ABSTRACT

KRISHNAKUMAR, SREE VIDHYA LAKSHMI. Caching Strategies for More Efficient Generational Garbage Collection. (Under the direction of Dr. Edward F. Gehringer)

Until the advent of generational garbage collection, page faults caused by garbage collection were a major source of bottleneck. Generational garbage collectors, by collecting smaller regions of the heap called generations, reduce the memory footprint of the collector and therefore the number of page faults caused by it. With page faults out of the way, the focus now is on cache misses due to garbage collection. The gap between the processor and memory cycle time is widening each year. Projections indicate that this increase is likely to continue. This makes cache performance an attractive area to study in order to tune the performance of a program.

In one such study, a strategy has been proposed to improve cache performance of generational garbage collectors by pinning the youngest generation in the L2 cache. In this thesis, we study an anomaly associated with this strategy, and propose a new realization of the pinning strategy that removes this anomaly, thereby making it more attractive. We apply the idea of an SNT (selectively non-temporal) cache to garbage collection. This helps reduce cache pollution and conflict misses in a direct mapped cache due to non-temporal accesses during garbage collection.

Simulation results show an average miss-rate reduction of 10% for 16 KB and 32 KB direct mapped L1 caches with SNT support. The improvement is greater in benchmarks with a large amount of live data.

Caching Strategies for More Efficient Generational Garbage Collection

by

Sree Vidhya Lakshmi Krishnakumar

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

Department of Computer Science

Raleigh

2004

Approved By:

Dr. Vincent W. Freeh	Dr. Yan Solihin

Dr. Edward F. Gehringer
Chair of Advisory Committee

To Amma, Nayna, Sagar and Prathap

Biography

Sree Vidhya Lakshmi Krishnakumar grew up in a little town in south India, dreamt of becoming a museum curator, and ended up with a degree in Computer Science and Engineering from the University of Madras. She worked for Bosch in Bangalore and Stuttgart before she moved to Raleigh. With the completion of this thesis, she is receiving her Master's in Computer Science.

Acknowledgements

My sincere thanks to Dr. Gehringer for his guidance and support with this work.

Thanks also to Dr. Solihin and Dr. Freeh for being on my thesis committee and for their valuable suggestions.

Thanks to Vimal Reddy for helping me understand prior work done by him in this area.

Thanks to the folks at the IBM Smalltalk group for all their help.

Thanks to Dr. Rotenberg for the condor pool that was used to run the simulations.

Amma, Nayna and Sagar… for everything they are to me.

Grandparents, aunties, uncles, cousins, friends and teachers… for enriching my life.

My friends at 2516 Avent Ferry Rd, Partners I and Li’l Dino… for the good times here.

Prathap, *you* talked me into this. But I forgive you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Memory is a finite resource. If a process does not use it carefully, it will soon run out of the memory allocated to it. Therefore programs need to recycle memory that is no longer in use.

1.1 Storage Allocation

There are three ways in which storage is allocated by programming languages, viz., static allocation, stack allocation and dynamic allocation. Programs often use a mix of these allocation methods. Once the allocated data become dead, the memory locations can be reused. The lifetimes of all static and stack allocated data are uniformly defined, unlike the lifetime of dynamically allocated data. Dynamically allocated data live from the time they are allocated until they are deallocated. This deallocation could be done either explicitly by the programmer in his code or automatically by the language run-time system.

1.2 Dynamic Memory Management and Garbage Collection

With simple programs, dynamic memory can be managed using explicit allocation and deallocation statements. However, as programs grow more and more complex, this task becomes a huge burden on the programmer. It is easy to introduce bugs in the program by prematurely freeing data structures (dangling references) or not freeing them (memory leaks). It is estimated that such memory management and related debugging efforts take up about 40% of programming time [8].

Languages with support for automatic garbage collection identify, during run- time, dynamically allocated data structures that have become unreachable, and deallocate them, so that memory can be reused. Automatic garbage collection thus frees the programmer of the burden of managing dynamic memory and lets him concentrate on the application program.

1.3 Garbage Collection and the Cache

There are, however, overheads associated with garbage collection. They consist of processor overhead associated with executing the extra instructions in order to garbage-collect, and the overhead that the garbage collection process imposes on memory subsystems like caches. Garbage collection is a memory-intensive activity. It interferes with the working set of the program and introduces page faults and cache misses.

Until the advent of generational garbage collection [9, 10], page faults caused by garbage collection were a major source of bottleneck. Generational collectors, by collecting smaller portions of the heap called generations, have reduced the memory footprint of the garbage collector and therefore the number of page faults caused by it. With page faults out of the way, the bottleneck now is cache misses due to garbage collection.

In this thesis we strive to study and explore ways to reduce the cache-system overhead imposed by a copying-style generational garbage collector.

1.4 Motivation

On the one hand, there is the growing importance of garbage collection in modern object-oriented languages. Object orientation is the strongest growing area of interest in analysis, design and programming today. The key to good software engineering is the control of complexity. One of the ways that object-oriented design achieves this goal is the encapsulation of abstractions into objects that communicate through clearly defined interfaces. Programmer-controlled storage management inhibits this modularity. For this reason, most modern object-oriented languages such as Smalltalk, Eiffel, Java and Dylan, use garbage collection. [5] On the other hand, garbage collection suffers from bottlenecks due to memory subsystem overheads.

With the recent trends in the growth of processing power, the gap between the speeds of the processor and memory subsystem is widening each year. Projections indicate that this gap is likely to increase even more in the coming years. Memory performance is one interesting area to look at in order to improve the performance of a program.

1.5 Contribution and Organization of Thesis

In a previous study [6], Reddy proposed a strategy to pin the youngest generation in the L2 cache to improve performance. In this thesis we study an anomaly associated with Reddy's strategy, and propose new realization of the pinning strategy that removes this anomaly.

We show that by making caching decisions based on the non-temporal nature of the garbage collector, we can reduce conflict misses and cache pollution. We introduce a selective non-temporal (SNT) cache and outline the OS/hardware support that would be required to realize this in a real system.

We perform a simulation-based study of our strategies on IBM VisualAge Smalltalk [14], which uses a copying-style generational collector. We believe these results could be extended to any system that uses a similar collector.

The organization of the thesis is as follows. Chapter 2 introduces garbage collection and the mapping strategy that has been proposed to pin the youngest generation in the L2 cache. Our realization of the pinning strategy that removes the L1 miss-rate anomaly associated with this is covered in Chapter 3. Chapter 4 shows how conflict misses and cache pollution due to garbage collection can be reduced using the SNT cache. Chapter 5 discusses the experimental setup, simulator and benchmarks used in this study. Chapter 6 surveys related work, followed by conclusions and future work in Chapter 7.

Chapter 2

Background

This chapter introduces garbage collection in general and the garbage collector used in IBM Smalltalk. It also introduces the mapping strategy [6] proposed by Reddy to pin the youngest generation in the L2 cache.

2.1 Terminology Used

In this thesis, we will be using the following terms.

Root set Root set is a set that contains object pointers from stacks, registers and global (static) variables.

Mutator Following Dijkstra's terminology, the user program is called mutator in order to distinguish it from the garbage collector.

2.2 Garbage Collection Algorithms

There is no single best way to garbage collect. The classical garbage collection algorithms are the reference-counting algorithm, the mark-and-sweep algorithm and the copying algorithm. The collector could be a stop-the-world collector or an incremental/concurrent collector. It could collect the entire heap or smaller portions of the heaps called generations.

The reference-counting algorithm is based on counting the number of references to each object from other active objects. This count is stored in an extra field associated with each object. Each time a pointer is set to refer to this object, its reference count is incremented by one. When a reference to it is deleted, the reference count is decremented by one. When the reference count of an object becomes zero, it indicates that the object is no longer in use and can be reclaimed. Since the reference count is updated during pointer writes by the mutator, reference-counting is a naturally incremental method of garbage collection.

The mark-and-sweep algorithm is a *tracing* method of garbage collection, because an object is not reclaimed as soon as it becomes unreachable. It lies around as garbage typically until there is no more space left for allocation on the heap. When the heap is unable to satisfy a request for allocation, the garbage collector is invoked. The collector begins from the root set, traces all objects that are reachable from it, and marks them. In the next phase, it *sweeps*, or reclaims, all objects in the heap that were not marked during the mark phase. Since this leaves chunks of free space interspersed among memory that is in use, the mark-and-sweep algorithm will cause fragmentation in the heap. This increases allocation cost and also interacts poorly with the virtual memory and the cache. A variant of this algorithm, called the mark-and-compact algorithm, overcomes this problem by compacting the free space.

The copying algorithm is also a tracing collector. It divides the heap into two semi-spaces called *tospace* (containing the current data) and *fromspace* (containing obsolete data). Allocations are always done in tospace. When the heap can find no more free space to satisfy an allocation request, the garbage collector is invoked. It begins collection by flipping the two semi-spaces and then copying all active objects from fromspace into tospace.

Some of the factors to be considered while choosing a particular algorithm are the overall garbage collection time, pause times, allocation costs, overheads imposed on mutator operations (pointer writes in the case of a reference-counting algorithm), space overheads and the effect of the algorithm on the performance of the memory subsystem.

A more detailed treatment of these garbage collection algorithms can be found in the book by Jones and Lins [5]. Also interesting is the survey by Paul Wilson [11]. This study focuses on copying-style generational garbage collection. In the next section we examine in detail the copying garbage collection algorithm, and in the following section we discuss generational garbage collection.

2.3 Copying Garbage Collection

There are many ways to implement a copying garbage collector. Cheney's algorithm, where copying is done iteratively rather than recursively, is the most widely used. The IBM Smalltalk system used in this study uses this algorithm. In this section, we briefly discuss how Cheney's semi-space copying collector [2] works. All allocations are done in tospace.

When tospace has no room left to satisfy a request for allocation, the garbage collector is invoked. At the beginning of a garbage collection cycle, the two semi-spaces are flipped. What was tospace becomes fromspace, and what was fromspace, tospace. After the flip, fromspace is scanned for live objects, which are then copied over to tospace. Once all live objects have been copied , the garbage collection cycle ends, and the mutator resumes.

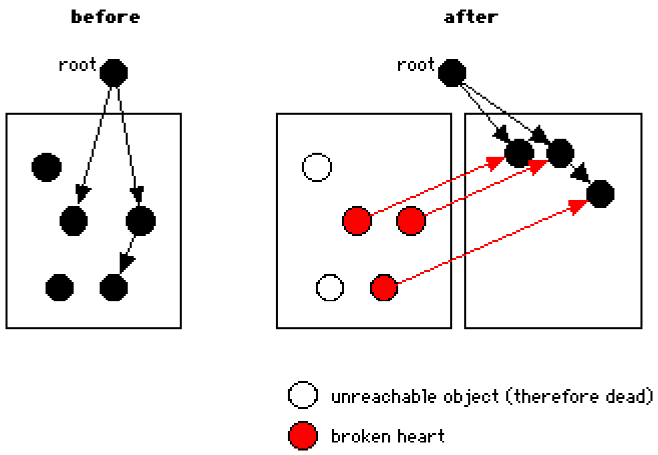


Figure 2. 1 [18] – Semi-space copying collection.

Let us look at the copying process in detail. Cheney's algorithm uses two pointers, called the *free pointer* and the *scan pointer*, to keep track of copying. At the start of a garbage collection cycle, after the spaces are flipped, both the pointers point to the start of tospace. First, the root set is copied to tospace. Then each object in tospace is examined for any pointers it might contain to fromspace. These objects are live and are therefore copied into tospace. The process of examining a tospace object for any pointers to fromspace (and copying those objects into tospace) is called scanning. When all objects in tospace have been scanned, the garbage collection cycle is complete.

As objects are copied into tospace, the free pointer is incremented to point to the first location in tospace that is free for allocation. The scan pointer points to the last object in tospace that has been scanned. Every time a tospace object is scanned, it is incremented to point to the next tospace object. When the scan pointer meets the free pointer, it indicates that there are no more objects left to scan in tospace. Copying is now complete. In order to avoid any redundant copying of objects, every time an object is copied from fromspace to tospace, it leaves a forwarding address (the address of the object in tospace) in fromspace. This way, when an object in tospace points to an already copied object in fromspace, it will not be copied more than once

The advantages of a semi-space copying collector are its ease of implementation and the fact that the heap is compacted at the end of every collection. This avoids fragmentation and reduces allocation cost. It also preserves the locality of the program. The disadvantages are the prohibitive cost of copying large objects, and repeated copying of long-lived objects. Since the semi-space collector touches almost the entire heap, its memory footprint is large, resulting in poor virtual memory and cache behavior.

2.4 Generational Garbage Collection

Some of the problems of the semi-space collector can be addressed by using a generational collector. Generational garbage collection [9, 10] works by dividing the heap into areas called generations, classified mainly by the age of objects residing in them. An object is created in a space called the youngest generation (or the "nursery") and as it gets older, is promoted to older generations. This process is called *tenuring*. A generational garbage collector has several parameters such as the algorithms used for collection, the number of generations, and the tenuring policy.

The idea behind generational garbage collection is to reduce collection effort by segregating large objects and long-lived objects from the mainstream. The assumption is that most objects die young, whereas an object that has survived for a while is likely to live longer. Therefore it is expeditious to focus collection effort on the area of the heap that holds the recently allocated objects (youngest generation). This is because there will be more garbage in the youngest generation and therefore more space will be reclaimed. But it also means that collections will be more frequent. While a semi-space collector can wait until half the heap is filled before it garbage-collects, a generational collector needs to collect every time its youngest generation becomes full. But the advantage is that even though scavenges are frequent, the garbage collection time of a generational garbage collector is less than the collection time of a semi-space collector. This means the mutator does not suffer long pauses.

Also, the footprint of the generational collector is small compared to the semi-space collector. This reduces page faults and therefore the overall execution time.

2.5 Garbage Collection in IBM Smalltalk

The IBM Smalltalk [14] system uses a generational garbage collector, dividing its heap into four regions called *allocate* space, *survivor* space, *old* space and *fixed* space. This is shown in Figure 2.2.

Allocate space is the region of heap where all objects are allocated. Exceptions are large and static objects which are allocated in areas reserved for them. The survivor space is the equivalent of tospace that we saw while discussing garbage collection algorithms. *Survivor* space is the one that flips roles with allocate space during every copying collection cycle. It is the equivalent of fromspace that we discussed earlier. Allocate space and survivor space are collectively known as *New* space. *Fixed* space stores large objects, and designated objects that the system cannot move by a garbage collection because their memory addresses have been passed to the operating system. *Old* space contains objects that are not expected to die soon. Objects that are known to be long-lived are allocated directly in the old space. Objects are also promoted from new space to old space when they are old enough. This way the collector will not waste its effort copying these objects repeatedly.

IBM Smalltalk uses different collection schemes for *scavenges* and *global garbage collections*. Every time allocate space becomes full, the garbage collector collects new space. This is called *scavenging*. A copying garbage collection algorithm is used for scavenging, which copies live objects from survivor space to allocate space. An object's age is the number of scavenges that it has survived. It is kept track of by incrementing it after each scavenge.

The tenuring policy used by the collector determines the age at which an object should be promoted to old space. The rationale behind this is that objects that survive a given number of scavenges are likely to live longer. When old space becomes full, the garbage collector collects the entire heap. This is called *global garbage collection*.

IBM Smalltalk uses a mark-and-sweep collector for global garbage collection. The size of the generations can be controlled by command-line parameters to IBM Smalltalk.

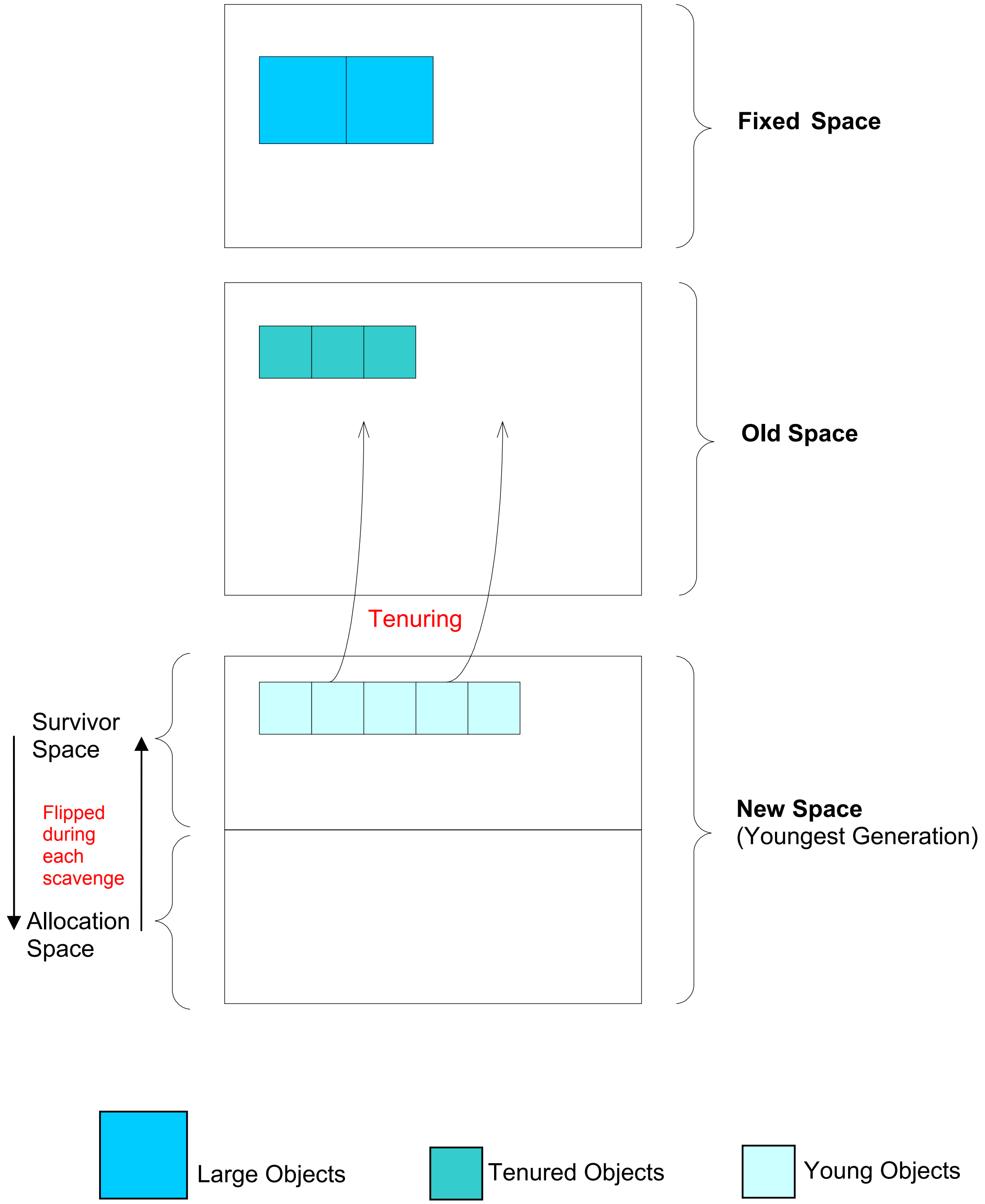


Figure 2. 2 [6] – Generational heap used in IBM Smalltalk.

2.6 Pinning Strategy

In this section we discuss the mapping strategy [6] used by Reddy in order to reduce cache misses due to garbage collection. The idea is to *pin* the youngest generation in the L2 cache; that is, arrange the virtual-to-physical memory mapping so that there are no conflict misses to lines from the youngest generation. We assume that we have an L2 cache that is large enough to hold the youngest generation. A portion of the cache is reserved for the youngest generation, and the page-fault handler is made aware of this. The page-fault handler identifies the page frames in physical memory that map to the reserved portion of the cache and makes sure that only pages from the youngest generation are placed in these frames. This prevents conflicts to the youngest generation in the cache, so it stays pinned.

Figure 2.3 describes a possible implementation in a real system. On startup, the Smalltalk system requests the operating system to reserve space in the cache for its youngest generation, passing the heap address range of its youngest generation as a parameter. Based on this, and on the cache and physical-memory parameters, the operating system determines the page frames in the physical memory that map to the reserved portion of the cache. These page frames are used exclusively to hold pages from the youngest generation. Let us call any contiguous region in physical memory that maps to the reserved portion of the cache a *bucket*. A bucket might consist of one or more pages. Of all the buckets in the physical memory that map to the reserved portion of the cache, Reddy's strategy selects a few to hold the youngest generation. This is done in such a way that there are no conflicts between youngest generation addresses in the cache.

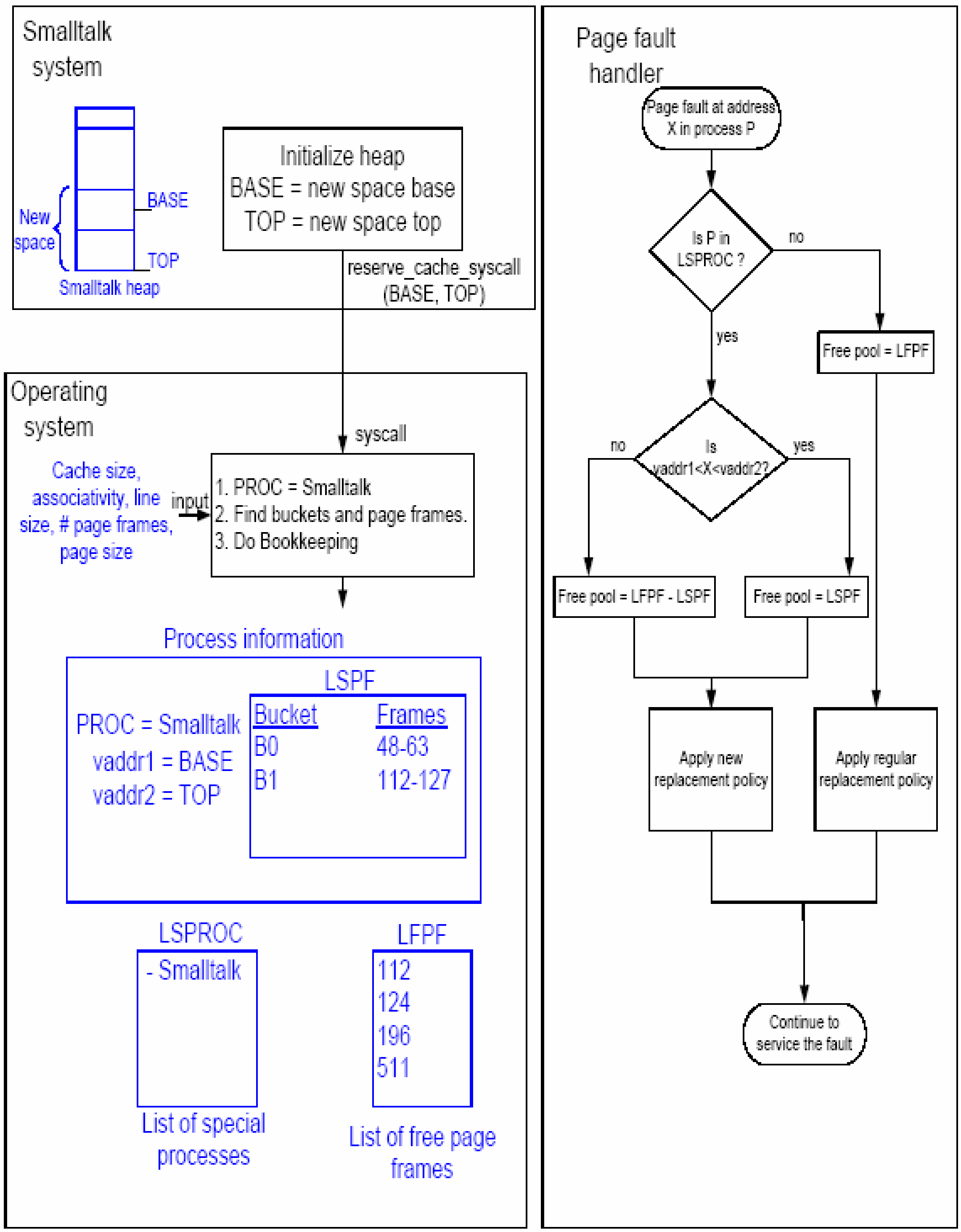


Figure 2. 3 [6] – An implementation of the mapping strategy in a real system.

The operating system maintains a list of *special processes* that have these special reservation requests. For each special process, it stores the address range which needs to be pinned in the cache, and also which page frames are exclusive to this address range.

For special processes, when the page-fault handler services a request for a page, if the address falls in the special address range, it is placed in one of the page frames reserved for it in physical memory. For all other pages, any frame from the free pool may be used, except for the reserved page frames. This ensures that the youngest generation is available in the L2 cache throughout program execution.

Simulation results show that this strategy reduces L2 miss rates by an average 45% for direct-mapped L2 and 15% for 2-way set-associative L2 caches. The gains decrease gradually with increasing associativity of the L2 cache. This is because associativity itself diminishes conflict misses. Hence there is less room for improvement by pinning the youngest generation in the cache. As for misses in the L1 cache, this strategy did not have much effect when a direct-mapped L2 was used. With an associative L2, however, L1 performance degraded considerably.

Figures 2.4 and 2.5 show the drop in L1 miss rate for a two-way set-associative L2 and four-way set-associative L2 respectively. In this thesis, we address this L1 miss-rate anomaly and present a new realization of the pinning strategy that will not degrade L1 performance.

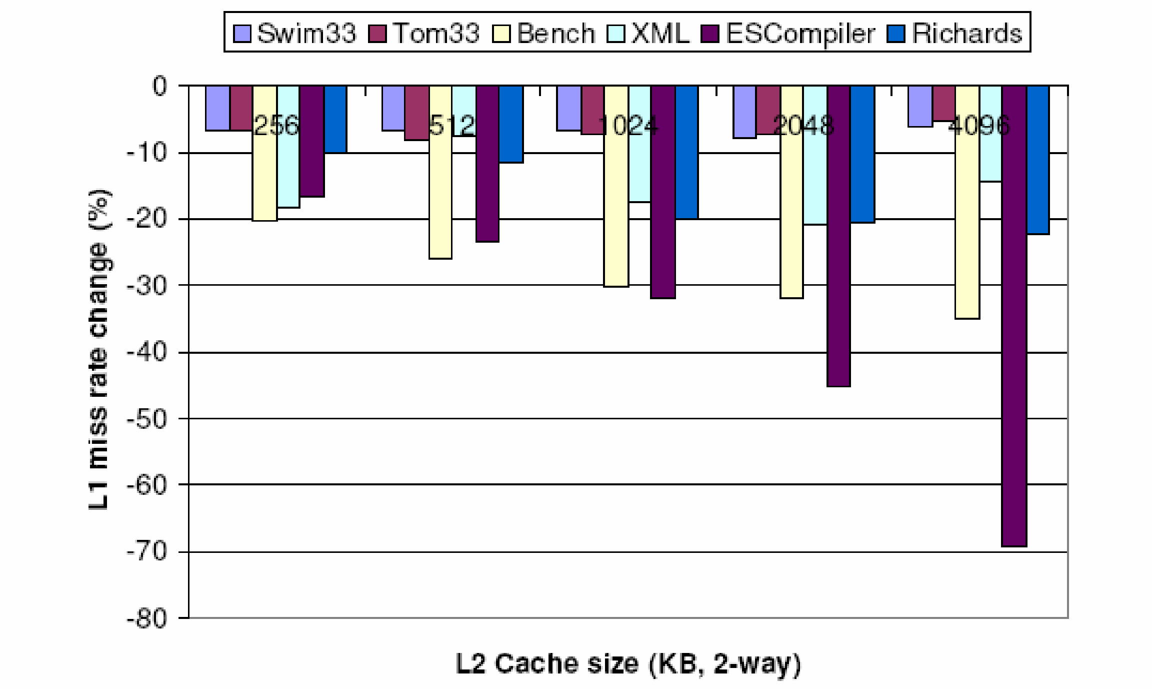


Figure 2. 4 [6] – L1 miss-rate change in a 2-level cache hierarchy with direct-mapped L1 and 2-way set-associative L2 cache when the pinning strategy was applied to the L2 cache, for all benchmarks.

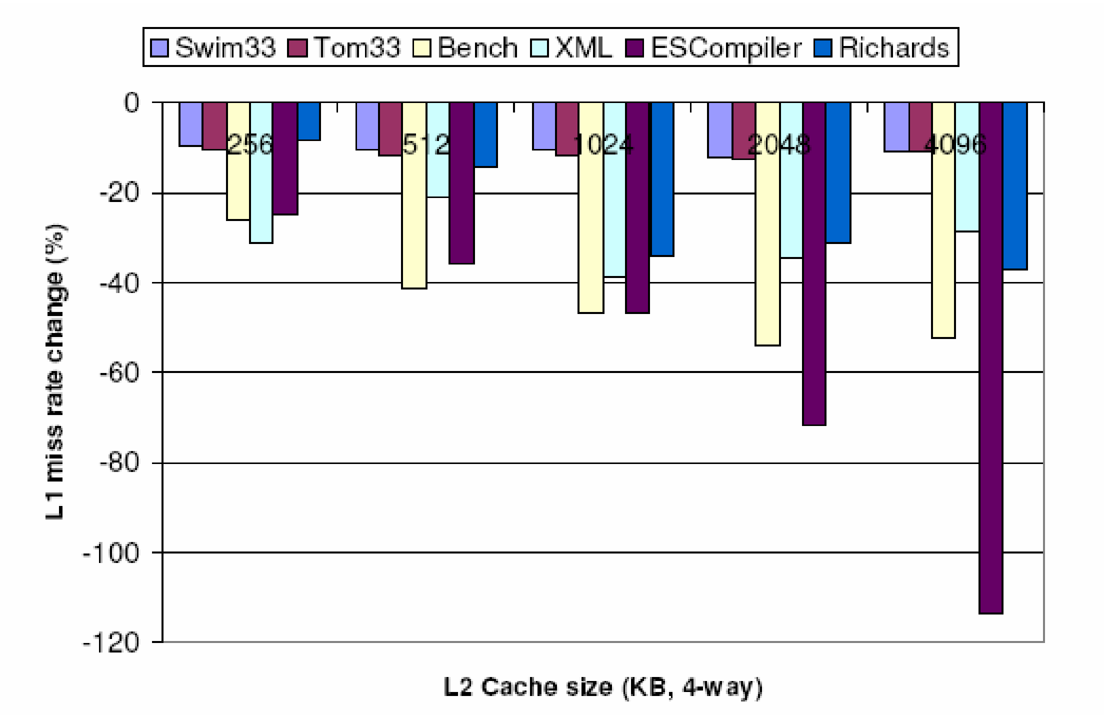


Figure 2. 5 [6] – L1 miss-rate change in a 2-level cache hierarchy with direct-mapped L1 and 4-way set-associative L2 cache when the pinning strategy was applied to the L2 cache, for all benchmarks.

Chapter 3

Reducing L1 Misses Due to Pinning

This chapter addresses the anomaly in the L1 miss rates seen in the study by Reddy [6], and proposes a new realization of the pinning strategy that removes this anomaly. Reddy's algorithm pins the youngest generation in the L2 cache. Intuitively, we expect this to improve the L2 hit rate and not affect the L1 hit rate. Results show that this holds good for direct-mapped L2 caches. But with set-associative L2 caches, pinning was seen to bring down the L1 hit rate.

3.1 Analysis

In order to understand this effect seen with set-associative L2 caches, let us see how the youngest generation was pinned in Reddy's study. Figures 3.1 and 3.2 illustrate the way the youngest generation was pinned in a direct-mapped L2 and a two way set-associative L2 respectively. The figures show details at the cache-line level for a line size of 32 bytes. In the figures, each line is qualified by three parameters, $S(x)$, $L(y)$ and $A(z)$. $S(x)$ indicates that the line falls in the x^{th} *set* of the cache. $L(y)$ indicates that the line is the y^{th} *line* within its set. $A(z)$ indicates that a block of 32 addresses in the youngest generation starting at virtual *address z* have been mapped to that cache line.

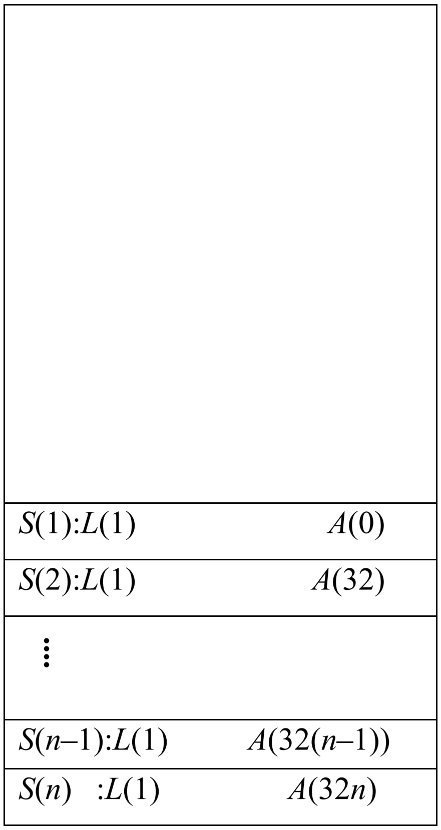


Figure 3. 1 – Reddy's layout of youngest generation addresses in a direct-mapped L2.

Line-level details shown. *S(x)* stands for set *x* of the cache, *L(y)* for line *y* within the set and *A(z)* for a block of 32 addresses in the youngest generation starting at virtual address *z*.

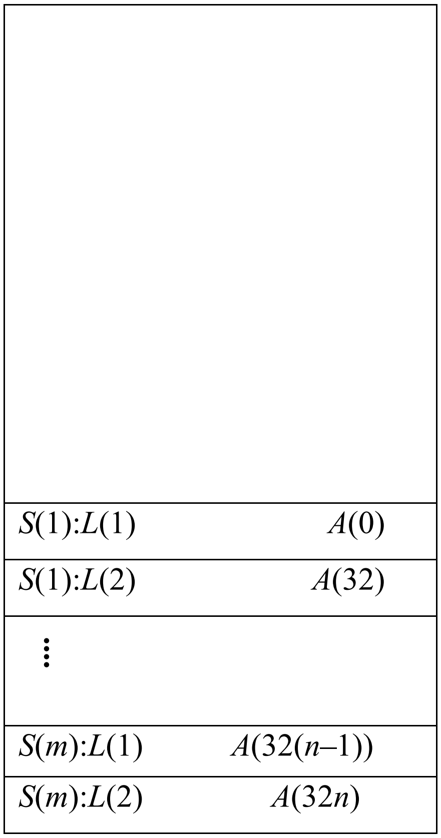


Figure 3. 2 – Reddy's Layout of youngest generation addresses in a 2-way set-associative L2. Line-level details shown. $S(x)$ stands for set x of the cache, $L(y)$ for line y within the set and $A(z)$ for a block of 32 addresses in the youngest generation starting at virtual address z.

We see that both for the direct-mapped and set-associative caches, the youngest generation addresses are laid out contiguously in the L2 cache. Let us examine the effect of this layout on an L1 cache. Assume a memory subsystem with a two-level cache hierarchy with the following parameters.

L1: 16 KB, direct-mapped, line size 32 bytes.

L2: 256 KB, two-way set-associative, line size 32 bytes.

Physical memory: 16 MB (# address bits = 24).

Figures 3.3 and 3.4 show the distribution of address bits for tag, index and offset in the L1 and L2 caches, respectively.

Tag	Index	Offset
23 14	13 5	4 0

Figure 3. 3 – Distribution of address bits for a 16 KB direct-mapped L1.

Tag	Index	Offset
23 17	16 5	4 0

Figure 3. 4 – Distribution of address bits for a 256 KB 2-way set-associative L2.

We see that the bits in L1's index field are a subset of the bits in L2's index field. For L2, index *x* refers to set *x* of the L2 cache. This being a two-way set-associative cache, addresses falling in either two lines of a set have the same index field. Since L1's index field is a subset of L2's index field, these two lines have the same index in L1. The L1 cache is direct mapped (one line per set). Therefore all lines of an L2 set map to the same line (i.e., conflict) in L1.

Due to the nature of garbage collection, tospace sees a sequential access pattern during garbage collection and during allocation when the mutator resumes. Since the youngest generation and therefore tospace are laid out contiguously in the L2 cache, the first two accesses tospace are mapped to the first line of L1, the next two to the second line, and so on. Thus we have a scenario where every other line that is fetched replaces the line that was fetched just before it. This leads to poor performance in L1 cache.

3.2 New Realization of the Pinning Strategy

In order to overcome this problem, we need to lay out the youngest generation in the L2 cache in such a way that it "spreads" over the L1 cache. That is, we map consecutive lines in physical memory to consecutive *sets* in the cache. Thus, one line will not map to the same set as the next line; indeed, the same set will not be reused until all sets have been used to cache a line.

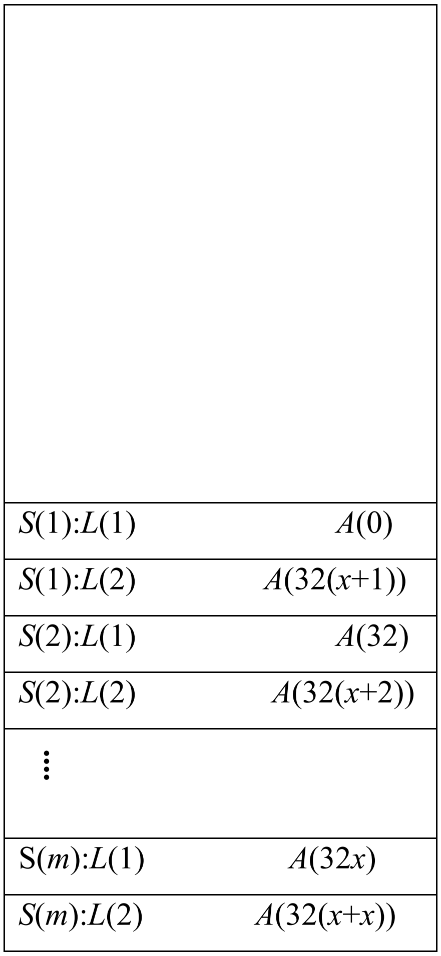

$S(1)$:$L(1)$ $A(0)$
$S(1)$:$L(2)$ $A(32(x+1))$
$S(2)$:$L(1)$ $A(32)$
$S(2)$:$L(2)$ $A(32(x+2))$
⋮
S(m):$L(1)$ $A(32x)$
$S(m)$:$L(2)$ $A(32(x+x))$

Figure 3. 5 – Pinning the youngest generation in a 2-way set-associative L2 cache with minimal L1 conflicts.

Figure 3.5 illustrates our method of arranging the YG addresses in the L2 cache. Not only does this prevent an increase in L1 miss rate, but this is the only realizable method to pin the youngest generation in L2. Let us see what was unnatural in Reddy's study about the layout of the youngest generation in the L2. With a 64 KB youngest generation and a page size of 4 KB, tospace occupies about 8 page frames in physical memory. Let us see how a single page in physical memory maps to the cache.

For ease of illustration, let us assume a mini-cache with a total of four sets and two lines per set, and a line size of two bytes. Let us also assume a miniature page (to match our miniature cache) of size 16 bytes. For the addresses that fall on one such page, figure 3.6 shows the distribution of the address bits for tag, index and offset in our mini-cache.

We see that not more than two (the line size) consecutive addresses share the same index. Therefore to have consecutive addresses of a page in the youngest generation fill up the lines of a set contiguously is unnatural. While this could be realized in a simulated environment, it would not be possible to implement such an algorithm in real life without changing the fundamental way in which either paging or cache mapping works.

T	I	O
0	00	0
0	00	1
0	01	0
0	01	1
0	10	0
0	10	1
0	11	0
0	11	1
1	00	0
1	00	1
1	01	0
1	01	1
1	10	0
1	10	1
1	11	0
1	11	1

Figure 3. 6 – Distribution of tag, index, and offset bits for addresses on a page.

3.3 Results

We performed simulation-based studies to test these hypotheses. The methodology for all simulations in this thesis is discussed in Chapter 5. The simulation-based study

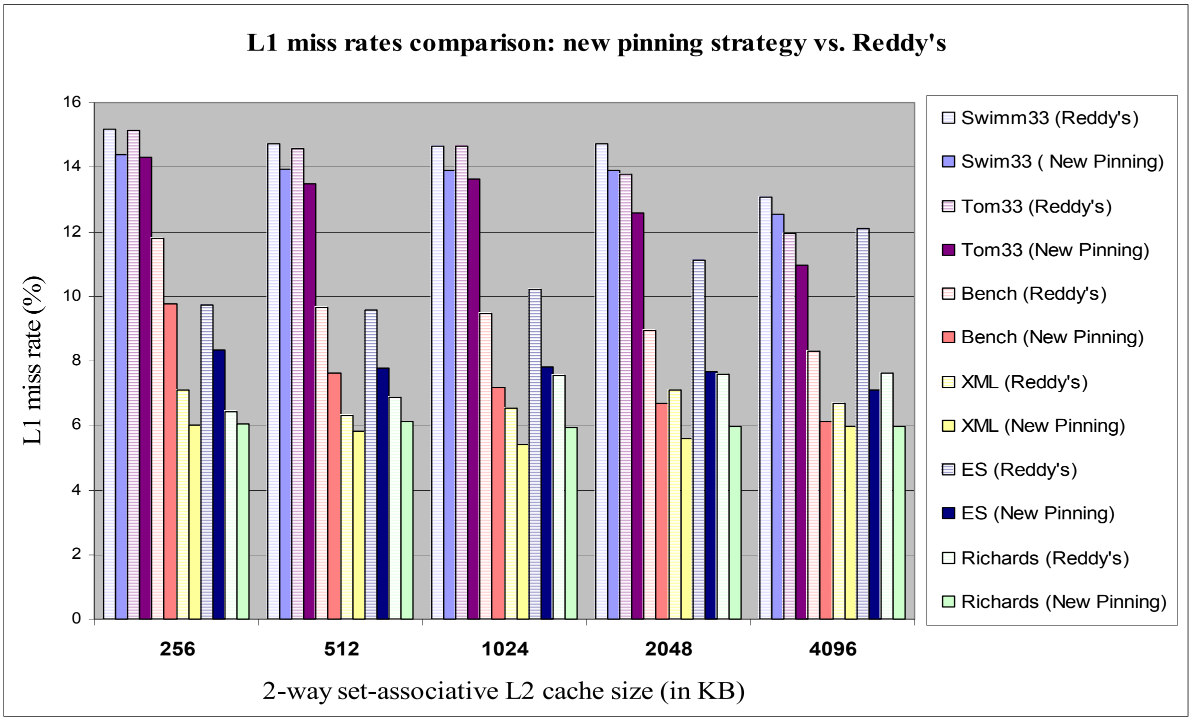


Figure 3. 7 – L1 miss-rate comparison (new pinning strategy vs. Reddy's) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *2-way* set-associative L2 cache.

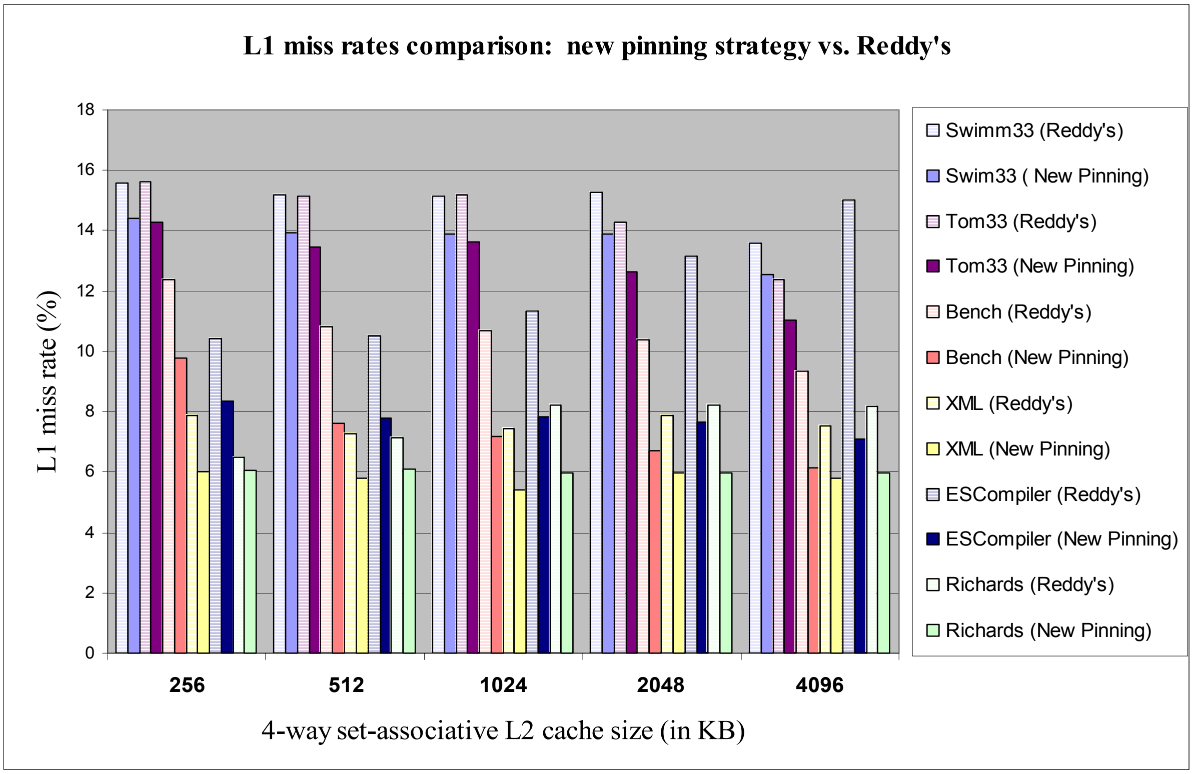


Figure 3. 8 – L1 miss-rate comparison (new pinning strategy vs. Reddy's) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *4-way* set-associative L2 cache.

confirms our belief. Figures 3.7 and 3.8 show a comparison of the L1 miss rates for new pinning strategy and Reddy's. We see that all benchmarks show a reduced miss rate with the new pinning strategy. This improvement makes the L1 miss rates of the new pinning strategy comparable to that of the base case (where the youngest generation was not pinned to L2). Figures 3.9 and 3.10 show this. Figure 3.9 shows a comparison of L1 miss rates obtained with the new pinning strategy and the base case for various sizes of two way set-associative L2 cache. Figure 3.10 compares the L1 miss rates of the new pinning strategy and the base case for various sizes of four way set-associative L2 cache. We see that the miss rates are comparable.

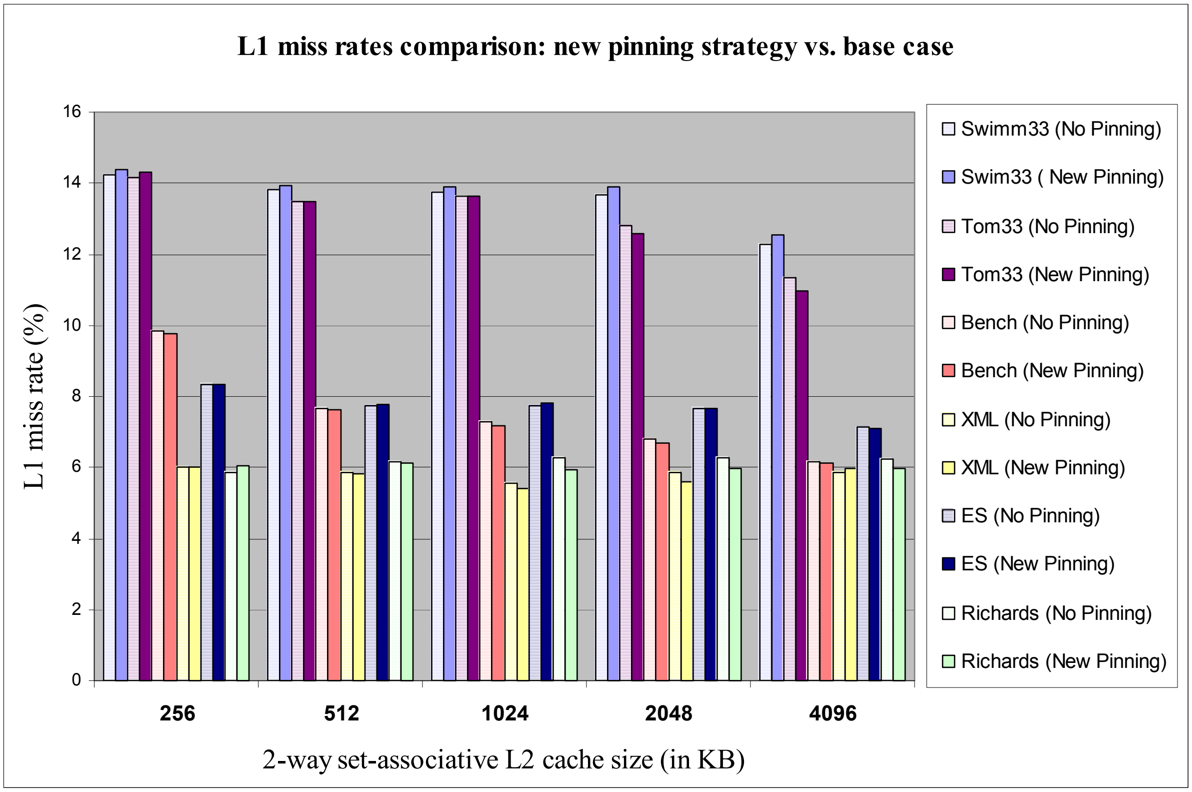


Figure 3. 9 – L1 miss-rate comparison (new pinning strategy vs. no pinning) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *2-way* set-associative L2 cache.

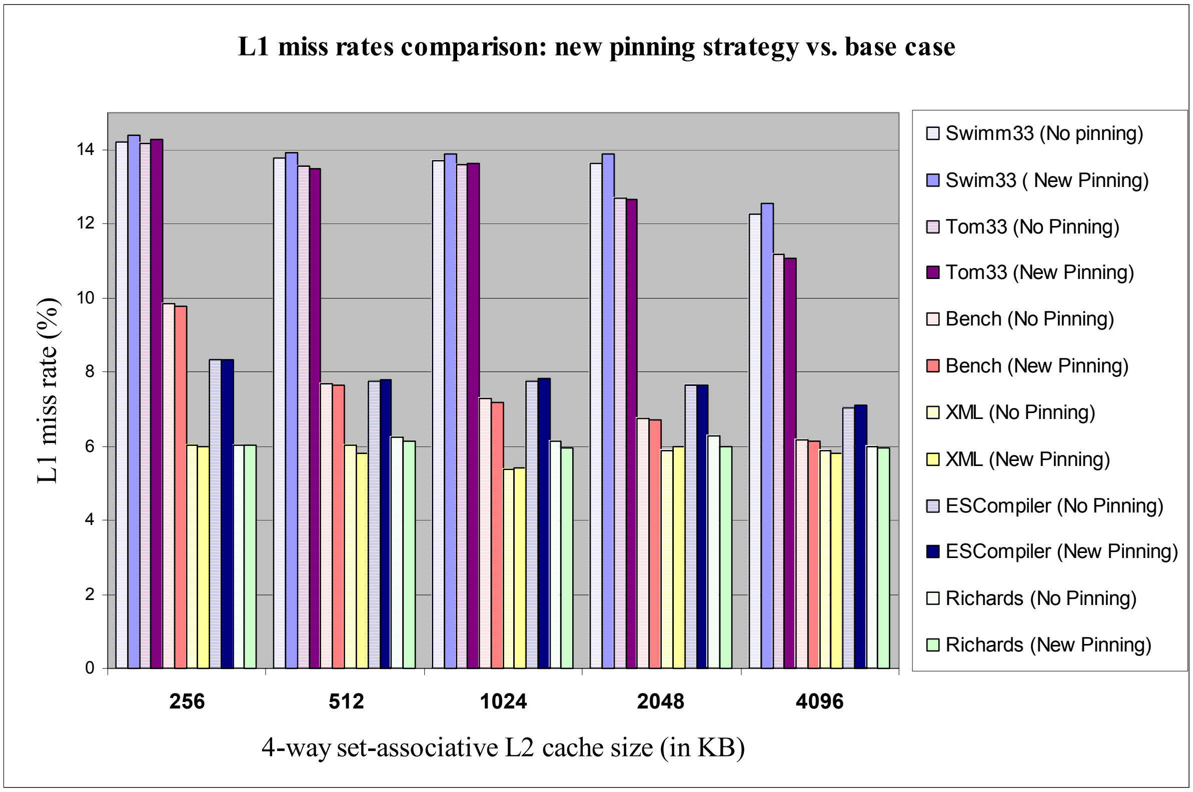


Figure 3. 10 – L1 miss-rate comparison (new pinning strategy vs. no pinning) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *4-way* set-associative L2 cache.

While the new pinning strategy improves the performance of L1, it does not degrade the performance of L2. Since the L1 hit rates are different for the two strategies, the number of accesses going to L2 is different and it would therefore be unfair to compare just the L2 miss rates of the two strategies. Hence we look at the global miss rates. Figures 3.11 and 3.12 show a comparison of the global miss rates obtained using Reddy's strategy and the new pinning strategy. We see that the global miss rate obtained using either strategy is almost the same for all benchmarks.

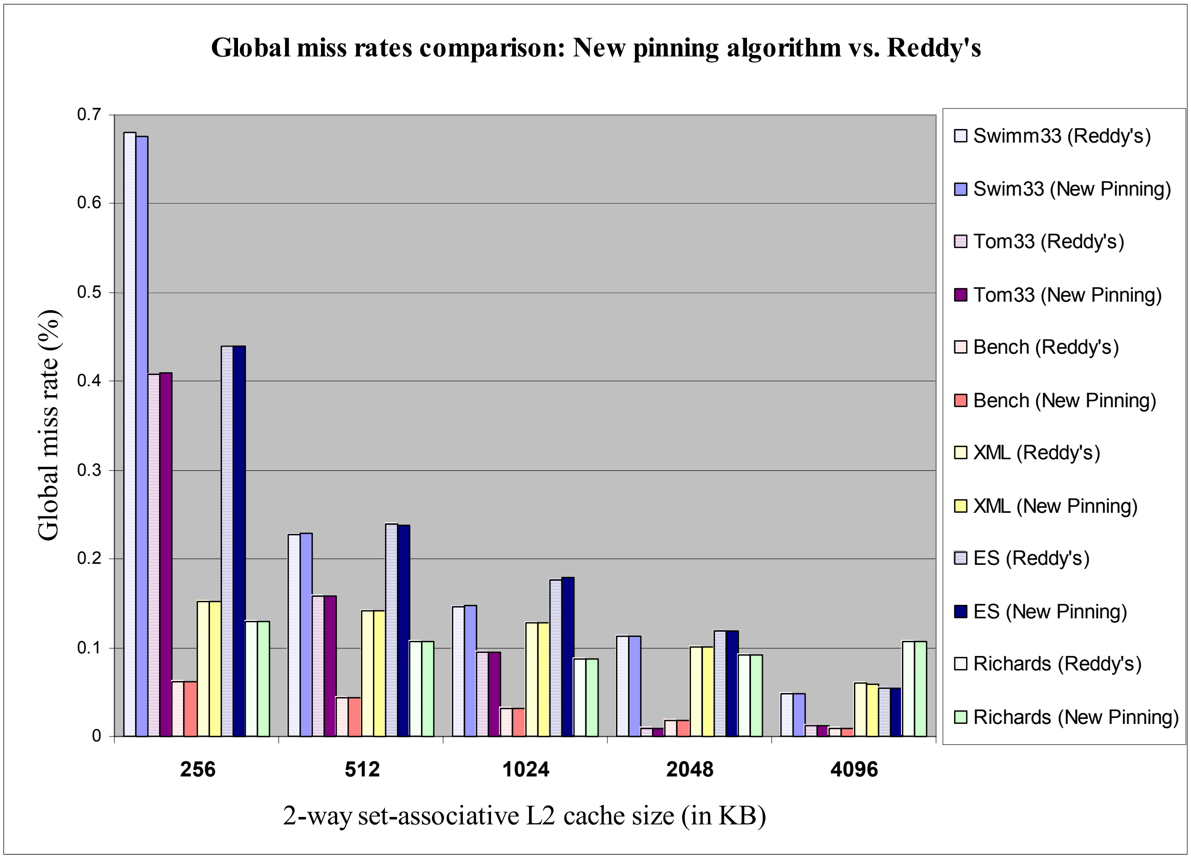


Figure 3. 11 – Global miss-rate comparison (new pinning strategy vs. Reddy's) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *2-way* set-associative L2 cache.

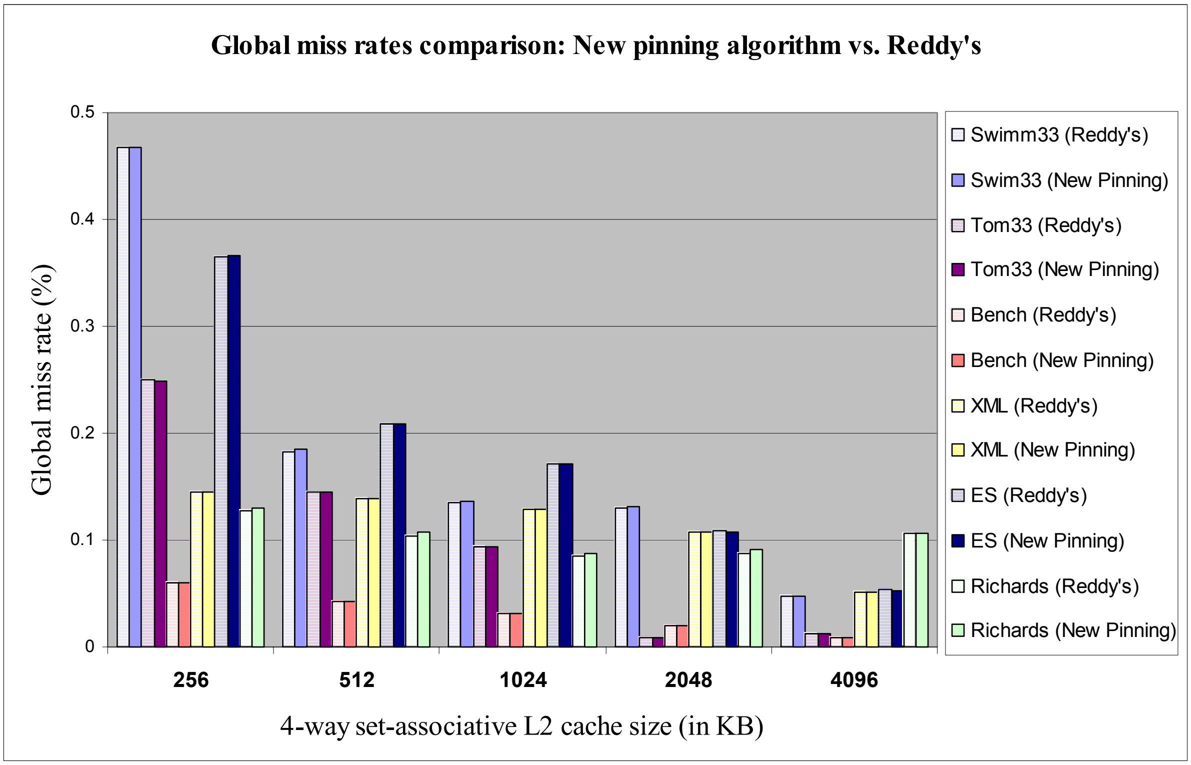


Figure 3. 12 – Global miss-rate comparison (new pinning strategy vs. Reddy's) in a 2-level cache hierarchy with direct-mapped L1 cache (16 KB) and *4-way* set-associative L2 cache.

3.4 Summary

Thus we see that the new realization of the pinning algorithm reduces L1 miss rates, making it comparable to that of the base case (with no pinning) while still maintaining the improvement in global miss rate seen with Reddy's strategy.

Chapter 4

Improving Hit Rates in Direct-Mapped Caches

Caches are designed to make replacement decisions that are reasonable for most programs. Because they have to operate on a wide variety of applications, they do not assume any knowledge of what lines will be referenced in the future, except that lines that have not been referenced recently probably will not be referenced soon again. However, if a program is performing generational garbage collection, we *do* have information on what lines will *not* be referenced in the future. Let us see if we can use this information to increase the hit rate of garbage-collected programs in direct-mapped caches.

4.1 Garbage Collection Overheads

The time spent by a program on garbage collection depends on factors like its rate of allocation, the size of the youngest generation etc. Studies by Reddy [6] show that for large data sets, the total time taken for garbage collection (scavenges + global garbage collection) can be greater than the mutator execution time. This can be seen in the cases of Tomcatv and Swim for grid sizes greater than 100×100 and youngest generation sizes of 4 MB and less. It is also seen that scavenge time makes up a far greater fraction of the total garbage collection time than global garbage collection. This is because the scavenger is invoked more frequently than the global garbage collector. With scavenges taking up a good chunk of the total garbage collection time, we want to make sure that the scavenger works well with the cache.

4.2 Assumptions of the Cache and Nature of the Scavenger

Let us see how the nature of the copying algorithm used by the scavenger interacts with the assumptions of the cache. Caches work on the principle of temporal and spatial locality. During a scavenge, accesses are made to fromspace and tospace.

- Tospace: Accesses are sequential in tospace during garbage collection. They exhibit high spatial locality. There is not much temporal locality during scavenge. But tospace blocks cached by the scavenger may be used by the mutator once it resumes. Assuming conservatively that tospace blocks accessed during scavenging stay in the cache until the mutator accesses them, we say they exhibit temporal locality.

- Fromspace: Accesses are generally to random locations in fromspace (to wherever objects in tospace point). For objects that span several words, when a word in fromspace is accessed, the words following it are likely to be accessed. Thus, scavenging exhibits short bursts of spatial locality. Temporal locality is poor. Fromspace addresses will not be accessed again until the next scavenge cycle when it plays the role of tospace. Given the typical size of an L1 cache and a youngest generation, we can assume that a fromspace address accessed during a scavenge cycle will rarely live in the cache long enough to see the next scavenge cycle when it will be accessed again. That is, these blocks exhibit no temporal locality.[1]

Thus we see that the assumptions of the conventional cache do not hold for fromspace accesses during scavenge cycles.

4.3 Nature of Conflicts in a Direct-Mapped Cache

Conflict misses are inevitable in a direct-mapped cache. But not all blocks that are fetched to replace others are useful. If we can categorize the blocks in the cache as temporal and non-temporal, with a temporal block being one whose words are accessed at least once during its lifetime in the cache after being fetched and a non-temporal block being one whose words are never accessed after the first time, then a conflict between two temporal blocks would lead to a useful fetch. However, if a non-temporal block conflicts with a temporal block, it “might” lead to a useless fetch. That is, if no other words of the block will be accessed (no spatial locality) it would have been just as fast to reference the word from main memory, without bringing it into the cache. To handle conflicts between non-temporal and temporal lines, we introduce a “selectively non-temporal” (SNT) cache. The SNT cache,

[1] Fromspace lines are not strictly non-temporal. See section 4.8

based on inputs from a process turns off temporal locality for select regions of its address space.

4.4 Best Candidate for Replacement During Scavenge

We earlier characterized fromspace lines as being non-temporal and burstily spatial. During a scavenge cycle, any conflicts of fromspace blocks with tospace blocks will be a conflict between a non-temporal and temporal block. However this does not call for entirely banning fromspace blocks from the cache. This is due to fromspace addresses exhibiting spatial locality, albeit in bursts. It seems like we would still have to fetch fromspace blocks into the cache. But it also seems that during a scavenge cycle, fromspace blocks are the best candidates to be replaced in the cache. With fromspace blocks being evicted from the cache as soon as they have been accessed, they could be confined to a limited area of the cache (say a line) so that they do not pollute the entire cache. This would eliminate conflict misses between fromspace and tospace blocks. Doing this ensures that we will be evicting blocks that we know for sure are not going to be used. This does not, of course, guarantee that the blocks we keep will be accessed again. We expect this to improve the performance of the scavenger (due to elimination of conflicts between fromspace and tospace blocks) as well as the mutator (since the non-temporal fromspace blocks have not polluted the cache)

4.5 Operating System and Hardware Support for SNT

During scavenge cycles we want to constrain fromspace blocks to a particular line in the cache. We will call this the non-temporal (NT) line. This will require support for selective non-temporality from the hardware and OS.

Let us say that fromspace falls in the virtual memory heap from address *n* to address *m*. At the beginning of a scavenge cycle, a Smalltalk process would request using a syscall that temporal locality be turned off for fromspace addresses in the L1 cache.

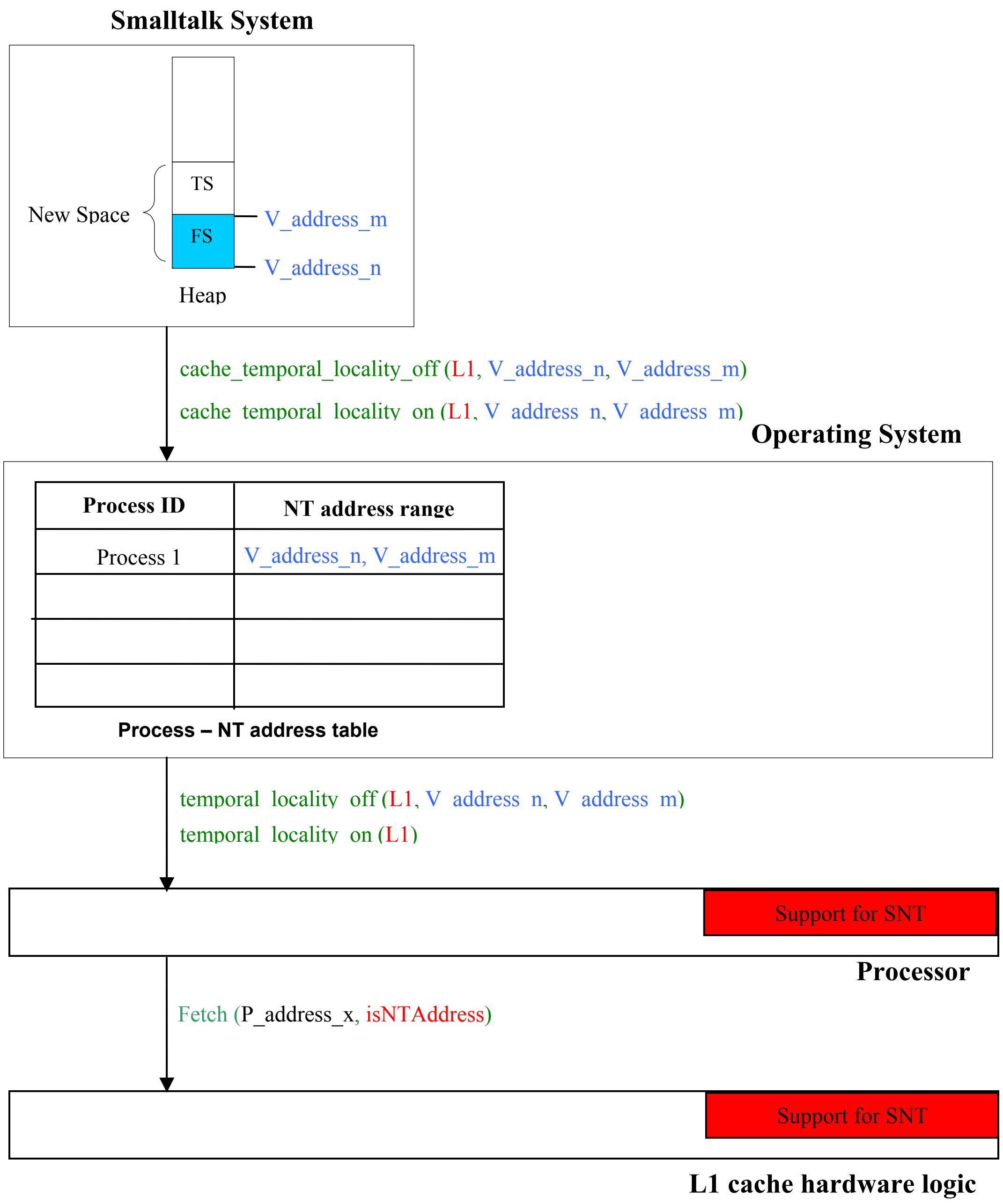
cache_temporal_locality_off(L1, *n*, *m*)

The operating system would maintain an NT address range associated with its processes. This syscall would cause the virtual addresses *n* and *m* to be marked by the operating system as NT range for the Smalltalk process that sent the request. The operating system would also indicate to the processor that temporal locality is to be turned off in the L1 cache for NT addresses. When temporal locality is turned off, every time the processor requests the cache to fetch a physical address that falls within the NT (virtual) address range, it indicates that it is a non-temporal address.

At the end of a scavenge cycle, a Smalltalk process would use a syscall to request that temporal locality be turned on again for fromspace addresses in the L1 cache.

cache_temporal_locality_on(L1, *n*, *m*)

This would cause the operating system to remove the corresponding NT address range associated with the requesting Smalltalk process. The operating system would also indicate to the processor hardware that temporal locality has been turned on for L1 cache. When temporal locality is turned on, the processor indicates that the address is temporal for all fetch requests to the cache.


Smalltalk System
New Space
TS
FS
V_address_m
V_address_n
Heap
cache_temporal_locality_off (L1, V_address_n, V_address_m)
cache_temporal_locality_on (L1, V_address_n, V_address_m)
Operating System
Process – NT address table
temporal_locality_off (L1, V_address_n, V_address_m)
temporal_locality_on (L1)
Support for SNT
Processor
Fetch (P_address_x, isNTAddress)
Support for SNT
L1 cache hardware logic

Process ID	NT address range
Process 1	V_address_n, V_address_m

Figure 4. 1 – OS and hardware support for SNT.

4.6 Cache Lookup Algorithm

The cache hardware would need to use a modified lookup algorithm in order to support SNT. This section describes the way the lookup algorithm should work for direct-mapped caches.

An address would first be looked up in the cache line it would normally have been mapped to. On a cache hit, all goes on normally. In the case of a cache miss, if the processor has indicated that the address is temporal, all goes on normally as well. If the processor has indicated that the address is non-temporal, then on a cache miss, the addresses should be looked up in the NT line. In the case of a miss there, the block should be fetched into the NT line.

What is fromspace during a scavenge cycle, is what was tospace during the mutator cycle preceding it. Therefore chances are some of fromspace blocks that we try to access during a scavenge cycle is already in the cache. Our experiments confirmed this. Just because they are not found in the NT line, we do not want to fetch fromspace blocks that are already in the cache once more. Checking an extra cache line is cheaper than fetching from the next level in the memory hierarchy. This is the reason for first looking up an NT address in the cache line it regularly would map to. This step could have been bypassed if our accesses were to write. But most of the accesses to fromspace during a scavenge cycle are read accesses.

The following flow-chart shows the modified cache lookup algorithm that would be used in a direct-mapped SNT cache.

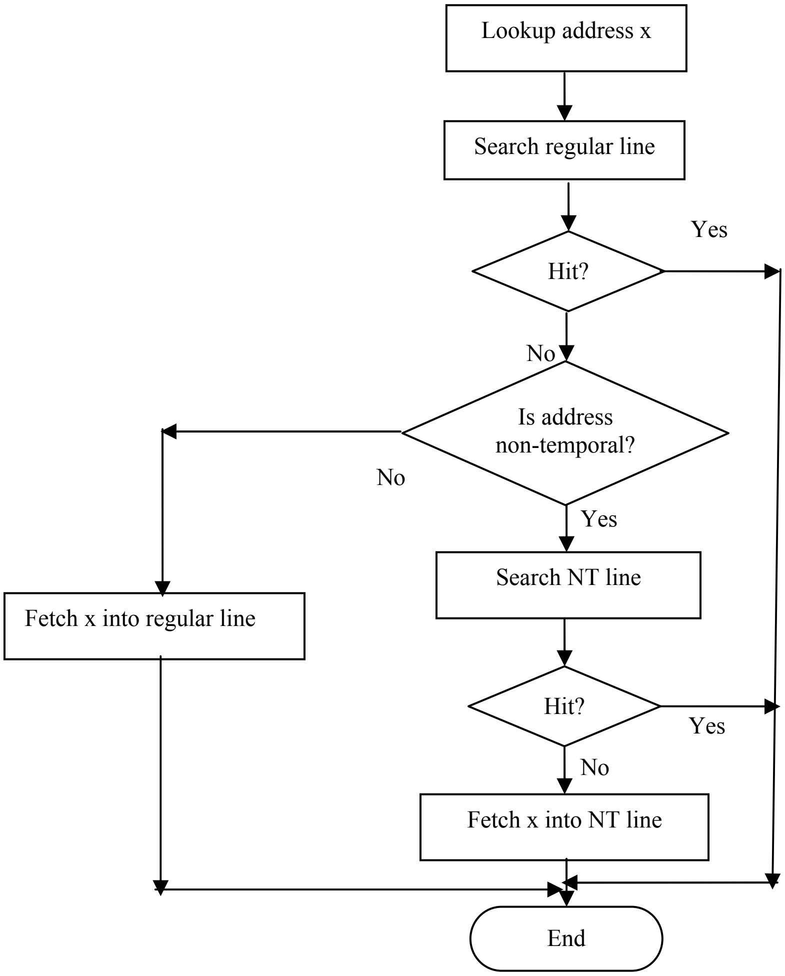


Figure 4. 2 – Lookup algorithm of a direct-mapped SNT cache.

4.7 Initial Results

For a 16 KB direct-mapped L1 cache, the first line of the cache was allotted as the NT line. It is to be noted here that *allotting* a line for NT addresses (fromspace addresses here), does not exclude addresses outside of fromspace being fetched into it. As the cache lookup algorithm indicates, temporal addresses are fetched into the line they normally map to, even if they map into the NT line. Making the allotted line of the cache exclusive to the NT addresses would introduce complications to the cache lookup algorithm.

Experiments were performed where, during scavenge cycles, all fromspace addresses were constrained to the NT line. The experiments were performed for various youngest generation sizes in the range of 64 KB to 1 MB. The results far from showing a reduction in the miss rates, showed an increase in miss rates for benchmarks in many cases.

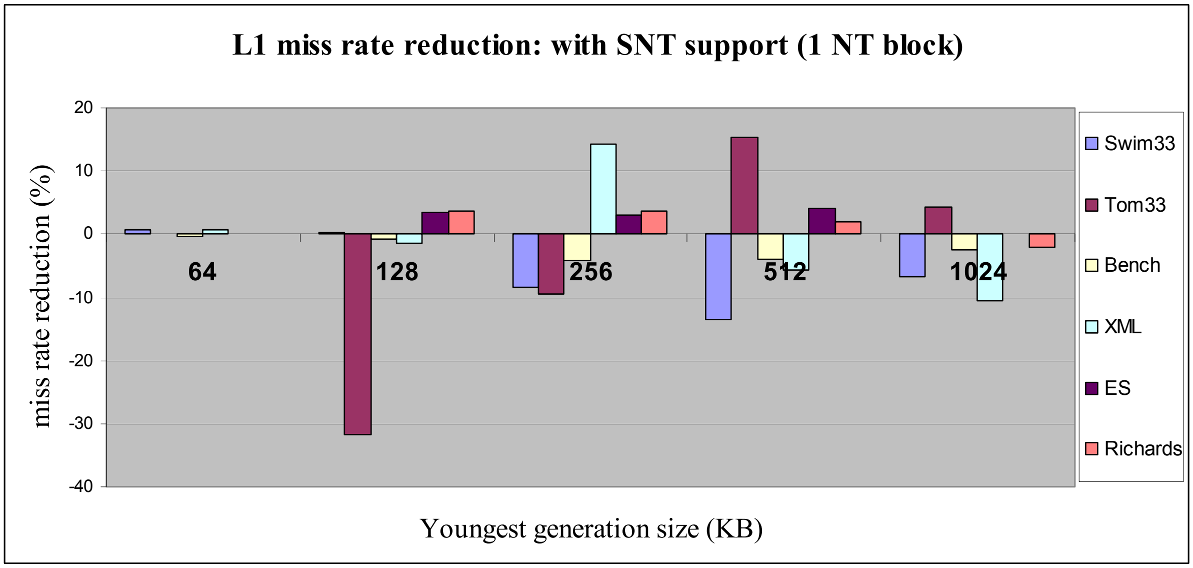


Figure 4. 3 – Miss-rate reduction in a 16 KB direct-mapped cache with SNT support during scavenge, where fromspace blocks were constrained to a single NT line in cache.

4.8 Analysis

In order to detect which part of the program suffered due to constraining NT blocks, we isolated the miss statistics for the scavenger and the mutator portions of the program. (The statistics for global garbage collection were not isolated. They were counted as part of mutator statistics, since no constraining was done during global garbage collection). As expected, the mutator portion of the program showed a slight improvement in hit rate. But the cache performance during scavenges was seen to deteriorate heavily. To further get to the root of the problem, we isolated the number of references and misses to fromspace and tospace during scavenges. It was seen that the accesses to fromspace suffered heavy misses, while accesses to tospace showed an improvement. The improvement in accesses to tospace was due to the elimination of conflicts from fromspace.

Constraining fromspace blocks to the NT line of the cache improved the hit rates seen by the mutator and tospace during scavenges. The overall negative results were due to the heavy increase in miss rates seen by fromspace during scavenges. Data showed that write accesses to fromspace were affected the most. The writes to fromspace occur when the collector writes the forwarding address of an object right after it has been copied to tospace. Thus a fromspace object that is accessed for copying needs to be retained in the cache until the forwarding address is written into it by the collector. For objects that span more than one cache line, the object is fetched into the cache in chunks as large as the line size. Since we have a single NT line allotted for fromspace objects, each block that is brought in, replaces the one that was brought in previously. Thus, after the object is being copied, when the collector tries to write the forwarding address in the object, the line to be written to is no longer in the cache. This is due to aggressive eviction of fromspace objects from the cache. We also see here that fromspace blocks are not strictly non-temporal. They are accessed just twice, and in close temporal proximity. To accommodate this, a less aggressive eviction policy should be used for fromspace blocks, giving them just enough time to live in the cache to be accessed for the second and last time. This would require allotting more than one NT line in the cache. The number of lines to be allotted would depend on the typical size of the objects used in a program.

4.9 Results

Simulations were performed that constrained fromspace addresses to four NT lines in the cache. Within the NT lines, FIFO replacement was used. This allowed for a lenient eviction of fromspace objects. As expected, this allowed objects to be alive in the cache until they were required again shortly following their first access. Fromspace accesses did not suffer this time and there was improvement in the mutator as well as scavenger portions of the program. Experiments were performed for all benchmarks with 16 KB and 32 KB direct-mapped L1 caches. The youngest generation sizes were varied from 64 KB to 1 MB. The miss-rate reduction shown in figure 4.4 is the maximum improvement seen among the various youngest generation sizes between this range.

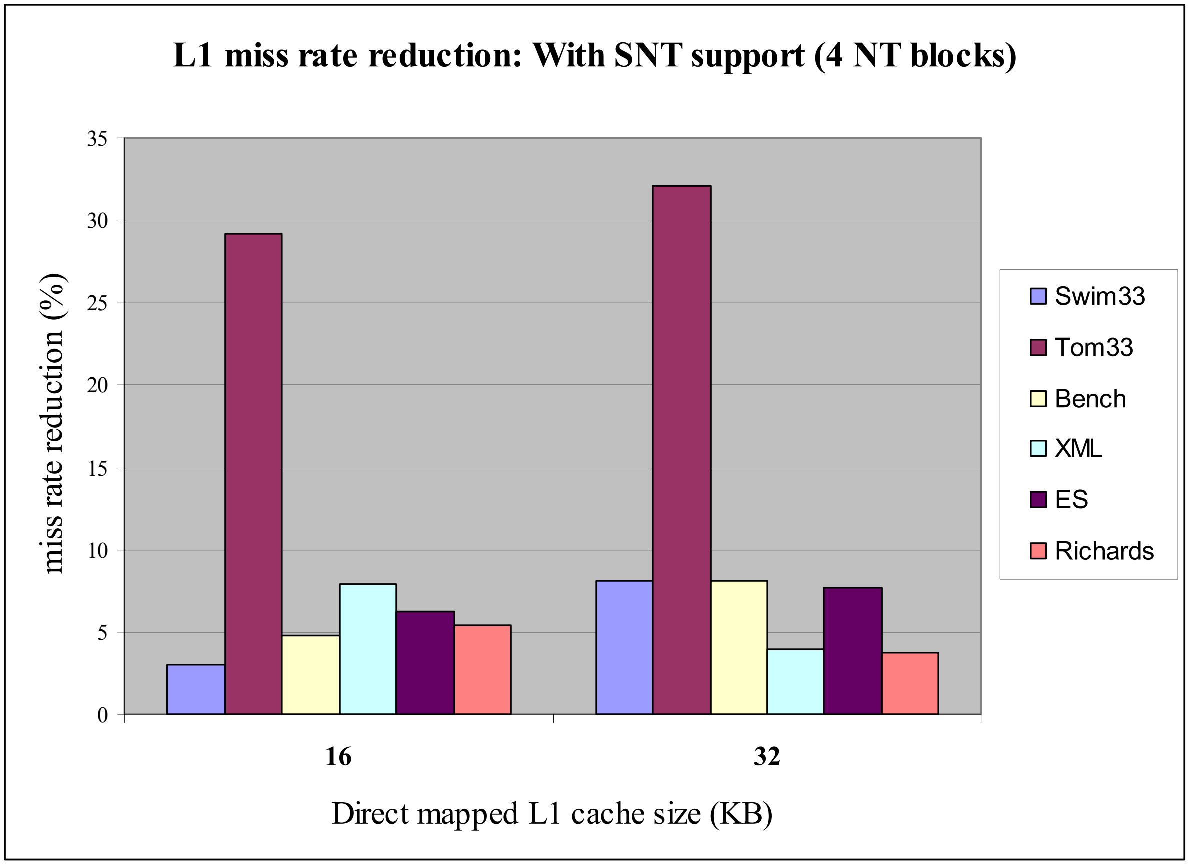


Figure 4. 4 – Miss-rate reduction in a 16 KB and 32 KB direct-mapped cache with SNT support during scavenge, where fromspace blocks were constrained to 4 NT lines in cache.

We see that the miss rates have gone down for all benchmarks. Tomcat showed the greatest improvement, with a miss-rate drop of 29.1% for a 16 KB L1 and 32.1% for a 32 KB L1. This is because Tomcat was one of the two benchmarks that had a large amount of live data, the other being Swim. Although Swim had a large amount of live data, the grid size that we used here (33×33) is small. Figure 4.5 shows that Swim shows a greater improvement when a larger grid size (151×151) is used. It is also to be noted here that with a 151×151 grid size the scavenge time for Swim exceeds the mutator execution time. Thus we see that there is more potential for improvement in applications with larger data sets, and where the scavenger takes up a larger fraction of the overall execution time than what the mutator does.

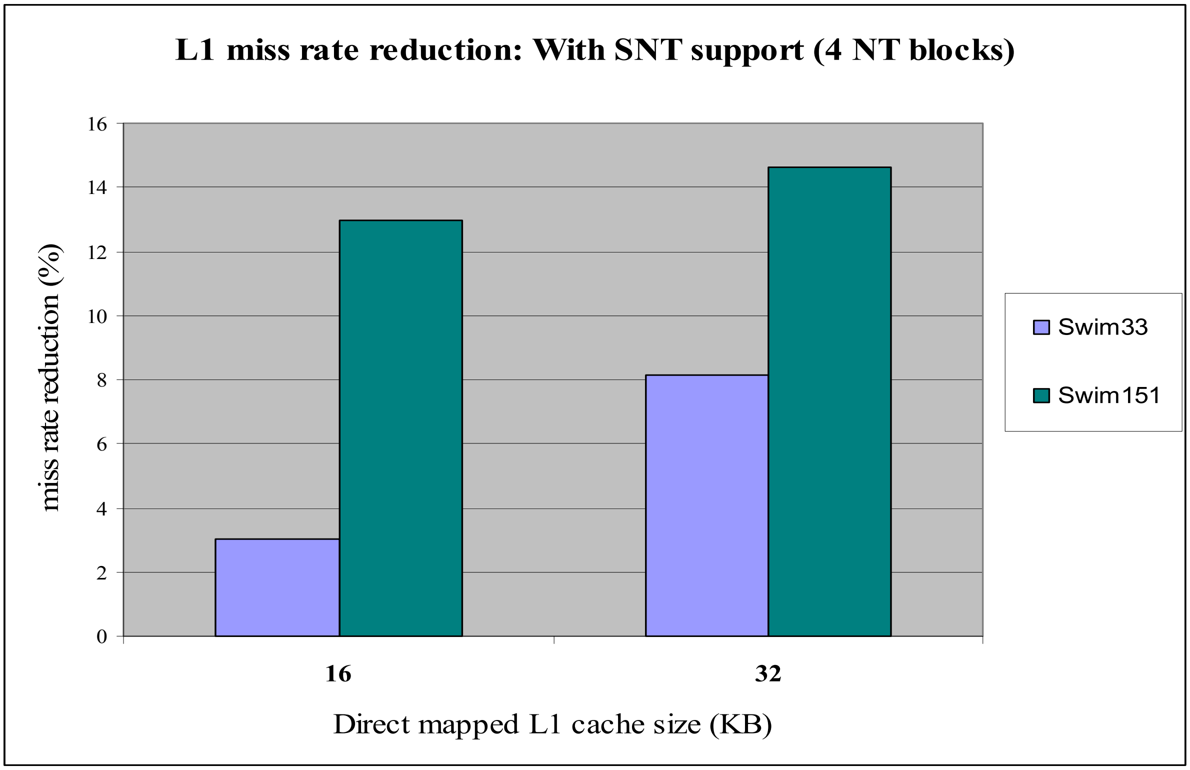


Figure 4. 5 – Comparison of the drop in miss rates (due to SNT) in Swim for grid sizes of 33×33 and 151×151.

Chapter 5

Experimental Setup

Experiments were conducted for the following 6 benchmarks: Swim, Tomcatv, Bench, XML Parser, ES Compiler and Richards. Shade was used for tracing the benchmarks and the traces were directly fed to analyzers which simulated a cache subsystem.

5.1 Shade

Shade [3,15] is an instruction-set simulator and custom trace generator from Sun Microsystems. Application programs are executed and traced under the control of a user-supplied trace analyzer. The version of Shade (v6) used in our study simulated execution of IBM Smalltalk [14] applications (compiled for SPARCv8 or SPARCv9) on a SPARCv9 host machine running Solaris 5.7

5.2 Cachesim5

Trace analyzers consume traces on the fly, thereby avoiding the need to collect large traces. For our study, we were interested in the cache performance of applications. Cachesim5 [15] is a trace analyzer from Sun Microsystems that can simulate multilevel caches. The number of cache levels, their size, associativity, replacement policy and other such parameters can be set from the command line [16]. The simulators used in this study are built upon Cachesim5 to reflect our mapping strategies and cache placement policies.

5.3 Methodology

5.3.1 Simulation in Shade

For our experiments, we ran IBM VisualAge Smalltalk benchmarks over Shade, which analyzed the benchmark as it was executed, using a cache simulator. Our cache simulators were adapted from Cachesime5, a multilevel cache simulator that comes with Shade. Cachesim5 simulates only the cache hierarchies and not the entire memory subsystem. Specifically, paging is not simulated. Cachesim5 uses the virtual-address references from the application that runs on Shade. In our case, the 32 bit virtual addresses from the Smalltalk application are used by the simulator to calculate the tag, index and offset fields.

To achieve the effect of pinning the youngest generation in the L2 cache, we made modifications to Cachesim5 so that the virtual addresses were translated to physical addresses with the following properties:

- All youngest generation pages map to the reserved portion of the cache.
- Non-youngest generation pages *do not* map to the reserved portion of the cache.

The target address space to which the virtual addresses are translated starts from zero and does not have an upper limit. This might make addresses large, but the cache behavior is not affected by this.

For SNT cache, to simulate the effect of constraining fromspace lines in a few NT blocks of the cache, we modified the cache lookup logic and the miss-handler of the cache simulator. During the mutator phases of the program, the regular lookup logic and miss-handler was used. During scavenge cycles the cache simulator used the modified lookup logic and miss-handler that was described in section 4.6.

5.3.2. Communication between the Smalltalk VM and Shade

During the course of simulation, the cache simulator needs certain inputs from the Smalltalk virtual machine. Two of these are the upper and lower bounds of new space (the youngest generation) in the heap. The Smalltalk virtual machine was made to write the heap details to a file. The simulator read the information from this file at the beginning of each run. If the size of new space does not change, its upper and lower bounds remain the same for every run. Therefore the VM had to write out its heap details just once, and this was used for future simulations.

Our experiments with SNT cache, required dynamic inputs from the virtual machine. Here the simulator needs to use a different cache lookup logic and miss-handler for fromspace addresses during a scavenge cycle. The virtual machine needs to let the simulator know the beginning and end of each scavenge cycle. Shade does not provide any means for the applications that run on it to communicate directly with an analyzer. We used a file to allow the virtual machine to communicate with the simulator. The Smalltalk virtual machine was instrumented to set a flag in a file at the beginning of every scavenge cycle and reset the flag at the end of it. For every memory access in the application simulated by it, the simulator had to check whether the application was in a scavenge cycle. This meant a lot of file I/O, and it slowed down the simulations considerably. Using memory-mapped I/O seemed a better choice. This helped bring down simulation time.

5.4 Cache Parameters

Instruction caches were not simulated in any of the experiments since our study pertains to just data caches. For studies involving two-level cache simulations, a 16 KB direct-mapped L1 cache with a write-through policy was used and a write-back-with-write-allocate policy was used for the L2 cache. For studies involving only L1 simulations, write-back-with-write-allocate policy was used. For set-associative caches, LRU replacement was used. A cache block size of 32 bytes was used in all experiments.

5.5 Benchmarks

This study closely follows the work of Reddy and therefore used the same set of benchmarks used in his study [6]. A set of six benchmarks were used. The input data set size affects the number of allocations and hence the number of garbage collection cycles and the execution time of the program. For purposes of comparison, our input data set sizes matched those used by Reddy. The selected input data sets ensure that we have reasonable simulation times while not compromising the validity of results. This was done by making sure that allocation was large enough to invoke several garbage collections. All benchmarks run in a non-interactive mode.

Swim is a scientific benchmark from the SPEC95 floating point benchmark suite. The program solves a system of shallow water equations using finite difference approximations on a $N \times N$ grid. The grid parameters size controls the size of the problem. It has several loop nests and scattered accesses to arrays. The Smalltalk version of Swim was written by Darko Stefanovic and was ported to IBM VisualAge Smalltalk by Reddy.

Tomcatv is from the SPEC95 floating point benchmark suite. It is a vectorized mesh generation program that is computation intensive. It works with two-dimensional arrays of data containing the X and Y coefficients, which are initialized from a file. Tomcatv contains a main loop that iterates 100 times. Each iteration contains several nested loops which accesses array elements. The Smalltalk version of Tomcatv was written by Darko Stefanovic and was ported to IBM VisualAge Smalltalk by Reddy.

Bench is a collection of programs that perform basic tasks like sorting, matrix multiplication and some logical problems like Queens and Puzzle. The programs were gathered into a benchmark suite by John Hennessy and modified by Peter Nye. The Smalltalk version was obtained from Greg Gritton and ported to IBM VisualAge Smalltalk by Reddy.

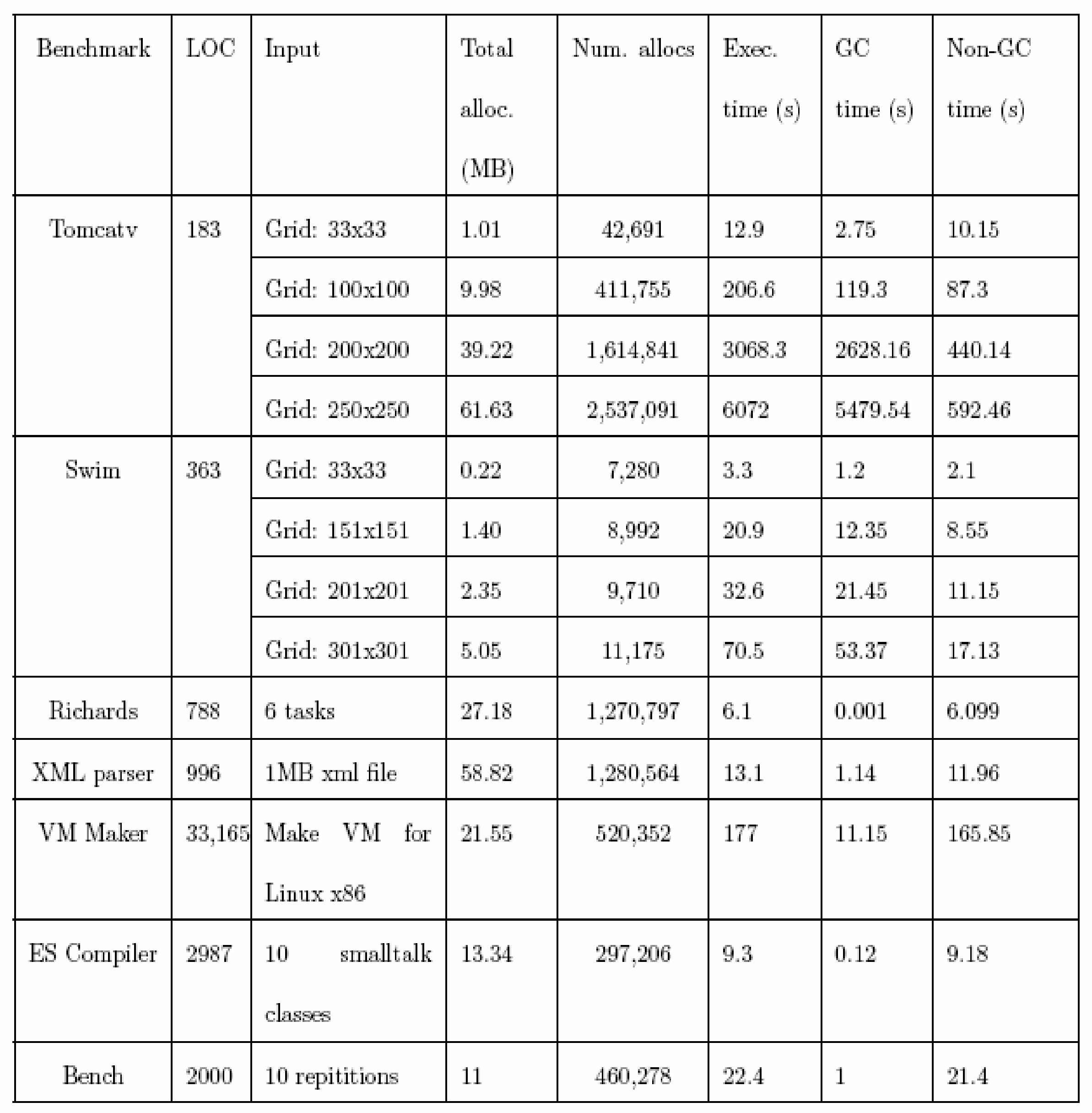
XML Parser is a tree-based SAX (Simple API for XML) parser from IBM VisualAge Smalltalk. It parses an XML document into an internal tree structure, allowing an application to navigate that tree. The file parsed in our experiments is an XML file containing insurance policy information. The XML file size was restricted to 5MB for simulation purposes.

ES compiler is a compiler for IBM VisualAge Smalltalk. It accepts Smalltalk code and returns its byte code representation. Code filed out for basic Smalltalk classes like Object, Rectangle, Point, Process and few more basic classes was compiled for our experiments. The number of input classes controlled the simulation time.

Richards is an operating-system benchmark due to Martin Richards from Cambridge University. It simulates the kernel task dispatcher of an operating system. We ran Richards with six tasks in the task list. This enabled a reasonable simulation time. The Smalltalk version was borrowed from the GNU Smalltalk package [17] and ported to IBM Visual Age Smalltalk by Reddy.

The benchmarks as characterized by Reddy are shown in table 5.1. The *input* column lists the data set for a particular benchmark. *Total allocation* shows the amount of memory allocated by the benchmark. The *number of allocations* column indicates the allocation rate of the benchmark. The total *execution time* shown is the wall-clock time and comprises the time taken by the garbage collector (*GC time*) and the mutator (*non-GC time*).

Table 5. 1 [6] – Characteristics of benchmarks.



Benchmark	LOC	Input	Total alloc. (MB)	Num. allocs	Exec. time (s)	GC time (s)	Non-GC time (s)
Tomcatv	183	Grid: 33x33	1.01	42,691	12.9	2.75	10.15
		Grid: 100x100	9.98	411,755	206.6	119.3	87.3
		Grid: 200x200	39.22	1,614,841	3068.3	2628.16	440.14
		Grid: 250x250	61.63	2,537,091	6072	5479.54	592.46
Swim	363	Grid: 33x33	0.22	7,280	3.3	1.2	2.1
		Grid: 151x151	1.40	8,992	20.9	12.35	8.55
		Grid: 201x201	2.35	9,710	32.6	21.45	11.15
		Grid: 301x301	5.05	11,175	70.5	53.37	17.13
Richards	788	6 tasks	27.18	1,270,797	6.1	0.001	6.099
XML parser	996	1MB xml file	58.82	1,280,564	13.1	1.14	11.96
VM Maker	33,165	Make VM for Linux x86	21.55	520,352	177	11.15	165.85
ES Compiler	2987	10 smalltalk classes	13.34	297,206	9.3	0.12	9.18
Bench	2000	10 repititions	11	460,278	22.4	1	21.4

Chapter 6

Related Work

Garbage collection has been a widely researched area. The advent of generational garbage collection improved the virtual-memory performance of garbage collectors and helped shift research interest to the cache performance of garbage collectors.

Zorn [13] compares the cache performance of generational mark-sweep and copying algorithms. His studies show that mark-sweep algorithms perform better on direct-mapped caches, while copying algorithms perform better with set-associative caches. He also finds that in caches large enough to hold the youngest generation, the parameters of the garbage collection algorithm can have a significant effect on the cache performance.

Wilson, Lam and Moher [12] attribute the cyclic memory access pattern of garbage-collected systems to the unusual ways in which they interact with the cache design. Their experiments measure the performance of garbage-collected systems on direct-mapped and set-associative caches.

Gonçalves and Appel [4] study the effect of fitting the allocation space in an L1 cache for fast-allocating ML programs that use generational garbage collection. They study the tradeoff between increasing garbage collection frequency and choosing an allocation space that fits in the cache. Smalltalk programs tend to have larger objects than ML programs, thus Smalltalk's allocation space is not able to fit into the L1 cache. Their study also differs from ours in that it does not employ a pinning strategy to eliminate cache conflicts.

Boehm [1] addresses the cache misses due to lack of temporal locality in garbage collection algorithms. He proposes a technique called "prefetch-on-grey" to deal with this issue.

Rivers and Davidson [7] propose a non-temporal streaming (NTS) cache to eliminate conflict misses in a direct-mapped cache due to non-temporal data. The NTS cache supplements the conventional direct-mapped cache with a parallel fully associative buffer.

Cache blocks that are non-temporal are identified dynamically and are allocated to the buffer. They use a hardware detection unit to monitor the temporal behavior of every cache block loaded into the main cache. Their results show that that an NTS Cache is as good as a conventional direct-mapped cache of about twice its size. However, they study the performance of scientific and numeric benchmarks on NTS caches with no special emphasis on garbage collection. We believe that garbage collection poses a special scenario in the following ways:

1. We know a priori the non-temporal nature of fromspace blocks.
2. Due to the nature of copying algorithm, fromspace and tospace are swapped during each scavenge cycle. This oscillating behavior might confuse a hardware detection unit that tries to classify their behavior as temporal or non-temporal.
3. Rivers and Davidson classify blocks as either temporal or non-temporal. We saw that fromspace blocks are not strictly non-temporal. This should be taken advantage of by using a non-aggressive eviction policy rather than just classifying them as temporal.

We believe that results similar to ours can be obtained for garbage-collected programs using the proposed NTS cache by extending it to accommodate these special concerns.

Chapter 7

Conclusions and Future Work

Prior work was done in the area of pinning the youngest generation in the L2 cache. While improving L2 hit rates, this exhibited an anomaly with set-associative L2 caches, where pinning brought down L1 hit rates. In this research, we studied this effect and proposed a realizable algorithm to pin the youngest generation in L2 that will not degrade L1 hit rates.

We also showed that by making caching decisions based on the knowledge of the lack of temporal locality in fromspace lines during scavenge cycles, misses due to conflicts between fromspace and tospace lines can be eliminated. We proposed a framework where the OS and the SNT cache hardware supports constraining non-temporal blocks to select cache lines for which temporal locality is turned off during scavenge cycles. Our simulations show that this has an effect on both the scavenger and mutator cycles. The scavenger saw less of conflict misses. The mutator cycle benefited due reduced cache pollution by non-temporal lines. For a 16 KB direct-mapped cache, the benchmarks showed a 3% to 29% drop in miss rates and an average drop of 9.41%. For a 32 KB direct-mapped cache, miss rates went down by 3.6% to 32%, with an average miss-rate reduction of 10.6%.

Future work includes intelligently choosing the number of non-temporal lines in the SNT cache to optimize performance, and extending SNT to set-associative L2 caches.

The generational garbage collector used in this study uses a fromspace and tospace in the youngest generation. We are interested in extending our strategies to generational collectors which use an eden-space and two survivor spaces in the youngest generation.

Bibliography

[1] Hans-J. Boehm. “Reducing garbage collector cache misses.” *Proceedings of the Second International Symposium on Memory Management*, October 2000, pages 59-64.

[2] C. J. Cheney. “A non-recursive list compacting algorithm.” *Communications of the ACM*, volume 13, issue 11, November 1970, pages 677-678.

[3] Robert Cmelik, David Keppel. “Shade: A fast instruction-set simulator for execution profiling.” *ACM SIGMETRICS Performance Evaluation Review*, volume 22, issue 1, May 1994, pages 128-137.

[4] Marcelo J. R. Gonçalves, Andrew W. Appel. “Cache performance of fast-allocating programs.” *ACM SIGPLAN/SIGARCH Conference on Functional Programming Languages and Computer Architecture*, June 1995.

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

[6] Vimal Reddy. “Caching strategies for improving generational garbage collection in Smalltalk.” *MS Thesis, North Carolina State University*, 2003.

[7] Jude A. Rivers and Edward S. Davidson. “Reducing conflicts in direct-mapped caches with a temporality-based design.” *In Proceedings of International Conference on Parallel Processing*, vol. 1, 1996, pages 154-63.

[8] Mark Sullivan, Ram Chillarege. “Software defects and their impact on system availability - a study of field failures in operating systems.” *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, 1991, pages 2-9.

[9] David Ungar. “Generation scavenging: A non-disruptive high performance storage reclamation algorithm.” *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, April 1984, pages 157-167.

[10] David Ungar. “The design and evaluation of a high performance smalltalk system.” *Distinguished dissertation, Massachusetts Institute of Technology, MIT press, Cambridge, MA*, March 1986.

[11] Paul Wilson. “Uniprocessor Garbage Collection Techniques.” *Proceedings of the 1992 International Workshop on Memory Management*, September 1992.

[12] Paul R. Wilson, Michael S. Lam, Thomas G. Moher. “Caching considerations for generational garbage collection.” *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, June 1992, pages 32-42.

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

[14] *IBM VisualAge Smalltalk Version 6.01*. Available at www-3.ibm.com/software/awdtools/smalltalk

[15] *Shade V6 and Cachesim5*. Available at www.sun.com/microelectronics/shade

[16] *Shade User's Manual V6.0*. Available at www.sun.com/microelectronics/shade

[17] *GNU Smalltalk Version 2.1*. Available at www.gnu.org/software/Smalltalk/smalltalk.html

[18] *The Memory Management Glossary* at www.memorymanagement.org