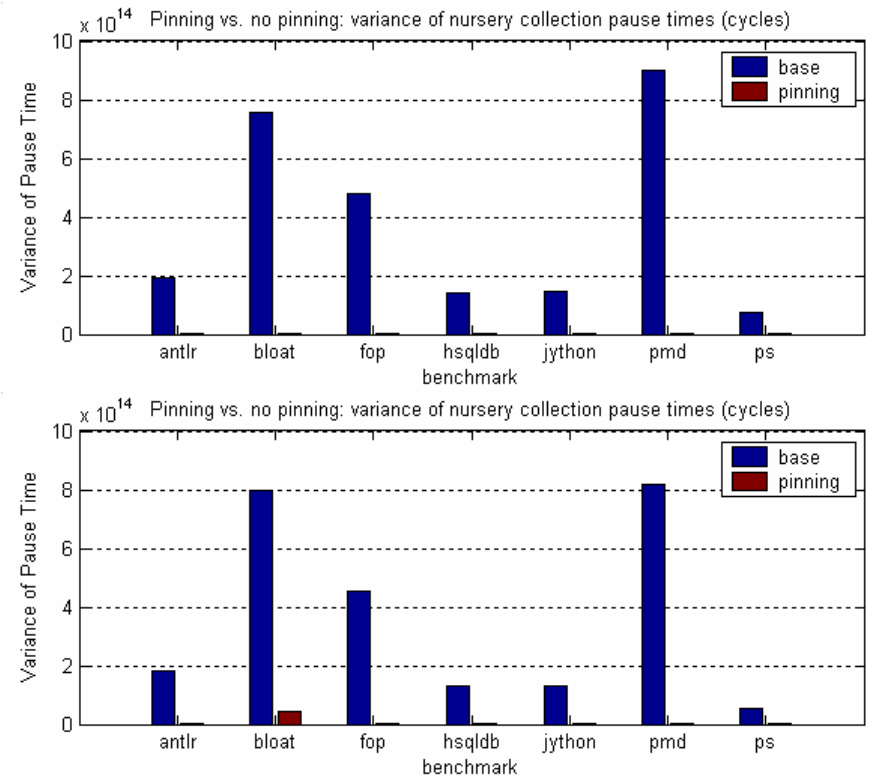

Figure 13: Pinning reduces the variance of nursery pause times. This especially beneficial for real-time programs.

5.5 Fraction of cache reserved

Pinning implies that part of the L2 cache is reserved for the nursery. In these experiments we tried to determine how much of the cache should be reserved. We considered three fractions: 12.5%, 25%, and 50%. The best fraction has the most miss-rate improvement. Figures 14, 15, and 16 demonstrate that the optimal fraction varies with cache

size and associativity. For direct-mapped caches 12.5% is the optimal fraction. 12.5% is also the optimal fraction for 2-way and 4-way caches 1MB or larger. A 25% portion is best for 512KB set-associative caches and 8-way or 16-way 1MB caches. For the remaining highly-associative large L2 caches (4-way to 16-way and 2MB to 4MB), it is best to reserve 50% of the cache for the nursery space. In the previous sections we reserved 25% of the L2 cache for pinning, but the results from this section show that 25% is not always the best choice.

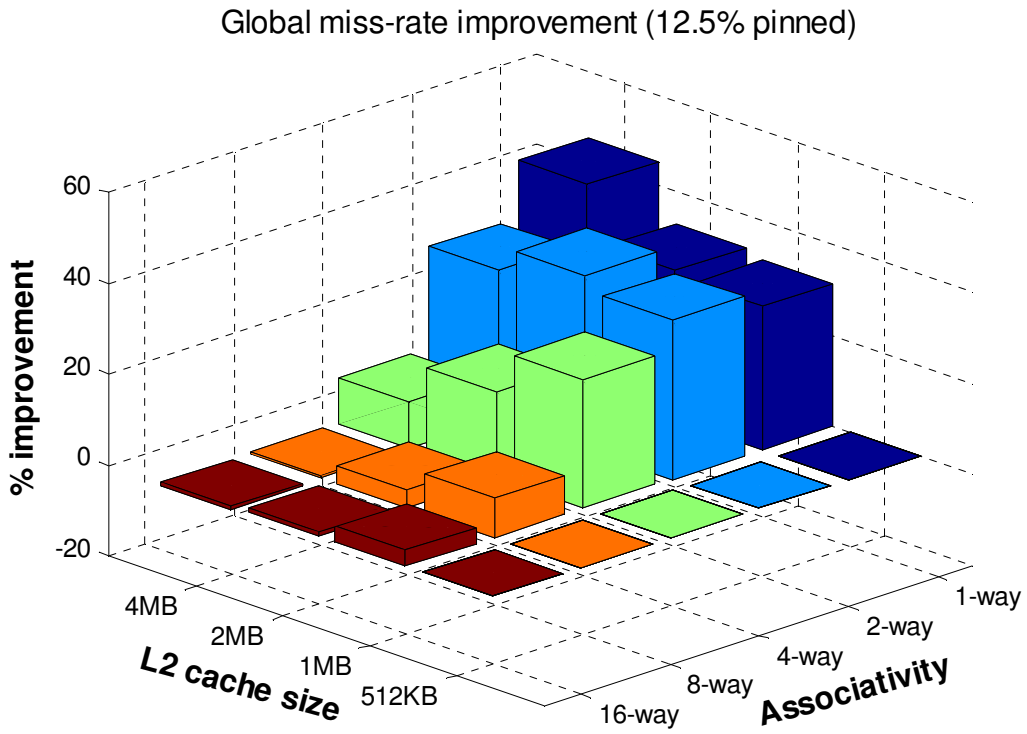

Figure 14: Global miss-rate improvement when pinning nursery in 12.5% of the L2 cache.

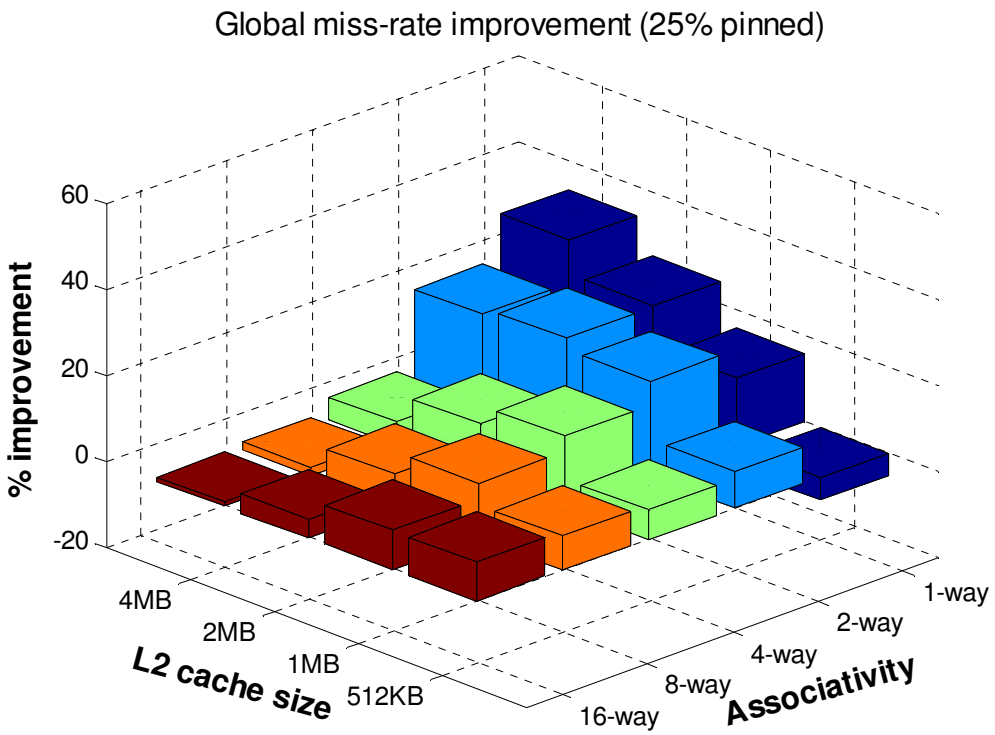

Figure 15: Global miss-rate improvement when pinning nursery in 25% of the L2 cache.

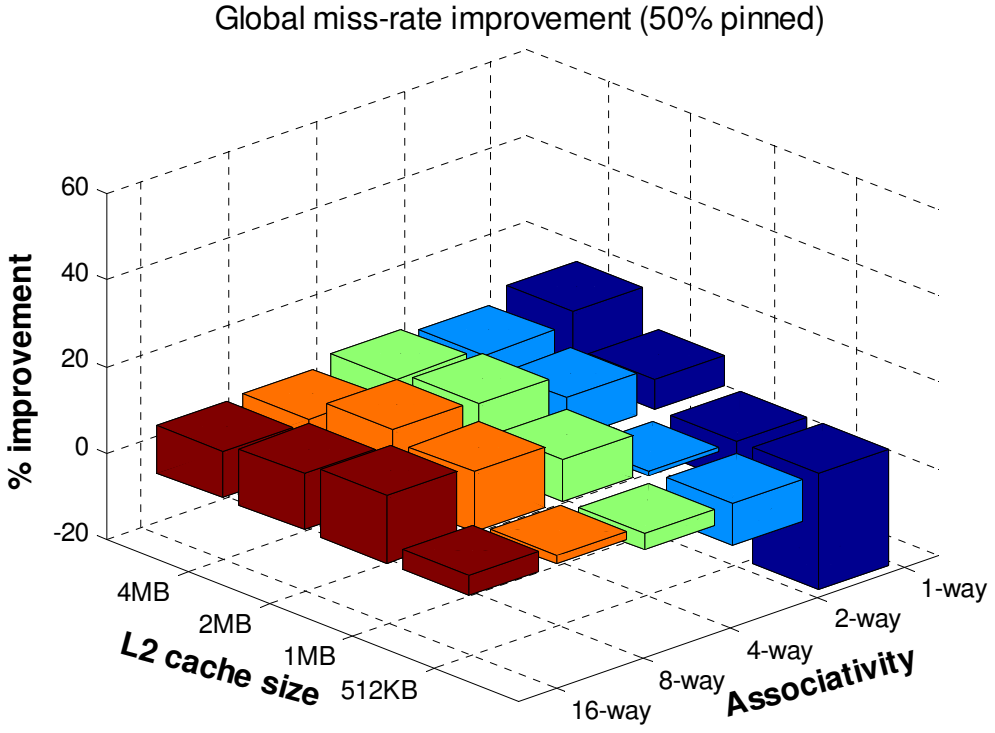

Figure 16: Global miss-rate improvement when pinning nursery in 50% of the L2 cache.

Chapter 6

Conclusion

Pinning is a page-mapping strategy that improves the cache miss rate of generational-garbage collectors. Our strategy eliminates L2 cache misses to nursery objects in a generational garbage collector by pinning the nursery to 25% of the L2 cache. We measured the global cache miss rates across a range of Java benchmarks from both the SPECjvm98 and DaCapo benchmark suites. Pinning is shown to outperform the competing virtual-memory mapping strategies of page coloring and bin hopping in reducing global miss rate.

Improvement in miss rate shortens overall execution time for practically every benchmark and every configuration. We measured the speedup of pinning using a cycle-accurate simulator. On an out-of-order processor with issue width 4, pinning provides a speedup of 5–7% for 2-way caches, 2–4% for 4-way caches, and 1–4% for 8-way caches. We also measured the speedup in execution time for memory configurations of the future. Pinning provides an average speedup of 2% for a large highly associative cache, 4MB 32-way L2 cache, and an L2 miss penalty of 400 cycles. These improvements are small, but they can be realized by a small change to software and seem likely to persist in future

generations of processor architectures. Moreover, any strategy that relies on a copying collector in the youngest generation (e.g., ulterior reference counting [2]) is amenable to improvement through this technique.

Pinning also reduces average pause time and variability of pause times. The reduction in the mean and variance of pause time is due to two factors: restricting the young generation size and eliminating L2 cache misses to young generation address space. It reduces the average nursery pause time by 56% to 92%, and decreases the variability of pause times by 95% to 99%. A shorter more predictable pause time is an important consideration especially for programs with real-time deadlines.

Future work includes examining the pinning strategy on a real system, such as a modified Linux kernel. It would also be useful to measure speedup for cache configurations where 12.5% or 50% of the L2 cache can provide greater improvement.

References

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

[2] Blackburn, S. M. and McKinley, K. S. 2003. Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA, October 26 - 30, 2003). OOPSLA '03. ACM Press, New York, NY, 344-358.

[3] Blackburn, S. M., Cheng, P., and McKinley, K. S. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th international Conference on Software Engineering* (May 23 - 28, 2004). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 137-146.

[4] Boehm, H. Reducing garbage collector cache misses. 2000. In *Proceedings of the 2nd international Symposium on Memory Management* (Minneapolis, Minnesota, United States, October 15 - 16, 2000). ISMM '00. ACM Press, New York, NY, 59-64.

[5] Chilimbi, T. M. and Larus, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st international Symposium on Memory Management* (Vancouver, British Columbia, Canada, October 17 - 19, 1998). ISMM '98. ACM Press, New York, NY, 37-48.

[6] Cooper, E., Nettles, S., and Subramanian, I. 1992. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, United States, June 22 - 24, 1992). LFP '92. ACM Press, New York, NY, 43-52.

[7] DaCapo benchmarks version beta050224 http://ali-www.cs.umass.edu/DaCapo/

[8] Diwan, A., Tarditi, D., and Moss, E. 1995. Memory system performance of programs with intensive heap allocation. *ACM Trans. Comput. Syst*. 13, 3 (Aug. 1995), 244-273.

[9] Dynamic SimpleScalar version 1.0.1 http://www-ali.cs.umass.edu/DSS/

[10] Hill, Mark D. and Smith, Alan Jay. 1989. "Evaluating Associativity in CPU Caches." IEEE Transaction on Computers, Vol. 38, No. 12, December 1989.

[11] Jikes Research Virtual Machine version 2.4.0 http://jikesrvm.sourceforge.net/

[12] Jones, Richard. and Lins, Rafael. 1996. *Garbage Collection: Algorithms for automatic dynamic memory management*. John Wiley & Sons, ISBN: 0471941484, 1996.

[13] Jouppi, N. P. 1993. Cache write policies and performance. In *Proceedings of the 20th Annual international Symposium on Computer Architecture* (San Diego, California, United States, May 16 - 19, 1993). ISCA '93. ACM Press, New York, NY, 191-201.

[14] Kessler, R. E. and Hill, M. D. 1992. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst*. 10, 4 (Nov. 1992), 338-359.

[15] Kim, J. and Hsu, Y. 2000. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems* (Santa Clara, California, United States, June 18 - 21, 2000). SIGMETRICS '00. ACM Press, New York, NY, 264-274.

[16] Krishnakumar, Sree Vidhya Lakshmi. 2004. "Caching Strategies for More Efficient Generational Garbage Collection." MS Thesis, North Carolina State University, 2004.

[17] Reddy, V. 2003. "Caching strategies for improving generational garbage collection in Smalltalk." MS Thesis, North Carolina State University, 2003.

[18] Reinhold, M. B. 1994. Cache performance of garbage-collected programs. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM Press, New York, NY, 206-217.

[19] SPECjvm98 benchmarks http://www.spec.org/jvm98/jvm98/doc/benchmarks/index.html

[20] Ungar, D. 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* SDE 1. ACM Press, New York, NY, 157-167.

[21] Wilson, P. R., Lam, M. S., and Moher, T. G. 1992. Caching considerations for generational garbage collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, United States, June 22 - 24, 1992). LFP '92. ACM Press, New York, NY, 32-42.

Appendix

Appendix A

A.1 Pause time histograms (1MB L2 cache)

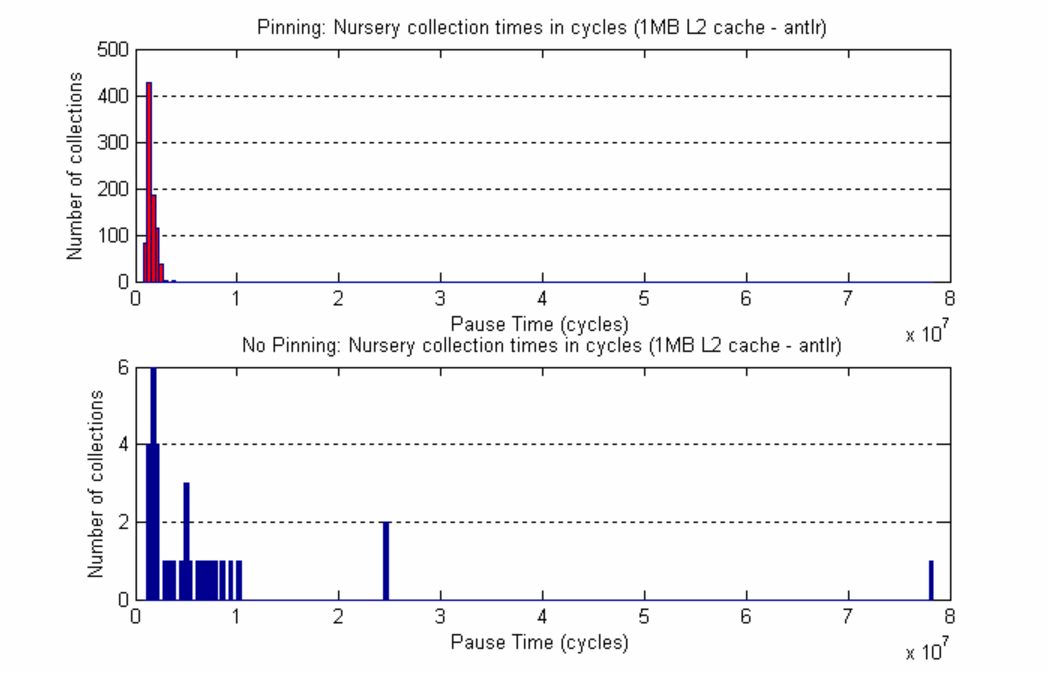

Figure A.1: Histrogram of pause time (1MB L2 cache, antlr benchmark)

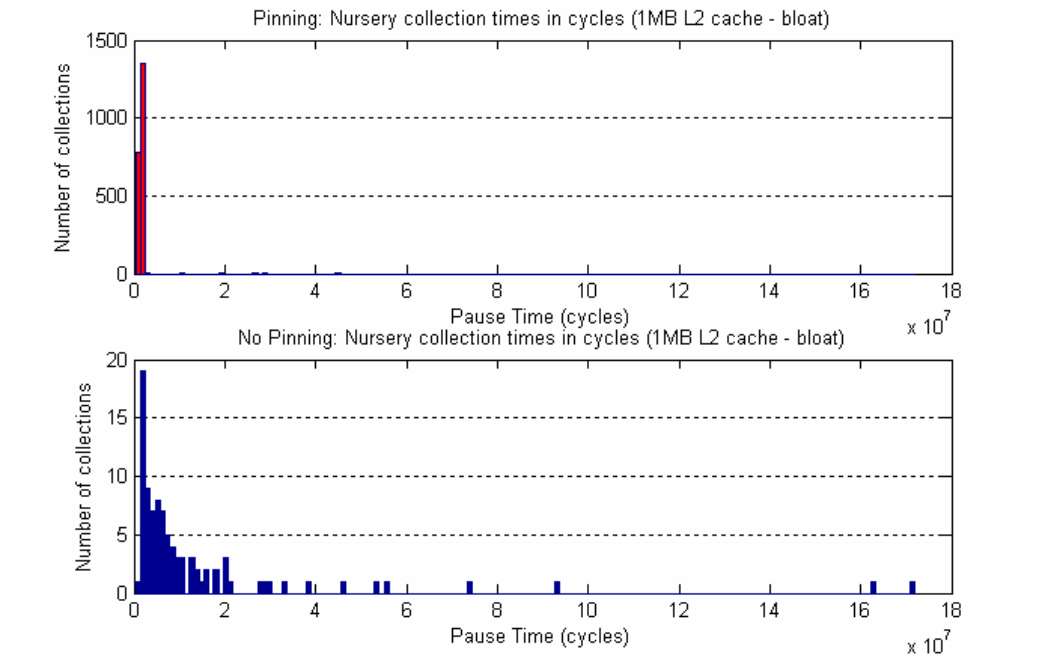

Figure A.2: Histrogram of pause time (1MB L2 cache, bloat benchmark)

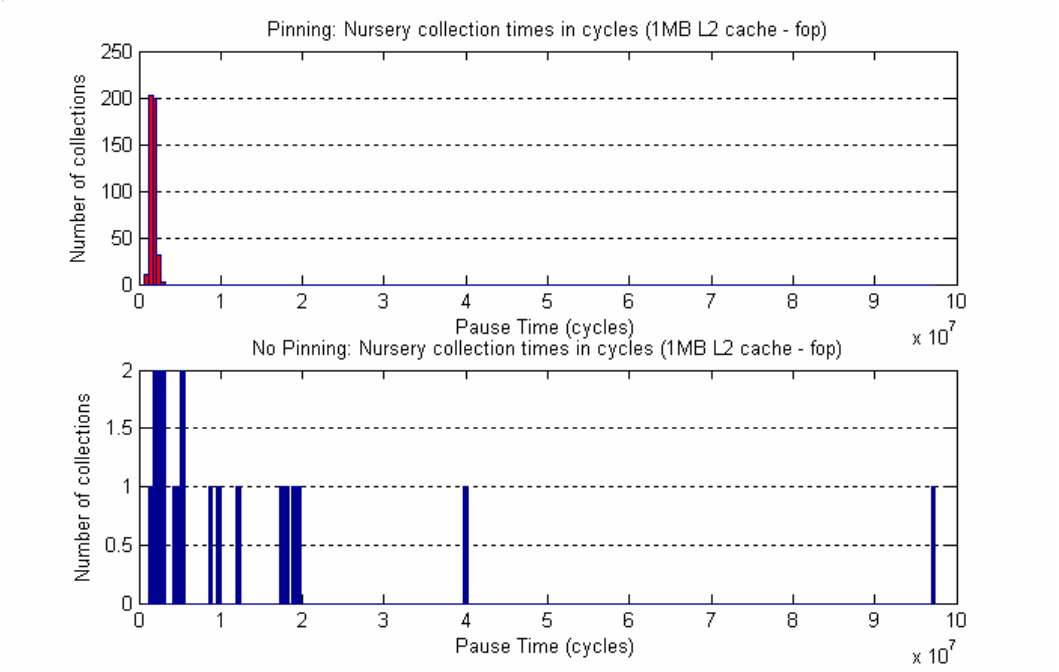

Figure A.3: Histrogram of pause time (1MB L2 cache, fop benchmark)

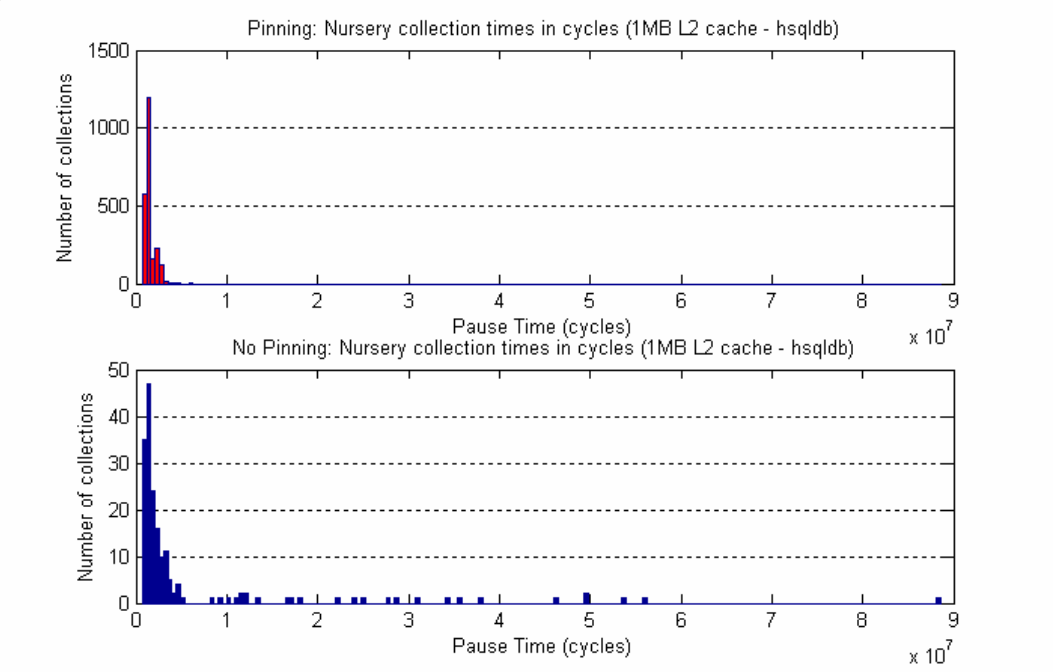

Figure A.4: Histrogram of pause time (1MB L2 cache, hsqldb benchmark)

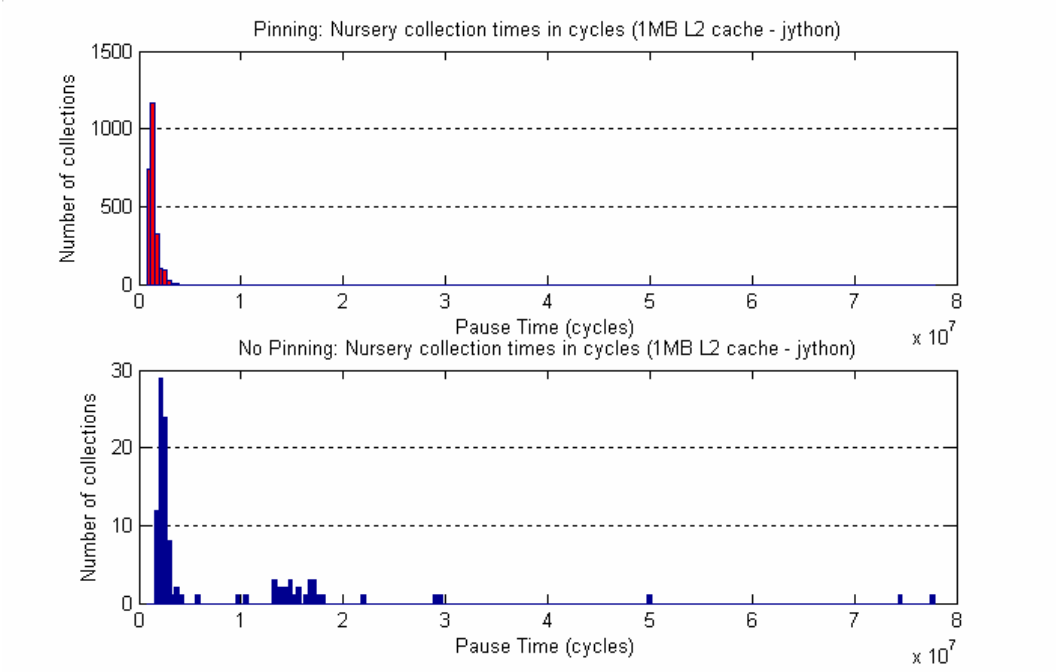

Figure A.5: Histrogram of pause time (1MB L2 cache, jython benchmark)

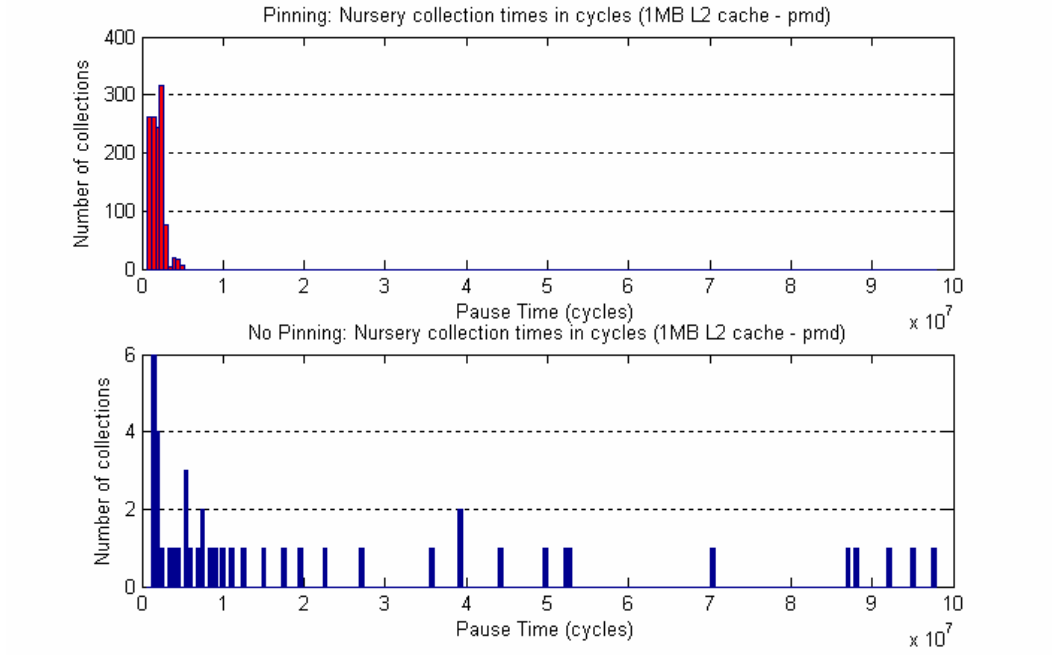

Figure A.6: Histrogram of pause time (1MB L2 cache, pmd benchmark)

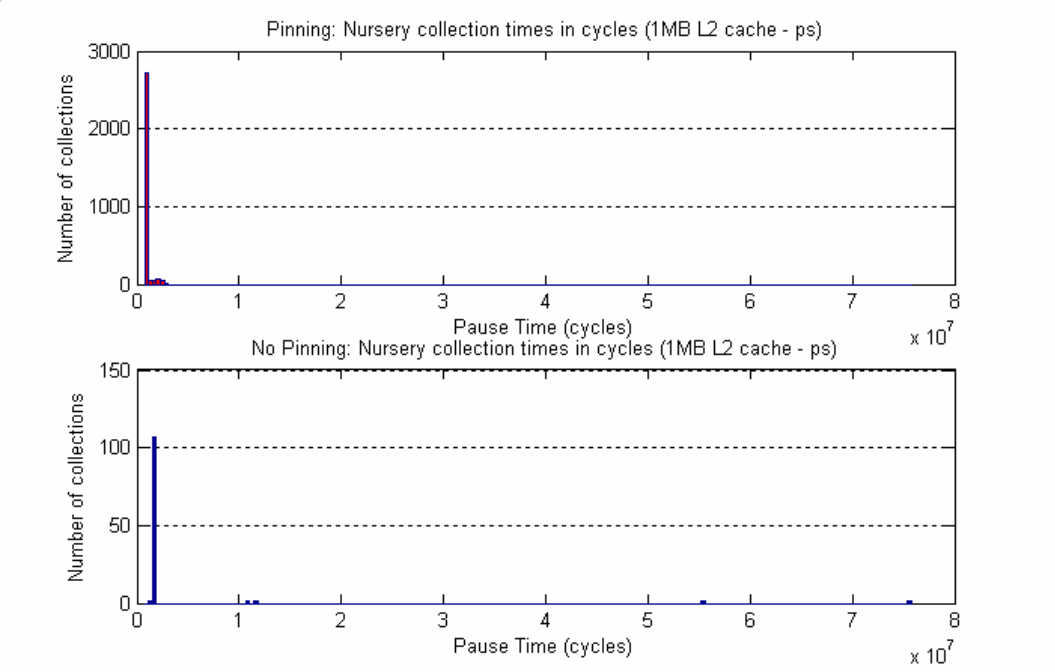

Figure A.7: Histrogram of pause time (1MB L2 cache, ps benchmark)

A.2 Pause time histograms (2MB L2 cache)

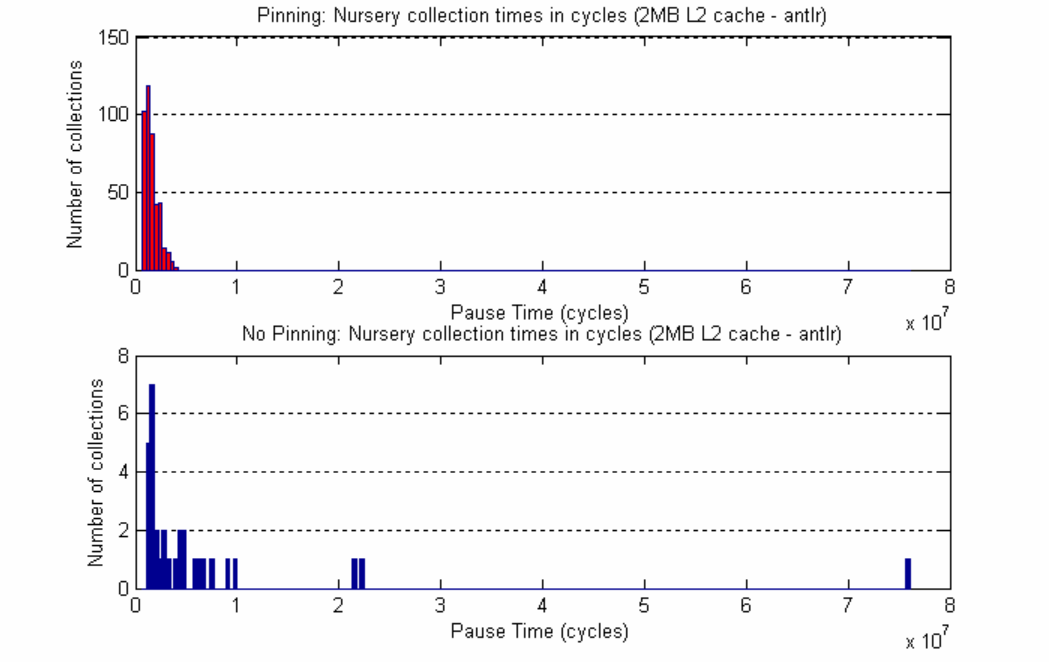

Figure A.8: Histrogram of pause time (2MB L2 cache, antlr benchmark)

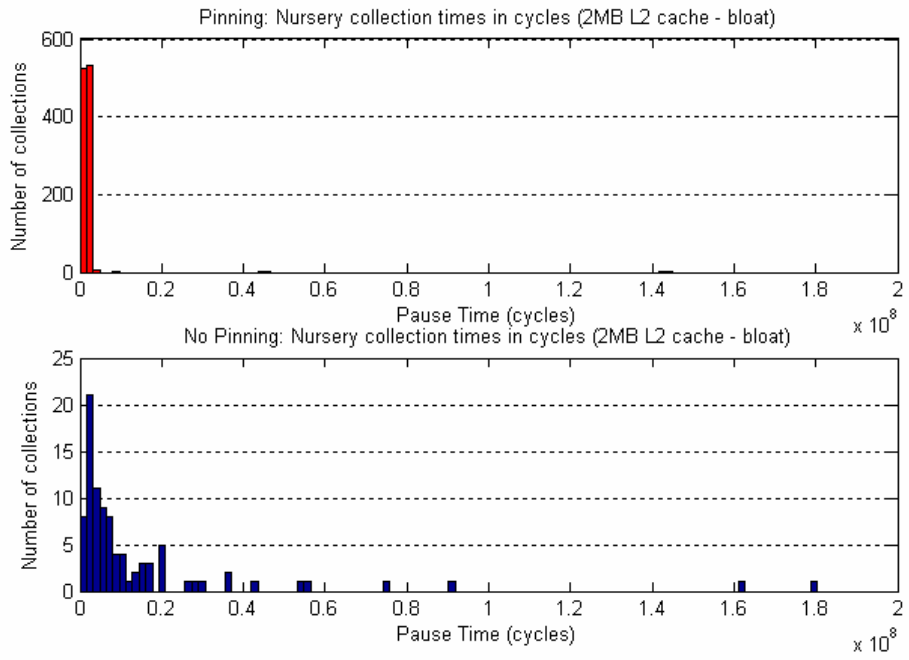

Figure A.9: Histrogram of pause time (2MB L2 cache, bloat benchmark)

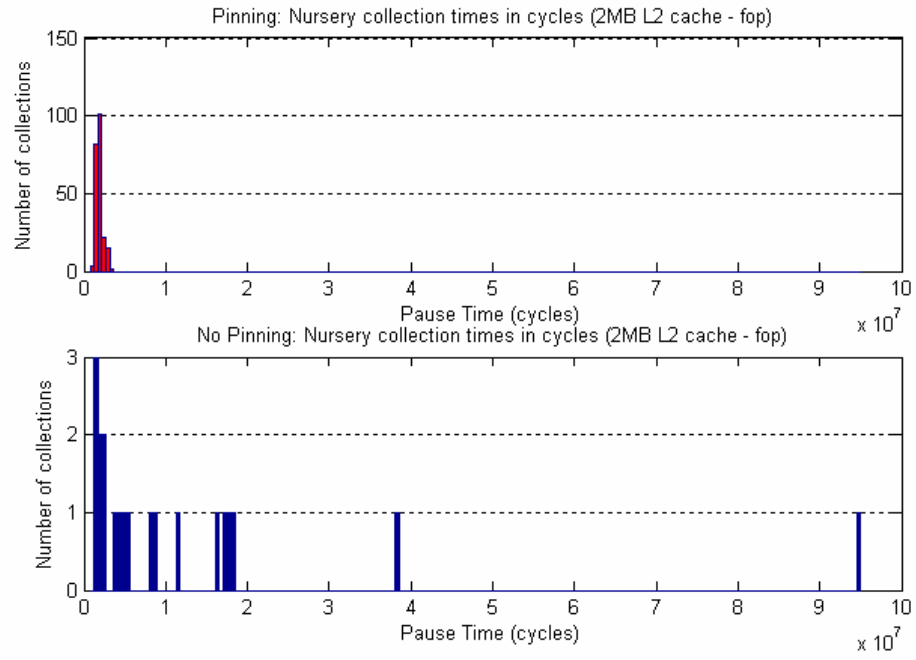

Figure A.10: Histrogram of pause time (2MB L2 cache, fop benchmark)

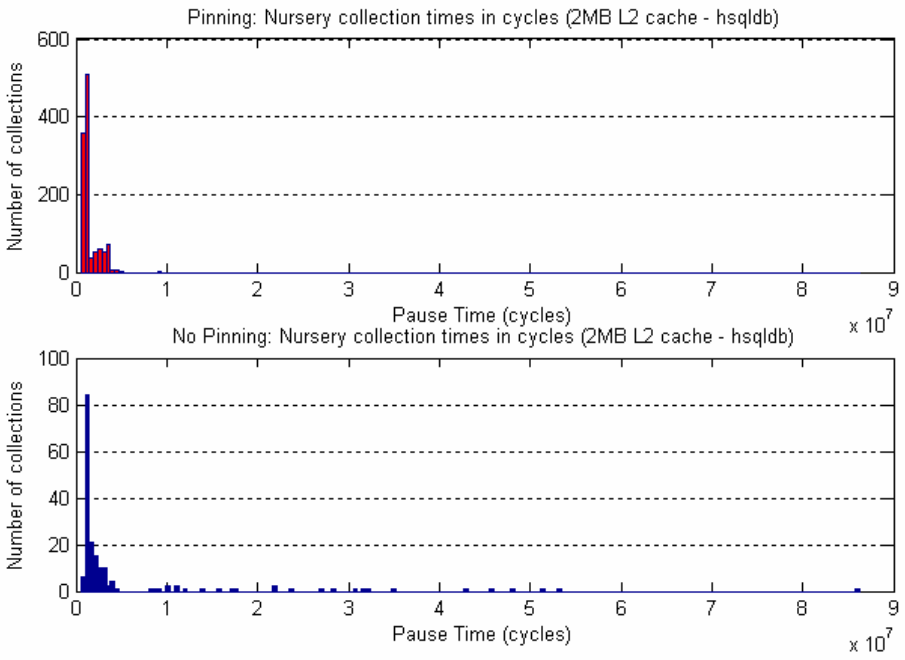

Figure A.11: Histrogram of pause time (2MB L2 cache, hsqldb benchmark)

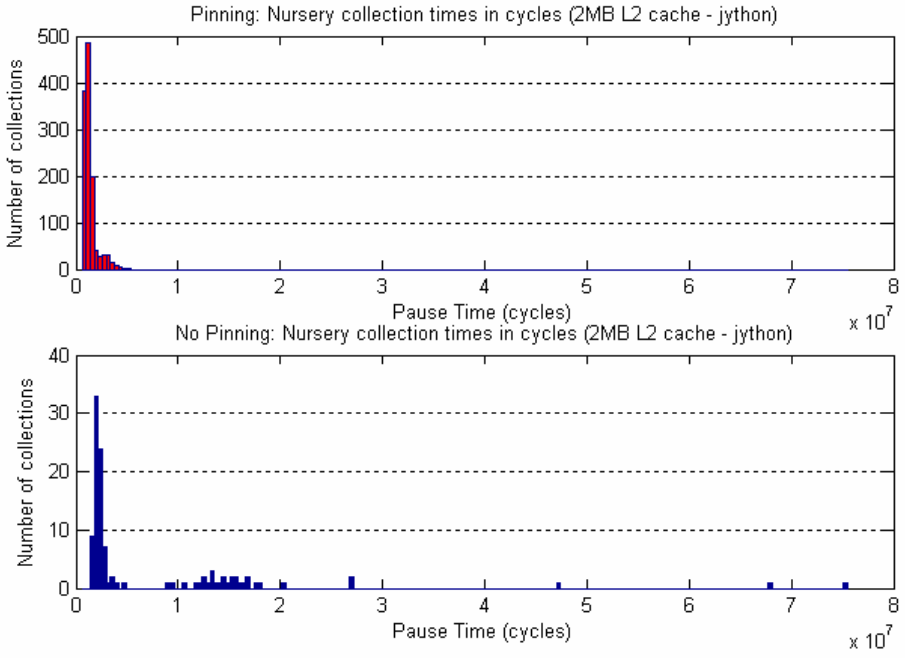

Figure A.12: Histrogram of pause time (2MB L2 cache, jython benchmark)

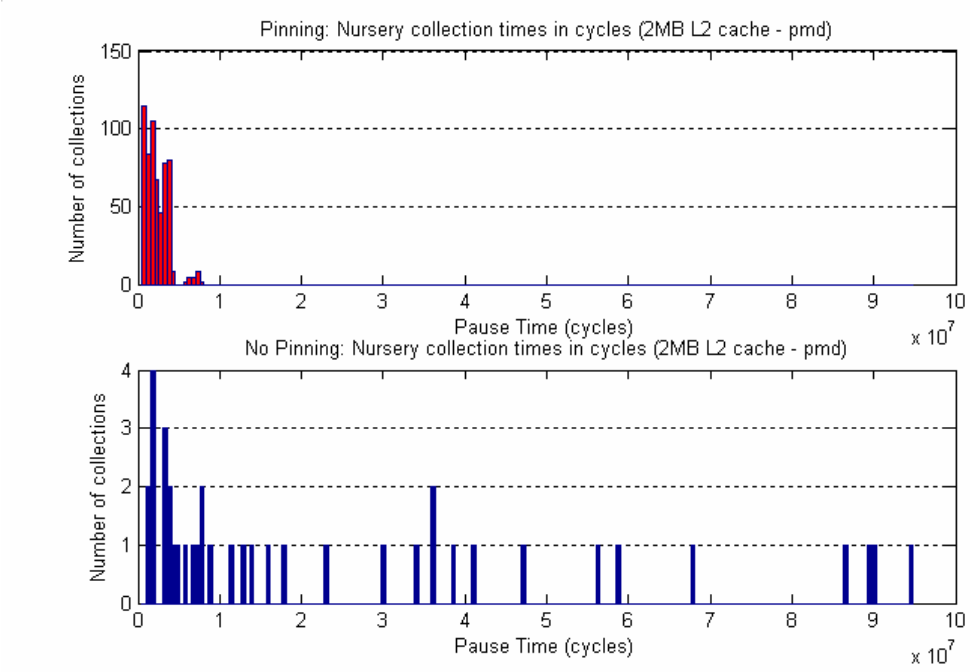

Figure A.13: Histrogram of pause time (2MB L2 cache, pmd benchmark)

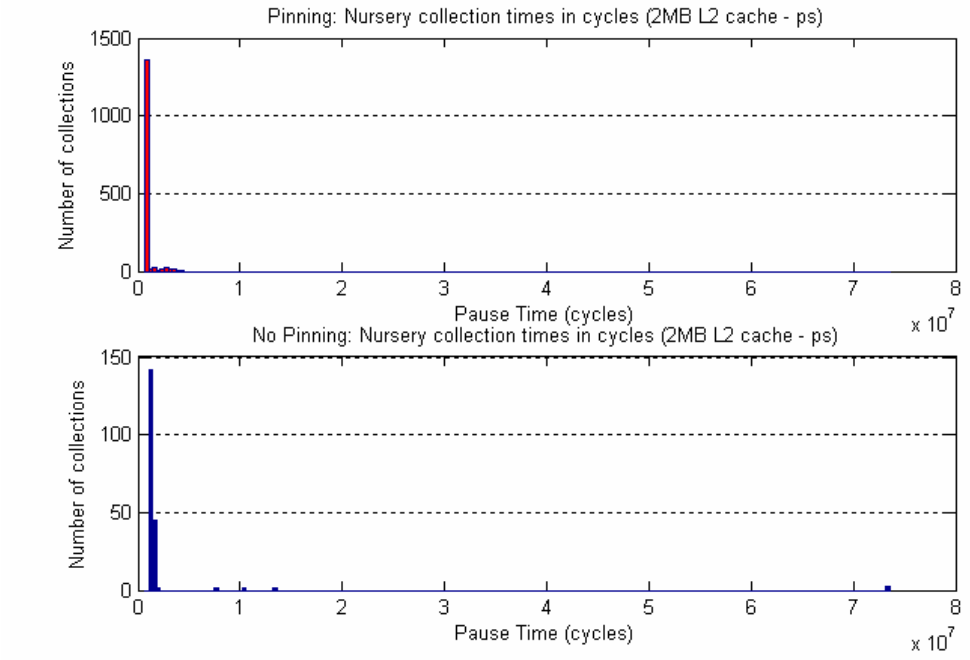

Figure A.14: Histrogram of pause time (2MB L2 cache, ps benchmark)

Appendix B

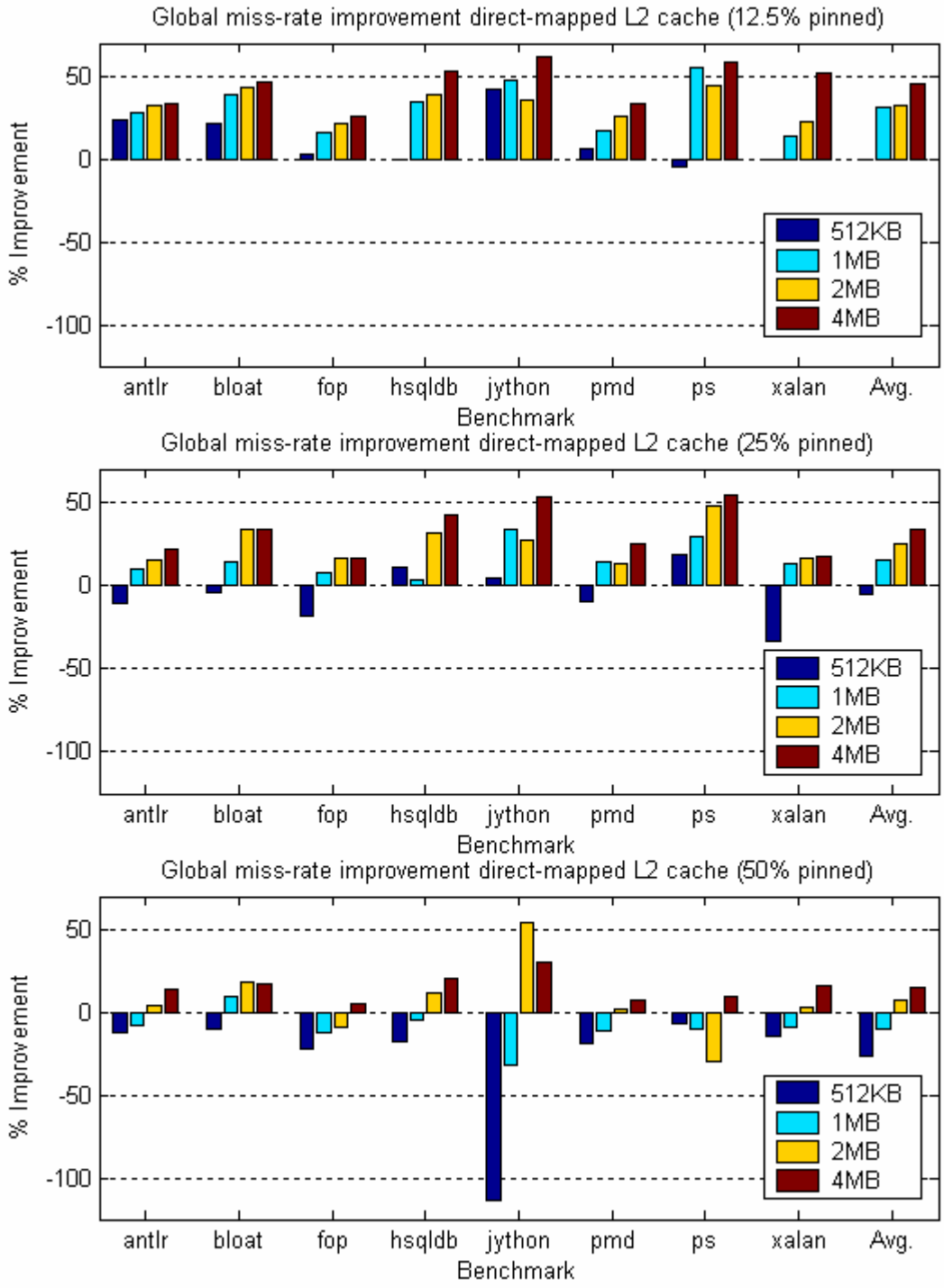

Figure B.1: Comparing global miss-rate improvement for 12.5%,25%, and 50% (direct-mapped)

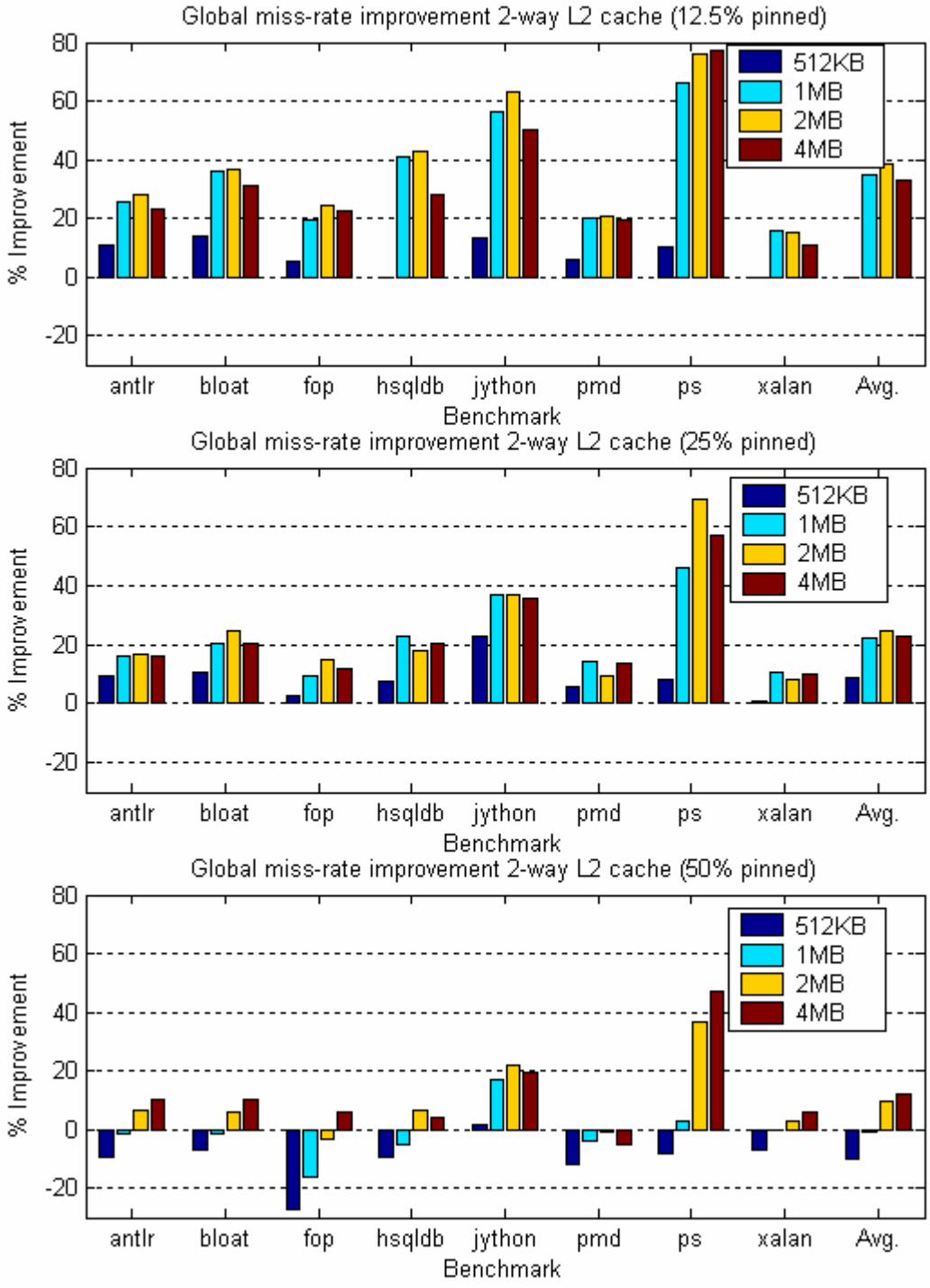

Figure B.2: Comparing global miss-rate improvement for 12.5%,25%, and 50% (2-way)

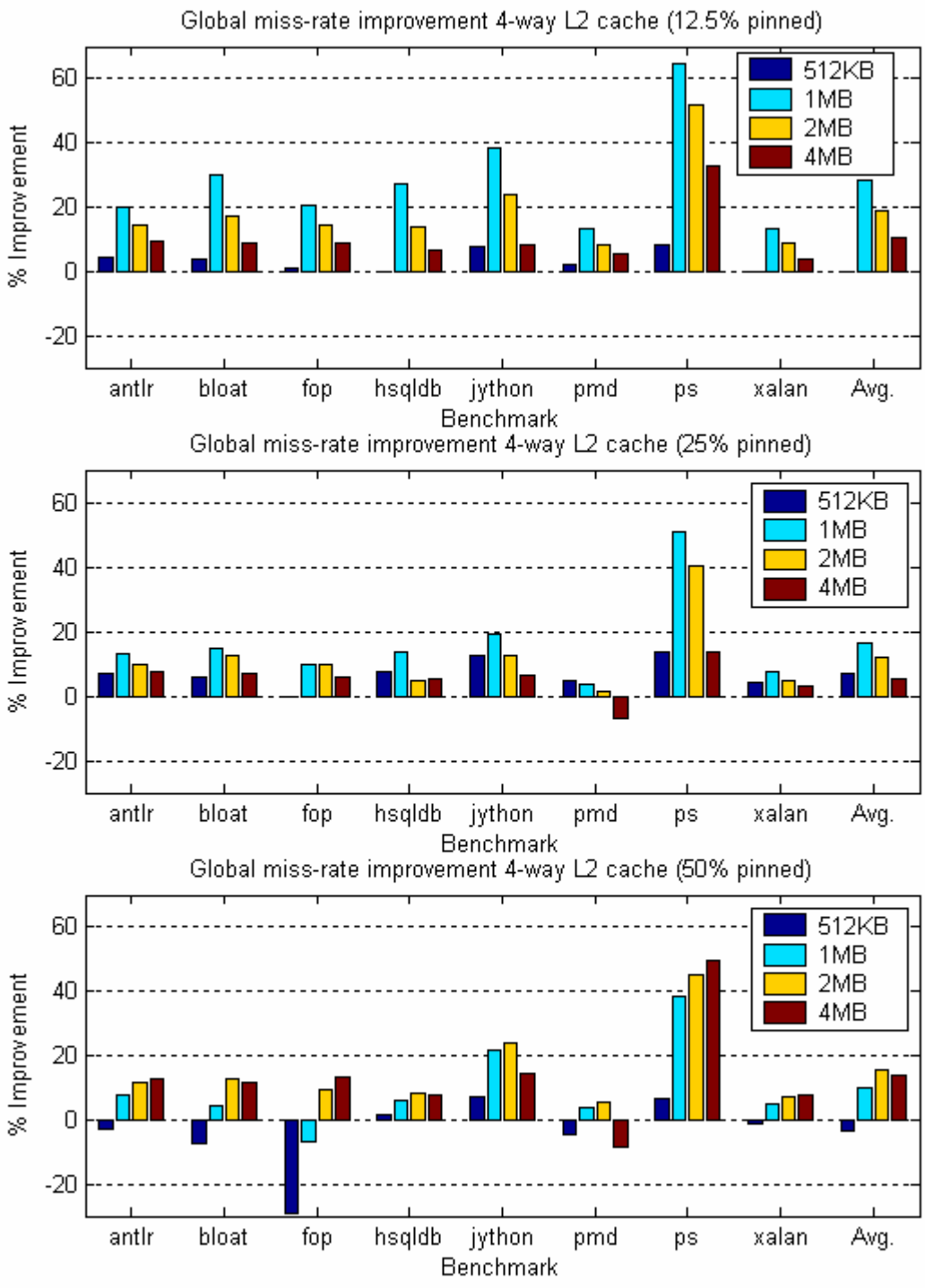

Figure B.3: Comparing global miss-rate improvement for 12.5%,25%, and 50% (4-way)

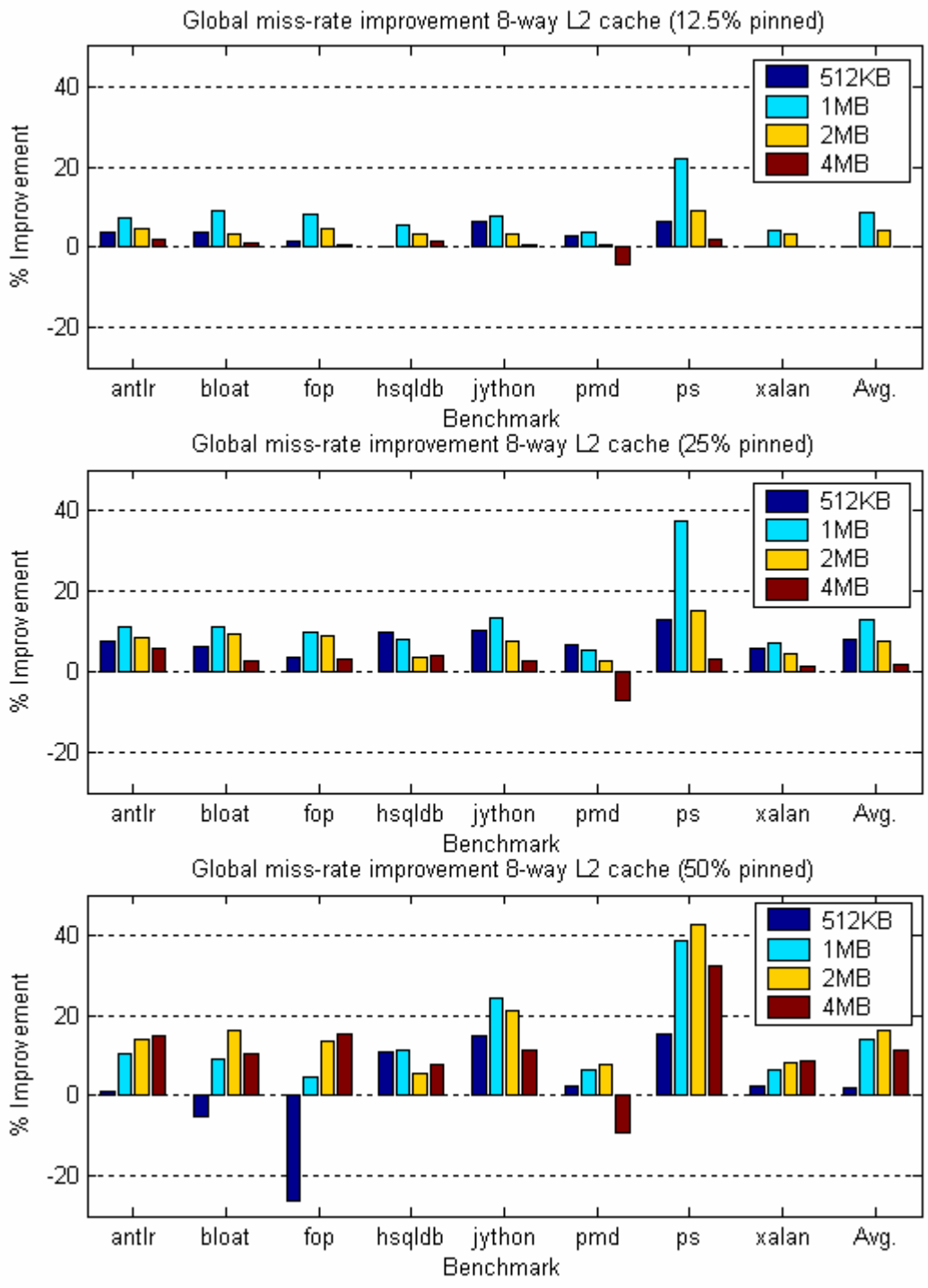

Figure B.4: Comparing global miss-rate improvement for 12.5%,25%, and 50% (8-way)

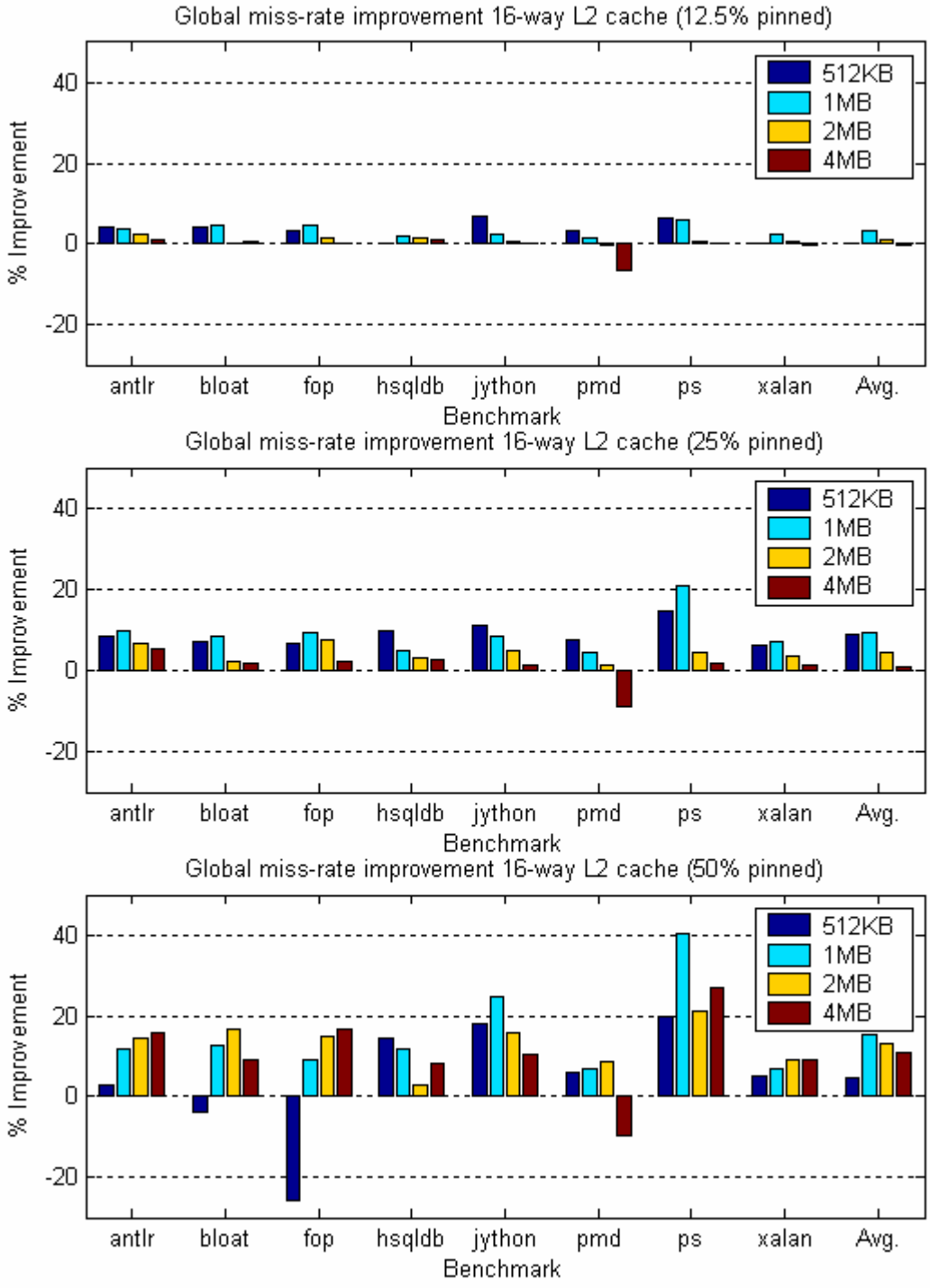

Figure B.5: Comparing global miss-rate improvement for 12.5%,25%, and 50% (16-way)