ABSTRACT

REDDY, VIMAL K. Caching strategies for improving generational garbage collection in Smalltalk. (Under the direction of Dr. Edward F. Gehringer).

Cache performance of programs is becoming increasingly important as processor speeds grow faster relative to main memory. Cache misses have become a major consideration for performance of garbage collected systems. This thesis explores a caching strategy for generational garbage collectors, the most prevalent form in use, which takes advantage of large caches available to modern-day processors. A portion of the cache is chosen to reserve the youngest generation and the page-fault manager is provided certain mapping rules that remove all conflicts to the youngest generation. The strategy can be completely realized in software which makes it an attractive solution to increase garbage collection performance. This "biased" cache mapping is shown to reduce cache misses and increase overall performance in the IBM VisualAge Smalltalk system, a high quality Smalltalk implementation that employs a generational copying garbage collector.

Favoring youngest generation in the mapping strategy is advantageous for the following reasons:

1. Languages like Smalltalk, where "everything" is an object, tend to allocate furiously. This is because they encourage a programming style where objects are created, used and shortly thereafter destroyed. This large number of allocations translates to initialization write misses if the allocated region is not cached. In generational heaps, all memory is allocated in the region containing youngest-generation objects.

2. A generational garbage collector focuses collection on the youngest generation, scavenging it to reclaim most garbage. It relies on empirical knowledge that most young objects die soon. This means the scavenger runs many times during a program lifetime, scanning the youngest generation for garbage. This process can lead to a large number of read and write cache misses if the youngest generation is not in cache.

3. Youngest generation objects form a major part of a program's working set. Making them available in the cache would also improve the mutator (i.e, user program) performance, making it immune to interference from the garbage collector.

4. Given that most young objects become garbage quickly, when a garbaged object is evicted from a writeback cache, an unneccessary writeback results. Caching the youngest generation would reduce traffic to memory.

We do a simulation-based study of our mapping strategies on IBM VisualAge Smalltalk, a generational copying garbage collected system. Our results show a 45% average drop in cache miss rates at the L2 level for direct-mapped caches and 15% average drop for $2 - way$ set-associative caches.

Caching strategies for improving generational garbage collection in Smalltalk

by

REDDY, VIMAL KODANDARAMA

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

Department of Electrical and Computer Engineering

Raleigh

2003

Approved By:

Dr. Gregory T. Byrd	Dr. Warren J. Jasper
Dr. Edward F. Gehringer Chair of Advisory Committee	Dr. Eric Rotenberg

to my parents, brother and “tony”

Biography

Vimal Kodandarama Reddy was born on 3rd November 1977 in Bangalore, India. In 1999, he graduated with a BE degree in "Electronics and Communications Engineering" from Bangalore University, Bangalore, India. Between 1999 and 2001, he worked as a Software Development Engineer at Siemens Public Communication Networks Ltd, Bangalore, India.

In Fall, 2001 he started his masters program in Computer Engineering at North Carolina State University. In Summer, 2002 he interned with IBM VisualAge Smalltalk in Raleigh, North Carolina.

Acknowledgements

First and foremost, I would like to thank my parents, Kodandarama Reddy and Viji Ram, for giving me life, educating me and supporting me throughout the way. I thank my brother Vikram for all the support and encouragement he gave. I continue to seek their advice and support in my doctoral pursuit.

This work would not have been possible without help from the IBM VisualAge Smalltalk group. I am indebted to Craig Chaney for the time he spent explaining to me anything and everything about IBM VisualAge Smalltalk. His expertise has certainly given direction to this work. He was always there to answer my questions and he got me through many a roadblock. Special thanks to him for packaging a Smalltalk virtual machine for me. It enabled me to run all experiments at our lab and saved a lot of effort. Thanks to Brian Hogan for writing the Xml parser and showing me how to use it. Thanks to John O'Keefe for listening to our ideas. Thanks to Ginny Ghezzo for allowing me to intern with her group in Summer 2002. I appreciate her confidence in me and thank her for allowing me to conduct this study.

I would like to thank Dr. Edward Gehringer for being my advisor and providing me an opportunity to do this thesis. The quality of this thesis improved manyfold due to his careful reviews. My sincere thanks to Dr. Eric Rotenberg for his constant guidance and support during my thesis. I have learned a lot about research and computer architecture from him. A big thanks to his group for allowing me to use the condor pool of machines to run my simulations. Without it this work could not have been completed in time. I would also like to thank my committee members for reviewing my thesis.

Several others have contributed indirectly. Thanks to Paolo Bonzini for creating GNU Smalltalk and answering all my beginner's questions. I learned a great deal about virtual machines, hacking and Smalltalk from GNU Smalltalk. Thanks to Dan Green and the ECE help staff for providing timely help. I would like to take this opportunity to thank my friend Clifton Fernandes for his help back home while graduate school kept me away. I wish him good luck in his future endeavors.

Finally, some graphs were produced using Jgraph by Jim Plank at Princeton University. Thanks to him for creating such an elegant package. Thanks to all LaTeX creators for the ultimate document preparation package.

Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

2 Background and Related Work **4**
2.1 Dynamic memory management 4
2.2 Copying garbage collection 5
2.3 Generational garbage collection 9
2.4 Related work 13

3 Design and Implementation **15**
3.1 Direct-mapped caches 15
3.2 Set-associative caches 18
3.3 Implementation 22

4 Simulator and measurement tools **24**
4.1 Shade and Cachesim5 24
4.2 Memory profiler 25

5 Methodology **28**
5.1 Mapping strategy in the simulator 28
5.2 Feeding heap information to the simulator 29

6 Benchmarks **30**
6.1 Benchmark expectations 30
6.2 Final benchmark set 31

7 Experiments, results and analysis **34**
7.1 Garbage collection overhead 34
7.2 Cache simulations 42
7.2.1 Finding the right portion of the cache to reserve 42

7.2.2 Applying the strategy to a direct-mapped L2 cache 48
7.2.3 Applying the strategy to a set-associative L2 cache 51
7.2.4 Effect on IPC 57

8 Conclusion and future work **64**

Bibliography **65**

List of Figures

2.1 Semispace copying garbage collection using Cheney's traversal algorithm. . 7
2.2 Semispace copying garbage collection (contd.) 8
2.3 Generational heap used in IBM VisualAge Smalltalk 10
2.4 Scavenge frequency in IBM VisualAge Smalltalk system, for Swim. 12
2.5 Global garbage collection frequency in IBM VisualAge Smalltalk, for Swim 12

3.1 Mapping strategy for a 32-byte line, 1 MB direct-mapped cache to reserve a 256KB large youngest generation in the cache. 17
3.2 Mapping strategy for a set-associative cache where a way of the set-associative cache is reserved for the youngest generation. 20
3.3 Mapping strategy for a 32-byte line, 4-way, 1 MB set-associative cache to reserve a 256 KB large youngest generation in the cache using a certain number of sets. 21
3.4 An implementation of the mapping strategy in a real system to reserve the youngest generation in the cache. 23

4.1 Method dictionary of a class before and after profiling. 27

7.1 New space size vs. Number of scavenges, for Xml parser, Swim, Tomcatv and Richards. 35
7.2 New space size vs. Number of scavenges, for Bench, VM Maker and ES Compiler. 36
7.3 Execution time distribution wrt New space size, for Swim. 38
7.4 Execution time distribution wrt New space size, for Tomcatv. 39
7.5 Execution time distribution wrt New space size, for Bench, VM Maker and ES Compiler. 40
7.6 Execution time distribution wrt New space size, for Xml parser and Richards. 41
7.7 Miss rate improvement of a direct-mapped L1 cache when new space is 50% of the cache size, for all benchmarks. 43
7.8 Miss rate improvement of a direct-mapped L1 cache when new space is 25% of the cache size, for all benchmarks. 43

7.9 Miss rate improvement of a direct-mapped L1 cache when new space is 12.5% of the cache size, for all benchmarks. 44
7.10 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Swim. 45
7.11 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Xml parser. 45
7.12 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for ES Compiler. 46
7.13 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Bench. 46
7.14 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Tomcatv. 47
7.15 Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for VM Maker. 47
7.16 Miss rate improvement for a direct-mapped L2 cache when new space is 25% of the L2 cache size, for all benchmarks. 49
7.17 L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and L2 caches when the mapping strategy is applied to the L2 cache, for all benchmarks. 49
7.18 Global miss rate improvement in a two-level cache hierarchy with a direct-mapped L1 and L2 cache when the mapping strategy is applied to the L2 cache. 50
7.19 Miss rate improvement for a $2-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks. 52
7.20 Miss rate improvement for a $4-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks. 52
7.21 Miss rate improvement for a $8-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks. 53
7.22 L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $2-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks. 55
7.23 L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $4-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks. 55
7.24 L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $8-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks. 56
7.25 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Swim. 58
7.26 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Tomcatv. 58
7.27 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for ES Compiler. 59

7.28 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Bench. 59
7.29 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Xml parser. 60
7.30 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Richards. 60
7.31 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Swim. 61
7.32 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Tomcatv. 61
7.33 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for ES Compiler. 62
7.34 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Bench. 62
7.35 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Xml parser. 63
7.36 Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Richards. 63

List of Tables

6.1 Benchmark characteristics 33

Chapter 1

Introduction

Automatic storage reclamation is a technique that simplifies memory management for the programmer. In systems using it, the runtime environment is able to track live data and efficiently reclaim garbaged memory for reuse. This enables a progammer to focus on writing a program to solve the real problem and not worry about memory-related bugs like memory leaks and premature freeing. One study of a non-garbage collected language estimates that 40% of the development time is used to implement memory management procedures and solve memory related bugs [26].

While garbage collection is effective in reducing development time and bugs, it has overheads. Additional instructions are executed to identify and reclaim garbaged objects. Garbage collection being a memory intensive operation, most instructions it executes are references to memory. The memory overhead introduced affects program performance in two ways. Firstly, increased memory references increase program execution time, especially in modern processors which have raised the memory wall [25]. Secondly, the garbage-collector can interfere with the working set of the actual program (called the mutator) and increase page faults and cache misses in the program.

Generational collection is the most prevalent garbage collection algorithm in use. This thesis studies the cache performance of a copying style generational garbage collector. Generational garbage collection [1, 2] is a strategy that mitigates the memory system overhead due to garbage collection. It relies on empirical knowledge that in languages like Smalltalk, where "everything" is an object, newly created objects (called young objects)

tends to become garbage quickly [2, 4]. By segregating the heap into regions based on age, and collecting regions containing young objects more often, generational collectors reclaim garbage with smaller pause times. Focusing collection on a small region of the address space reduces the memory footprint of the collector. This in turn reduces page faults and improves virtual memory performance. Generational collectors helped shift the focus of research on garbage collection performance from virtual memory to caches, which are crucial for program performance in modern architectures.

In a generational garbage-collected system, all memory is allocated in a region of the heap called *youngest generation*. This thesis attempts to improve cache performance of a generational copying garbage collector by favoring youngest-generation objects in the cache mapping strategy. The youngest generation of a generational heap is a memory hotspot in languages like Smalltalk. This is because the programming style encourages rapid creation of objects in transit to program completion. Most of these objects soon become unreferenced and hence garbage, a short while after creation. When the youngest generation of heap runs out of memory, the garbage collector "scavenges" [2] it to reclaim garbaged memory for future use. The process of rapid allocation and reclamation of memory in the youngest generation leads to lot of memory accesses to this region of the heap.

Traditional caching strategies do not work well in capturing the accesses for two main reasons. Firstly, the access patterns due to sequential allocation in the youngest generation depart from the rules of locality that make caches effective. Secondly, the working set of a program spreads across different generations, leading to inter-generation conflicts in the cache. The strategy employed in this thesis aims to make all youngest-generation accesses hit in the cache. In doing so, we have assumed caches that are sufficiently big to hold the youngest generation. This is a fair assumption for modern processors where megabyte second-level caches are becoming common. A cache big enough to fit the youngest generation cannot capture all accesses just by virtue of its size. This is because the placement of the youngest generation in physical memory may cause memory blocks containing them to conflict in the cache. Another reason is that inter-generation accesses can still lead to cache conflicts. The mapping strategy we adopt requires only a slight modification to the page-placement policy of the operating system. A portion of the cache, equal to the size of the youngest generation, is reserved for youngest generation objects. Page frames that map to the reserved region in cache are identified and used to aid page placement. The page-placement policy makes sure it does not place pages that do not contain youngest

generation objects in page frames mapping to the reserved region in cache.

In this work, we do a simulation-based study of our mapping strategy on the cache performance of IBM VisualAge Smalltalk in a non-interactive mode. The reasons for choosing IBM VisualAge Smalltalk over other Smalltalks were: firstly, availability of a high quality Smalltalk implementation; secondly, its compatibility with Shade, the simulator we used in our studies; and lastly, its use of a copying style generational garbage collector. We believe similar results could be obtained with other Smalltalk implementations that use a generational copying garbage collector.

A memory profiler was developed in Smalltalk to collect allocation statistics of benchmarks. The profiler adds instrumentation code to calculate memory allocated by a Smalltalk program. This information was used to collect the right set of benchmarks.

To summarize, this thesis makes the following contributions:

1. A mapping strategy was designed for direct-mapped and set-associative caches to make the youngest generation persistent in cache.

2. A simulation based study was performed on IBM VisualAge Smalltalk in a non-interactive mode.

3. Smalltalk benchmarks were collected and some of them ported to IBM VisualAge Smalltalk.

4. A memory profiler was developed in IBM VisualAge Smalltalk to collect allocation statistics of benchmarks.

The rest of the thesis is organized as follows. Chapter 2 introduces generational garbage collection and surveys related work on cache performance of garbage collected systems. Our mapping strategy is the topic of discussion in Chapter 3. Chapter 4 discusses the simulator and other tools used in this study. Chapter 5 presents the methodology followed in implementing the mapping strategy in the simulator. Chapter 6 describes the benchmarks collected and chapter 7 discusses experiments carried out and results obtained. Chapter 8 concludes by suggesting future work.

Chapter 2

Background and Related Work

This chapter discusses terminology related to this work and surveys related research. The organization of this chapter is as follows. Section 2.1 discusses dynamic memory management and introduces garbage collection. Section 2.2 introduces copying garbage collection and related terminology. Section 2.3 discusses the generational garbage collection algorithm that was used in this study. Section 2.4 looks at previous research on cache performance of garbage collectors and cache strategies related to our work.

2.1 Dynamic memory management

In a traditional programming model, a program uses three types of memory: static, automatic and dynamic [27]. Static memory (e.g. global data), is allocated at the beginning of the program and persists till the end. Automatic memory is allocated in a memory region called the *stack* and is used to store local variables and function arguments. Memory pertaining to each function is stored in a *stack frame*, which is allocated and deallocated automatically. Dynamic memory, as the name indicates, is used to allocate objects whose size is determined only at run time. The region of allocation is known as the free store or more commonly, the *heap*. Memory on the heap is allocated and deallocated explicitly by the user. The heap is generally managed by storing chunks of memory on one or more lists, called free lists. Allocation involves removing a chunk from the free list and deallocation,

adding an unused chunk back to the free list. The burden of finding when a chunk of memory (i.e., an object) becomes unused is left to the user. This can be a source of programming errors in the following ways. If a chunk is deallocated while still being used it leads to a dangling reference, whereas losing track of an unused chunk leads to memory leaks. The former has disastrous effects while the latter causes fragmentation and deems the program to be a "memory hog." This memory model is representative of manual storage management used in programming languages like C and C++. An alternative memory-management technique is automatic storage reclamation or *garbage collection*, which is of interest in this thesis. In systems using garbage collection, the task of deallocating objects is delegated to the runtime environment. The runtime uses software called a *garbage collector* to detect objects that are unused and reclaims them for reuse. The garbage collector uses program state information to determine an initial set of pointers to objects in the heap. This is called the *rootset*, and reachability from the rootset defines an object as "live." In order to find the rootset, the garbage collector scans for pointers to the heap in all objects that would affect the future course of the program. This would include all global data, current activation records and machine registers. The detected garbage can be reclaimed in many ways depending on the garbage collection algorithm.

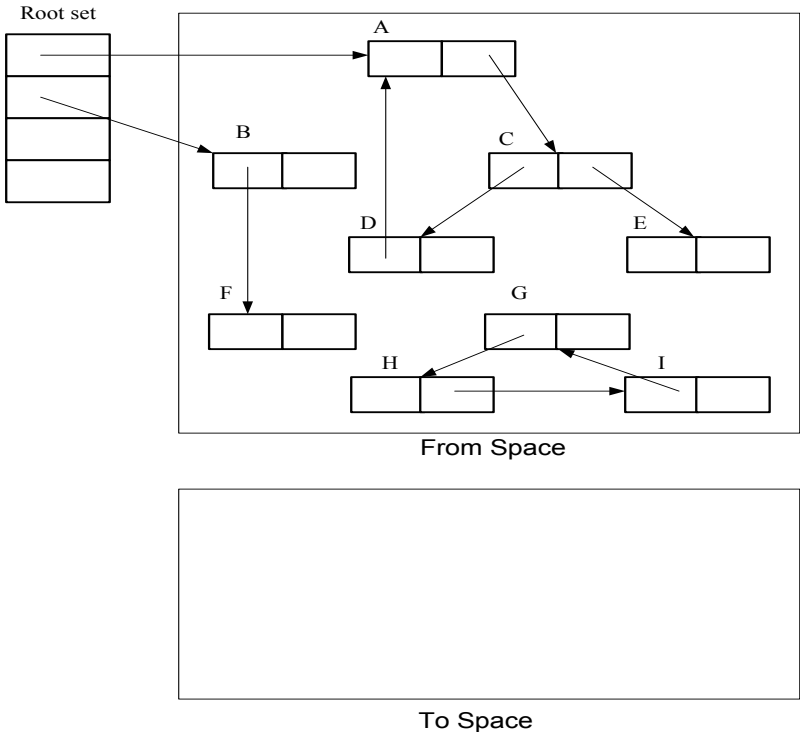
Garbage collection is a vastly researched topic. A good reference on garbage collection studies is Paul Wilson's survey [5]. A book on the subject has been written by Jones and Lins [6]. Sankaran [7] has compiled a comprehensive list of publications related to garbage collection. This thesis studies a copying-style generational garbage collection and is introduced in the next few sections. For information on other garbage-collection algorithms the reader is referred to the sources mentioned above.

2.2 Copying garbage collection

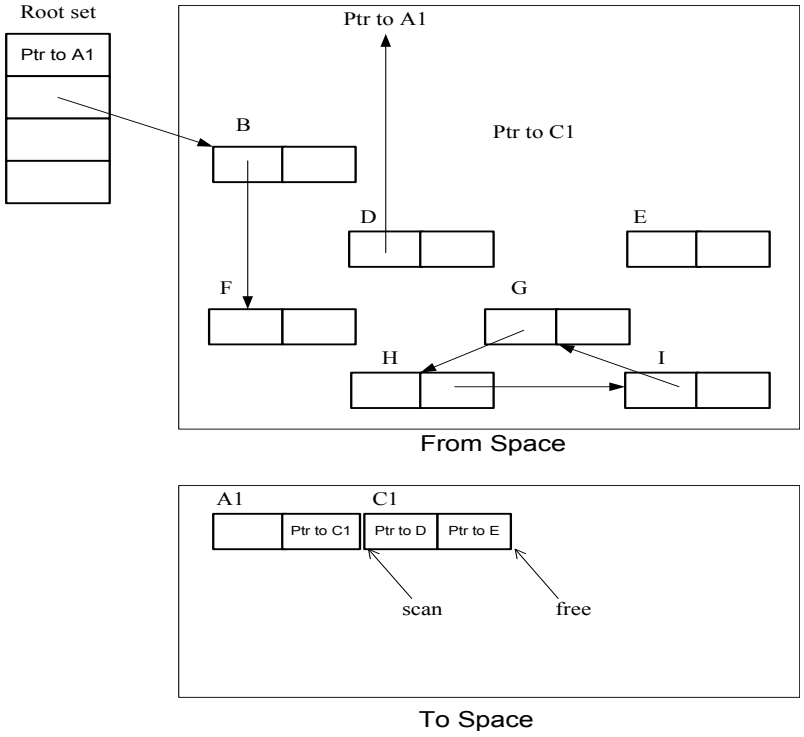
A copying garbage collector works by copying live objects to a different area in memory. After garbage collection, the new area is compact with live objects while the old area is reclaimed by the garbage collector. In a copying garbage collector, garbage detection is interleaved with the copying process. Live data detection involves traversing objects starting from the garbage-collector rootset. As live data is detected, it also gets copied into a new region in memory. Different algorithms exist for combining copying of live

data with object traversal. Among these Cheney's breadth-first traversal algorithm [31] is perhaps the most popular and widely used. The Smalltalk system used in this thesis [23] also uses this algorithm for its garbage collection purposes. Below I describe with an example, a simple semispace copying garbage collector [28] that uses Cheney's breadth first traversal algorithm.

A semispace garbage collector works by dividing the heap into two semispaces. Only one of the semispaces is active at any time; it contains the program data. The other space is considered to be free memory. Allocation is done by incrementing an allocation pointer on every allocation. When the active space runs out of memory, garbage collection is invoked. One possible way to detect this state is to place the heap top next to a guarded page [20]. When the allocation pointer gets incremented into the guarded page (write-protected page), the page fault handler invokes the garbage collector. The garbage collector then traverses objects reachable from the rootset. The free semispace now becomes the destination space for live objects detected in the active semispace. As live objects are detected, they are copied to the destination space. This process is called *scavenging*. For convenience, the active semispace is referred to as *From space* and the free semispace as *To space*. In Cheney's breadth-first algorithm, an implicit queue of live objects to be copied is maintained in the *To space*. A *scan* pointer points to the head of the queue and a *free* pointer to the tail. Both initially point to the base of *To space*. When a live object is detected in *From space*, it gets copied over to *To space*. The copying process involves incrementing the free pointer by the size of the object being copied. The copied object is next scanned for pointers to *From space*. Any such pointers are considered as pointing to live data, in which case they get copied over to *To space*. The scan pointer is incremented as objects are scanned. When an object gets copied, its *To space* address is left in its *From space* location. This forwarding pointer is required to prevent redundant copying of live objects when a scanned location points to an already copied object. When the scan and free pointers meet, the rootset is searched for any remaining pointers to *From space*. Garbage collection ends when the scan and free pointers are equal and the rootset has no more pointers to the *From space*. The *To space* now becomes the *From space* and continues to do so until it runs out of memory.

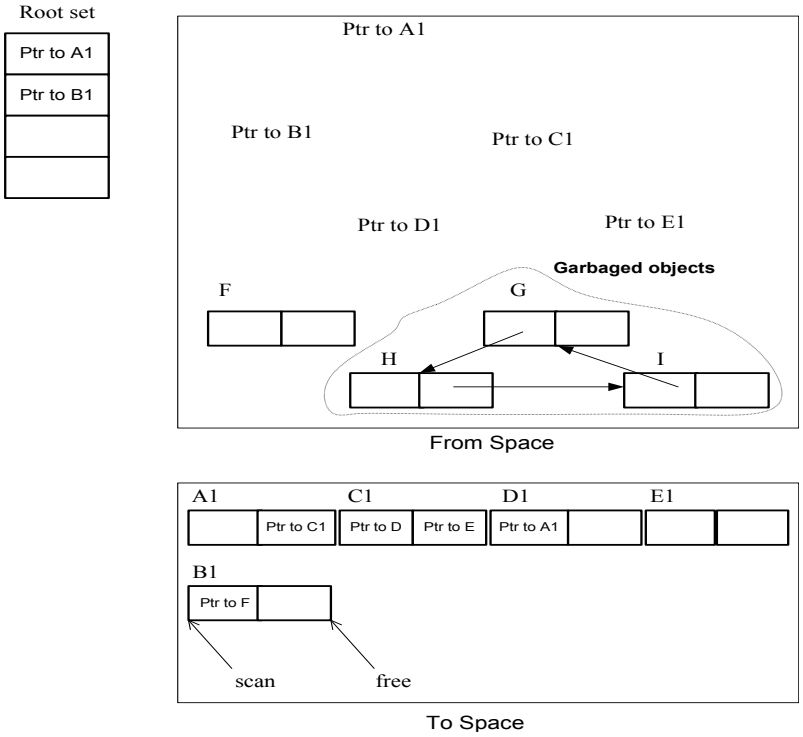


(a) Before garbage collection.

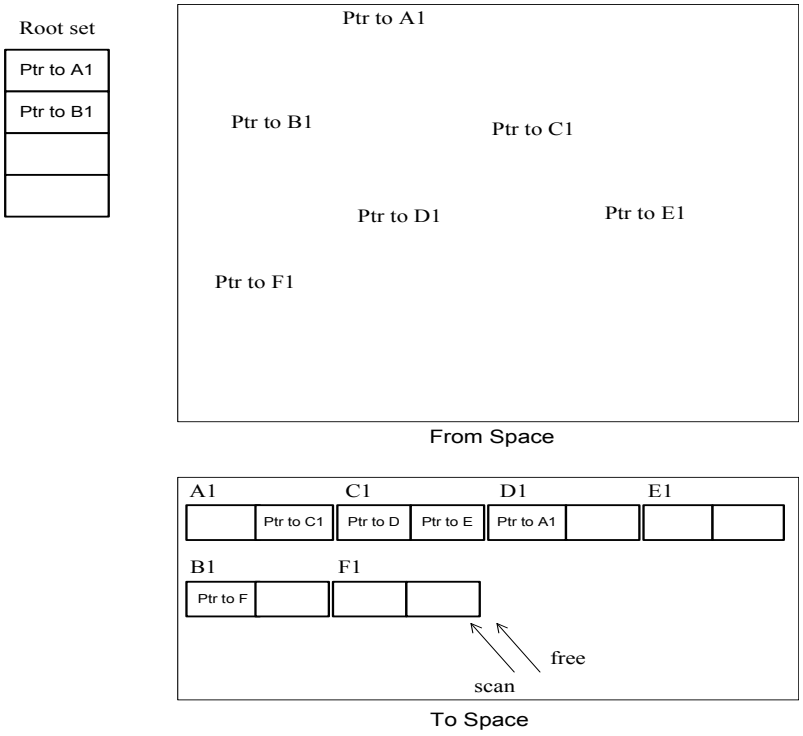


(b) Finished scanning object A.

Figure 2.1: Semispace copying garbage collection using Cheney's traversal algorithm.



(c) Finished scanning object B.



(d) After garbage collection.

Figure 2.2: Semispace copying garbage collection (contd.)

Figures 2.1 and 2.2 show a semispace copying garbage collector in action. The rootset has pointers to objects A and B. During garbage collection, objects are traversed starting from the root pointers and copied into the *To space*. The objects G, H and I are not reachable from the rootset and hence are garbage. The main advantages of copying garbage collection are ease of implementation and the availability of a compacted heap which makes allocation fast. The main disadvantage of copying collectors is that objects that live longer are copied over many times before they die. This increases garbage collection overhead and can slow down the main program. From the memory system point of view, semispace collectors cause too many page faults. This is because during garbage collection the entire heap is touched and there is a good chance that in betweeen collections, the idle *To space* has been paged out.

2.3 Generational garbage collection

Generational garbage collection divides the heap into subheaps (referred to as generations) based on age of the objects. An object is "born" in the youngest generation and gets promoted to older generations as it "lives." Generational collection works on the notion that in languages like Smalltalk, where objects are created rapidly, most objects do not live for a long time, i.e., they die young. By collecting younger generations more often, the garbage collection effort is focused on regions most likely to produce garbage. Objects that survive long enough get promoted to older generations. This process is called *tenuring*. The decision to collect younger generations more often is not a choice the garbage collector makes dynamically, but rather is built into the design. This is because object creation rate far exceeds the object survival rate and younger generations fill up more quickly than older generations. The number of generations used and the garbage collection algorithm used to collect them are parameters that define a generational garbage collector. Below I discuss the two generation garbage collector employed by the Smalltalk system [23] we used in our study.

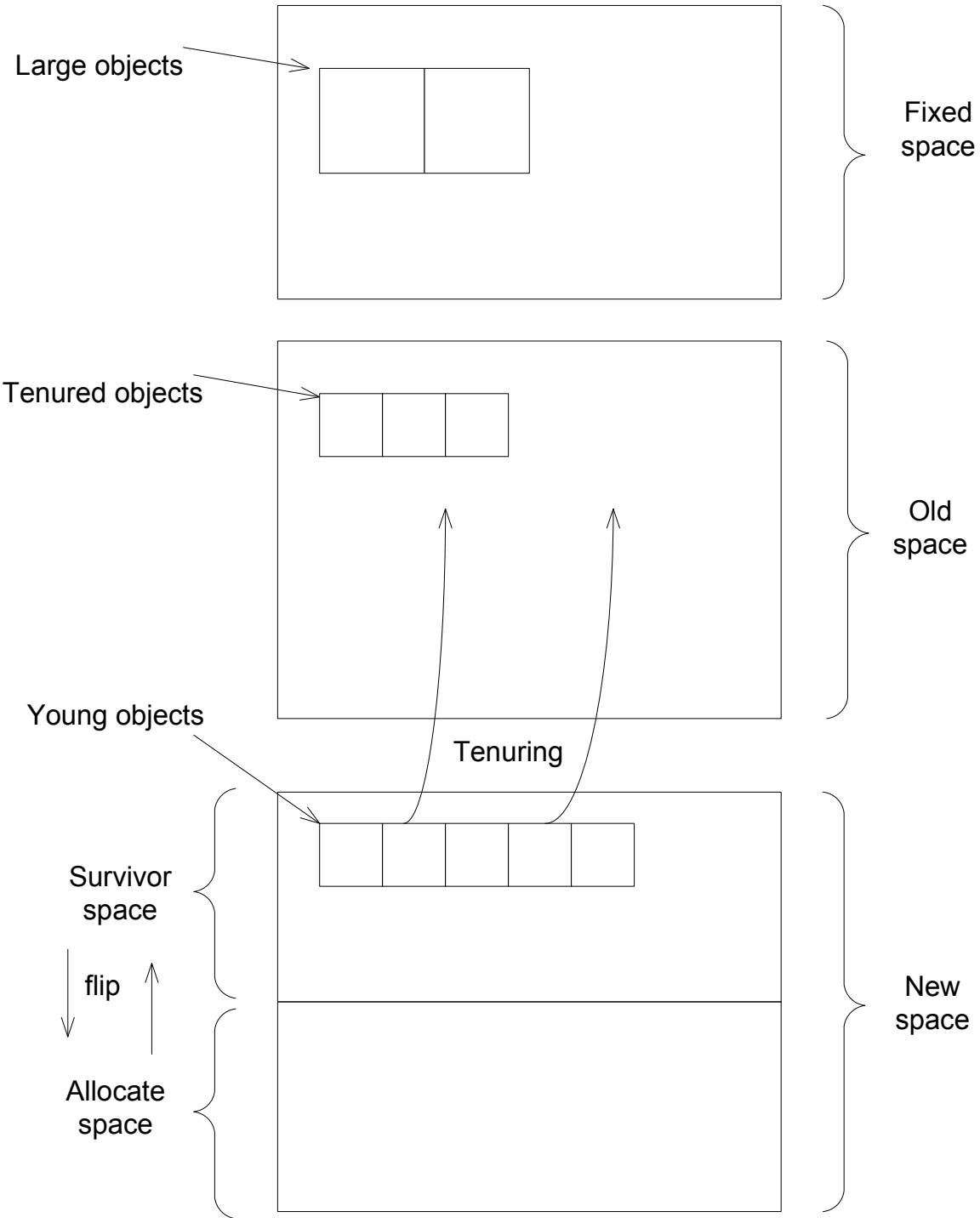


Figure 2.3: Generational heap used in IBM VisualAge Smalltalk

The IBM VisualAge Smalltalk system uses a generational garbage collector with four subheaps. Figure 2.3 shows the generational heap used. The four subheaps are referred to as: Allocate space, Survivor space, Old space and Fixed Space. The allocate and survivor spaces together are referred to as New space, as they constitute the region of heap that contains newly created objects. The old space contains objects that have been tenured from the new space. Some objects also get loaded directly into the Old space at Smalltalk system start up. The fixed space is mainly there to store large objects (like bitmaps) because it becomes really inefficient to move large objects around.

Two garbage collection schemes are used to manage the heap. A copying-style garbage collector is used to scavenge just the New space and a mark-sweep garbage collector [**?**] is used to "global garbage-collect" the entire heap. The copying garbage collector works on the lines of the semispace collector introduced in Section 2.2. The allocate and survivor spaces are analogous to *From space* and *To space*. They flip roles at the start of each scavenge. During every scavenge live objects are identified by object traversal starting from the rootset. Objects that survive a scavenge increase their age by one. The tenuring policy employed in this Smalltalk system is very simple. When an object becomes ten or older, it gets tenured to old space. More adaptive policies are possible [2]. The global garbage collector is a mark-sweep garbage collector that gets invoked when the old space fills up. Wilson and Jones [5, 6] discuss mark-sweep collectors in great detail. In this study we are interested in the more frequent garbage collection, the copying garbage collection. Figures 2.4 and 2.5 show the scavenge frequency and global garbage collection frequency respectively during execution of Swim, a SPEC95 benchmark used in our studies. This data was gathered by collecting heap information at the beginning and end of each garbage collection. The benchmark had to be run with a larger data set to collect global garbage collection information. As observed, during program execution scavenging occurs every few milliseconds compared to global garbage collection which occurs every few seconds. Also, the number of scavenges increases from an idle Smalltalk system to when the program begins to execute. This is seen by the dense lines that represent memory allocation in Figure 2.4.

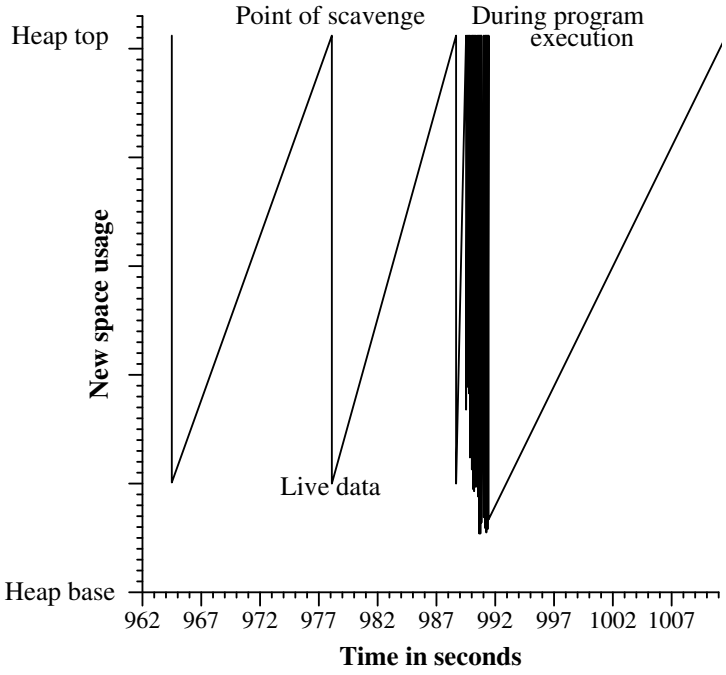


Figure 2.4: Scavenge frequency in IBM VisualAge Smalltalk system, for Swim.

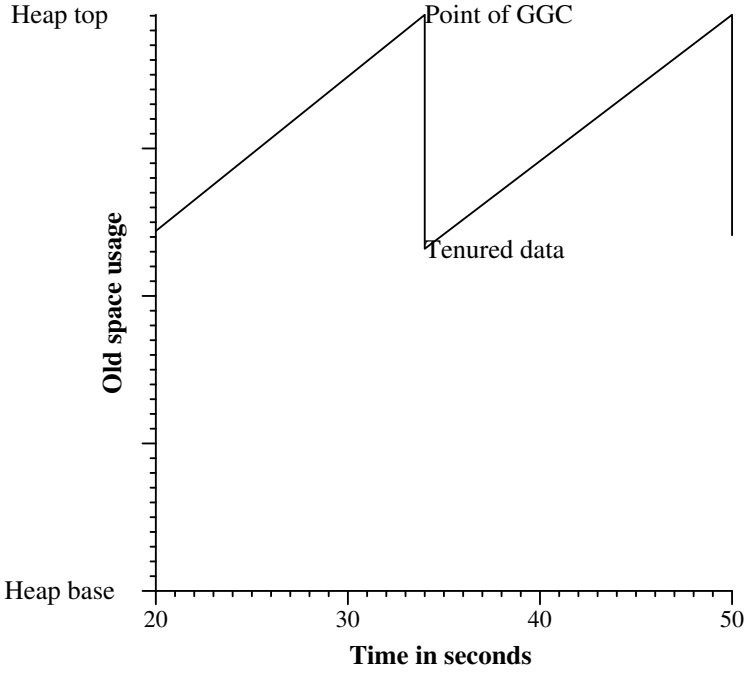


Figure 2.5: Global garbage collection frequency in IBM VisualAge Smalltalk, for Swim

2.4 Related work

This section discusses previous work related to our study. Our work relates with two research areas in the fields of computer architecture and memory management – garbage collection and caches. We propose to improve generational garbage collection performance through a strategy that reserves the youngest generation in a cache. The most relevant work to this thesis is by Wilson, Lam and Moher [8]. They recognize that the cyclic allocation pattern in generational garbage collected programs is the main cause for poor cache performance. They use a copying generational collector with two spaces like we do and study the effect on cache performance when using direct-mapped and set-associative caches larger than the memory reuse cycle. They show that while a direct-mapped cache is capable of removing capacity misses, set-associative caches do better by removing conflicts as well. Our study goes a step further and reserves the memory reuse cycle (youngest generation) in the cache. We report miss rate improvements over the base case of using a large enough cache and regular caching policies, hence indicating more room for improvement than results shown in [8].

Zorn evaluates the effect of garbage collection on cache performance [9]. He compares the cache performance of generational mark-sweep and copying algorithms and shows mark-sweep algorithms have better hit rates on direct-mapped caches while copying algorithms do well with set-associative caches. He also proposes allocation threshold (where scavenging starts only after a certain amount of memory is allocated) as a parameter for tuning cache performance and finds that in data caches large enough to hold a program's youngest generation, the configuration of the garbage collection algorithm can significantly alter the cache performance. In our study we do assume data caches large enough to fit the youngest generation, but we do not explore changes to garbage collection algorithms to improve cache performance. Our strategy exploits a fundamental aspect of any generational garbage collection algorithm to suggest a caching mechanism that would reduce cache misses. Hence, we study a more general solution to improving garbage collection performance.

Goncalves and Appel [10] study the effect of fitting the allocation space in a cache for Standard ML programs with intensive heap allocation that use generational garbage collection. They assume smaller caches and study the trade off between increasing garbage collection frequency and choosing an allocation space that fits in the cache. We assume

caches large enough to hold the allocation space which is more common today than it was at the time when their study was conducted.

Boehm proposes two techniques to reduce garbage collector misses: "prefetch-on-grey" and lazy sweeping for mark-sweep collectors [11]. While his study focuses on modifying the garbage collection algorithm, ours is more at the archiectural level.

Other studies have also suggested hardware changes to improve performance in programs that are allocation intensive [13]. We propose a software only solution that makes it attractive to implement in existing systems.

Chapter 3

Design and Implementation

The goal of my thesis is to make the youngest generation available in the cache from program start to end. To do so, I assume a cache size greater than the youngest generation. The fraction of cache reserved for the youngest generation is a parameter in my experiments. The strategy works by reserving a portion of the cache and finding page frames that would map to this region of the cache. The page replacement policy is made aware of this information so that only pages containing the youngest generation are placed in these frames. This removes conflicts to this reserved region of cache and hence prevents the youngest generation from being evicted. No changes are required in the caching policies. The page-fault manager needs a slight modification in that it has to be aware of page frames reserved for the youngest generation while swapping a page into memory. In the following sections, the different strategies adopted are described along with a possible implementation in a real system. Section 3.1 discusses the strategy for direct-mapped caches and Section 3.2 describes the same for set-associative caches. Section 3.3 explores an implementation of the strategies in a real system.

3.1 Direct-mapped caches

In direct mapped caches, we map the youngest generation to a certain number of cache lines whose size is large enough to hold the generational garbage collector's new

space. In our implementation we choose to reserve the bottom (highest numbered) few lines of the cache. Next, we identify page frames which contain memory blocks that map to the reserved cache lines. This is obtained by dividing the physical address space into regions containing as many lines as the cache. The bottom few lines of each region (equaling the number of lines reserved for youngest generation) map to the reserved region of the cache. This is due to the mapping policy of a direct-mapped cache. We call the collection of such lines in each region a "bucket." The goal of our strategy is to place the youngest generation in these buckets. For a direct mapped cache, the size of each bucket is equal to the size of the youngest generation and the buckets conflict in cache. For a set-associative cache this situation is different as will be seen in the next section. To avoid cache conflicts to the youngest generation, one of the buckets is used to hold the youngest generation and the rest are left as unusable for this Smalltalk process. The page frames comprising each bucket are identified and passed on to the page-fault handler of the operating system, which uses it in its page replacement logic. The page replacement logic has to be altered in the following way:

1. To service a page-fault from a non-youngest generation address, it should not pick any listed page-frame.

2. To service a page-fault from a youngest generation address, it should only pick a page-frame belonging to one of the buckets.

Figure 3.1 shows an example of applying this strategy to a 1MB direct-mapped cache on a system with a physical address space of 4096 page frames, each 4KB large. A youngest generation of size 256 KB will require 64 pages for storage. We reserve the bottom 256 KB (8192 lines) of the cache for the youngest generation. A bucket consists of the last 64 page frames occuring at any 1MB (size of the cache) boundary in physical address space. Only one of these buckets is chosen to contain the youngest generation. The remaining buckets are unused to avoid conflicts. The list of buckets and the page frames they contain are passed to the page-fault handler to implement the replacement policy described above.

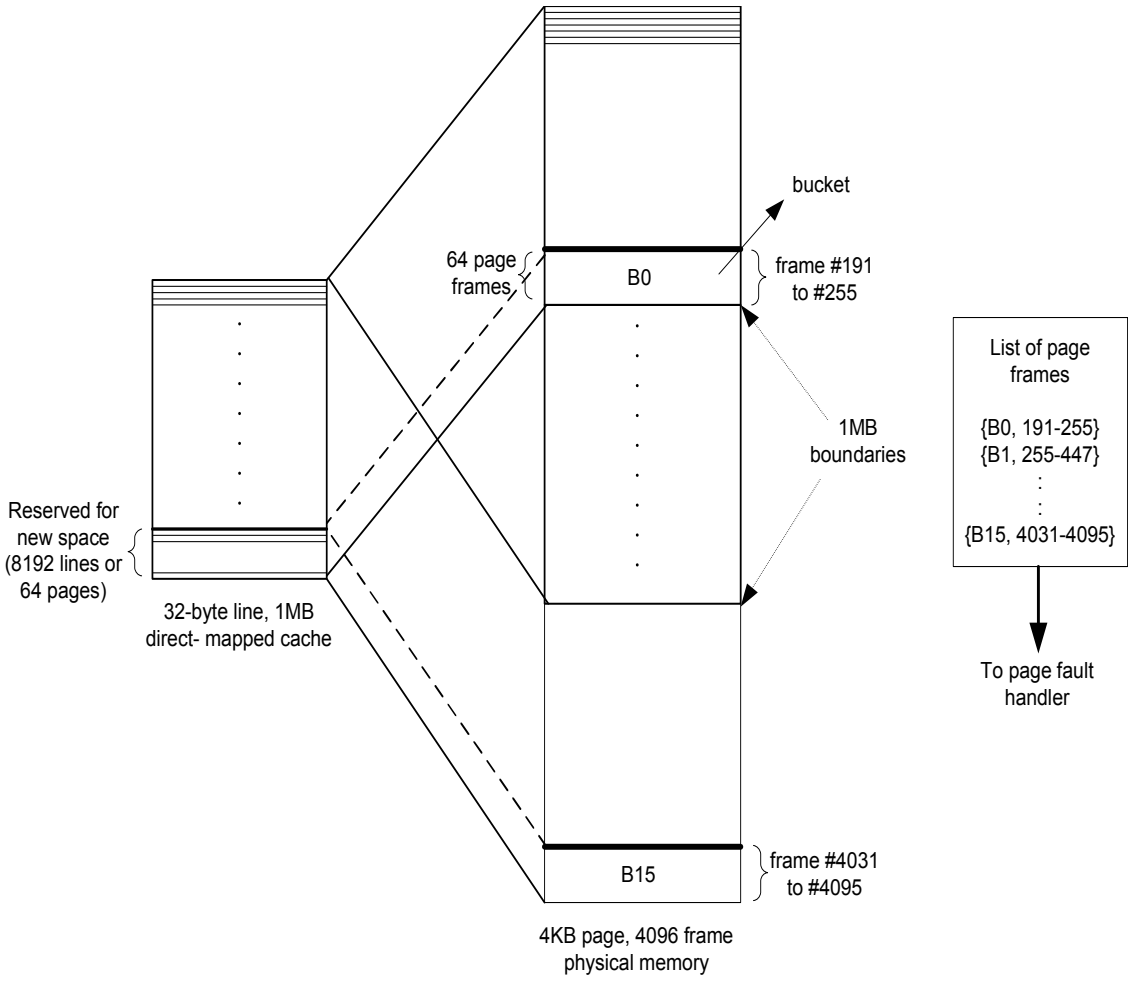


Figure 3.1: Mapping strategy for a 32-byte line, 1 MB direct-mapped cache to reserve a 256KB large youngest generation in the cache. Here the bottom 8192 lines are reserved for the youngest generation. Each bucket is formed by taking the bottom 64 page frames (8192 lines) at 1 MB boundaries (size of the cache). Only one bucket among the fifteen buckets contains the youngest generation. The choice is left to the page-fault handler.

3.2 Set-associative caches

This section extends our design to set-associative caches, which are more common at the L2 cache level. Our design assumes a cache large enough to hold the youngest generation. Modern L2 caches meet that requirement. We explored two ideas to reserve a portion of a set-associative cache for the youngest generation. The first was to reserve a certain number of "ways" of a n-way set-associative cache. The problem with this approach is that it assigns the cache hardware the task of ensuring that the reserved cache line is filled only with a youngest generation block. For example in Figure 3.2, one way of a 4-way set-associative cache is reserved. Memory blocks M1, M2, M3, and M4 map to set S1. Suppose M1 is defined to contain only a youngest generation block. Even if the page-replacement policy [1] is made aware of and it religiously follows the rule, M2, M3 and M4 have an opportunity to fill the reserved line in set S1. One way to prevent this is not to allow page placement in frames containing M2, M3 and M4. This would leave a lot of memory unusable if it is applied for each reserved line. The other way is to modify the cache replacement policy to not evict M1 from cache. Since such policies are implemented in hardware we dropped this approach.

Instead we adopt an approach where a certain number of sets, large enough to fit the youngest generation, are reserved in the cache. We choose to reserve the bottom few sets of the cache in our implementation. This approach is very similar to the direct-mapped approach; here the physical address space is divided into regions containing as many lines as there are sets in the cache. A bucket in each region contains lines that map to reserved sets in the cache. However, unlike the direct-mapped case, the size of a bucket is not equal to that of the youngest generation. Rather, a certain number of buckets, equal to the associativity of the cache, make up the youngest generation. The other buckets remain unused. The process of finding page frames containing the buckets remains the same. The only difference is that the page-fault handler can pick frames from more than one bucket to service a non-youngest generation miss. The page-fault handler has to be altered the following way:

1. To service a page-fault from a non-youngest generation address, it should not pick any listed page-frame.

[1] The placement policy would be made aware that the frame containing M1 is to be used only by a youngest generation page.

2. To service a page-fault from a youngest generation address, n being the associativity of the cache, it should only pick a page-frame belonging to any one of n buckets.

Figure 3.2 shows an example of applying the above strategy to a 32-byte line, 1MB, 4-way set-associative cache for a system with a physical address space containing 4096 page frames, each 4KB large. The bottom 2048 sets (256KB) of the cache are reserved for the youngest generation. As shown in the figure, a bucket occurs at 256KB boundaries and contains 2048 lines (16 pages). Four such buckets make up the youngest generation and the remaining buckets go unused.

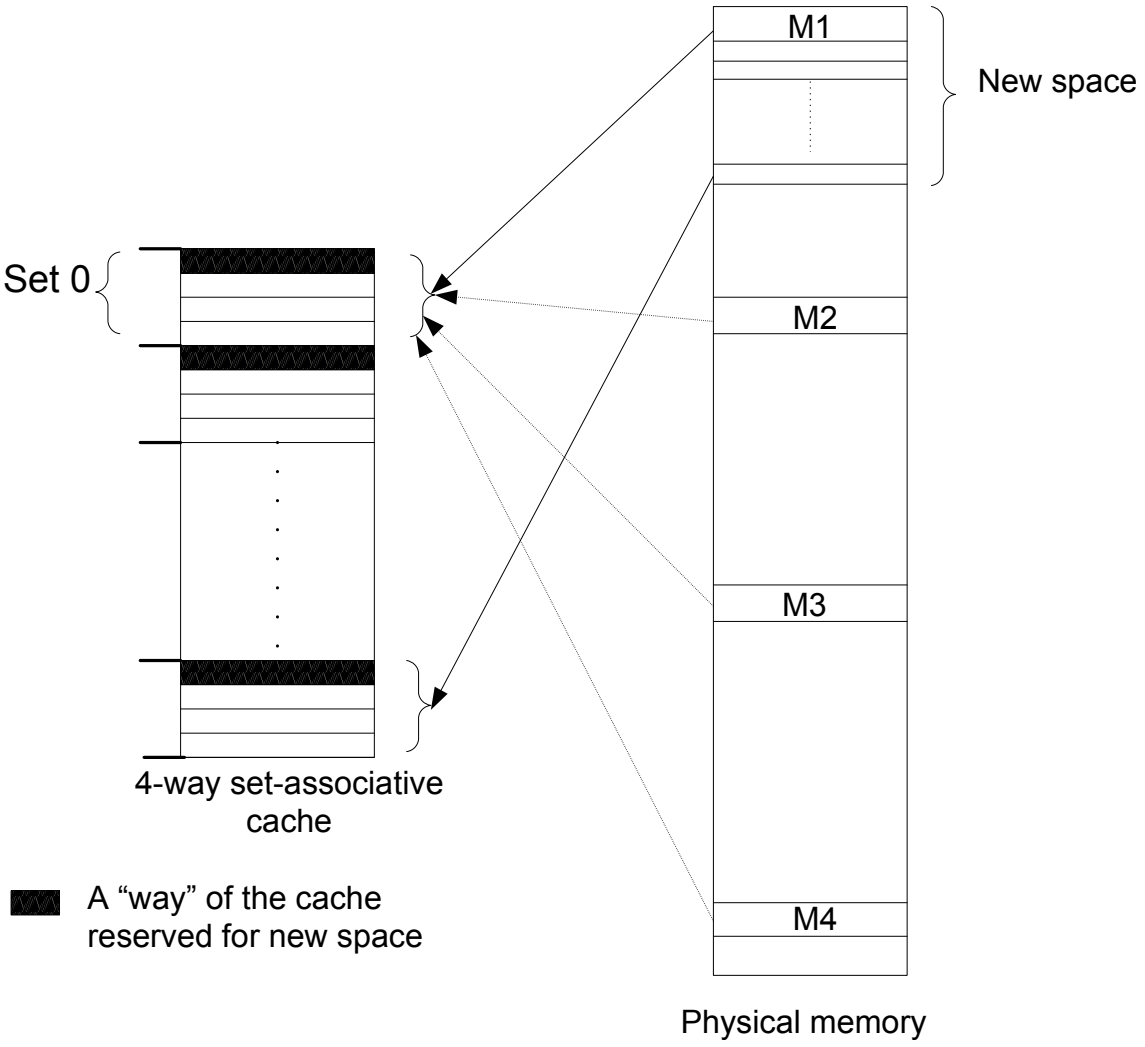


Figure 3.2: Mapping strategy for a set-associative cache where a way of the set-associative cache is reserved for the youngest generation. Here memory block M1 belongs to new space whereas M2, M3 and M4 are memory blocks that map to the same set as M1. If a way of the cache is reserved, then M2, M3 and M4 have an opportunity to evict M1 and occupy the reserved way. Leaving blocks like M2, M3 and M4 unused will prevent this at the cost of lot of memory. The other way is to make the cache hardware aware of M1's mapping.

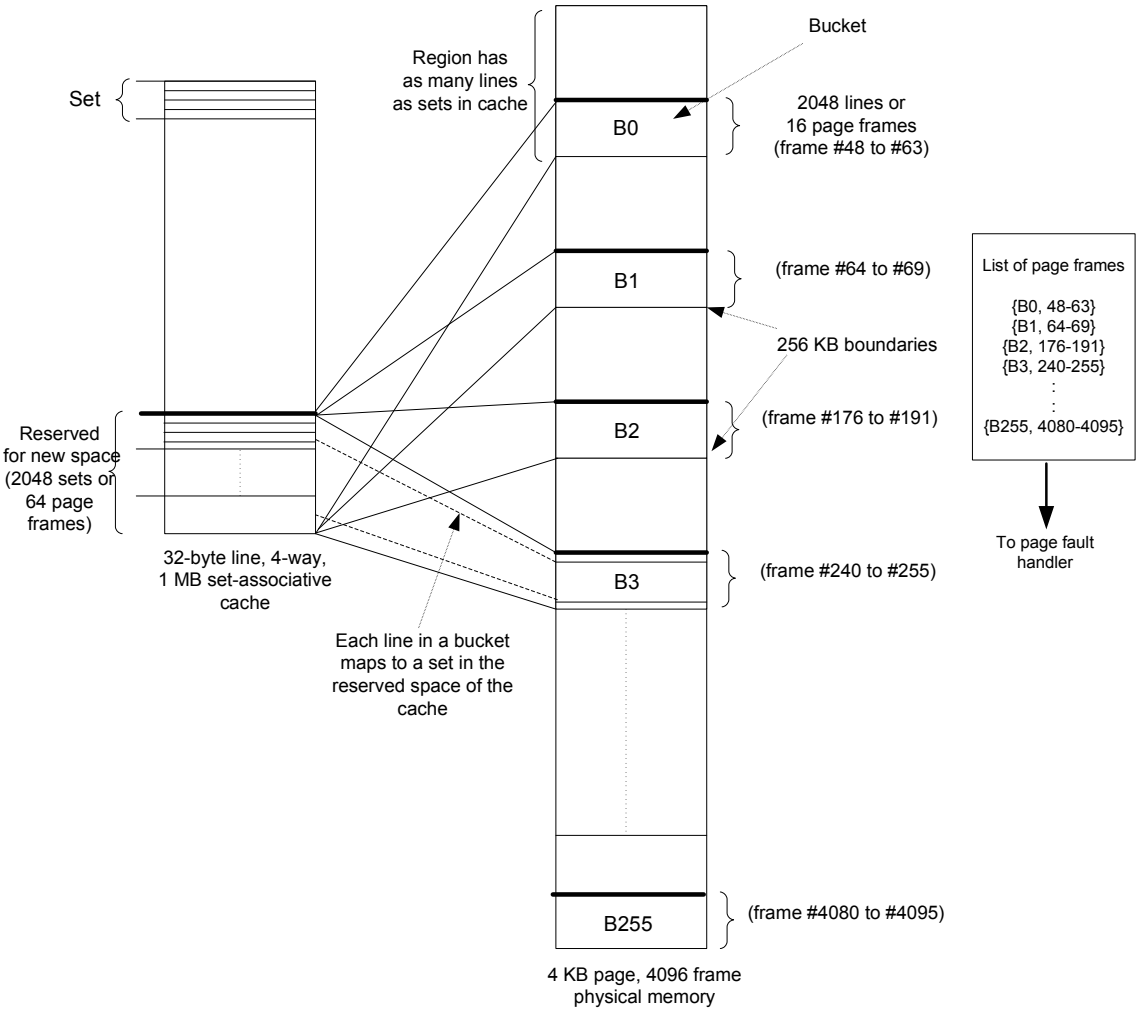


Figure 3.3: Mapping strategy for a 32-byte line, 4-way, 1 MB set-associative cache to reserve a 256 KB large youngest generation in the cache. Here the bottom 2048 sets are reserved for the youngest generation. A bucket is made up of as many lines as the reserved number of sets (2048 here). Four buckets are required to store the youngest generation. These could be any four from B0 to B255; the page fault handler makes this choice. In general, as many buckets as the associativity of the cache make up the youngest generation.

3.3 Implementation

In this section we discuss how our design could be implemented in an actual system. There are two main steps in our strategy that need implementation:

Step 1 Finding out the page frames that map to the reserved region of cache.

Step 2 To use information from Step 1 while making page replacement decisions.

Step 1 requires information about the cache, viz., cache size, associativity, and line size. It also requires information about the number of page frames available in the system and the page size. This step could be either implemented within the Smalltalk system or through a syscall to the operating system. Implementing it at the operating system level would be more appropriate as it would keep a programmer away from low-level details. The operating system will have to maintain additional information for the invoking process (in this case, the Smaltallk process) besides its page table, process control block etc. There are two types of information that need to be tracked: (a) A table containing page frames which should be used only for certain "special" virtual page numbers; (b) the range of virtual addresses (in our case, the top and base of the new space heap) that form the "special" pages mentioned in (a). Step 2 needs some coding in the page-fault handler of the operating system. Basically, the page-handler has to use the information stored in Step 1 while making its page-replacement decisions. If there is a page fault from a process that performed Step 1, the handler would check to see if the faulting address lies in the range of "special" addresses. If it does not, the "special" page frames are removed from the pool of frames considered to service the page fault. If it is a "special" address, only the "special" page frames are used to service the fault. Figure 3.3 is an outline of the implementation discussed above. As seen, our strategy could be implemented without any hardware changes, which makes it an attractive solution to reduce cache misses in an existing garbage collected system. The software modifications required are all at the operating system level and not very difficult to implement.

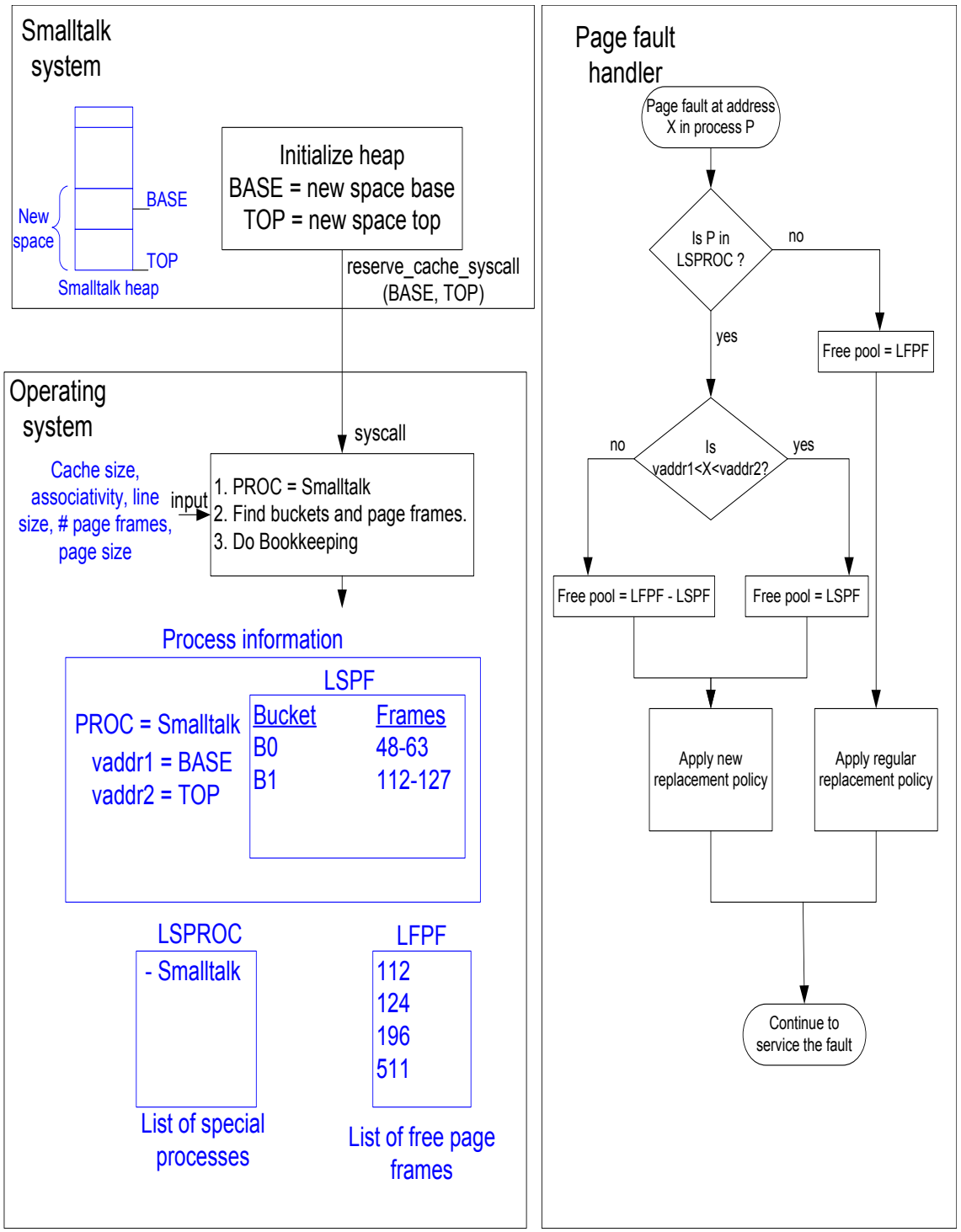


Figure 3.4: An implementation of the mapping strategy in a real system to reserve the youngest generation in the cache.

Chapter 4

Simulator and measurement tools

This chapter discusses the simulator and tools used in our studies. The organization of this chapter is as follows. Section 4.1 introduces Shade [29] simulator and Cachesim5 analyzer. Section 4.2 discusses the memory profiler that was developed by the author in IBM VisualAge Smalltalk.

4.1 Shade and Cachesim5

Shade is a fast instruction-set simulator from Sun Microsystems Inc. It reads instructions from a target application and dynamically compiles them to a set of host instructions, which get executed in a simulated environment. Our version of Shade simulated a target application compiled for SPARCv8 or SPARCv9 and executed on a SPARCv9 host machine running Solaris 5.7. Shade was suitable for our studies for various reasons. Firstly, Shade dynamically compiles target instructions to host instructions. This meant we just had to compile our code (including dynamic libraries) to SPARCv8, which was a supported target platform in Shade. For instance, we tried to use the Simplescalar architecture simulator which expects MIPS-like PISA binaries [30]. But the Smalltalk system we used linked with libraries that were not ported to Simplescalar. Porting would have been a big effort. Secondly, Shade is execution-driven, which saved us from collecting large traces. Target instructions are compiled to the host platform and executed in a simulated environment.

The instructions and their outputs are added to a buffer which an analyzer consumes. The amount of information added to the buffer can be controlled and affects simulation speed (lesser information traced, the faster the simulation runs). We did our cache studies using Cachesim5 analyzer that is part of the Shade simulation suite. It is a cache simulator capable of simulating multi-level caches. Modifications were made to Cachesim5 to include our cache mapping strategy.

4.2 Memory profiler

A memory profiler was integrated into the Smalltalk system to collect memory allocation statistics. The goal was to calculate the amount of memory allocated by a Smalltalk program. This helped our studies in the following ways. 1) We were able to choose benchmarks that were allocation intensive. 2) It helped us decide on simulation parameters, e.g. cache size and new space size. In IBM VAST, allocation takes place by calling any of the four methods implemented in the Smalltalk class `Behavior>>basicNew, Behavior>>new, Behavior>>basicNew:, Behavior>>new:`. The idea was to capture calls made to these methods by the application being profiled. To do so, the profiler exploits the Smalltalk message dispatch model and inserts profiling code to methods of "interesting" classes that make up the target application. To understand the profiler, an explanation of the Smalltalk message dispatch model is due.

In Smalltalk, methods belonging to a class are organized in a method dictionary. The dictionary is a one-to-one mapping between each method symbol (i.e., message selector) and the address of its compiled byte-code representation. Every Smalltalk class contains a reference to its method dictionary. When a message is sent to an instance I of class C, the runtime environment looks up the method dictionary of C. If the message is found, the corresponding compiled method is returned and executed. If it's not found, the method dictionary of C's superclass is searched and so on until the message is found somewhere up in the class hierarchy.

To profile a class, its existing method dictionary is replaced with a new method dictionary. In the new method dictionary, the original message selectors are mapped to instrumented compiled methods [24]. The instrumented methods contain profiling code wrapped around a call to a selector that maps to the original method. For example, consider

the method dictionary for class C shown in Figure 4.2. The message selector `sel1` is associated with `method1` in the original method dictionary. After profiling the new method dictionary contains additional entries. `key1_sel1` is now associated with `method1` and `sel1` is associated with an instrumented method. A unique prefix is required for each selector (as in key1, key2 etc.). This is because if a class overrides a method from the superclass, a call to the original method from an instrumented method in the superclass will end up being handled at its subclass. This instrumentation is applied to every class in the profiled application. To measure memory allocation, a flag is checked in each of the four allocation methods to enquire if the request is from one of the "interesting" classes, in which case the allocation size is recorded.

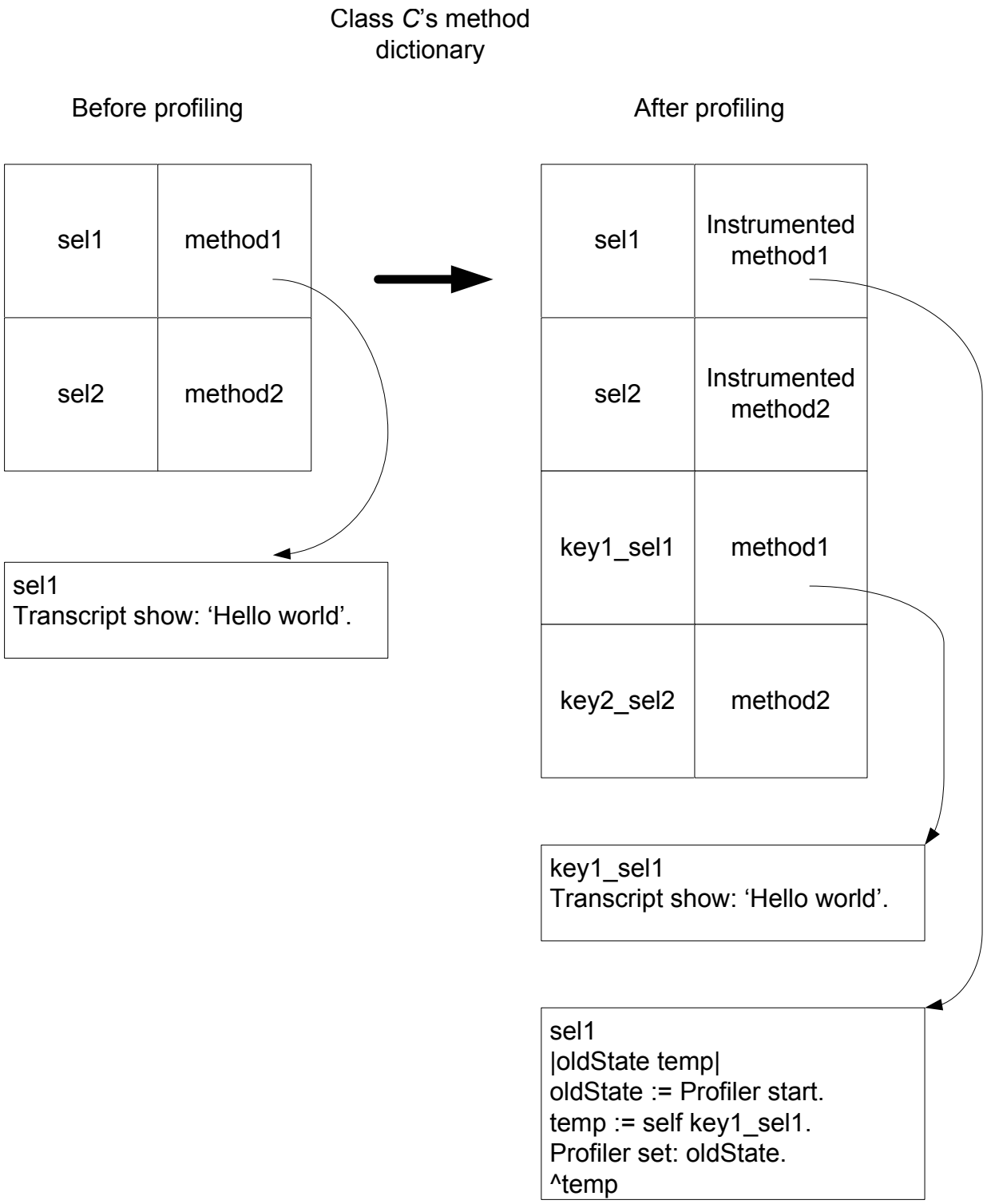


Figure 4.1: Method dictionary of a class before and after profiling.

Chapter 5

Methodology

This chapter discusses the simulation methodology used in our study. Section 5.1 discusses the implementation of our mapping strategy in the simulator. Section 5.2 discusses our method of acquiring heap information from the Smalltalk VM and feed it to the simulator.

5.1 Mapping strategy in the simulator

We do a simulation based study where we run IBM VisualAge Smalltalk on a cache simulator and measure cache performance with and without the mapping strategy. The cache simulator used in our studies (see Section 4.1) does not simulate the entire memory system; only the cache hierarchy is simulated. Also, virtual address references from the simulated application are used for cache simulation; that is, there is no virtual to physical address conversion. The *tag, index* and *offset* fields required to access a cache line are extracted directly from a 32-bit virtual address. Our mapping strategy, as explained in Chapter 3, involves two main tasks: (1) Locating page frames that map to the reserved region in cache, and (2) using this information to pick only certain "special" page frames for youngest-generation addresses, while leaving them out in picking a frame for other addresses. The outcome of this process is to convert virtual addresses to physical addresses with the following properties:

- The youngest generation addresses map to the reserved region of the cache.
- Other addresses *do not* map to the reserved region and hence, do not conflict with the youngest generation.

In our simulator, we simulate the behavior of the mapping strategy by translating virtual addresses to "real" addresses that have the appropriate properties. The target address space to which the virtual addresses are translated starts from zero and does not have an upper limit (like 4GB in 32-bit virtual memory). This might make addresses large, but the cache behavior is not affected by this. The cache will just require a larger tag. We made the *tag* field large enough to not overflow for any translated address. The translation process is a simple equation that takes a virtual address and returns an addresses conforming to the properties mentioned above. The code snippet below shows this in a `C`-like language for a direct-mapped cache.

```
get_phys_addr(vaddr) {
start_address = cache_size - size_to_reserve ;
if( heap_base <= vaddr <= heap_top) {
paddr = start_address +  (vaddr - heap_base);
} else {
paddr = vaddr + (vaddr/start_address) * size_to_reserve;
}
}
```

5.2 Feeding heap information to the simulator

The simulator requires the address range of the new space heap to implement the mapping strategy. The IBM VisualAge Smalltalk virtual machine was made to dump its heap details to a file. The simulator reads out the heap information from this file at the beginning of each run. For a given new space size, the new space address information remains the same for every run. One simulation run is required to just dump the heap details, which are used for future simulations. This avoided any direct interaction between the simulated program and the simulator to exchange heap details.

Chapter 6

Benchmarks

This chapter presents the benchmarks used in this study and the criteria used to select them. Section 6.1 discusses our benchmark expectations and Section 6.2 presents our final benchmark suite.

6.1 Benchmark expectations

Finding good benchmarks is a challenging task in any research. This is particularly true for Smalltalk where no established benchmark suites like SPEC exist. Our mapping strategy relies on a fact that a typical Smalltalk application allocates rapidly and frequent garbage collections are required on the youngest generation. A toy benchmark might not bring out the effects of our mapping strategy if it uses very little memory. Our main criterion for choosing a benchmark was memory allocation. We wanted benchmarks that allocated enough to warrant frequent scavenges for a reasonable new space size. The second criterion was that programs do useful work representative of real-world applications. Generally programs shorter than 100 lines of code were rejected. Smalltalk has been around for a long time and many flavors of Smalltalk have evolved. This meant some programs, though they had the characteristics we desired, could not be chosen because they were tightly bound to a particular flavor of Smalltalk and hard to port. The next section discusses our final benchmark set.

6.2 Final benchmark set

A set of seven benchmarks has been assembled. In most cases a port was required to IBM VAST. But some benchmarks were directly picked from IBM VAST and needed slight modifications to run in a non-interactive mode. The input data sets to the benchmarks had to be changed to have reasonable simulation times while not comprimising the validity of results. We picked data sets for which allocation was large enough to invoke several garbage collections. Below is a description of each benchmark and its input data set—

Tomcatv is a vectorized mesh generation program and is part of the SPEC95 floating point benchmark suite. It is compute and data intensive and uses very little I/O. The size of the problem can be varied by the number of elements in the 2-D data arrays. The arrays are initialized from a file containing X and Y coeffecients. Tomcatv contains a main loop that iterates 100 times. Each iteration contains several loop nests with many array accesses. The Smalltalk version of Tomcatv was written by Darko Stefanovic [21] and needed a port to IBM VAST. We chose grid sizes of 33, 100, 200 and 250 to have reasonable simulation times.

Swim is a scientific benchmark from the SPEC95 floating point benchmark suite. The program solves the system of shallow water equations using finite difference approximations on a NxN grid. The grid parameters $N1$ and $N2$ can be used to control the size of the problem. It has several loop nests and scattered accessses to arrays. The Smalltalk version of Swim was written by Darko Stefanovic [21] and needed a port to IBM VAST. We chose grid sizes of 33, 151, 201 and 301 to have reasonable simulation times.

Richards is an operating-system event-driven simulation benchmark due to Martin Richards from Cambridge University. It simulates the kernel task dispatcher of an operating system. The Smalltalk version was borrowed from the GNU Smalltalk package [22] and required a port to IBM VAST. We ran Richards with six tasks in the task list. This enabled a reasonable simulation time.

XML Parser Parser is a tree-based SAX (Simple API for XML) parser from IBM VisualAge Smalltalk [23]. It parses an XML document into an internal tree structure, allowing an application to navigate that tree. The file parsed in our experiments is

a XML file containing insurance policy information. After parsing the file, the nodes are traversed to emulate a typical XML application. The XML file size was restricted to 5MB for simulation purposes.

VM Maker is an IBM VisualAge Smalltalk [23] application that generates the Smalltalk virtual machine files in C and assembly for a specified architecture. In our experiments, we generate selected virtual machine files for x86 architecture. The number of files generated helped us control our data set and simulation time.

ES compiler is a compiler for IBM VisualAge Smalltalk. It accepts Smalltalk code and returns their bytecode representation. We compiled Smalltalk code filed out for basic Smalltalk classes like Object, Rectangle, Point, Process and few more basic classes. The number of input classes helped us control our simulation time.

Bench is a collection of useful programs that perform basic tasks like sorting, matrix multiply 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 posted on the message board `comp.lang.smalltalk` by Greg Gritton. A port was required to IBM VAST. (Should I include the list of programs here?)

Table lists the characteristics of all the benchmarks. Input column lists the data set for a particular benchmark. Benchmarks were run with different data sets to identify a suitable data set that allowed simulating the benchmark in reasonable time. Total allocation measures the memory allocated by the Smalltalk program from start to end. Number of allocations reflects the allocation rate of the benchmark. Both these statistics were collected using the memory profiler discussed in Section 4.2. Total execution time is the wall-clock time required to execute the benchmark and is comprised of garbage collection time and the mutator execution time (non-gc time).

Table 6.1: Benchmark characteristics

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 7

Experiments, results and analysis

This chapter discusses the experiments carried out in this study and the results obtained.

7.1 Garbage collection overhead

Garbage collection has overheads as many extra memory references are made to reclaim garbage. This experiment attempts to find where the overhead lies in a generational garbage collected program. The experiment was carried out on a Sun Ultra-250 machine with 1 GB of main memory. This allows us to keep page faults out from our analysis.

We increase the size of the new space until very few or no garbage collections[1] occur. Figures 7.1 and 7.2 plot the number of scavenges in a program against the size of the new space for all benchmarks. The main point the graph makes is that garbage collection activity reduces as new space size is increased. This occurs because there is more allocation space for a program before it runs out of memory.

[1] In our study, garbage collection is composed of scavenges to the new space and global garbage collections over the entire heap.

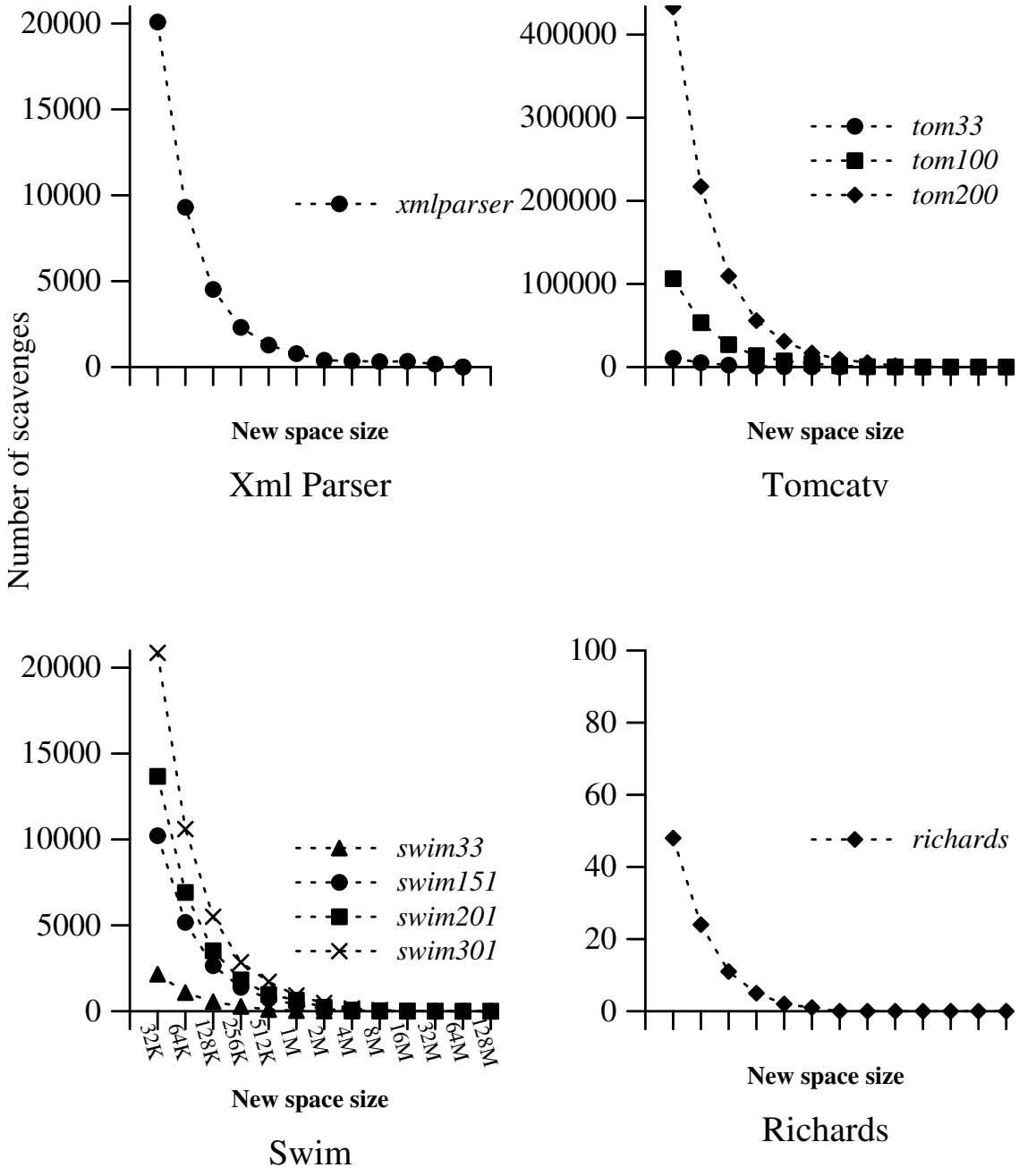


Figure 7.1: New space size vs. Number of scavenges, for Xml parser, Swim, Tomcatv and Richards.

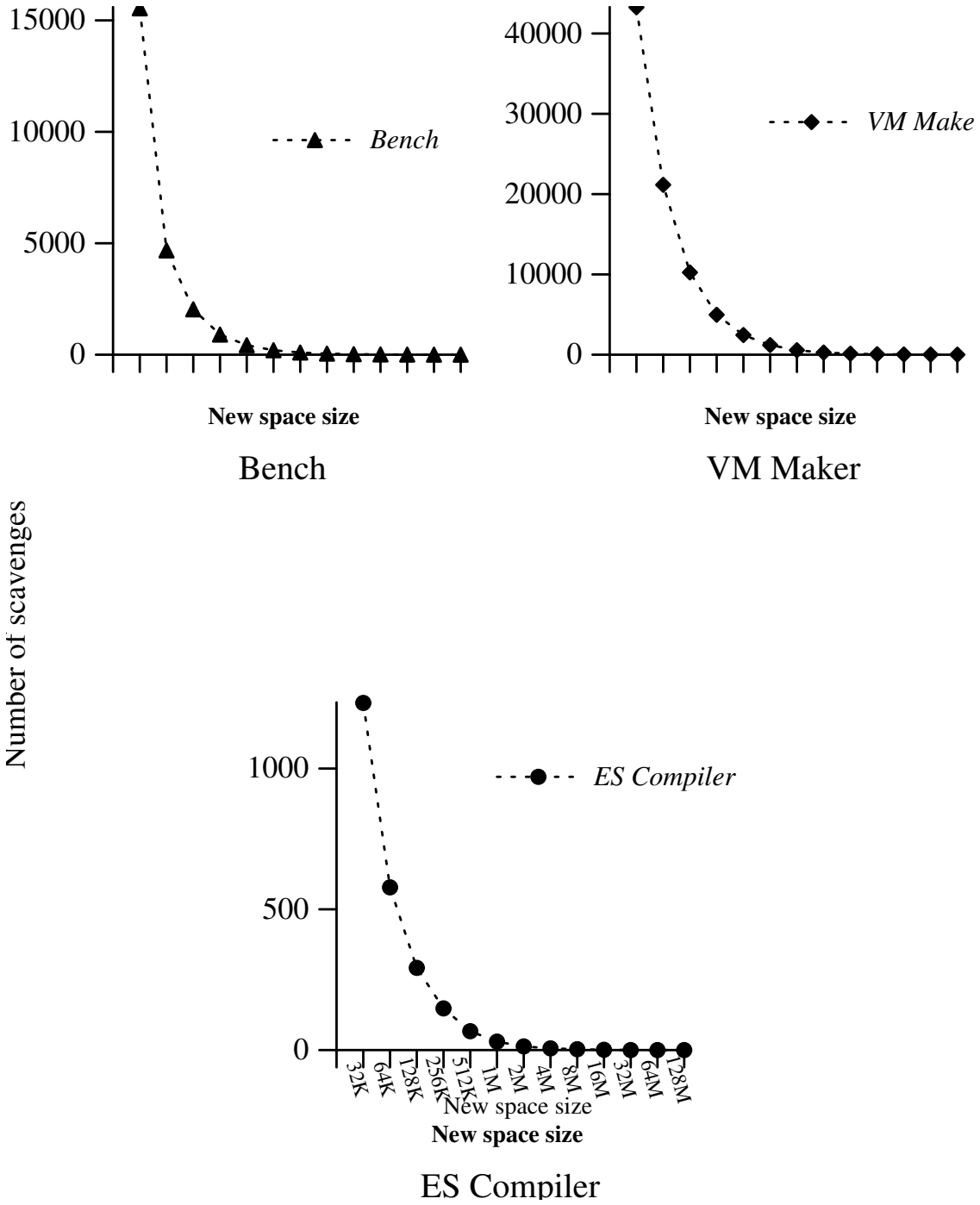


Figure 7.2: New space size vs. Number of scavenges, for Bench, VM Maker and ES Compiler.

In the next experiment, we profile the execution time of the programs for different new space sizes. The execution time is composed of different quantities –
Total exec. time - Wall-clock time to execute the entire program from start to end.
Total GC time - Time spent garbage collecting (any part of the heap) during execution.
Scavenge time - Time spent only in scavenging the new space.
GGC time - Time spent only in global garbage collection (entire heap).
Non-GC time - Time spent in actual program execution.
The relationship between them can be summed up as follows –

$$\begin{aligned} Total\ exec.\ time &= Total\ GC\ time + NonGC\ time \\ Total\ GC\ time &= Scavenge\ time + GGC\ time \end{aligned}$$

Figures 7.3, 7.4 and 7.5 show the execution time distribution for all the benchmarks. Space between the non-gc time curve and total execution time curve represents overhead. As seen, scavenge time forms a major fraction of the garbage collection overhead. The time spent in global garbage collection is only a fraction of the time spent scavenging. For larger data sets (like grid size more than 100x100 for Swim and Tomcatv) the scavenge time is many times the actual program execution time in the case of mid-level new space sizes. For smaller data sets (like Richards) the overhead is not prominent. This is because the garbage collector is invoked fewer times due to lesser allocation. Garbage collection overhead reduces as new space size increases. As shown in the previous experiment, this is due to fewer scavenges in a large new space. Another interesting sidenote is the increase in execution time for a very large new space. This occurs due to a loss of locality in the new space when garbaged data fragments the live data. As there are almost no garbage collections for a large new space this effect is prominent enough to increase execution time.

Overall, for all benchmarks, scavenge time is the most important variant in the total execution time. Scavenging is a memory intensive activity that mainly comprises of memory references to the new space. Making these references hit in the cache definitely promises an improvement in performance. This experiment confirms our belief that making the new space persistent in the cache would reduce garbage collection time and improve performance. The mapping strategies introduced in this thesis aim to do exactly this.

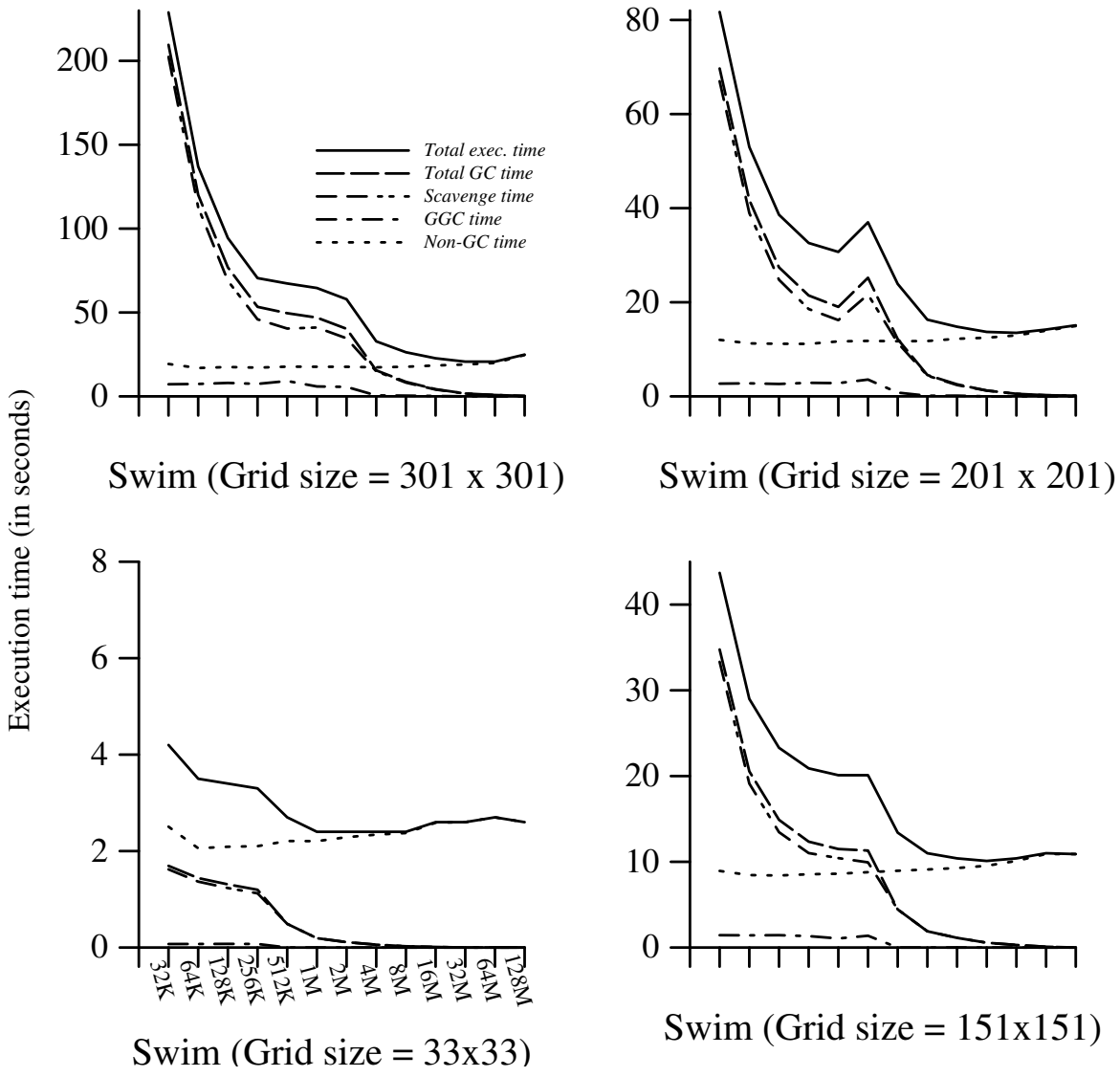


Figure 7.3: Execution time distribution wrt New space size, for Swim.

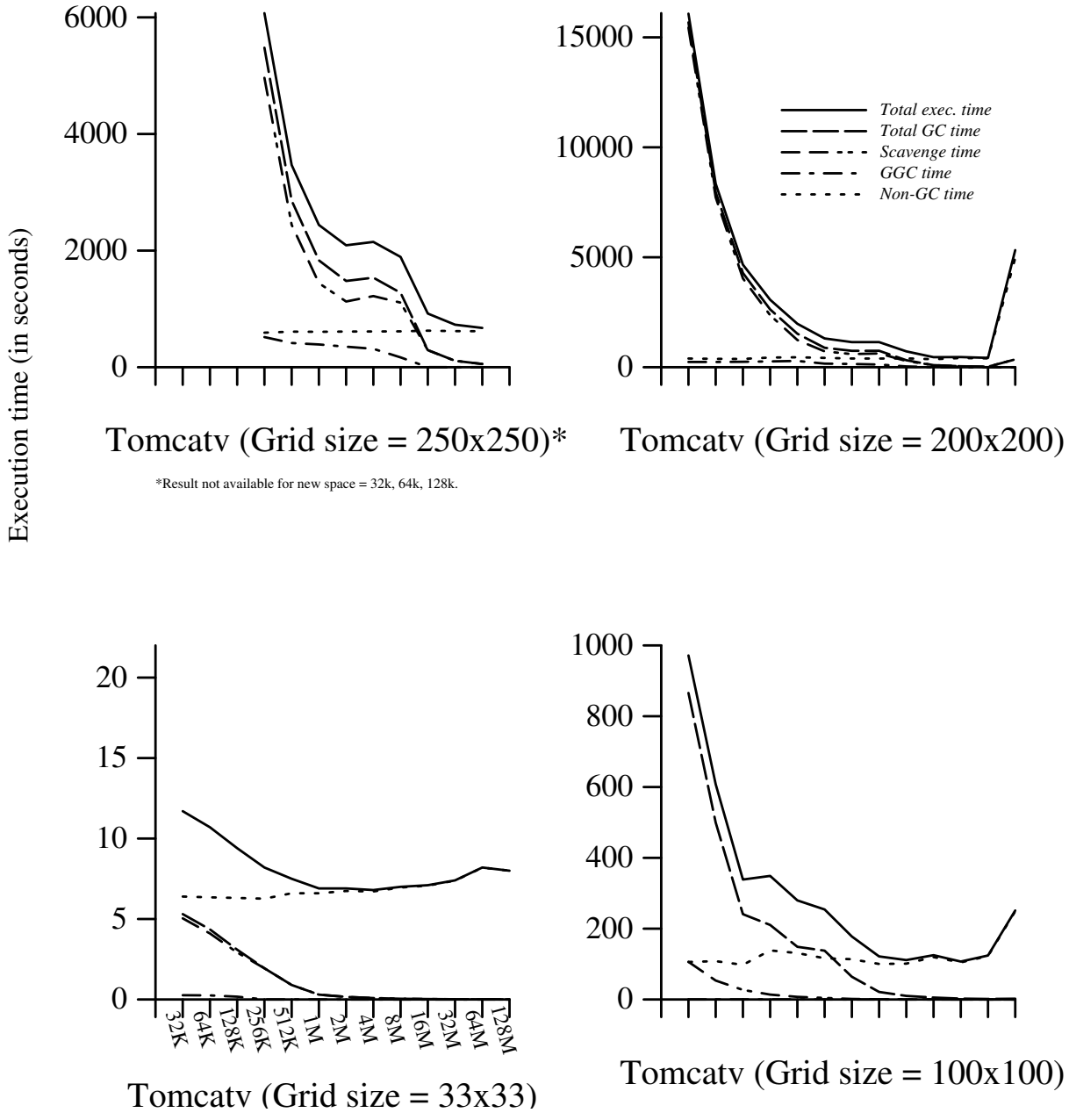


Figure 7.4: Execution time distribution wrt New space size, for Tomcatv.

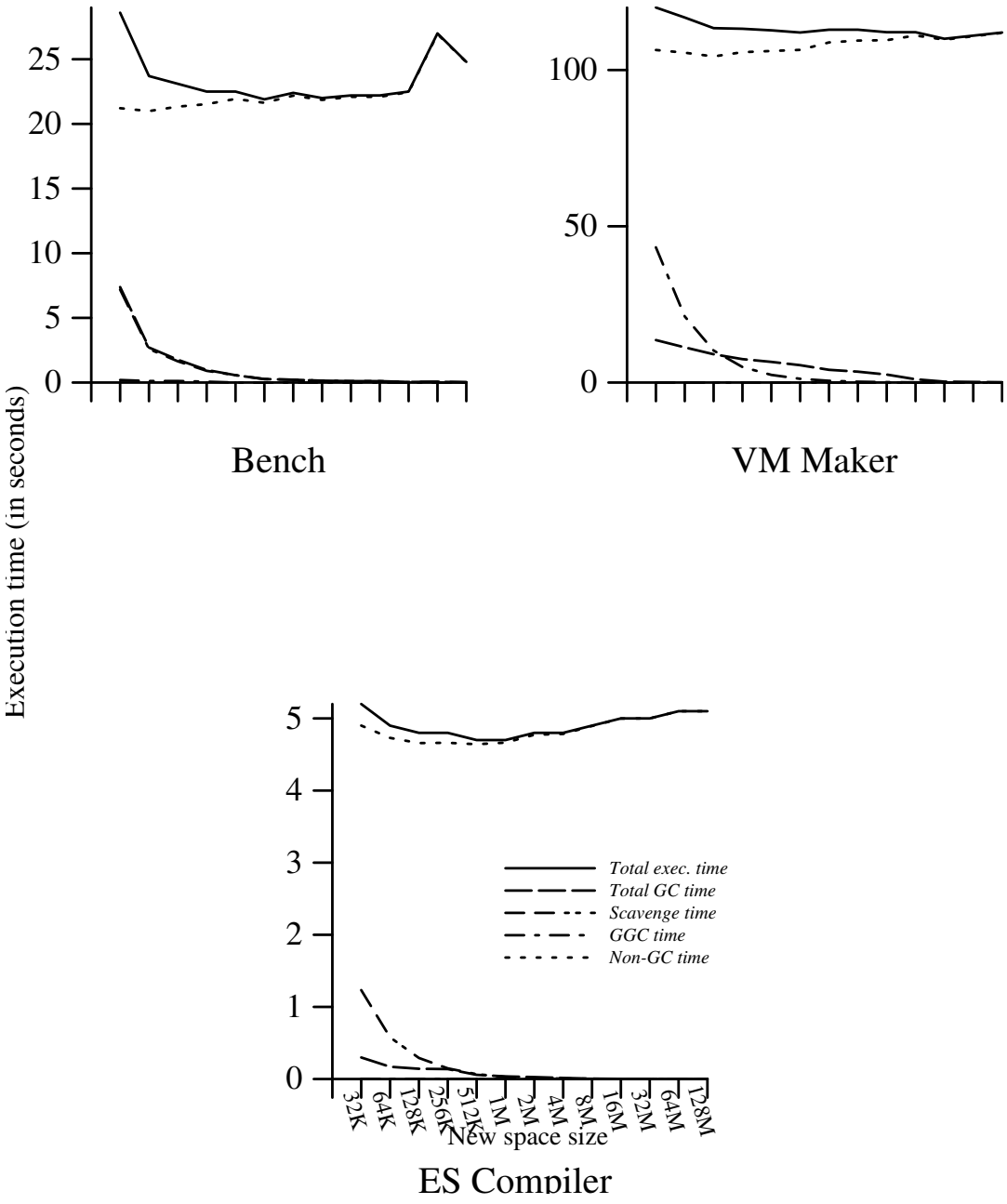


Figure 7.5: Execution time distribution wrt New space size, for Bench, VM Maker and ES Compiler.

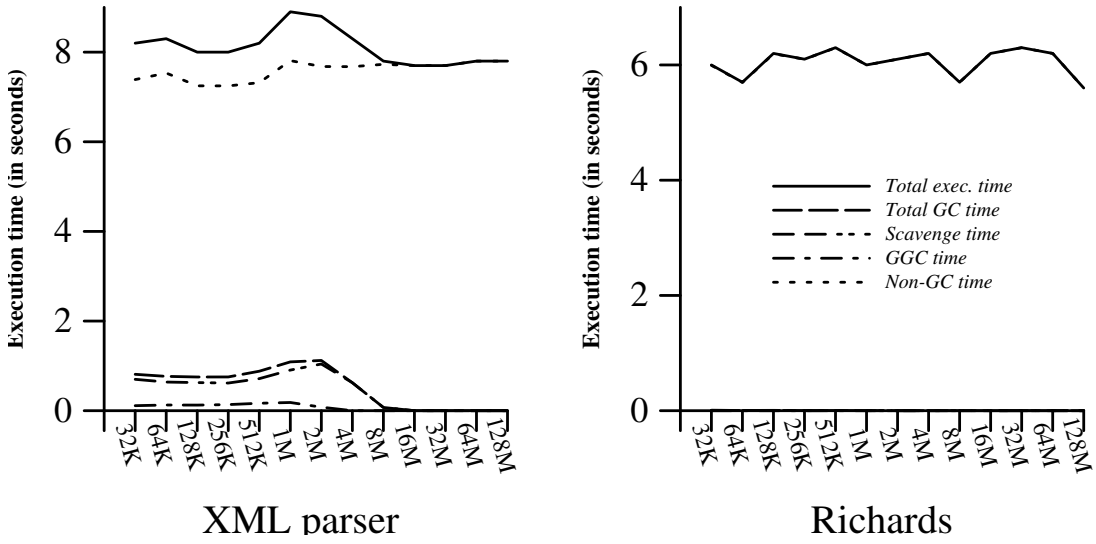


Figure 7.6: Execution time distribution wrt New space size, for Xml parser and Richards.

7.2 Cache simulations

The following assumptions are made in the cache simulations, unless otherwise specified. Instruction caches are not simulated in any of the experiments since we study our mapping strategy on data caches. A cache block size of 32 bytes is assumed for all experiments. For set-associative caches, a Least Recently Used (LRU) replacement algorithm is employed. For level1-only simulations, a write-back-with-allocate write miss policy is used. For level2 simulations, the level1 cache is assumed to be a direct-mapped, 16KB cache which employs a write-through write miss policy. The L2 cache in this case is assumed to have a write-back-with-allocate write miss policy.

7.2.1 Finding the right portion of the cache to reserve

In this experiment, we try to find the optimum portion of a cache that should be reserved for the new space. Only L1 caches are simulated and the effect of reserving 50%, 25% and 12.5% of the cache are observed. Figures 7.7, 7.8 and 7.9 show the L1 miss-rate improvements for the three cases. We find that reserving 25% of the cache provides more consistent gains than the other two cases. Reserving 50% of the cache seems to affect the locality of references to regions other than the new space. Figures 7.10 to 7.15 also show the variation in miss-rate percentages for the three cases. As seen, for most benchmarks reserving 25% of the cache seems to be a good choice.

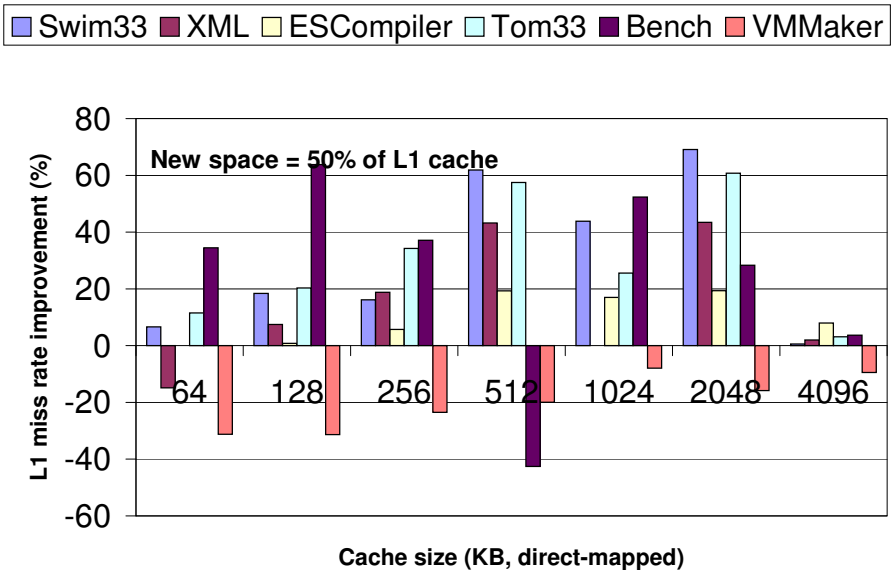


Figure 7.7: Cache miss rate improvement when new space is 50% of the cache size in a direct-mapped L1 cache.

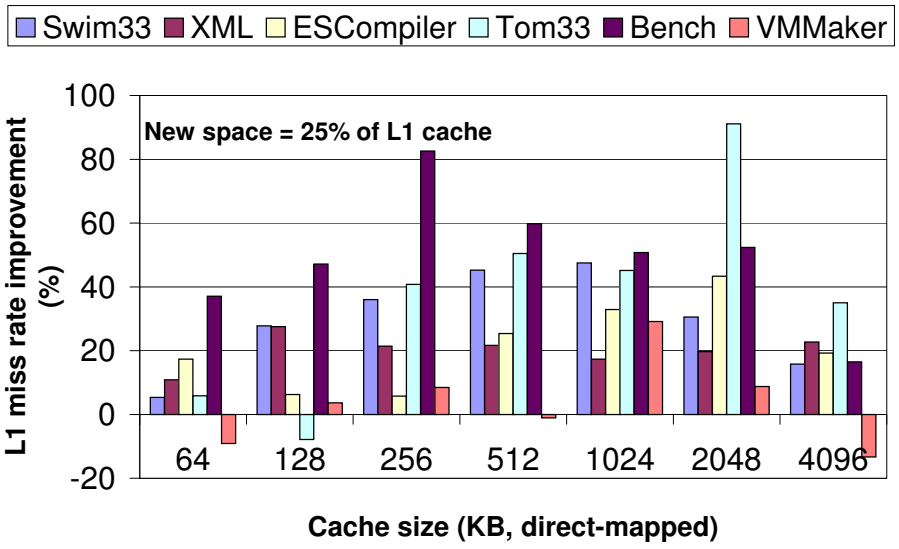


Figure 7.8: Cache miss rate improvement when new space is 25% of the cache size in a direct-mapped L1 cache.

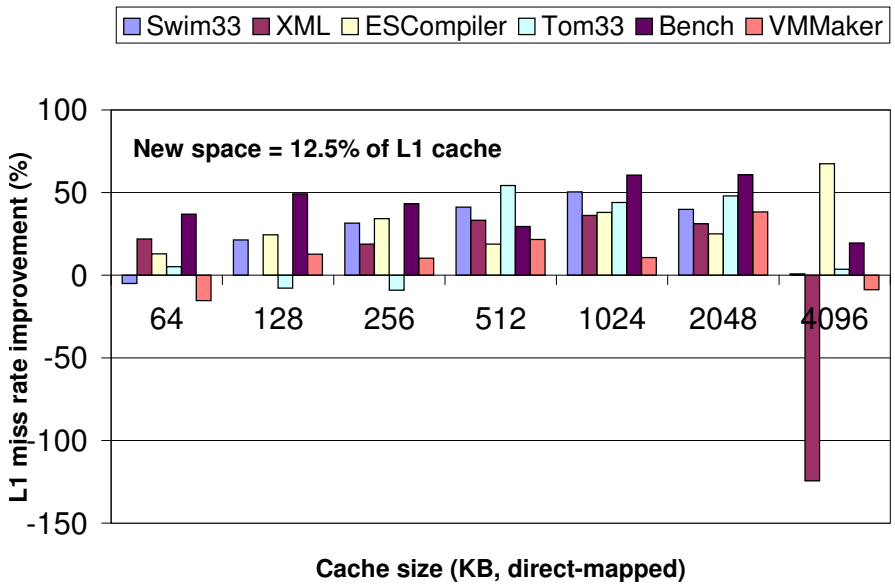


Figure 7.9: Cache miss rate improvement when new space is 12.5% of the cache size in a direct-mapped L1 cache.

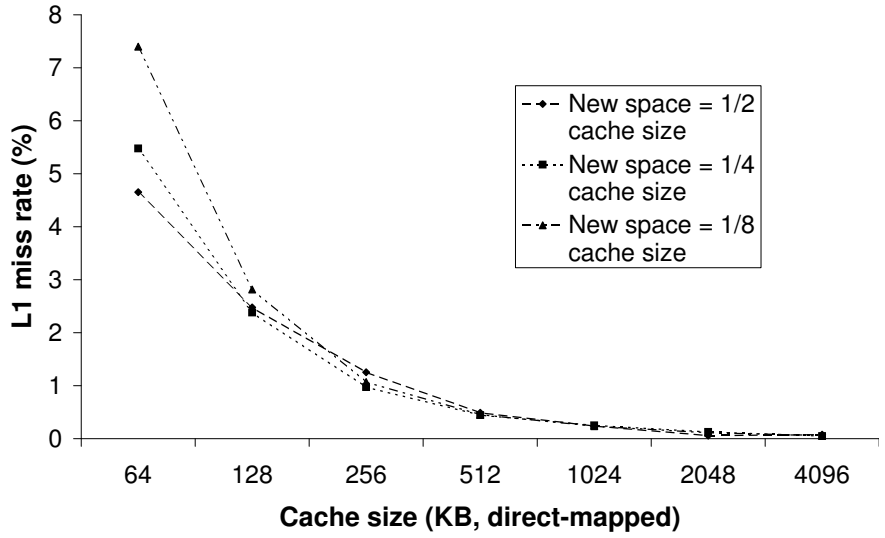


Figure 7.10: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Swim.

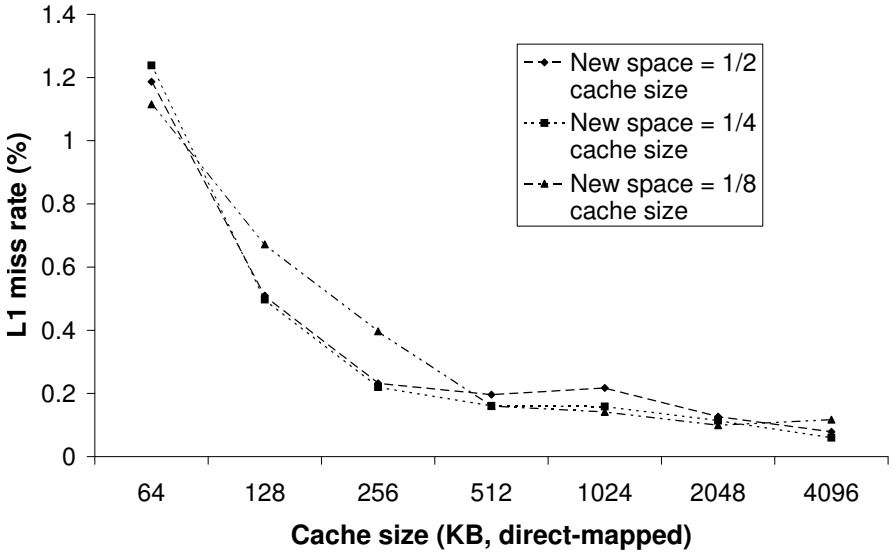


Figure 7.11: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Xml parser.

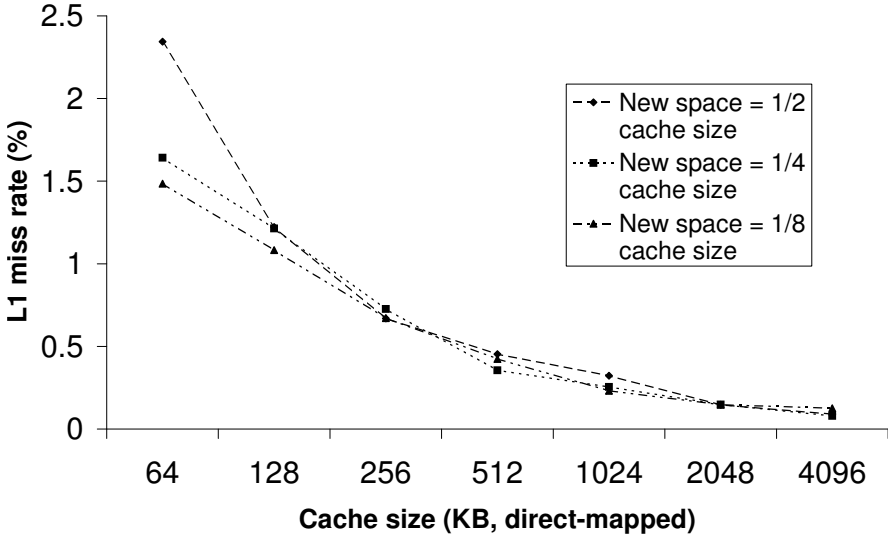


Figure 7.12: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for ES Compiler.

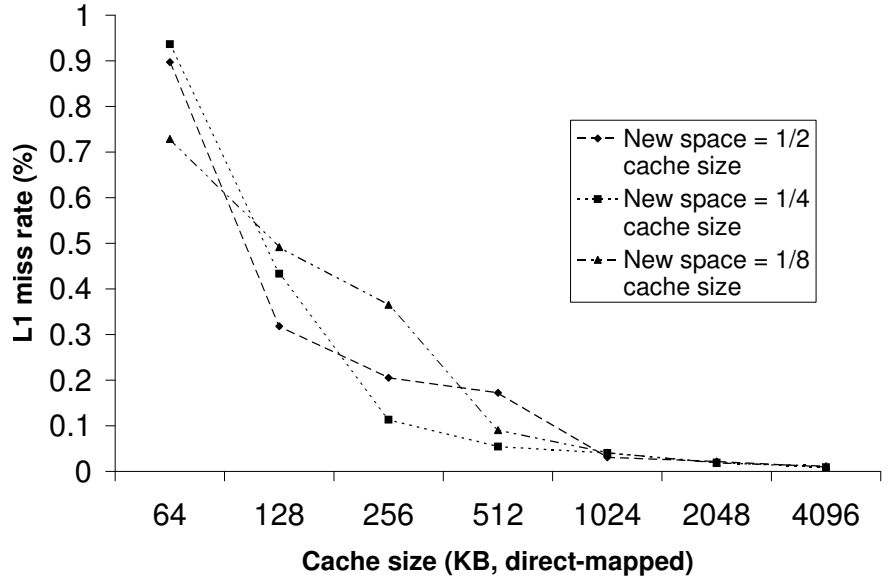


Figure 7.13: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Bench.

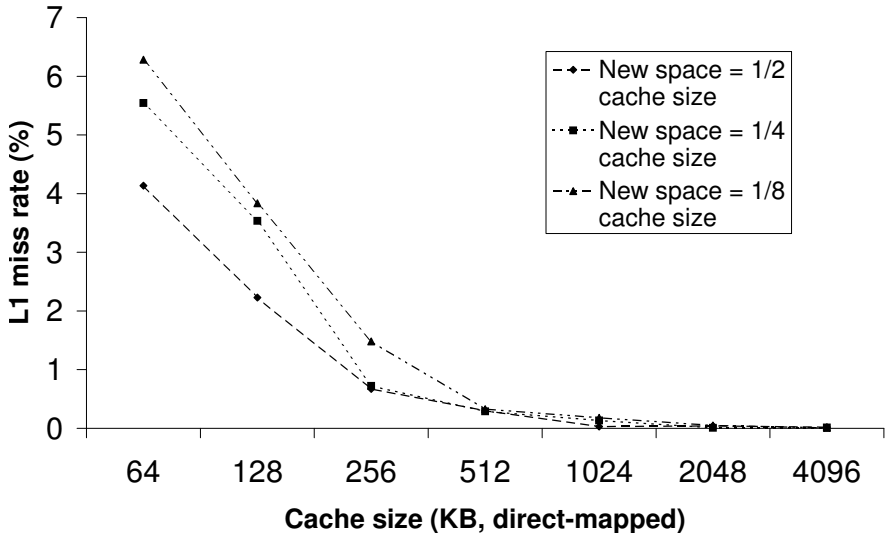


Figure 7.14: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for Tomcatv.

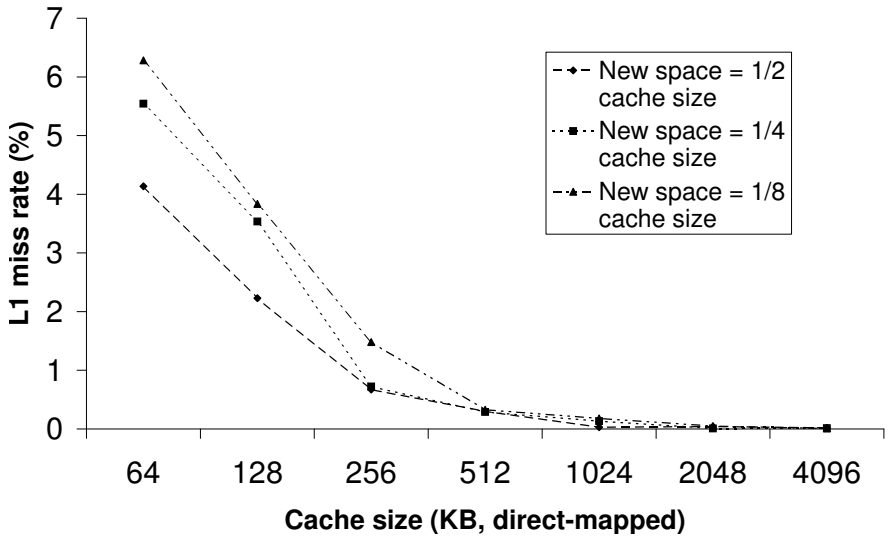


Figure 7.15: Miss rate comparison when new space is 50%, 25% and 12.5% of a direct-mapped L1 cache, for VM Maker.

7.2.2 Applying the strategy to a direct-mapped L2 cache

Figure 7.16 shows the miss-rate improvement when the strategy is applied to a direct-mapped L2 cache of various sizes. As seen, all benchmarks show a significant improvement in miss rates. The mapping is successful in removing all of the conflicts to the youngest generation. Tomcatv shows a 90% improvement in L2 miss rate. The average miss rate improvement is 45%, which is significant at the L2 cache level. Systems that use direct-mapped L2 caches (e.g. Sun Ultrasparcs I, II, IIi and AMD K6) would benefit from this strategy.

Figure 7.17 shows the effect of applying the strategy to the L2 cache on L1 cache miss rate. As seen, for most benchmarks there is not much change in the L1 miss rate, except for Xml parser. Xml parser reads a 1MB file and creates objects that live till the end of the program. Hence the liveness is more and percentage of objects getting tenured to old space is more. The mapping strategy for a direct-mapped cache places the youngest generation in one bucket. The mapping of this bucket to the L1 cache can have an effect on the L1 miss rates. In case of Xml parser, it seems like the conflicts between the reserved bucket and the old generation are more, which is degrading the miss rate.

Figure 7.18 shows the global miss rate change. There is an improvement in all cases, which shows the mapping strategy at the L2 level is able to reduce the number of accesses going to memory and hence reduce the memory traffic.

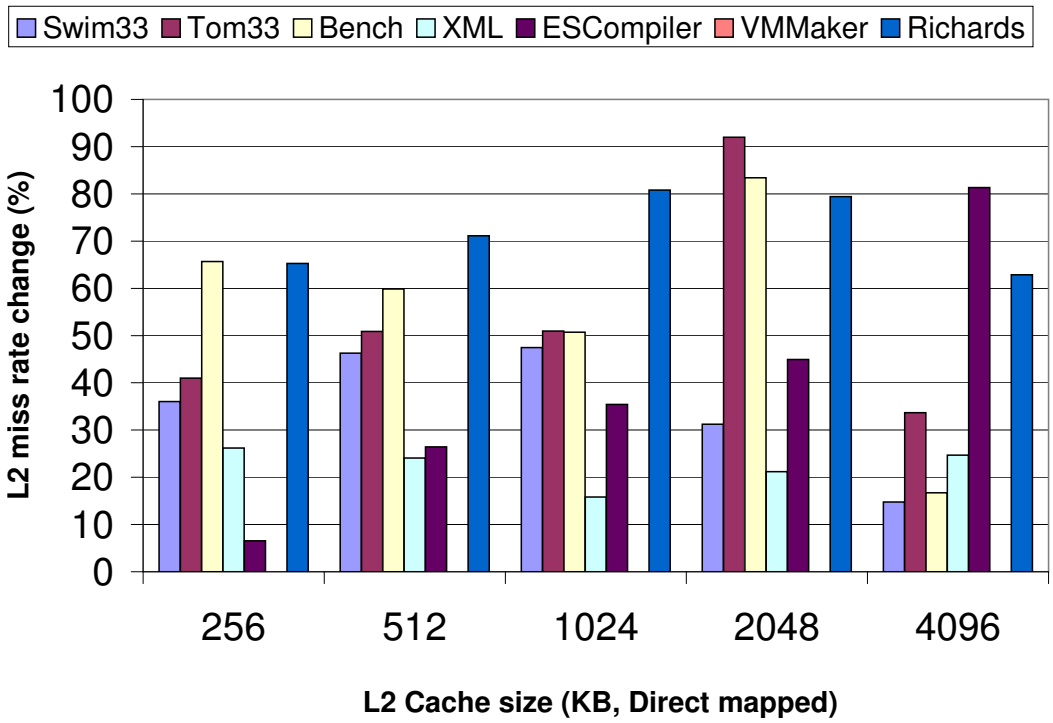


Figure 7.16: Miss rate improvement for a direct-mapped L2 cache when new space is 25% of the L2 cache size, for all benchmarks.

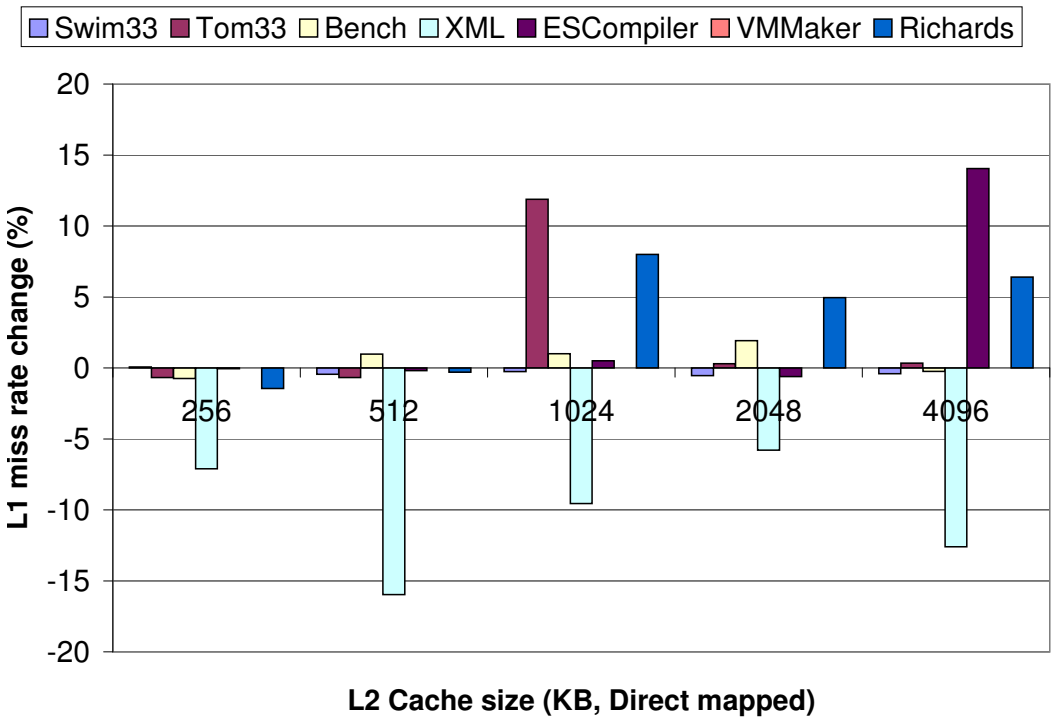


Figure 7.17: L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and L2 caches when the mapping strategy is applied to the L2 cache, for all benchmarks.

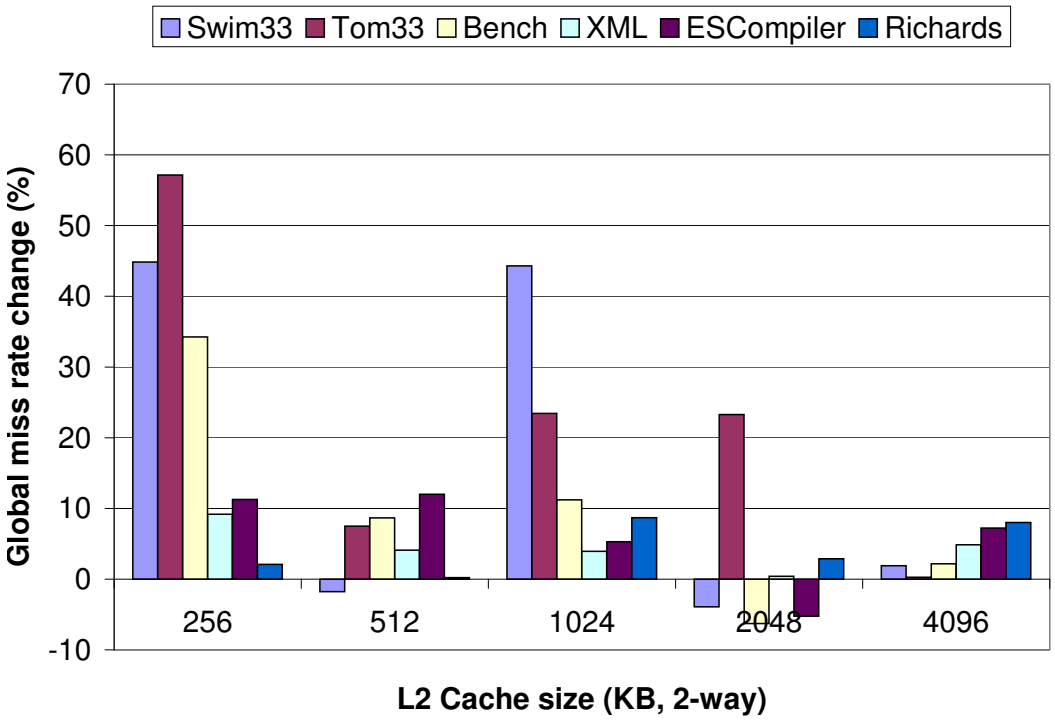


Figure 7.18: Global miss rate improvement in a two-level cache hierarchy with a direct-mapped L1 and L2 cache when the mapping strategy is applied to the L2 cache.

7.2.3 Applying the strategy to a set-associative L2 cache

Figures 7.19, 7.20 and 7.21 show the L2 miss rate improvements when the strategy is applied for $2-way$, $4-way$ and $8-way$ set-associative caches respectively. As seen, the gain in miss-rates decreases with associativity. The mapping strategy is effective because it removes all conflicts to the youngest generation. The strategy is very effective in direct-mapped caches where there are more conflicts to the youngest generation. Set-associativity removes some of the conflicts in the youngest generation and hence the benefits of the strategy decrease. The effect of associativity on cache misses in the youngest generation is discussed by Wilson et al [8] and can be seen in our experiments. However, our simulation does not measure a particular source of conflicts – cache misses due to conflicts between youngest generation pages that are randomly placed. A page-level simulation would reveal these conflicts. These conflicts would not affect results measured with the mapping strategy. This is because the strategy ensures non-occurrence of pages that conflict with the youngest generation pages. Results show an average 15% improvement for $2-way$ set-associative caches. Systems like MIPS R10000 and DEC Alpha 21264 that use $2-way$ set-associative caches would benefit from this strategy.

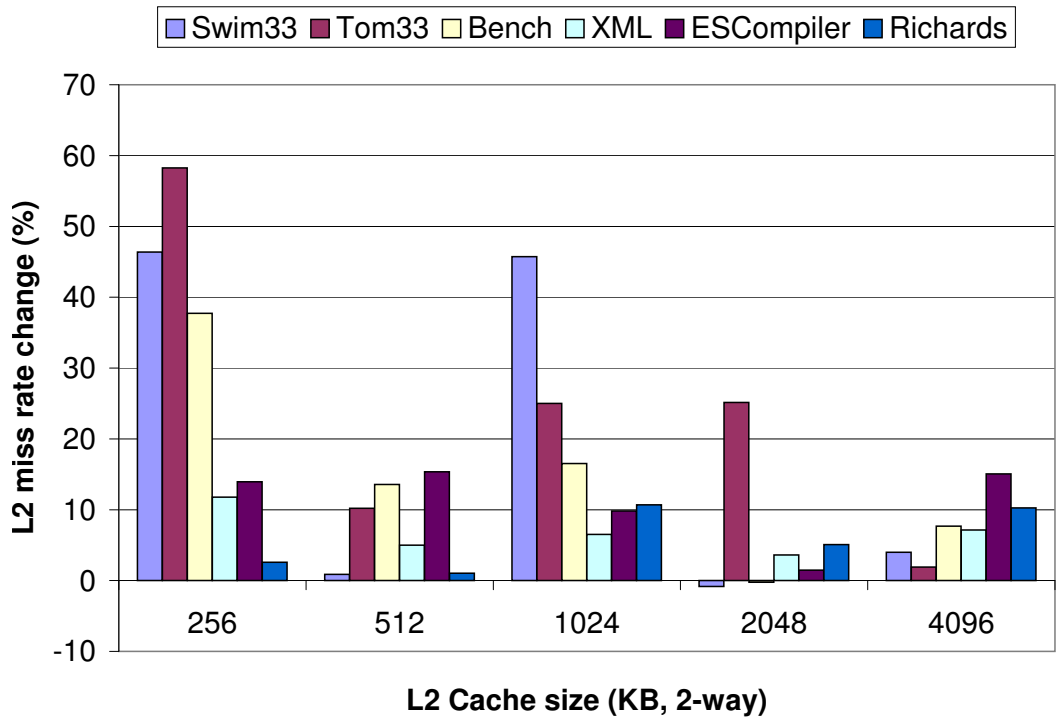


Figure 7.19: Miss rate improvement for a $2-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks.

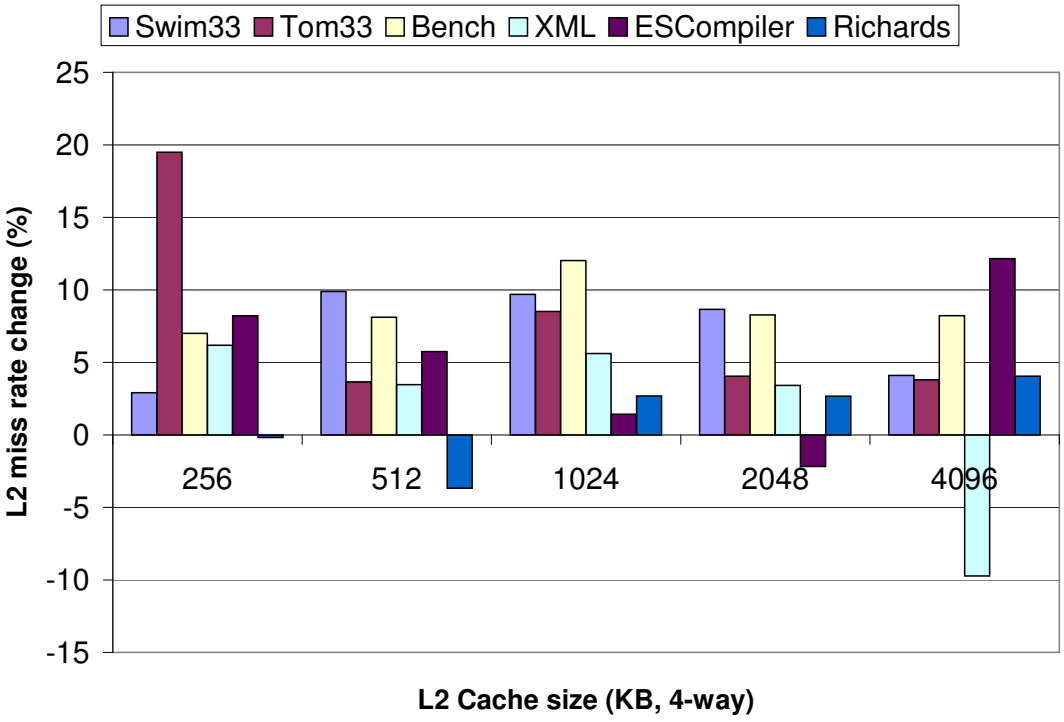


Figure 7.20: Miss rate improvement for a $4-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks.

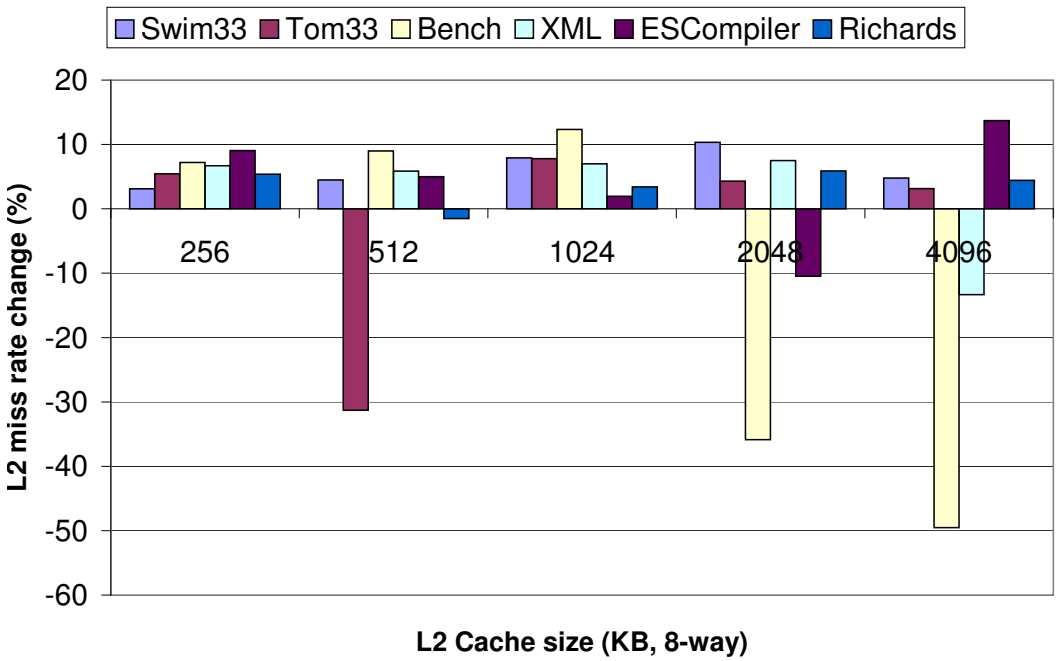


Figure 7.21: Miss rate improvement for a $8-way$ set-associative L2 cache when new space is 25% of the L2 cache size, for all benchmarks.

Figures 7.22, 7.23 and 7.24 show the change in L1 miss rates when the strategy is applied to the L2 cache for $2-way$, $4-way$ and $8-way$ associativities. There is a significant increase in miss rates for all cases. This is in stark contrast to L1 miss rate changes when the strategy is applied to direct-mapped L2 caches as shown in Figure 7.17. This conflict in L1 cache for set-associative L2 caches is due to accesses across buckets for each stride in the new space. Referring back to Figure 1 in Section 3.2, the youngest generation is mapped to as many buckets as the associcativity of the cache. Each successive line of the youngest generation maps to an alternate bucket in main memory. During sequential allocation, a stride through the new space causes accesses to different buckets. If the buckets are as large or larger than the L1 cache, this access pattern will cause buckets to thrash each other out of the cache. As seen, this conflicting pattern results in an increase in L1 miss rates. To prevent this thrashing, the placement of the youngest generation should be changed such that its lines completely fill up a bucket before overflowing into another. This would ensure a pattern that is similar to our base case simulation[2] and would prevent the increases in L1 miss rates. The strategy restricts the placement of the youngest generation to certain buckets. But within the buckets, there is freedom to place the youngest generation in any style.

[2]In our base case simulation, virtual addresses are used for cache simulation. In virtual memory, the youngest generation is sequential and hence a stride in new space produces a sequential access pattern. When applying the strategy, organizing the new space as described would produce a similar access pattern.

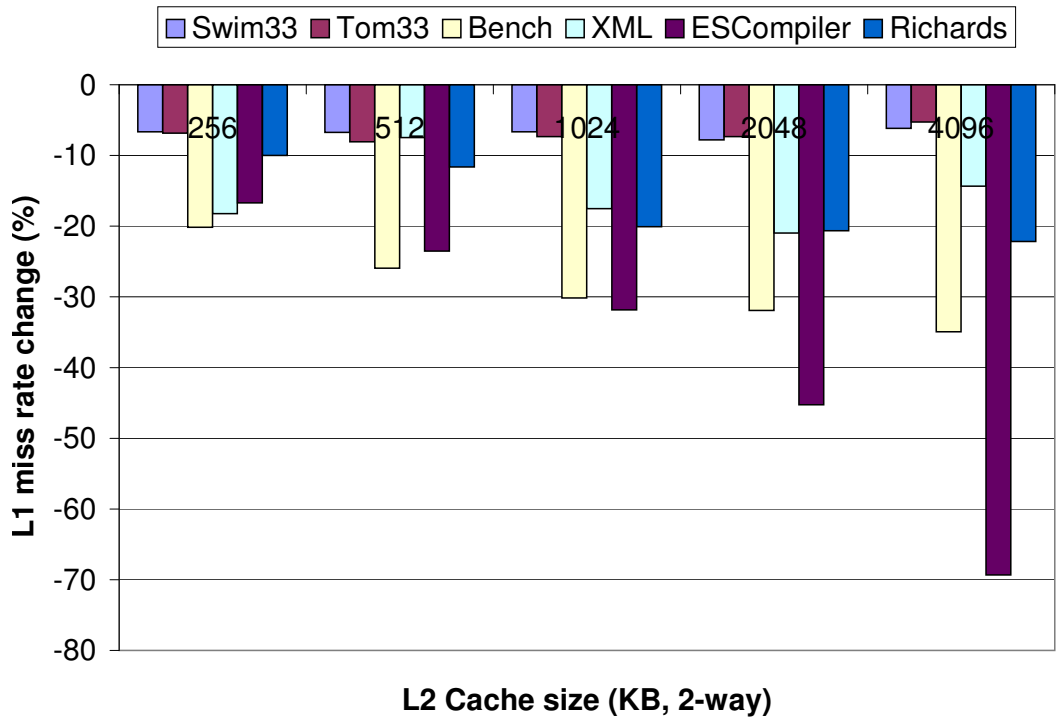


Figure 7.22: L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $2-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks.

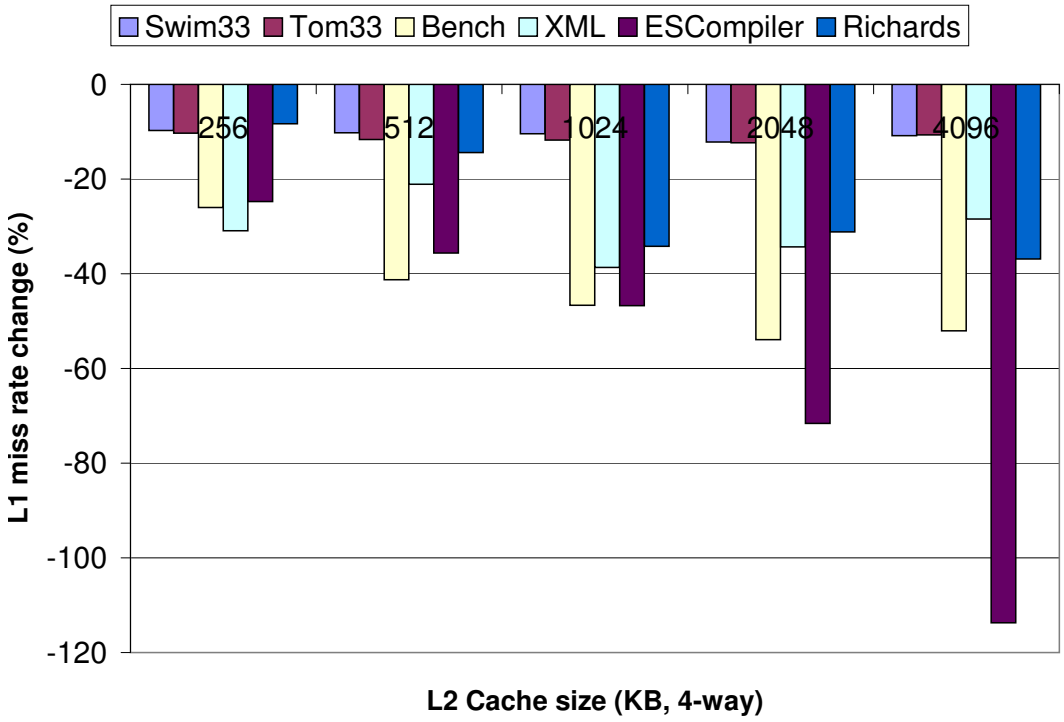


Figure 7.23: L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $4-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks.

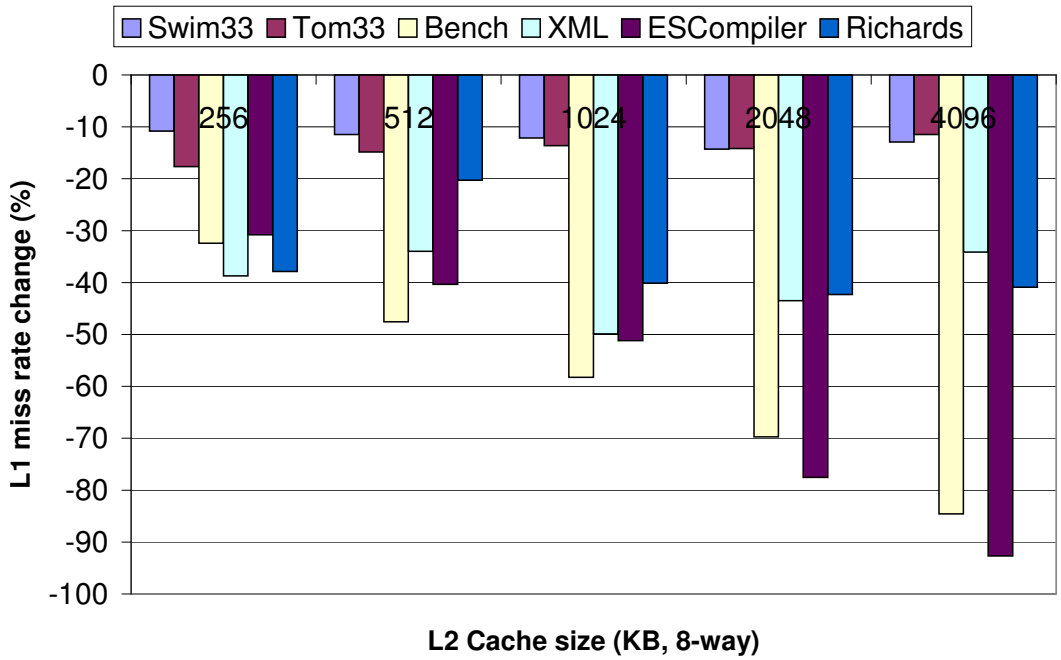


Figure 7.24: L1 miss rate change in a two-level cache hierarchy with direct-mapped L1 and $8-way$ set-associative L2 cache when the mapping strategy is applied to the L2 cache, for all benchmarks.

7.2.4 Effect on IPC

The strategy discussed in this thesis is targeted at L2 caches which are large enough to hold the youngest generation. Our experiments show that the strategy is effective in reducing upto 45% of misses in direct-mapped and 15% in $2-way$ set-associative caches. In this section we try to estimate the effect of the miss rate improvement on the IPC count (instructions per cycle) of a superscalar processor.

The IPC was modeled in the following way. A $4-issue$ superscalar processor is assumed. Branch mispredictions are assumed to lower the IPC to 3. We also assumed that a L2 miss causes a processor to stall long enough that the entire processor goes idle. Overlapping L2 misses are ignored. Then IPC^1, the new count that accounts for L2 misses, is represented by the following equation –

$$IPC^1 = \frac{I}{\frac{I}{IPC} + (Number\ of\ L2misses)\ \times\ (L2\ miss\ penalty)} \tag{7.1}$$

where I is the number of instructions executed and IPC is the original count without accounting for L2 misses (3 in this case).

Figures 7.25 to 7.30 show the effect of the strategy on system IPC for a L2 cache size of 256KB. It plots the IPC count for both cases; using the mapping strategy and not using it. The IPC curves are shown for different L2 miss penalties as indicated in the brackets. As seen, the improvement in IPC is better for direct-mapped caches and decreases with associativity. Figures 7.31 to 7.36 plot the same for a 512KB L2 cache.

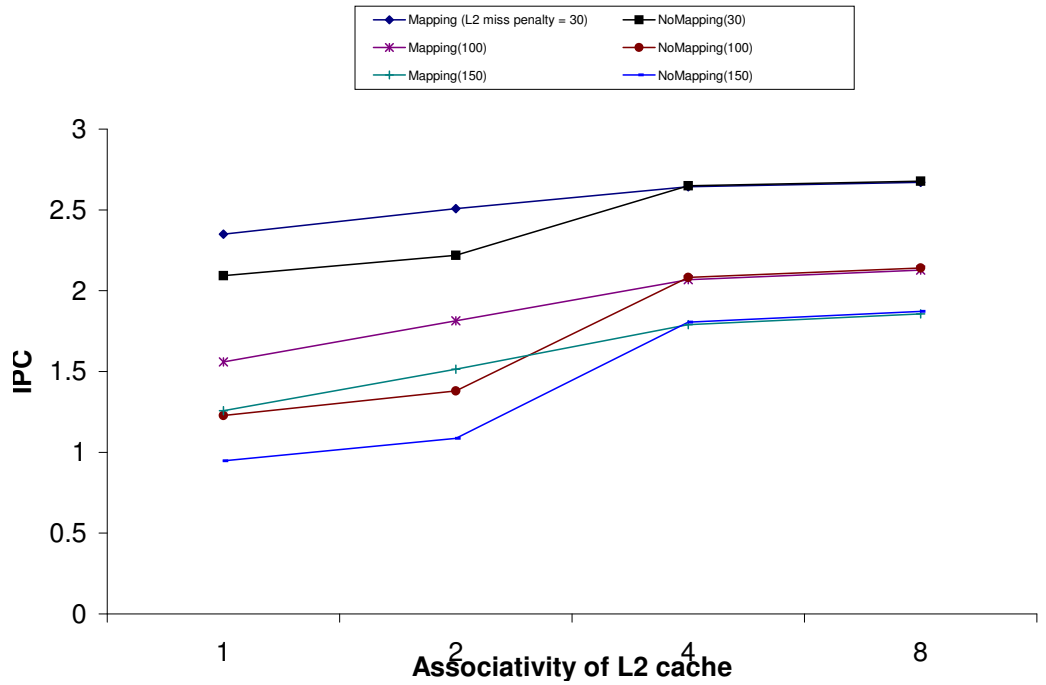


Figure 7.25: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Swim.

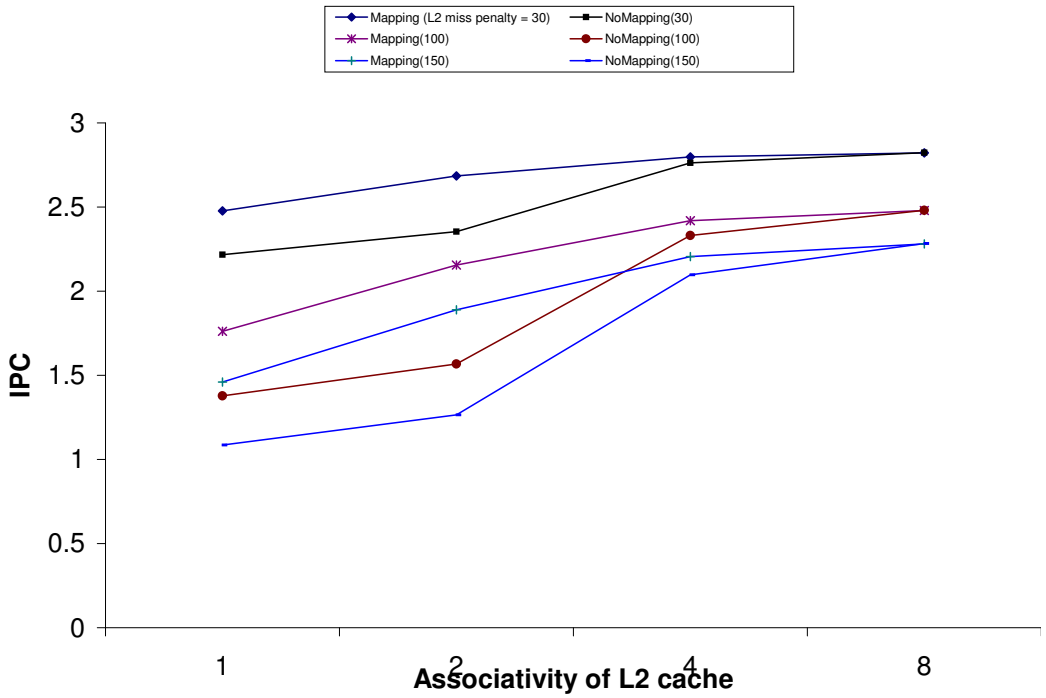


Figure 7.26: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Tomcatv.

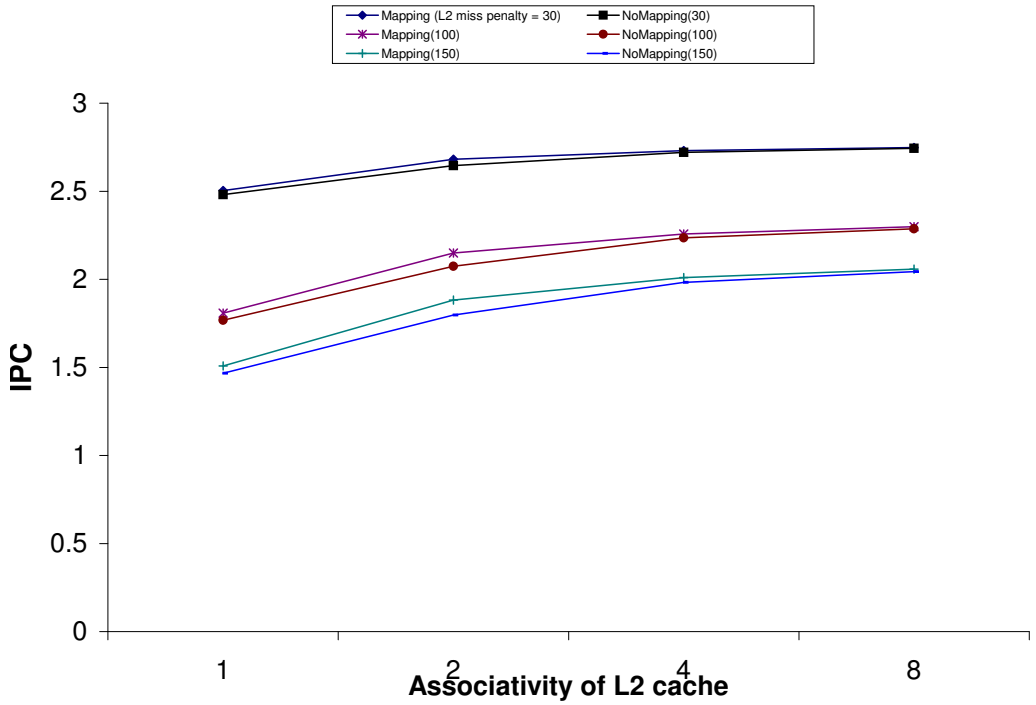


Figure 7.27: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for ES Compiler.

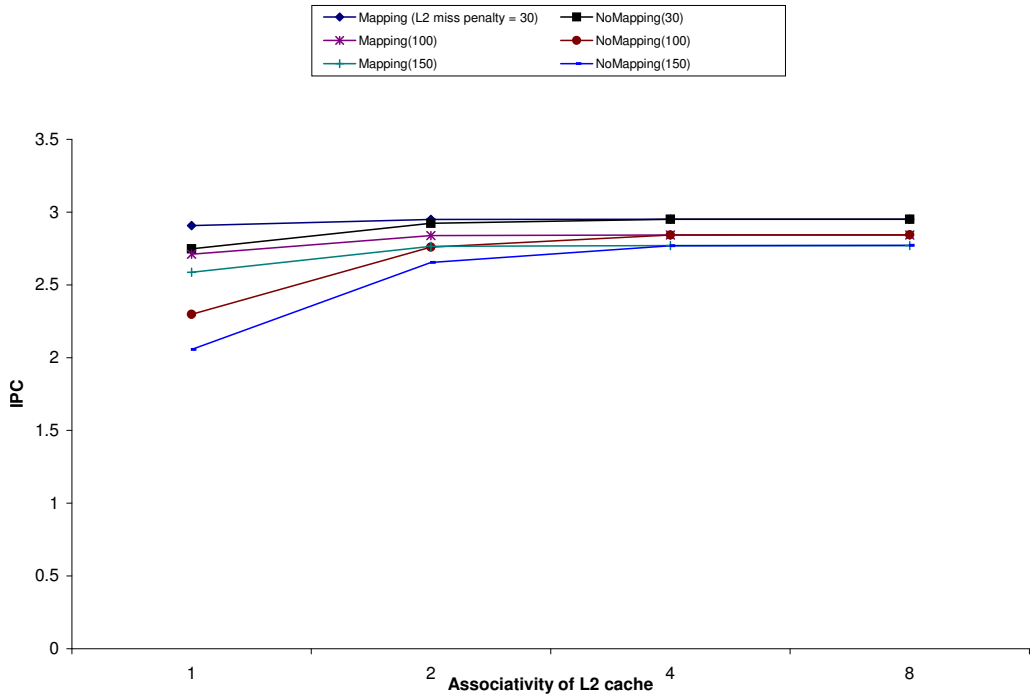


Figure 7.28: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Bench.

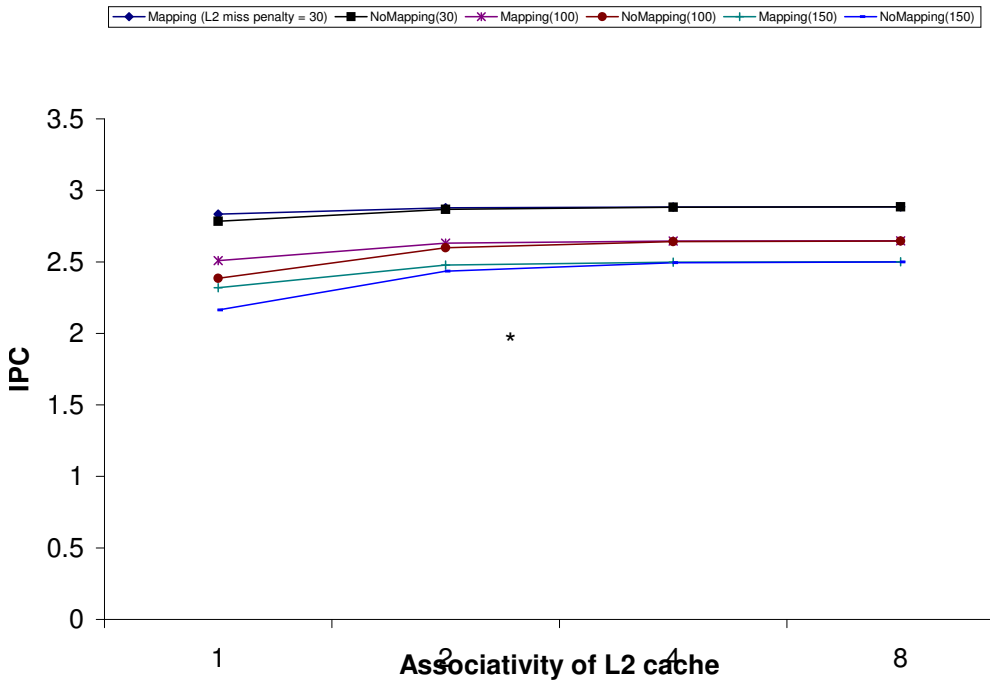


Figure 7.29: Effect of the mapping strategy on IPC for a $n - way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Xml parser.

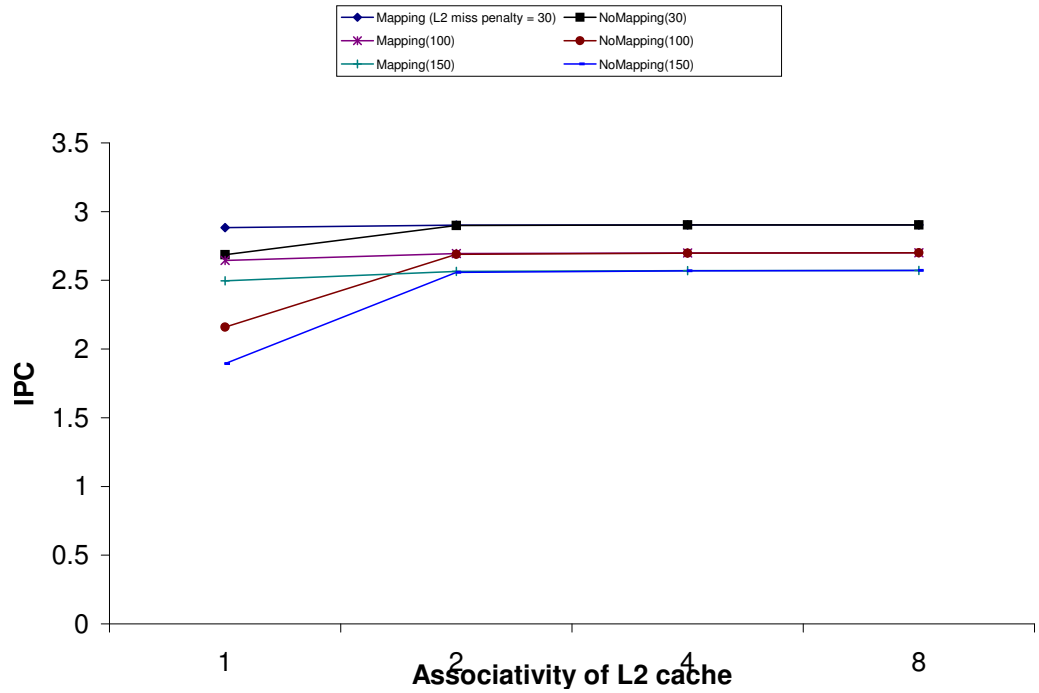


Figure 7.30: Effect of the mapping strategy on IPC for a $n - way$ set-associative L2 cache (n=1,2,4,8) of size 256KB, for Richards.

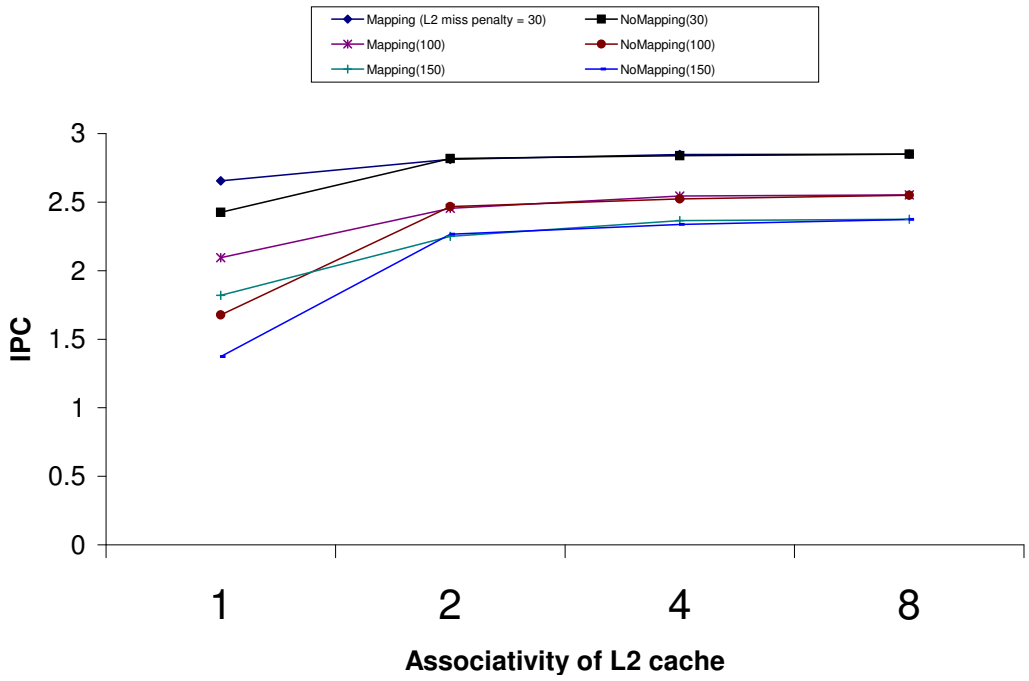


Figure 7.31: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Swim.

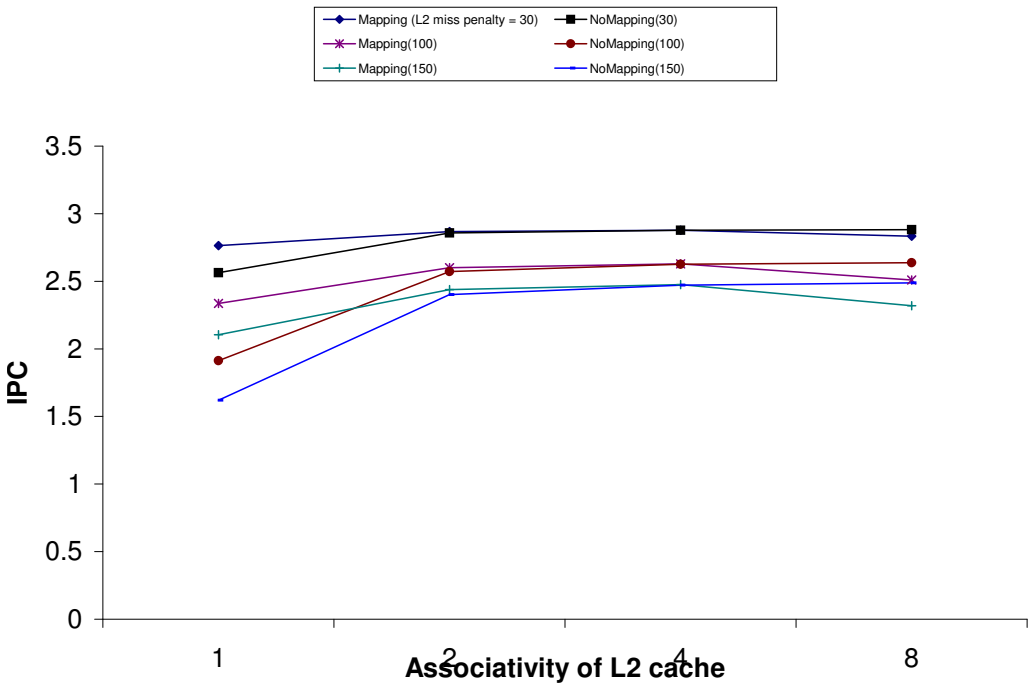


Figure 7.32: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Tomcatv.

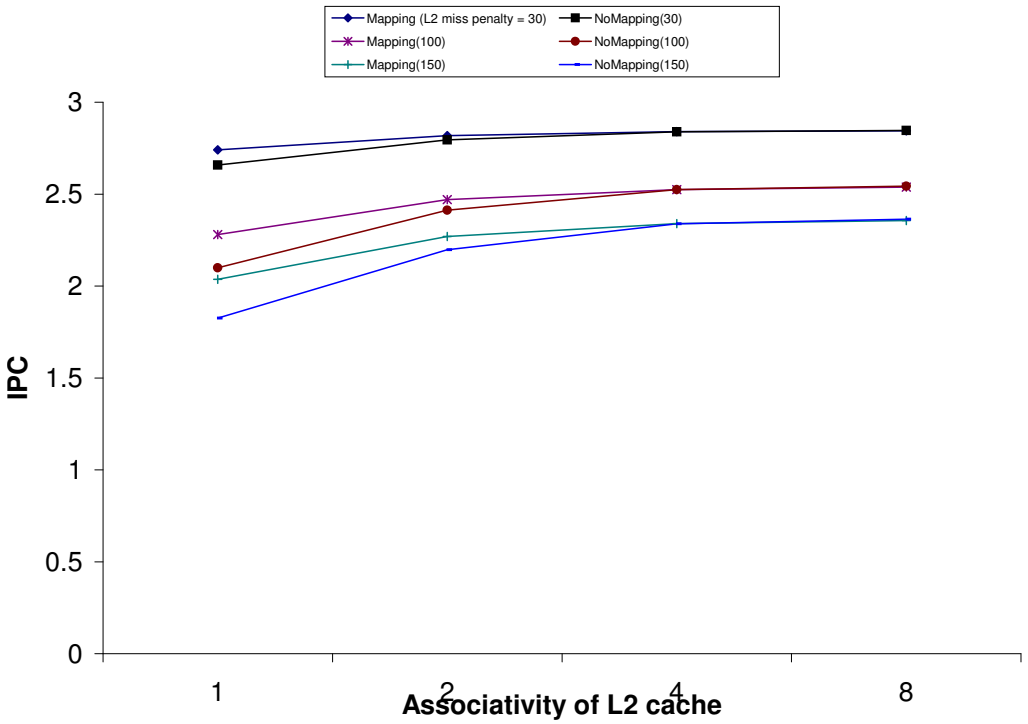


Figure 7.33: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for ES Compiler.

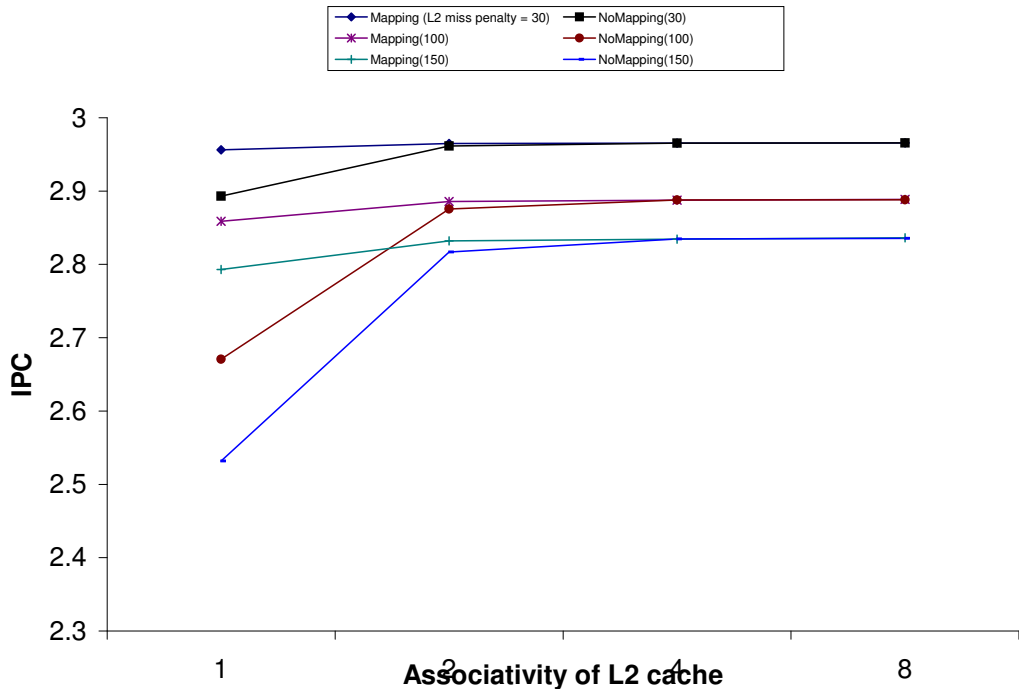


Figure 7.34: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Bench.

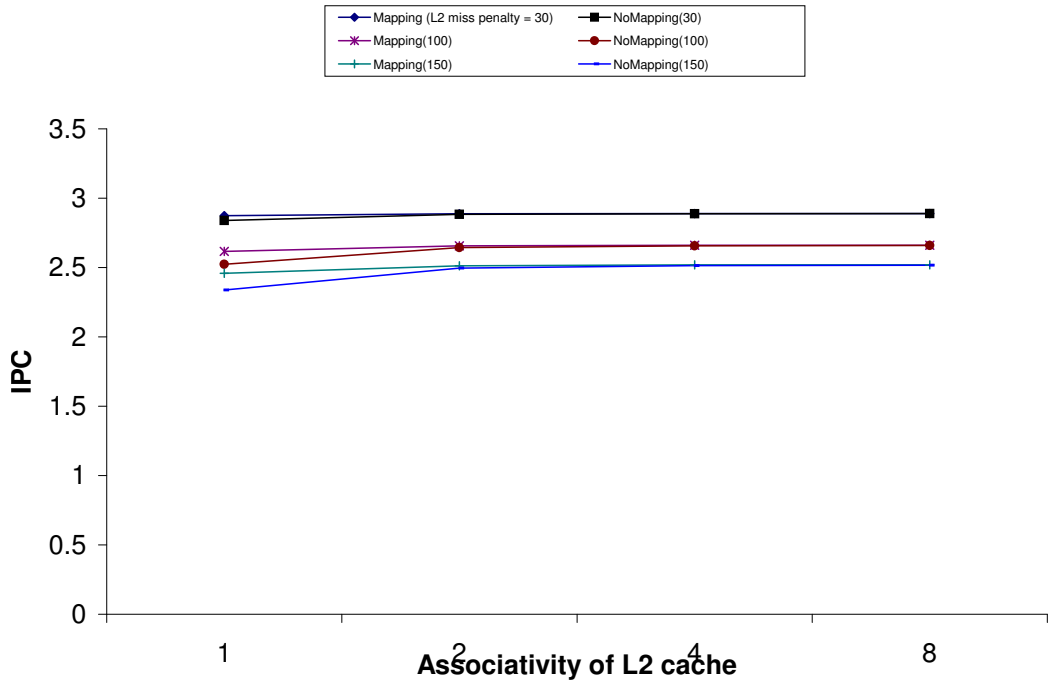


Figure 7.35: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Xml parser.

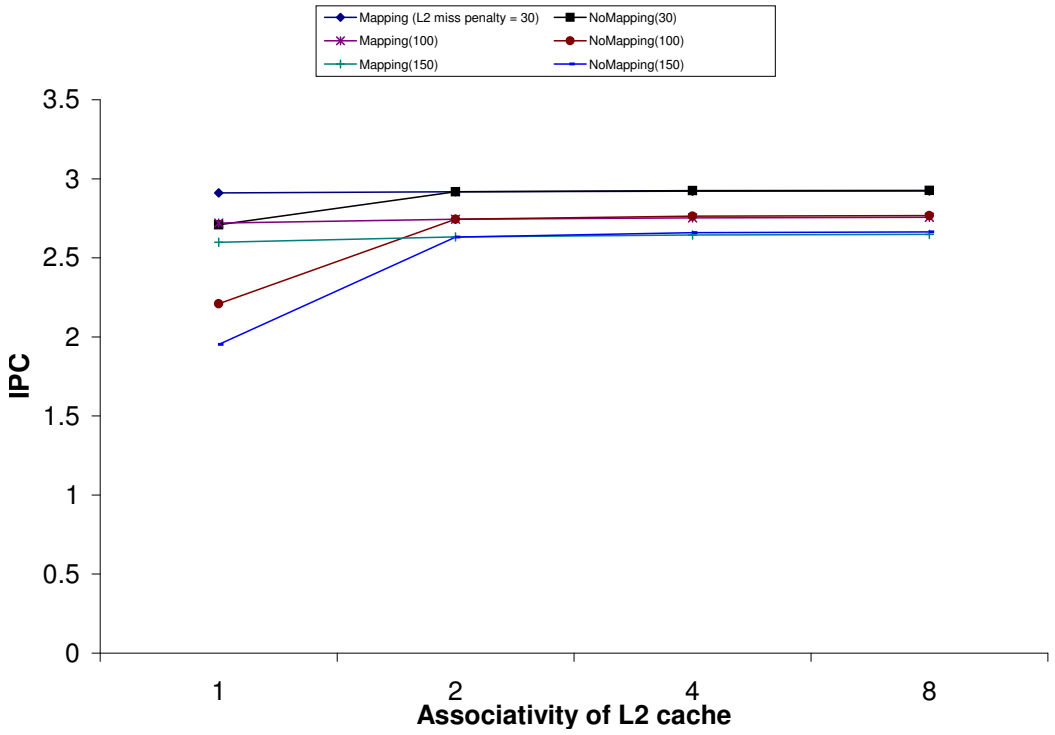


Figure 7.36: Effect of the mapping strategy on IPC for a $n-way$ set-associative L2 cache (n=1,2,4,8) of size 512KB, for Richards.

Chapter 8

Conclusion and future work

We designed a strategy to reserve the youngest generation of a generational copying-garbage collector in the cache. A software-only implementation was proposed which requires minor changes to the page-fault manager of an operating system. This makes it an attractive solution to improve performance of generational-garbage collected systems. Our simulation results show that the removal of conflicts to the youngest generation reduces cache miss rates by an average 45% for direct-mapped caches and 15% for $2 - way$ set-associative caches. We expect better improvement in a real system where randomly placed youngest generation pages could conflict in the cache.

Future work involves implementing the strategy in a real system. This would involve making changes to an existing page-fault manager to incorporate a new replacement policy. A few changes in the operating system are required as well.

The same strategy could also be applied to Smalltalk instructions which are actually data at the machine level. This possibility seems attractive and needs to be explored. Another area to explore is choosing of page-frames intelligently from the identified buckets. For instance, when the mapping strategy is applied to the L2 cache, choosing odd page frames for *From space* and even page frames for *To space* could remove conflicts in the L1 cache during scavenging,

Bibliography

[1] Henry Lieberman, Carl Hewitt. *A Real-Time garbage collector based on the lifetime of objects.* Communications of the ACM, volume 26, issue 6, pages 419–429, June 1983.

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

[3] David Ungar. *Tenuring policies for generation-based storage reclamation.* OOPSLA Conference Proceedings, ACM SIGPLAN Notices, volume 23, issue 11, pages 1-17, September 1988.

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

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

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

[7] Nandakumar Sankaran. *A bibliography on garbage collection.* Technical Report, Clemson University, Clemson, South Carolina, February 1994.

[8] 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, pages 32–42, June 1992.

[9] 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.

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

[11] Hans-J. Boehm. *Reducing garbage collector cache misses.* Proceedings of the second International Symposium on Memory Management, pages 59–64, October 2000.

[12] C. J. Peng and G. S. Sohi. *Cache memory design considerations to support languages with dynamic heap allocation.* Technical report 860, Computer Sciences Dept., University of Wisconsin Madison, Madison, Wisconsin, July 1989.

[13] Jarrod A. Lewis, Bryan Black, Mikko H. Lipasti. *Avoiding initialization misses to the heap.* 29th Annual International Symposium on Computer Architecture, May 2002.

[14] Trishul M. Chilimbi, James R. Larus. *Using generational garbage collection to implement cache-conscious data placement.* Proceedings of the first International Symposium on Memory Management, pages 37–48, October 1998.

[15] Amer Diwan, David Tarditi, Eliot Moss. *Memory subsystem performance of programs using copying garbage collection.* Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 1–14, January 1994.

[16] Dirk Grunwald, Benjamin Zorn, Robert Henderson. *Improving the cache locality of memory allocation.* ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

[17] Robert Courts. *Improving locality of reference in a garbage collecting memory management system.* Communications of the ACM, volume 31, issue 9, pages 1128–1138, September 1988.

[18] Mark B. Reinhold. *Cache performance of garbage-collected programming languages.* Technical Report MIT/LCS/TR–581, Massachusetts Institue of Technology, Cambridge, Massachusetts, September 1993.

[19] Andrew W. Appel. *Garbage collection can be faster than stack allocation.* Information Processing Letters, volume 25, issue 4, June 1987.

[20] Andrew W. Appel. *Simple generational garbage collection and fast allocation.* Software Practice and Experience, volume 19, issue 2, pages 171–183, February 1989.

[21] Darko Stefanovic. *Properties of age-based automatic memory reclamation algorithms.* PhD Thesis, University of Massachusetts Amherst, Amherst, Massachusetts, February 1999.

[22] GNU Smalltalk Version 2.1. Available from URL `http://www.gnu.org/software/Smalltalk/smalltalk.html`

[23] IBM VisualAge Smalltalk Version 6.01. Available from URL `http://www-3.ibm.com/software/awdtools/smalltalk`

[24] Jingyu Sun, Edward F. Gehringer. *A Smalltalk memory profiler and its performance enhancement.* Presented at OOPSLA '97 Memory Managemnt and Garbage Collection Workshop, October 1995.

[25] William A. Wulf, Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious.* Computer Architecture News, volume 23, issue 1, pages 20–24, 1995.

[26] 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, pages 2–9, 1991.

[27] Bjarne Stroustrup. *The C++ Programming langauge.* Third edition, Addison-Wesley Publishing Company, ISBN: 0201700735, 1997.

[28] Robert R. Fenichel, Jerome C. Yochelson. *A LISP garbage-collector for virtual-memory computer systems.* Communications of the ACM, volume 12, issue 11, pages 611–612, November 1969.

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

[30] Doug Burger, Todd M. Austin. *The Simplescalar toolset, Version 2.0.* URL http://www.simplescalar.com

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