The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

TOWARDS SELF-OPTIMIZING MEMORY MANAGEMENT

A Thesis in

Computer Science and Engineering

by

Gokul B. Kandiraju

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2004

We approve the thesis of Gokul B. Kandiraju.

Date of Signature

Anand Sivasubramaniam
Associate Professor of Computer Science and Engineering
Thesis Adviser
Chair of Committee

Mary Jane Irwin
Distinguished Professor of Computer Science and Engineering

Natarajan Gautam
Associate Professor of Industrial Engineering

Mahmut Kandemir
Assistant Professor of Computer Science and Engineering

Ekanadham Kattamuri
Research Staff Member of IBM T. J. Watson Research Center
Special Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Abstract

Systems today are application driven. Increasing application sizes re-iterate the importance of memory management and increasing application complexity stresses the need for self-management. At the same time, different memory requirements of different applications require that optimizations for memory management be done from a complete system perspective. In the view of this, the goal of this thesis is to take a step towards self-optimizing memory management at all the different levels of the memory hierarchy.

This thesis makes three main contributions to the memory management system. First, it undertakes a thorough characterization study for the TLBs and proposes a novel prefetching mechanism that is simple, powerful and adapts to the applications. Second, it presents a dynamic memory allocator that tunes itself to the applications. Finally, towards the goal of developing a self-optimizing VMM, it finds the important VMM parameters that govern the system performance, relates the influence of these parameters to the application/OS characteristics, and provides a solid motivation to set these parameters dynamically.

Table of Contents

List of Tables

List of Figures

Acknowledgments

It has been a pleasure and an honor doing my PhD at Penn State. Looking back, I am glad I did my PhD here and my heart is filled with gratitude for all those who made it happen. I always believed that there is Divine Grace behind everything we do. My gratefulness for this Grace, which I know in the form of *Guru* (or Baba) cannot be expressed in words.

I am deeply thankful to my advisor Dr. Anand Sivasubramaniam for everything he has given me in the last five years. I owe all my success to him. He is a great researcher, a perfect mentor and above all, a great friend. His encouragement, persistence, aggression and enthusiasm have been the driving force for me to learn, develop skills and accomplish. Of course, there have been moments of complain and mis-understanding from my side, but in retrospect, I only feel more grateful for his understanding, maturity and foresight. His technical expertise in analyzing intricacies and his sharp intellect where he saw through things which I could not, have taught me a lot. His criticism was perfect. It made me progress. He also played a major role in placing me at IBM T. J. Watson Research Center. More than anything else, I feel fortunate because I got to work with someone who really cares about me. I hope to collaborate with him in future and seek his advice when necessary.

The three summers that I spent in the IBM T. J. Watson Research Center have been a learning experience for me. I would like to thank my mentor, Dr. Ekanadham Kattamuri, for it. In many ways, he has changed my perspective on research from a grass-root level. Our collaboration has gone well beyond three summers and I hope to continue this with him.

Next, my gratitude goes to my referees: Dr. Mary Jane Irwin, Dr. Natarajan Gautam, and Dr. Raj Acharya, who took time from their busy schedules, writing strong recommendation letters for me. Their strong support has made my job search much easier.

I would also like to thank the members in my Ph.D committee: Dr. Mary Jane Irwin, Dr. Mahmut Kandemir, Dr. Natarajan Gautam and Dr. Ekanadham Kattamuri. They gave me a lot of valuable comments on my thesis work, which helped me refine my dissertation.

In the last five years, I realised that during the process of a PhD, one not only learns from one's advisors but also from one's peers. I have been extremely lucky to have a group of excellent lab-mates and house-mates who have made my stay at Penn State very colorful. Murali Vilayannur and Deepak Ramrakhyani have been amazing friends and the first persons for me to approach for anything. Shailabh Nagar, Ning An and Yanyong Zhang have helped me throughout my PhD, as seniors, to make many important decisions. Chun Liu, Sudhanva Gurumurthi, Partho Nath, and Jianyong Zhang have been really great and sweet friends, who were always ready to help. Naveen Cherukuri, Saurabh Agarwal, Ananth Indrakanti, Ashish Kulkarni and Rajdeep Pradhan have been roomies who taught me a lot.

After five years, I realise that PhD is not an achievement of a single person. There are many people that contribute to one's success - every PhD is an organizational effort. All the administrative people at Penn State, especially those from the Department of Computer Science and Engineering have played a significant role in my PhD. I am especially grateful to Vicki Keller who has been so helpful from the first day at Penn State and to Eric Prescott who has spent significant time helping me to get the right set of resources.

A very important phase of my PhD began after I did the *Art of Living* Course. There was a distinct difference in the quality of my days after Art of Living. I am very grateful to Kaushik

Narayanan and Birjoo Vaishnav for making me aware of this course. Birjoo Vaishnav has been an inspiration for me as a person filled with love and one who spends his time for others. In retrospect, the highlight of my PhD is getting involved in Art of Living.

Last but not the least, I am extremely grateful to my parents and my brother. They have been with me all the five years (although I have not visited them), and encouraged me in every decision I made, even though it may be painful to them. Their unwavering encouragement and enthusiasm pulled me through hard times.

This acknowledgment list is by no means complete. There are many people not mentioned above who have contributed to my success. Whether I went through joyful moments or depressing moments, it all happened for good. I believe that I have become aware of, and hence became immune to, many ups and downs that are possible in life. I am leaving Penn State filled with confidence, enthusiasm and love to take on the rest of my life as a challenge. Now I know why this is called Doctor of Philosophy and not Doctor of Computer Science.

Chapter 1

Introduction

Systems today are becoming application driven. Application performance is critical in various fields like biology, genetics, military, space exploration etc. For instance, the protein folding problem that could provide a cure to Alzheimer's disease requires massive computational power and IBM is building the Bluegene system [21] in an attempt to solve this problem. Several large-scale projects in industry and academia have applications as their primary motivation today. With regard to applications themselves, two fundamental trends can be observed in their evolution: (i) Application sizes are increasing and (ii) Application complexity is increasing.

Today's applications are much larger than those from a decade ago. Common desktop applications like KDE [10], Netscape [16] or Mozilla [13] need about 20-32 megabytes(MB) of memory. High-end server applications like WebSphere [9] need a minimum of 256MB to run and a few gigabytes of memory is recommended. It is not only a question of fitting applications in the processor caches but also that of fitting them in the main memory. As the distance between the processor and memory increases, with disk access times showing few signs of improvement, optimizations to reduce the data access time by having the right data at the right time in the right level of this hierarchy become all the more important. At the same time, diversity of application working-sets can mandate different optimizations. For example, desktop applications generally fit in memory. For such applications, optimizations done in the processor memory management structures, the caches and the TLBs, can have greater impact on their performance compared to

those done in the operating system. Thus, techniques to increase the performance of these on-chip structures will help. On the other hand, if we consider high-end applications like DBT3 [17] or Websphere [9], the bottleneck for these applications is the disk access. Though optimizations done in the processor-memory path might help, optimizations done to manage data effectively in the main memory, thereby preventing disk accesses, will have much greater benefits. Thus, for such applications, memory management techniques employed in the operating system become more important. Therefore, due to diverse requirements of today's applications, it is necessary to provide optimizations at all levels in this hierarchy. In other words, memory management optimizations need to be done from a complete system perspective.

At the same time, phenomenal growth of computing power has also resulted in making applications more complex. Complexity has not only increased for a single application in terms of the number of lines of code, bugs, error conditions etc. [2], but also in the diversity of applications that are being developed. In fact, it is not just application complexity that has increased, but systems in general have become more complex. Today, a lot of time and effort is spent in managing systems. For instance, large companies spend about 40% of their investment in just managing systems [62, 58]. As the proliferation of the Internet continues and device complexities increase, managing systems will become an indomitable task. The goal of *Autonomic Computing* [62] is to reduce the increasing complexity of managing larger computing systems by making them self-managing. If we take a typical computer system, there are a number of variables on which performance of applications depend. For instance, the parameters within an operating system have significant effect on the application runtime. Humans cannot be expected to sit and tune such parameters for every application - this is not economical both in terms of time and cost. Thus, in the light of emerging technologies, memory management needs to be

self-managing. While there are several aspects of Autonomic Computing, this thesis specifically focuses on making memory management *self-optimizing*, i.e. the ability of a system to adapt itself to the workload behavior in order to provide the best performance at anytime.

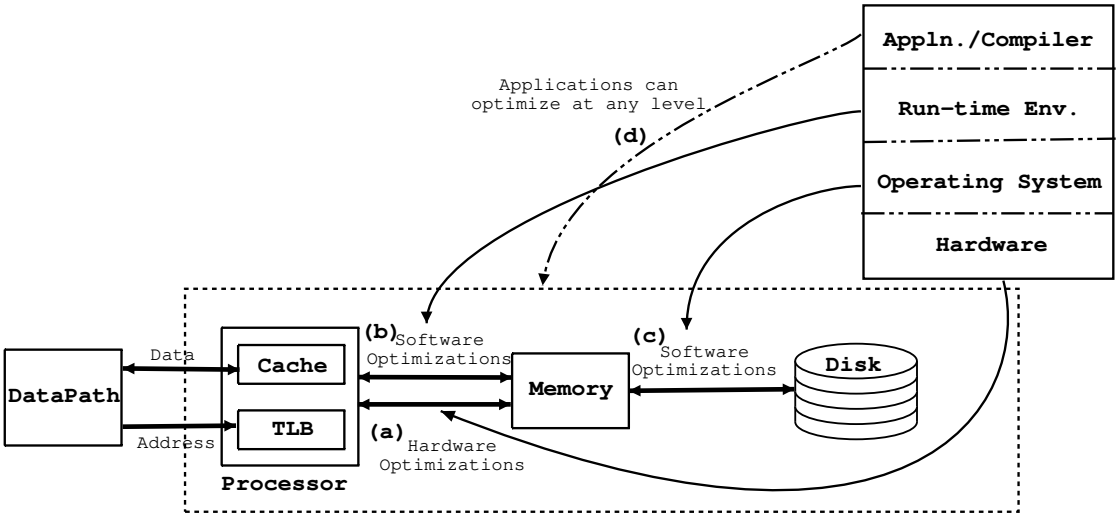


Fig. 1.1. Figure shows optimization methods for memory management.

Increasing application sizes reiterate the importance of efficient and effective memory management. Differing memory requirements of different applications imply the need for memory management optimizations to be done at all the levels. Increasing application complexity stresses the need for self-optimization. The goal of this thesis is to take a step towards making memory management self-optimizing - this being done from a complete system perspective at all the levels in the memory hierarchy without requiring applications to be modified. While one could think of writing an application by keeping the complete memory hierarchy in perspective, we do not want to burden application programmers to do this. Also, as applications become

more complex, this task will become non-trivial. This thesis takes a step towards proposing adaptive mechanisms for optimizing the Memory Management System(MMS) at different levels - hardware, operating system and runtime libraries, to improve application performance.

1.1 Hardware Enhancements for the TLB

On the hardware front (shown as (a) in Figure 1.1), the caches and the TLBs are responsible for providing efficient data access and protection to the processor core. The performance of these two structures has direct implication on the application. Numerous efforts have been undertaken to improve the performance of these structures, especially for caches. Methods to improve performance of data caches include design of exclusive caches [70], adaptive caches [90], stream buffers [69], column associative caches [22] etc. Methods have also been proposed on the software side to improve the performance of caches [52, 72, 98, 40, 35]. Techniques have been proposed to improve the performance of instruction caches as well [95]. Some of these proposals have even been implemented in commercial microprocessors. Caches have also been investigated quite extensively with regard to prefetching [42, 75, 48, 108]. In all, a plethora of literature exists for caches in the past while TLBs have received much less attention. Most of the optimizations for TLBs in the past have been mainly been limited to structural optimizations like Multi-level TLBs[84] and Superpaging[104]. There has not even been a characterization study that has been undertaken for the TLBs (where as such studies have been done extensively for caches [28, 36, 60], for different kinds of workloads). Since the access time of TLB is extremely important, there has been reluctance to propose novel enhancements in case it affects the design of the critical path. Further, since this is an on-chip structure, use of elaborate hardware

to monitor and optimize the datapath reference patterns would be limited by space and power constraints.

Several previous studies [63, 94, 97, 23] have noted the importance of the TLB. Some studies report that TLB overhead could constitute as much as 40% of the execution time in extreme cases with an average overhead of 8-9% [63]. TLB overhead could be upto 80% of a kernel's computation time [94] and studies with commercial and scientific applications [97] have shown that the TLB miss rate can account for over 10% of their execution time even with an optimistic 30-50 cycle miss overhead. It has also been observed in the past [23] that TLB handler is the most frequently executed kernel service.

Keeping the current trends in mind, it is important to provide optimizations for the TLBs due to several reasons: First, the growing gap between the processor and the memory can only increase the TLB overhead in case the translation is not cached. Second, as application working-sets get larger it is difficult for the TLB to cover the working-set effectively, even with large TLBs. For instance, many of the SPEC CPU 2000 benchmarks have working sets larger than the TLB coverage provided by any of the processors today. In fact, it might even become difficult to find the translations in the cache. Third, as pipelines get deeper, overhead of software handled TLBs will increase due to the increased cost of flushing the pipelines. Therefore, the issue of improving TLB performance will continue to remain an important problem to be looked at even in future.

Before proposing enhancements, it is important to understand how applications behave and then justify the extra hardware based on the cost-benefit trade-offs. Such an approach has been used in the design and analysis of other system components (eg. [28] for caches, [100] for I/O, [111] for processors etc.), and to our knowledge there is no other prior study that has done

the same for the TLBs. In our quest for a self-optimizing MMS, we begin with such a study (Chapter 2), studying numerous applications to find out what is it that applications need from the TLB, and whether we can provide it in a fashion that is not specifically tailored for an application while still dynamically adapting itself to the application characteristics. Our thorough characterization study yields several good insights into the application characteristics based on which we develop a novel TLB prefetching technique Our technique is simple, yet powerful, and it adapts to applications as they execute.

1.2 Software Memory Management

Optimizations to the memory structures can also be done in software Software optimizations ((b) in Figure 1.1) for caches, which can also have an effect on the TLBs have used compiler transformations and data reorganization to enhance locality [40, 98, 72, 52]. Increasing complexity of applications (and hence that of application code) can make compiler optimizations more difficult in future. Compiler optimizations may not be possible when we do not have access to the source code. Also, most of these optimizations target static data and arrays but not dynamic memory, which is managed by the runtime environment. Many applications today spend significant amount of time in managing dynamic memory. For example common utility applications like GNU Make [6] and GNU Awk [4] spend about 5% of their runtime in dynamic memory allocation. Cfrac, a prime number factoring application, spends about 15% of its execution time in memory allocation. SPEC benchmarks twolf, vpr and SPEC JBB [18] spend about 1-3% of application runtime in memory allocation. Efficient dynamic memory allocation is a necessity for all these applications. Although operating systems ship libraries that do have allocators, applications do not trust or use them. The reason being that these allocators are much more generic

than what is required by the applications and hence are not optimized for a given application. At the same time, these allocators need to be generic enough to work with any given application. In order to prevent the application slowdown using generic memory allocators, many applications today come with their own *custom allocators* e.g. The Apache Webserver [1], MySQL Database [15], Natural Language Processing Benchmark parser [18], GNU C Compiler [5] etc. As the application complexity and working set sizes increase, application writers cannot be expected to supply an efficient allocator with every application. This is not good software engineering practice, may not be feasible at times and can also result in duplication of work. Also, use of custom allocators often precludes the use of memory leak detection tools [30] and makes debugging more difficult. There are also several other disadvantages with traditional custom allocators and these have been discussed in [30] in detail. The solution rather lies in building an adaptive memory allocator that tunes itself to application behavior. As a next step in a self-optimizing MMS, this thesis proposes an allocator that *observes* incoming allocation/deallocation events and their properties and optimizes itself to the application. Before designing such an adaptive memory allocator, we first look at application allocation/deallocation characteristics. These characteristics show promise for an adaptive allocator. We then build an adaptive memory allocator and compare it with traditional allocators which are similar to custom allocators. Our results indicate that adaptivity can help reduce application runtime in many cases but can also hurt in a few. We conclude that adaptivity does not come for free. The overhead involved in monitoring application characteristics might overshadow the performance gain using adaptivity. We finally recommend an infrastructure that is a combination of our adaptive allocator and a traditional custom allocator that can reap the benefits of both.

1.3 Adaptivity in the Operating System

Applications whose working-sets fit into the memory would benefit from the optimizations discussed above that are in the processor-memory path. Many high-end server applications today do not fit in memory. For such applications, disk accesses become the bottleneck and it is important to minimize the need to go to the disk. The operating system's Virtual Memory Manager (VMM) is responsible for moving the data in and out of the disk(represented by (c) in Figure 1.1) and keeping the important data in the memory. VMM is composed of two main components: the policy and the parameters of the policy. The policy determines how the pages are managed by the operating system and is typically fixed. The parameters used by the policy, on the other hand, have influence on its performance and can be varied dynamically as the system is running. Operating system developers spend a lot of time tuning these parameters. During development, these parameters are set 'intuitively' based on the developer's experience. After the systems get deployed, if the users start observing poor application performance, these parameters are again tuned after massive experimentation. It is neither possible to think of every application when fixing these parameters, nor are the same values good for all classes of applications. Developers spend days to set these parameters after hundreds of experiments. Finally, 'hooks' are provided to change these parameters while the system is running if the user wishes to. Most of the users do not have any idea about the internals of the system and do not know how to set such parameters. This motivates the need for a self-tuning system that can tune the parameters by itself when different applications are running, to deliver the best performance at any time. Such a system would adapt to different applications automatically and no external tuning would be required.

Observing application characteristics within an operating system and tuning parameters dynamically using these characteristics is a very challenging problem. Towards this goal, investigating the influence of important VMM parameters on the application execution, coming up with methods to capture application characteristics in the operating system (where not all information is available), and correlating these characteristics to the influence of parameters is a good starting point. In this thesis, we undertake a thorough characterization study and compare pure application characteristics with those that can be inferred in the OS. We show that a few parameters have significant impact on the performance of the applications and finally show how these characteristics impact performance. The insights gained in this study provide a solid background to investigate different methods for designing a self-tuning VMM.

1.4 Application-controlled Memory Management

Finally, we would like to point out that memory management can also be done by applications themselves ((d) in Figure 1.1). Application developers can pass hints or use operating system/architectural features explicitly to manage the memory more efficiently. This would definitely place a large burden on the programmers. Instead, in the light of emerging system and application complexities, this thesis takes a different approach by proposing to self-optimize Memory Management at different levels - hardware, operating system and runtime environment - to save application programmers from this burden of application-level memory management.

1.5 Roadmap

The rest of the thesis is organized as follows: Chapter 2 goes over the issues on TLB characterization and optimization. Chapter 3 discusses the design and evaluation of the adaptive runtime memory allocator and Chapter 4 discusses the memory management issues in an operating system. Finally, Chapter 5 presents concluding remarks.

Chapter 2

Hardware Enhancements for the TLB

Virtual to physical address translation is one of the most critical operations in computer systems since this is invoked on every instruction fetch and data reference. To speed up address translation, computer systems that use page based virtual memory provide a cache of recent translations called the Translation Lookaside Buffer (TLB). The TLB is usually a small structure indexed by the virtual page number that can be quickly looked up by the memory management unit (MMU) to get the corresponding physical frame number. The importance of the TLB has always been recognized, and emerging technological and application trends only reaffirm its dominance in determining system performance. Increases in instruction level parallelism, higher clock frequencies, and the growing demands for larger working sets by applications continue to make TLB design and implementation critical in current and future microprocessors.

Several studies have quantified the importance of TLB performance on system execution [66]. Anderson et al. [23] show that TLB miss handling has an important consequence on performance, and this is the most frequently executed kernel service. TLB miss handling has been shown to constitute as much as 40% of execution time [63]. and upto 90% of a kernel's computation [94]. Studies with specific applications [97] have also shown that the TLB miss rate can account for over 10% of their execution time even with an optimistic 30-50 cycle miss overhead.

Processor	***i*-TLB**	***d*-TLB**	**TLB Miss Handler**
MIPS R10000	8-entry FA	64-paired-entry FA	Software
Alpha 21164	48-entry FA	64-entry FA	Software
PowerPC 604	128-entry 2-way	128-entry 2-way	Hardware
HP PA-RISC 2.0	4-entry Micro-TLB for Instructions 96-entry FA Main-TLB		Software
Sun UltraSparc-I	64-entry FA	64-entry FA	Software
Intel Pentium II	32-entry 4-way	64-entry 4-way	Hardware
Intel Itanium	64-entry FA	32-entry level1 FA 96-entry level2 FA	Hardware/Software
AMD Athlon (Thunderbird)	24-Entry level1 FA 256-Entry level2 FA	32-Entry level1 FA 256-Entry level2 FA	Hardware
AMD Athlon (Palomino)	24-Entry level1 FA 256-Entry level2 FA (Exclusive)	40-Entry level1 FA 256-Entry level2 FA (Exclusive)	Hardware

Table 2.1. TLBs in Commercial Microprocessors

With its importance widely recognized, the TLB has been the target for several optimizations to reduce access latency, miss rates and miss handling overheads. With regard to TLB structures themselves, there have been investigations on suitable organizations in terms of size, associativities and multi-level organizations [104, 84, 41, 25]. Superpaging is a concept that has been proposed to boost TLB coverage. The basic idea is to use a single entry to map several pages, thereby increasing the number of translations within the TLB. Hardware and software techniques for supporting this mechanism have come under a lot of scrutiny [104, 105, 103, 56, 88, 93]. Most prior work in TLB optimizations has targeted lowering miss rates or miss handling costs. It is only recently [97, 89, 26] that the issue of prefetching TLB entries to hide all or some of the miss costs has started drawing interest. Many research findings on TLBs have also made their way into commercial offerings. Today's microprocessors exhibit a wide range of TLB organizations and miss handling capabilities as is shown in Table 2.1. A very nice survey of several of these TLB structures can be found in [65]. In all, a good deal of research has been undertaken on TLB design and evaluation. However, to our knowledge, there has not been any study looking at characterizing the TLB support that is required from an application's perspective. At the same time, different studies have used different workloads to evaluate their designs/improvements, and the lack of a common ground makes a uniform comparison difficult. Further, one may ignore one or more issues when optimizing any particular detail, and such omissions can play an important role in the effectiveness of the optimizations. For instance, in the recent study on prefetching benefits [97], it is not clear if there is a reasonable window of chance for performing the prefetching. It is thus essential to uniformly examine a wide range of application characteristics at the same time (from an application's perspective) to gain better insights on:

- How can TLB structures/organization be optimized to reduce misses?
- How much scope is there to benefit from superpaging?
- Can we use the program's static code structure and/or dynamic instruction stream to trigger optimizations?
- Would the flexibility provided by software TLB management[1] offset the higher overheads of miss handling compared to a hardwired approach?
- How well suited are applications to prefetching TLB entries? Which prefetching techniques should be employed, and under what circumstances?
- Do we have a large enough window to benefit from prefetching? If not, what other techniques should we employ for latency tolerance/reduction?

Characterizing the behavior of an application is crucial in any systems design endeavor as many application-driven studies have shown [67, 47, 46]. Application characteristics can specify what is really important from an application's perspective, identify bottlenecks in the execution for a closer look, help evaluate current/proposed designs with real workloads, and even suggest architectural/OS enhancements for better performance (in fact, an examination of some of our characterization results have really led to the development of a new TLB prefetching mechanism called Distance Prefetching [73]) While there have been characterization efforts in the context of other processor features, caches [46], I/O and interconnects, this issue has not been looked at previously for TLBs.

[1] We would like to differentiate between the terms software TLB management and software TLB handling in this thesis. We use the latter to denote that the miss handling is done by the software i.e. the operating system, and the former term is used to denote more sophisticated software that can control placement, pinning and replacement of TLB entries. While many current systems provide software miss handlers (see Table 2.1), software TLB management has not been investigated extensively.

Towards addressing this deficiency, this thesis first sets out to examine the different characteristics of applications from the SPEC2000 suite that affect TLB performance to answer many of the questions raised above. The SPEC2000 [45, 61] suite contains 26 candidate C, C++ and Fortran applications for CPU evaluations, that encompass a wide spectrum of application domains containing interesting and important problems for several users. We have used all the applications from this suite in this study to quantify their TLB behavior for different configurations, but focus specifically on those incurring the highest miss rates for the characterization effort. We limit ourselves to d-TLB (only data references) in this study since data references are usually much less well-behaved than instructions in terms of misses (i-TLB miss rates in our experiments are very low) Several other studies have also focussed primarily on data misses [97].

2.1 Related Work

TLB design, implementation and management has always been and continues to be the target for different optimizations because of its presence in the critical path of program execution. As was mentioned earlier, many studies [43, 65, 23, 63, 94] have pointed out the importance of the TLB and the necessity of speeding up the miss handling process.

Several studies [104, 65, 84] have looked at hardware TLB structures/organization and their impact on system performance in terms of capacity and/or associativity. While some of these have focussed on single (monolithic) TLBs, there have been studies which have investigated the benefits of multi-level TLBs [41, 25]. There are also implementations of multi-level TLBs in commercial processors such as MIPS R4000, Hal's SPARC64, IBM AS/400 PowerPC, AMD K-7 and Intel Itanium. With instruction level parallelism (ILP) being exploited by most

current processors, there is a need a provide multi-ported TLBs to allow several concurrent instruction streams to access the TLB. Austin and Sohi [25] show how multiple ports can impact access latencies, and argue for interleaved and multi-level designs. They show that combining requests at the TLB access port (called piggybacking), to reduce the number of ports, can provide significant benefits.

TLB miss handling costs need to be kept extremely low for good performance. Commercial processors use either a hardware mechanism or a software mechanism to fill the TLB on a miss. Unlike hardware managed TLB misses which have a relatively small refill penalty, handlers for software managed TLBs need to be carefully crafted to both reduce misses as well as reduce the miss handling costs. Nagle et al. [84] study the influence of the operating system on the software managed MIPS R2000 TLB, and investigate the impact of size, associativity and partitioning of TLB entries (between OS and application). They point out that the operating system has a considerable influence on the number and nature of misses. Bala et al. [26] focus in specifically on interprocess communication activities, and illustrate software techniques for lowering miss penalties on software managed TLBs.

Superpaging is another well investigated technique to boost the coverage of the TLB and better utilize its capacity [104, 105, 103, 55]. Studies have looked at hardware and operating system support for providing superpage translations in the TLB. Recent work in this area [55] is investigating memory controller support for re-mapping pages so that there is more scope for creating superpage entries (without incurring the overheads of copying).

Application	i-TLB Missrate	Application	i-TLB Missrate	Application	i-TLB Missrate	Application	i-TLB Missrate
galgel	1.4×10^{-8}	mcf	4.4×10^{-9}	ammp	7.3×10^{-9}	apsi	2.46×10^{-7}
vpr	7.2×10^{-9}	lucas	1.1×10^{-8}	twolf	1.28×10^{-8}	facerec	8.8×10^{-8}
art	3×10^{-9}	bzip2	4.4×10^{-9}	parser	8.28×10^{-8}	vortex	2.74×10^{-4}
crafty	1.8×10^{-8}	swim	1.3×10^{-8}	applu	5.75×10^{-8}	gcc	1.08×10^{-4}
mesa	3.6×10^{-8}	mgrid	1.5×10^{-8}	equake	5.55×10^{-9}	perlbmk	3.35×10^{-5}
wupwise	1.2×10^{-8}	sixtrack	8.8×10^{-8}	gap	2.34×10^{-8}	fma3d	3.42×10^{-5}
		gzip	5×10^{-9}	eon	6.8×10^{-8}		

Table 2.2. i-TLB Missrates for all the applications using a 64-entry, 4-way set-associative i-TLB for 7 billion instructions

2.2 Experimental Setup

We have studied the TLB behavior for all 26 applications from the SPEC2000 suite. The benchmarks were compiled on an Alpha 21264 machine using Compaq's cc V5.9-008, cxx V6.2-024, f77 V5.3-915 and f90 V5.3-915 compilers using -O4(-O5 for Fortran) optimization flags which enable loop unrolling, software pipelining using dependency analysis, vectorization of some loops, inline expansion of small functions etc. All the simulations are conducted using the SimpleScalar-3.0 toolset [34] that simulates the Alpha architecture. Since we are mainly interested in the TLB behavior, we have modified `sim-cache` component of this toolset, by adding a TLB simulator that traps all memory references. `sim-cache` does a functional (not a cycle-by-cycle) simulation of instructions, and we only examine the memory references for the d-TLB investigation. While there could be some possible effects due to instructions retiring in a different order than that with a cycle accurate simulator, we do not feel that this will substantially change the results given in this thesis since we usually find that TLB misses are reasonably

spaced to affect the relative ordering of the misses. We also found that differences in what gets replaced is also not significantly affected by the coarser simulation granularity.

As was mentioned earlier, we are only examining the data references (d-TLB). Table 2.2 shows the i-TLB miss rates which are very low for these applications. Some studies have pointed out the influence of the OS on TLB behavior [84]. However, similar to what many other studies [97, 41] have done, in this thesis we examine only application TLB references(we do not simulate the OS) since our focus is more on investigating application level characteristics. Issues about the interference between application and OS TLB entries, or reserving some entries for the OS are not considered here (we understand that these issues can and have been shown to have a considerable influence on TLB performance). The effect of coexisting applications and context switching is not considered, i.e. TLB fills after a context switch. One could perhaps think of the TLB being saved and restored at context switches to capture purely application effects. Simulations have been conducted with different TLB configurations - sizes (64, 128, 256, and 512 entries), and associativities (full, 4-way and 2-way) and we assume a page size of 4KB. We do not consider the effect of page faults on TLB behavior (the entry needs to be invalidated on a page replacement), since we believe that page faults are much less frequent than TLB misses to have a meaningful influence. Further, all these applications take less than 256MB of space, which can be accommodated in most current systems.

Simulation of these benchmarks is very time-consuming as has been pointed out by others as well [87, 37]. In fact, the recent study on quantifying cache performance for these benchmarks mentions that it *takes 30 CPU years*, and their study used several workstations over several days to conduct this evaluation. In this study, we do not attempt to execute these applications to completion. Rather, we have examined the TLB behavior over *five billion instructions after*

skipping the first two billion instructions of the execution for each application. We believe that this ignores the initialization/start-up properties of the application, and captures the representative facets of the main execution. The simulated instructions constitute around 1%(`parser`)-12%(`mcf`) of the application run time [37].

For `gzip/bzip2`,`perlbmk` and `vortex` the input files that we used are `input.source`, `diffmail.pl` and `lendian3.raw` respectively.

The term, *miss rate*, which is often used in the following discussion is defined as the number of TLB misses expressed as a percentage of the total number of memory references.

2.3 Characterization Results

2.3.1 What is the impact of TLB structure?

As a starting step, we first examine the overall TLB performance for the different applications for 9 different TLB configurations (combinations of three sizes - 64, 128, and 256, and three associativities - fully associative (FA), 4-way and 2-way). The resulting miss rates are shown in Figure 2.1, and we observe the following:

- A diverse spectrum of TLB characteristics is exhibited by these applications. We have applications such as `gzip`, `eon`, `perlbmk`, `gap`, `vortex`, `wupwise`, `swim`, `mgrid`, `applu`, `mesa`, `art`, `equake`, `facerec` and `fma3d` which incur few TLB misses. On the other hand, applications such as `vpr`, `mcf`, `twolf`, `galgel`, `ammp`, `lucas`, `sixtrack` and `apsi` have a significant number (greater than 1%) of TLB misses (at least in some of the configurations). TLB miss penalties can be quite significant. For instance, the IBM PowerPC 603 that handles misses in software, incurs a latency of 32

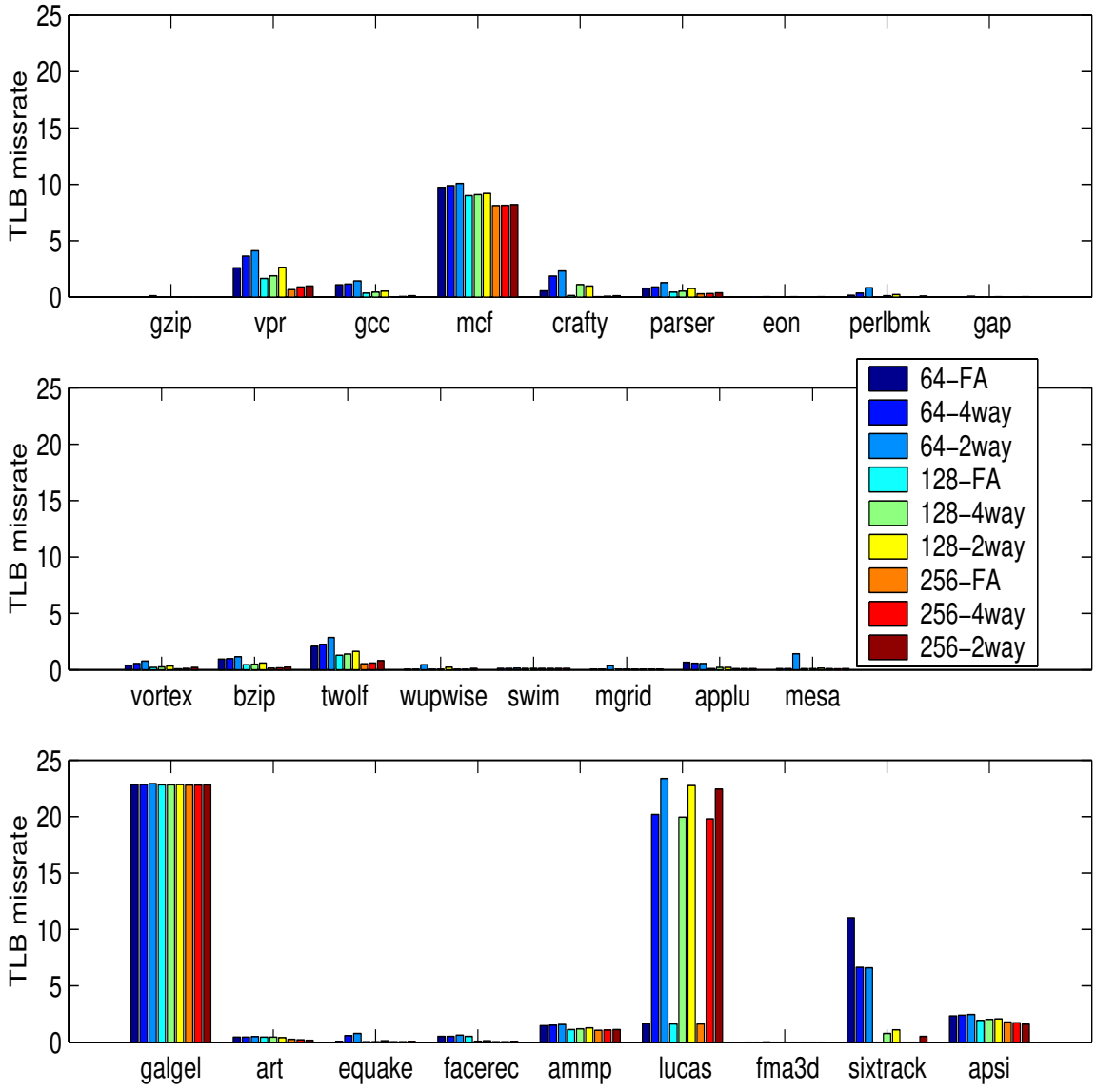


Fig. 2.1. Figure showing TLB miss rates of all the SPEC2000 applications with different TLB sizes (64, 128, 256) and three different associativities (2-way, 4-way and FA).

cycles just to invoke and return from the handler, leave alone the cost of loading the TLBs [53]. Even if we optimistically assume a TLB miss penalty of 30 cycles, a 1% TLB miss rate can result in significant overheads - around 10% of the overall execution time for a CPI of 1.0. Other studies have pointed out that TLB miss handling overheads can be even higher [55, 97, 56]. Applications such as `bzip`, `gcc`, `crafty` and `parser` fall between these extremes. We focus on the six applications which incur higher TLB miss rates.

- Even though making the TLB larger helps, the benefit is not very significant in most of these applications except `vpr`, `twolf` and `mcf`.

- Applications such `vpr`, `mcf`, `twolf`, `lucas` and `apsi` are more influenced by the associativity than the others.

These results reiterate the importance of the TLB in the execution of these applications. Nearly a fourth of the SPEC2000 suite have reasonably high TLB miss rates to be affected by its miss penalty. While capacity and associativity do help in cases, they are not the universal solutions to address the TLB bottleneck.

2.3.2 Will a multi-level TLB help?

Processors, such as the Itanium IA-64 (32-entry L1, 96-entry L2), AMD Athlon (32-entry L1, 256-entry L2) etc. provide Multi-level TLB structures, instead of a single monolithic TLB, i.e. lookup in a smaller first level TLB, and only on a miss there do we go to the second level TLB. With a smaller first level TLB, overall TLB lookup time can become much lower as long as we have good hit rates in the first level. Performance of 2-level TLBs has been conducted by others [41], but its benefit for SPEC2000 workloads has not been investigated to our knowledge.

An in-depth investigation of the impact of multi-level TLBs requires varying numerous parameters for each of the levels. Rather than undertake such a full factorial experiment, we limit our study to a 2-level TLB with a 32 entry fully associative 1st level and a 96 entry fully associative 2nd level that is similar to the Itanium's(IA-64) structure. We also assume that the 2-level TLB obeys the containment property(whatever is in the first level is duplicated in the second level). Table 2.3 shows the hit and miss rates with this 2-level structure for the applications.

Application	1^{st} Level-1 TLB		2^{nd} Level-2 TLB		Overall Miss Rate	Monolithic TLB	
	Hits	Misses	Hits	Misses		Hits	Misses
ammp	98.07%	1.93%	38%	62%	1.2%	98.87%	1.13%
mcf	89.5%	10.5%	11.2%	88.8%	9.24%	91.01%	8.99%
twolf	96.57%	3.43%	53.2%	46.8%	1.60%	98.71%	1.29%
vpr	94.5%	4.5%	56.6%	43.4%	1.95%	98.36%	1.64%
lucas	98.33%	1.67%	2%	98%	1.64%	98.37%	1.63%
apsi	97.39%	2.61%	12.6%	87.4%	2.28%	98.04%	1.96%
galgel	77.1%	22.9%	0.05%	99.95%	22.8%	77.2%	22.8%

Table 2.3. Hit and Miss Rates for the 2-level TLB configuration. Table shows the hits and miss rates for each of the 2-levels as a percentage of the number of references to that level, as well as the overall miss rate which is the percentage of references that do not find a translation in either of the levels. Also shown are the miss rates for a single monolithic TLB of the same size.

To study the impact of a hierarchical TLB, we compare these results with that of a single monolithic TLB of the same size (32 + 96 = 128 entries) in Table 2.3. It is to be expected that the 2-level TLB is making less effective use of the overall space because of the containment property and as a result we do find slightly higher miss rates for the seven applications (especially `twolf`, `vpr`, `apsi` and `mcf`). On the other hand, the benefit of a multi-level TLB would be felt due to

the much lower access time of the 1st level TLB and the avoidance of accessing the second level - which is inturn determined by the miss rates for the 1st level TLB.

Assume that the access time for a single monolithic TLB (i.e. 128 entry in this situation) is a. Let the access time for the 1st and 2nd level TLBs in the hierarchical alternative be a_1 and a_2 respectively. Let us denote the miss fraction of the monolithic TLB, the 1st level and 2nd level of the hierarchical TLB to be m, m_1, and m_2 respectively. Also, let the cost of fetching a translation that is not in the TLB be denoted by C. Then, the cost of translating an address in the monolithic structure (C_m) is calculated by

$$C_m = a + m \times C \tag{2.1}$$

The cost of translating an address in the 2-level TLB (C_s) is given by

$$C_s = a_1 + m_1 \times (a_2 + m_2 \times C) \tag{2.2}$$

The 2-level TLB is a better alternative when

$$a_1 + m_1 \times (a_2 + m_2 \times C) < a + m \times C \tag{2.3}$$

i.e.,

$$m_1 < \frac{a - a_1 + m \times C}{a_2 + m_2 \times C} \tag{2.4}$$

Actual access times for associative memories of different sizes are hardware sensitive and fairly accurate models have been developed. We use the model described in [82]. According

to this model, access times for 32, 96 and 128-entry TLBs would approximately be $2.7ns$, $2.9ns$ and $3.0ns$ respectively. Plugging in these values and assuming an optimistic 30 cycle miss penalty(C = 30), we find that the miss rate (m_1) in the first level TLB has to be less than 2.97%, 10.15%, 4.05%, 4.97%, 2.44%, 3.04% and 21.71% for `ammp`, `mcf`, `twolf`, `vpr`, `lucas`, `apsi` and `galgel` respectively. If we compare these numbers with the actual miss rates in the 1st level TLB in Table 2.3(column 3), multi-level structure is definitely a better choice for `ammp`, `twolf`, `vpr` and `apsi`. For `mcf` and `galgel`, the choice is questionable since the difference in miss rate is not significant (and we have used a very optimistic 30 cycle miss penalty). It is only for `lucas` that level-2 TLB does not do as well.

2.3.3 Can we improve TLB coverage? (Superpaging)

The previous set of results showed how much we can gain by improving the structure (i.e. capacity and associativity) of the TLB, and these gains are obtained with an increase in hardware complexity. The recent trend [104] to improve performance without significantly increasing hardware costs is through the concept of superpaging. TLBs that support superpaging use a single entry for translating a set of consecutive virtual pages - the number of pages for a single entry is typically a power of two - as long as these pages are located physically contiguous as well. As a result, pages that are accessed at around the same time and which are adjacent to each other virtually can share the same TLB entry, thus freeing up slots for other translations (to improve TLB coverage).

To study the potential benefits of superpaging for these applications, we next examine the contents of the TLB during the course of execution. In particular, we are interested in finding out how small a TLB would suffice to hold all the translations if it supported superpaging. At every

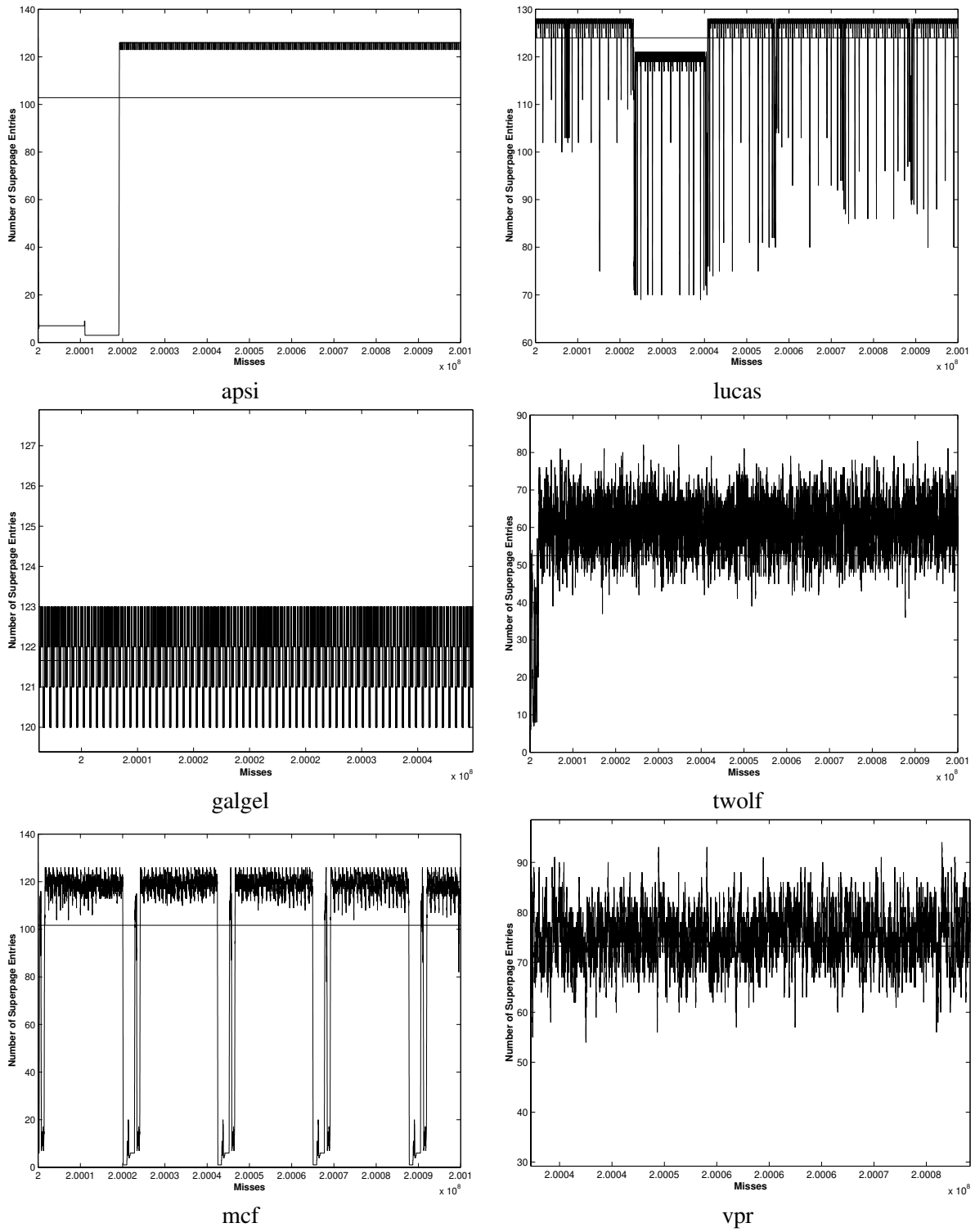


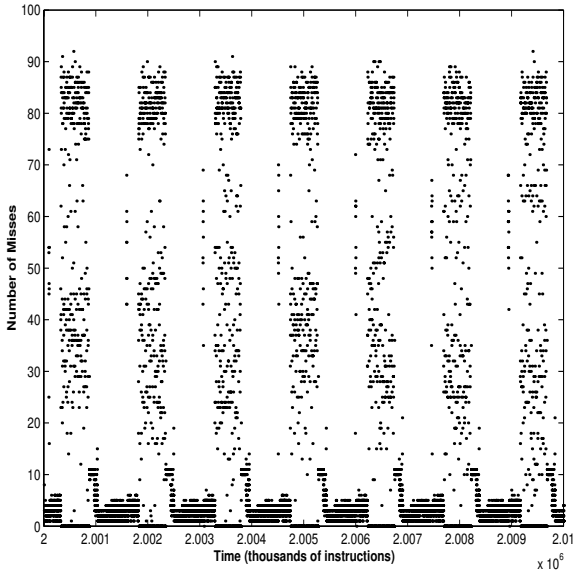
Fig. 2.2. Figure showing what size TLB would suffice when we combine contiguous virtual page translations with superpage entries. A 128 entry fully associative TLB is used.

miss (since TLB changes only on a miss), we examine all the TLB entries and find out how many of them can be combined into a single one (they are virtually contiguous pages). The number of resulting entries is plotted in Figure 2.2 during the course of execution (in terms of the misses in the x-axis). It should be noted that we are not restricting ourselves to a power of two and are assuming that the operating system will automatically allocate these virtually contiguous pages in contiguous physical locations, since our goal here is to examine the potential of superpaging from an application perspective. For better clarity, only a small window of the execution is plotted (similar execution is seen over a much larger window).

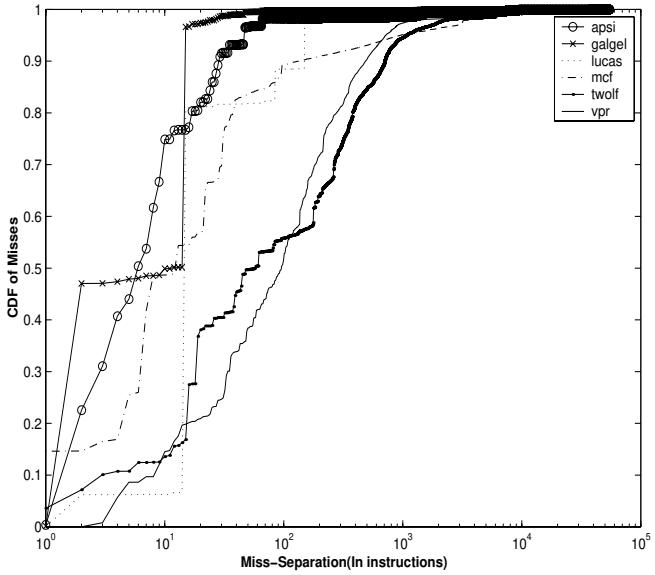
We find quite different behaviors across the applications. Applications like `lucas`, `mcf`, `galgel` and `apsi` may not benefit much from superpaging. These applications require at-least around 100 TLB entries on the average without too much scope for combining these entries. Provisioning superpage support may not show much improvement in the performance while adding hardware complexity. On the other hand, applications like `ammp`, `twolf` and `vpr` show the potential to benefit from superpaging mechanisms. Of these, the working set in `ammp` at any time can be covered by just a handful of superpage entries (since it just sequentially scans a lot of data).

2.3.4 Where are the Misses Occurring?

We move on to examining how the TLB misses are distributed over the execution of the program. This can help us identify hot spots in the execution that are worth a closer look for optimizations. Dynamically chosen/triggered optimizations based on different phases of the execution may perhaps be beneficial, if we do find well-defined phases.



(a) TLB misses as the application is executing(for `mcf`). x axis denotes the number of instructions executed in thousands, and y axis denotes the number of misses for every thousand instructions.



(b) Cumulative Density Function (CDF) of miss separation/interval. x-axis shows the number of instructions between successive misses in increasing order, and y-axis denotes the fraction of misses that have miss separations of at most this many instructions.

Since we are working with the `sim-cache` version of SimpleScalar, we do not have the time stamps for the memory references and the TLB misses. Instead, we plot the TLB misses as a function of the number of executed instructions (in thousands). Note that the scale of the graph can make it look as though there are multiple y values for a given x, while this is actually not the case.

From the dynamic optimization viewpoint, `mcf` a candidate that shows phases of varying TLB activity as shown in the Figure 2.3.4(a). The variance over time for the others is not as significant, and dynamic triggers in those cases may not be very helpful.

2.3.5 How far apart are the misses temporally?

The temporal separation of misses is an important characteristic for examining possible optimizations. If most misses are fairly close to each other temporally, then mechanisms that prefetch a TLB entry based on the previous miss may not enjoy a large enough window of overlap to completely hide miss latencies. In such situations, we should explore techniques for bulk loading/prefetching TLB entries. A significantly larger window suggests that more sophisticated prefetching techniques (which may take more time) can be viable. To our knowledge, no previous work has looked at the temporal separation of TLB misses previously. Figure 2.3.4(b) shows the cumulative density distribution of temporal miss separations (i.e. the y-axis shows the fraction of misses that are separated by at most a given value on the x-axis), where the separations are expressed in terms of instructions. A steep curve indicates that more misses have very short temporal separations.

For nearly all applications, a large portion of misses are separated by at most 30-50 instructions. For instance, `ammp`, `galgel` and `apsi` have over 90% of misses separated by

less than 50 instructions. `mcf` also has small temporal separation with around 70% of misses falling within 50 instructions. Only `lucas`, `twolf` and `vpr` have less steep curves. In some applications (`lucas` and `galgel`), a few miss-distances dominate the execution, suggesting that the TLB misses are occurring very periodically/regularly.

These results provide insight on the constraint window that prefetch mechanisms need to operate under. As CPIs get smaller (with higher ILPs) and processor speeds increase (making memory from where the TLB entries are fetched an even bigger problem), this window can shrink even further.

2.3.6 Where are the misses occurring? - Static

If there are not very well demarcated phases of execution to trigger actions dynamically based on the phase, then it may perhaps be more rewarding to examine the (static) structure of the application programs to find out where compiler or user-directed optimizations need to concentrate on. To identify the points in the program contributing to the TLB misses, we plot the number of data TLB misses incurred by each program counter (PC) value in Figure 2.3. Overall, we find that there are only a handful of instructions that contribute to the bulk of TLB misses. These memory reference instructions are typically called repetitively (in a loop), and in the following discussion we briefly go through each application describing the points in the program corresponding to these instructions.

2.3.7 How do procedure calls influence TLB behavior?

Another possible piece of information that may be useful is how the dynamic procedure activations are affecting TLB performance. Just as context switches are wiping out locality

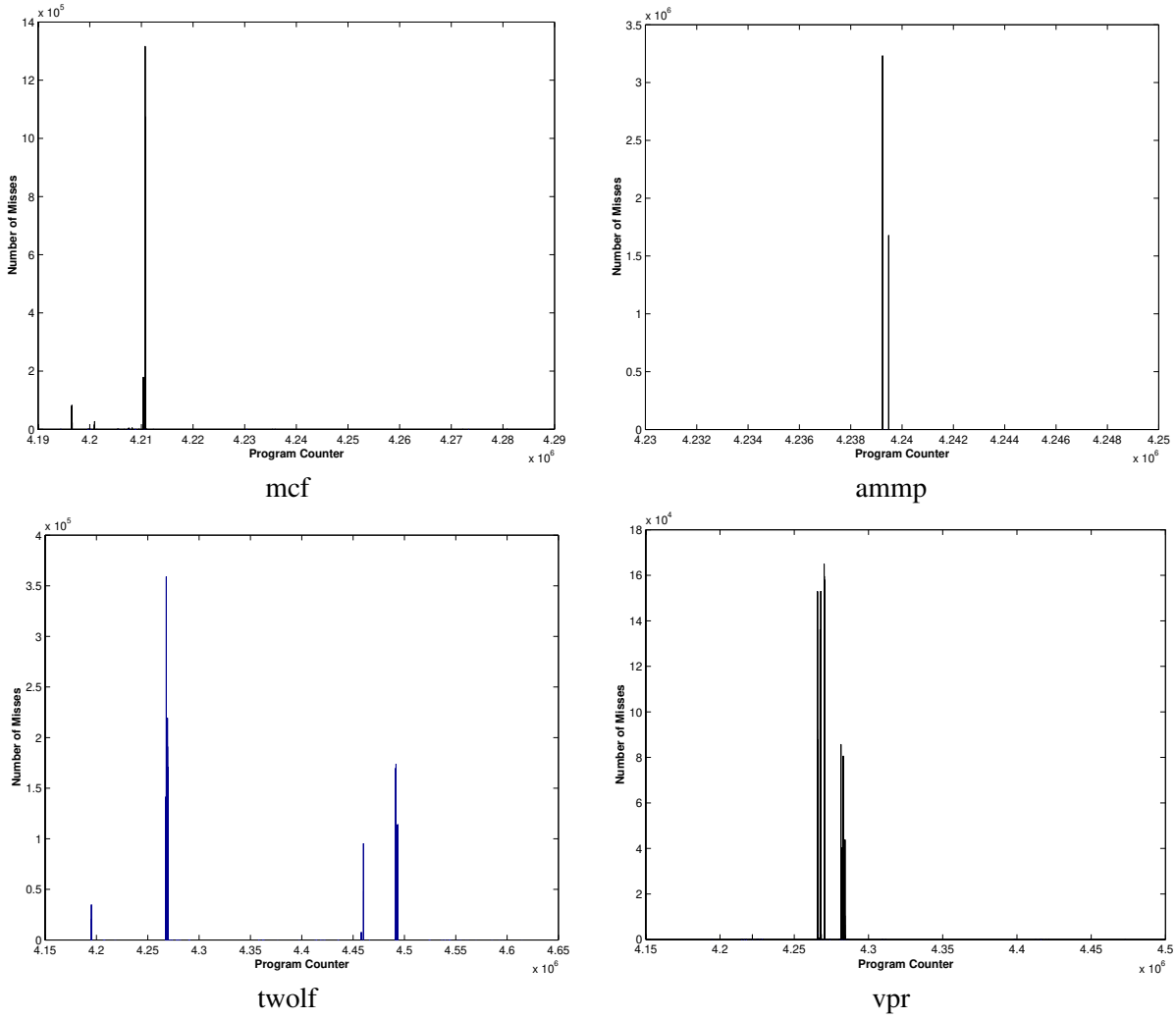
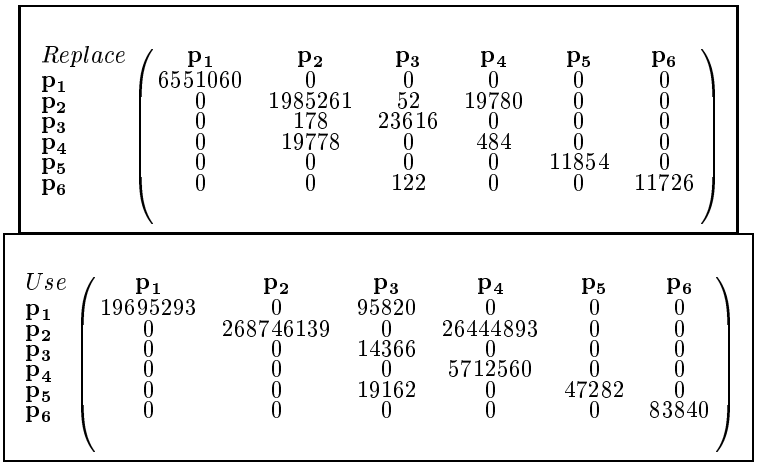
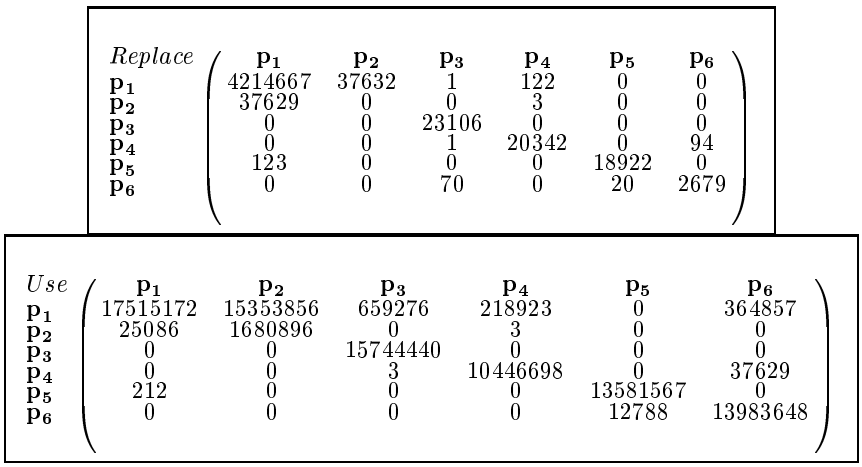


Fig. 2.3. PC Values Incurring TLB Misses

ammp



apsi



galgel

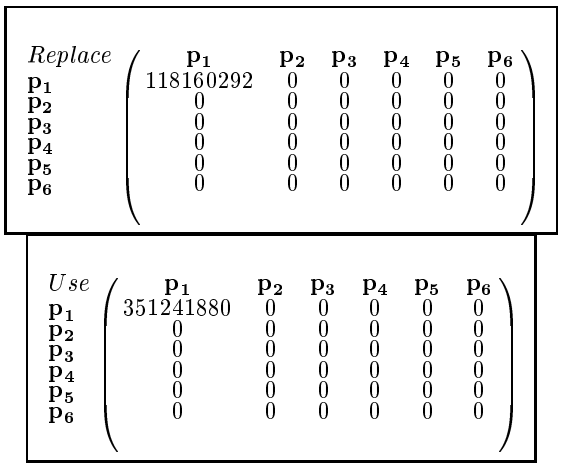


Fig. 2.4. Figure showing Use and Replace matrices for different procedures. Applications like `galgel` will not benefit from procedure based TLB allocation.

lucas

Replace

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	2134568	0	115	3	1	3
p_2	128	1354750	0	0	0	0
p_3	0	115	8071	2	3	0
p_4	0	2	3	0	0	0
p_5	0	3	1	0	0	0
p_6	0	0	3	0	0	0

Use

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	88043027	6205515	12583409	74	228	32
p_2	8	116268798	0	0	0	0
p_3	1	1377055	8380144	0	2	0
p_4	0	1	0	53	1	0
p_5	0	0	0	16	47	0
p_6	0	0	0	68	75	8

mcf

Replace

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	11515218	84758	3442	0	0	1400
p_2	37495	5309290	2825387	294807	39965	34437
p_3	22710	2642559	1253810	167969	23458	19619
p_4	22596	394896	37779	0	7467	2
p_5	3994	62796	3621	0	95966	951
p_6	2723	47108	6101	0	473	1

Use

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	341317243	33233	4325	0	0	1400
p_2	42784	5684912	93639	217971	159982	5450
p_3	691	11717955	26204887	555515	120620	213272
p_4	168504	678494	0	2600412	1622580	36553
p_5	31769	20039	1144	0	757815	397
p_6	1400	49470	184	0	12915	81318

vpr

Replace

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	1534418	665568	389960	205716	185082	31914
p_2	671606	295406	171910	87269	79305	10895
p_3	397985	170213	93952	50505	48062	7159
p_4	202392	88042	51153	26917	25409	4563
p_5	181524	82155	51514	24179	21356	2921
p_6	24720	14994	9401	3902	4435	1202

Use

	p_1	p_2	p_3	p_4	p_5	p_6
p_1	115982826	18038095	13586974	18778831	31114123	0
p_2	844524	3615068	1110638	0	442	0
p_3	906738	925468	6021517	0	1011	0
p_4	13584480	0	0	11316157	0	0
p_5	15325	0	43204	0	11045628	0
p_6	53010370	4102716	685043	10258594	2896227	6222597

Fig. 2.5. Figure showing Use and Replace matrices for different procedures. `mcf` is a potential candidate because of interaction between p_2, p_3 and p_4.

from one application to another, one could hypothesize that the working sets may also change when a procedure A invokes B. At the same time, both procedures may also share several data structures. These two factors can lead to interference and sharing of TLB entries between these procedures. Identifying such characteristics could be useful in different ways. If the interference is significantly higher, but the locality carries over from the previous time this procedure was invoked, then one could save the TLB upon a procedure call and restore the TLB from the last exit out of that procedure. With a little hardware support (to switch/load/save TLBs), the software could invoke mechanisms appropriately to facilitate this. If we find a strong case for having separate sets of TLBs, then we could even advocate supporting more than one TLB in hardware (even though there is only one active TLB at any time), and switching from one to the other at runtime. A strong sharing across different procedures may indicate that these enhancements are not going to be very useful (and could in fact hurt performance). Finally, if some procedures share entries significantly, but not all or if there is significant interference with other procedures, then we have a TLB assignment problem for each procedure (with a small number of TLBs how do I assign a procedure to a specific TLB) that can be interesting for future research. To investigate this issue, we tracked how many times a procedure B brought in an entry that evicted an entry brought in earlier by procedure A. We call this the `Replace` matrix, and show this for the top 6 procedures incurring misses. Similarly, we also have a `Use` matrix, that shows how many time procedure B used an entry that was brought in by procedure A. The Replace and Use matrices are given in the Figures 2.4 and 2.5.

In general, we find no strong case advocating separate TLBs or saving/restoring TLBs. When an element in the Use matrix is high, the corresponding element in the Replace matrix is also high in most of the cases. There is less sharing or interference across procedures for `ammp`,

`apsi`, `galgel`, and `lucas`, compared to the others. In fact, we even conducted experiments with separate TLBs to explore any benefits, but the miss rates were not affected significantly.

2.3.8 How would software TLB management help?

Application	64-entry			128-entry			256-entry		
	OPT'	LRU	OPT'/LRU	OPT'	LRU	OPT'/LRU	OPT'	LRU	OPT'/LRU
ammp	1.13%	1.47%	0.77	0.98%	1.13%	0.87	0.89%	1.06%	0.84
mcf	9.3%	9.73%	0.955	8.86%	8.99%	0.985	7.34%	8.1%	0.90
twolf	1.3%	2.08%	0.625	0.65%	1.29%	0.503	0.2%	0.53%	0.377
vpr	1.54%	2.58%	0.596	0.80%	1.64%	0.487	0.23%	0.66%	0.348
lucas	1.64%	1.64%	1.0	1.62%	1.63%	0.99	1.57%	1.61%	0.975
apsi	1.85%	2.34%	0.79	1.38%	1.96%	0.704	0.77%	1.79%	0.43
galgel	22.5%	22.8%	0.986	19.8%	22.8%	0.868	14.4%	22.7%	0.634

Table 2.4. Comparing miss rates for OPT' and LRU replacement policies

TLB management is essentially trying to (i) bring the entry that is needed into the TLB at some appropriate time, and (ii) evict an entry currently present to make room for this incoming entry. These two actions play a key role in determining performance. The first action determines the overhead that is incurred when a miss is encountered, while the latter affects the miss rate itself (the choice of replacement candidate will determine future misses). Hardware TLB management, though fast, hardwires these mechanisms allowing little scope for flexibility. Typically, the missing entry is brought in on demand at the time of a miss (if we discount any prefetching), and the replacement is determined usually based on LRU (either global in terms of fully associative, or within a set in case of set associative). On the other hand, selecting the replacement candidate in software - either by the application or by the compiler - can potentially

choose better alternatives than a hardwired LRU. This is an issue that has not been investigated in prior research to our knowledge in the context of TLBs. Most current TLBs do not provide such capabilities, but any strong supporting arguments in favor of this could influence future offerings. Prefetching TLB entries to tolerate/hide the miss latency is discussed later in section 2.4.

A detailed investigation of software directed replacement techniques is well beyond the scope of this thesis. Rather, we would like to find out what is the best that we can ever do, so that we can get a lower bound of achievable miss rates with any software directed replacement scheme. Such a study can be conducted by simulating the OPT replacement policy, i.e. at any instant replace the entry whose next usage is furthest in the future. It can be proved that no other replacement scheme can give lower miss rates than OPT (which is itself not practical/implementable). We simulate a TLB using OPT and compare the results with that for a fully associative TLB using LRU of the same size. We would like to mention that simulation of OPT exactly is extremely difficult (pointed out by others [101]) since it involves looking into the future (infinitely). Instead, at every miss, we look ahead one million references in our simulation (which we call OPT'). We have tried looking further into the future and the results are not very different, leading us to believe that the results with OPT' are very close to that of OPT. Table 2.4 compares the miss rates for OPT' and the TLB using LRU for three different sizes (particularly note the columns denoting the ratio of miss rates between the two approaches).

When the TLB size is very small, either scheme incurs a lot of misses, and the differences between the schemes are less significant than at larger TLB sizes (in many cases). At the other end, when the TLB size gets large enough that the working set fits in it, the differences between the schemes again become less prominent. We find that in these applications, a 256 entry TLB

is still small enough that the benefits of OPT' over LRU continue to increase with size. `ammp` and `mcf` are exceptions at small TLB sizes.

There are some interesting observations/suggestions we can make from these results. First, we find that in many situations, OPT' gives substantially lower miss rates than LRU, suggesting that software directed replacement needs to be explored further. In applications such as `ammp`, the reference behavior is statically analyzable, making compiler/application directed optimizations worthy for exploration. While the dynamic reference behavior in other applications may not be readily amenable for static optimizations, our results show that it is definitely worthwhile to explore a combination of static and runtime approaches to bring the miss rates closer to OPT'.

Second, despite the significant improvements with OPT', we still find that there is still a reasonably large miss rate that any software directed replacement scheme cannot reduce. This motivates the need for miss latency tolerance techniques such as prefetching (either by software or hardware) which is explored later in section 2.4.

2.3.9 How far apart are the misses spatially?

Having seen the temporal separation of misses, we next move on to examining the spatial distribution of the misses in terms of the virtual addresses themselves. Any observable patterns/periodicity would be extremely useful for prefetching TLB entries from both the architectural and compiler perspectives. Figure 2.6 shows what virtual addresses (on the y-axis) incur misses during the execution of the applications (expressed in terms of the misses on the x-axis). In consideration of the scale of the graph, the x-axis only shows a small representative window of the execution so that any observable patterns are more noticeable. Again, the scale of the

graph may make it appear as though there are two y-values for a given x-value, though this is not really the case. It should be noted that there are earlier studies which have drawn such graphs [41, 80] for application reference behaviors. The difference is that we are looking at the misses (not the references) since that is what TLB prefetch engines work with as is discussed next, and also in that no one has looked at these graphs for SPEC2000 applications.

We observe that many applications exhibit patterns in the addresses that miss, which repeat over time. For applications such as `ammp`, `galgel`, `apsi`, and to a lesser extent `mcf` and `lucas`, the patterns are more discernable. In `twolf` and `vpr`, the patterns if any, are not very clear. These patterns are a result of the iterative nature of many of these executions that repeat reference behavior after a while.

2.4 Prefetching TLB Entries

There are several approaches to improve the delivered performance of TLBs. On the software side - at the application, compiler or operating system level - optimizations for improving locality can help lower the number of TLB entries needed to cover the working set of the execution at any instant. On the hardware side, TLB structure in terms of its size and associativity has a significant impact on both the miss rates as well as on the access times [104, 41, 25]. A technique that is employed in some commercial CPUs (MIPS R4000 [83], Hal's SPARC64 [59], IBM AS/400 PowerPC [33], AMD K-7 [70] and Intel Itanium [64]) based on the trade-offs between miss rates and access times, is to build the TLB as a multi-level structure. Such a structure can reduce the access times for the most frequent lookups without significantly affecting miss rates compared to a monolithic implementation. Trade-offs when increasing the number of TLB ports for multiple issue machines have also been investigated [25]. Another solution to boost

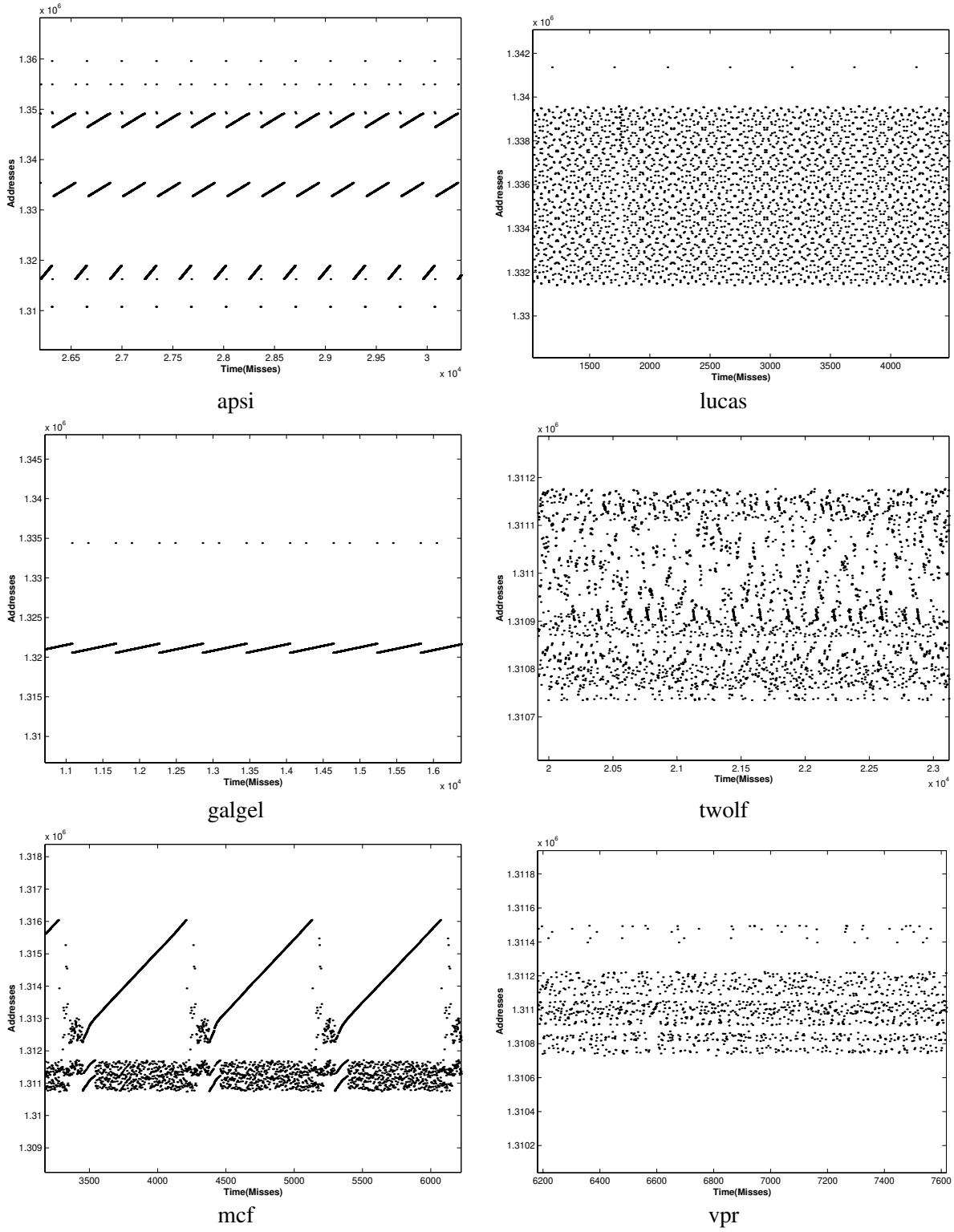


Fig. 2.6. Figure showing the data addresses that miss during the course of program execution shown in the x-axis in terms of misses (Time).

TLB coverage is by the use of superpaging [104, 105, 103, 55, 56, 88, 93]. The general idea is to find contiguous virtual page translations in the TLB and combine them into a single entry. OS issues for facilitating this and memory controller designs to find more opportunities for superpaging have been investigated. Finally, on the miss handling side, a considerable amount of effort has been expended on tuning software miss handlers [94] or for performing the necessary actions in hardware [64, 70].

However, it is only recently [97, 89, 26] that the issue of prefetching/preloading TLB entries to hide all or some of the miss costs has started drawing interest. Some of these [26, 89] consider prefetching TLB entries only for the cold starts, which in many long running programs (such as the SPEC 2000 suite) constitute a much smaller fraction of the misses. The first work on prefetching TLB entries for capacity related misses has been undertaken in [97]. Despite the voluminous literature on prefetching techniques available for other levels of the memory hierarchy, prefetching TLB entries has not gained much attention. This is, perhaps, due to the fear of slowing down the critical path of TLB accesses (which is usually much more important than the other levels of the memory hierarchy) and the possible cost/space of the additional real-estate (one could make a less strong argument about this with the ability to pack in billions of transistors on chip, though there is still the issue of power consumption and distribution that needs to be considered) that may need to be provisioned on-chip. However, we need to understand the benefits and ramifications of prefetching TLB entries inorder to be able to make these trade-offs. In this thesis, we specifically focus on the data TLB (d-TLB), which is usually much more of a problem than instruction TLB (i-TLB) in terms of miss rates (many previous studies [97, 55] specifically target d-TLB as well).

Addressing the critical path issue, Saulsbury et al. [97] propose a new mechanism, called Recency-based Prefetching (RP), that maintains an LRU stack of page references and prefetches the pages adjacent to the one currently referenced (on either side of the stack). The associated logic is placed after the TLB, i.e. it has the privilege of examining only the misses from the TLB (and does not look at the actual reference stream). They can afford to do that since the TLB itself keeps (and replaces) entries in LRU order in hardware automatically (assuming a fully associative TLB). The entries that are prefetched are not loaded into the TLB directly. Rather, they are kept in a prefetch buffer which is concurrently looked up with the TLB (and then moved over to the TLB if actually referenced). As a result, this cannot increase the miss rate of the original TLB. However, RP requires maintenance of the LRU stack in the page table (which resides in the memory hierarchy), requiring two additional pointers for each entry (page table size can get very large). Further, removal of an entry from the stack (when brought in upon a miss) and adding an entry to the top of stack (when evicted from TLB), each require two pointers to be manipulated. Even though this can go on in the background, there can be a lot of additional memory system traffic that can be generated, which may not only interfere with the normal TLB miss loading memory traffic but also with the rest of the data references in the program. These issues have not been investigated in prior research.

A number of prefetching mechanisms have been proposed in the context of caches [108, 42, 68, 48, 69, 75, 49] and I/O [77]. To our knowledge, no prior study has investigated the suitability of these earlier proposals for TLB prefetching. It would be very interesting to see how these earlier proposals would work with the miss stream coming out of the TLB. While many of these schemes may require a little more logic/real-estate on-chip than RP, they usually do not impose as much storage and bandwidth requirements as RP.

It is well beyond the scope of this thesis to cover a detailed survey/classification of prefetching mechanisms or to evaluate all of them (if one is interested, a survey of these can be found in [108]). Rather, we want to cover some representative points of the spectrum of mechanisms in the context of TLB prefetching. In a broad sense, prefetching mechanisms can be viewed in two classes: ones that capture strided reference patterns (using less history, such as sequential prefetching or arbitrary stride prefetching (ASP) [48, 42]), and those that base their decisions on a much longer history (such as markov prefetching (MP)[68] or even the recency based mechanism (RP) discussed above). Reference behavior can also be viewed as following broadly one of these categories: (a) showing regular/strided accesses to several data items that are touched only once; (b) showing regular/strided accesses to several data items that are touched several times; (c) showing regular/strided accesses to several data items, but the stride itself can change over time for the same data item; (d) not having constant strided accesses (either keeps changing constantly or there is no regularity in the stride at all), but repeating the same irregularity from one access to another for the same data item over time; and (e) not having any regularity either in strides and not obeying previous history either. Usually stride based schemes are a better alternative than history based schemes for (a) (there is no history established here), while both categories can do well for (b). Some of the more intelligent/adaptive stride based schemes such as ASP can track (c) also fairly well, but the history based schemes are not as good for such behavior. On the other hand, history based schemes can do a much better job than stride based schemes for (d). In (e), it is very difficult for any prefetching scheme to be able to do a good job.

As we can observe, neither of the classes can do well across all of (a) through (d). Instead, we propose a new prefetching mechanism called *Distance Prefetching (DP)*[2] in this thesis that tries to get the better of both approaches. The idea is to approximate the behavior of stride based mechanisms whenever there are very regular strided accesses (and capture first time references as well which are not possible in a history based mechanism), and track the history of strides (that is indexed by the stride itself). The hope is that whenever the stride changes, the changes themselves form a historical pattern and we can refer to this history to make better predictions. We find that DP can do fairly well (approximating the better of the two classes) for all of (a) through (d). DP is a general prefetching technique, that can be used in several situations (for caches, I/O etc.). In this thesis, apart from proposing this new general purpose technique, we specifically illustrate its design and use for tracking TLB misses (placed after the TLB) to make good predictions. It takes space that is comparable to that of some of the earlier history based mechanisms such as Markov (usually a 256 entry direct mapped table suffices), while making much more accurate predictions. It also incurs much less memory traffic compared to RP which is the only other prefetching technique proposed and evaluated specifically for TLBs.

Using a wide range of diverse applications spanning several benchmark suites (26 applications from SPEC CPU2000 [45], 20 applications from MediaBench [76], 5 applications from the Etch traces [107], and 5 applications from the Pointer Intensive Benchmark suite [102]), this thesis makes the following contributions:

[2]Distance Prefetching also tracks strides to make predictions. In the interest of distinguishing this mechanism clearly from the earlier stride based mechanisms, we give it an entirely different name using the term "distance". "Distance" and "stride" mean the same thing and refer to the spatial separation (could be positive or negative) between any two successive references.

- We compare the performance of two previously proposed prefetching mechanisms (ASP and MP) that have been evaluated in the context of caches, by studying their potential in making predictions based on the miss reference string coming out of the TLB. These mechanisms are compared with the only other existing TLB prefetching mechanism - RP - that has been evaluated for d-TLBs.

- We present a new prefetching mechanism, called distance prefetching (DP), that can do a better job of capturing a wider spectrum of reference behaviors than some stride-based (ASP) and history-based (MP and RP) mechanisms.

- We demonstrate DP to be a much better alternative than the other mechanisms for TLB prefetching. It is able to make good predictions at a reasonable hardware cost, and incurs much lower memory traffic overheads than RP.

- We also conduct several investigations into the design choices for implementing DP.

2.5 Prefetching Mechanisms

Since we extensively refer and compare against previously proposed prefetching mechanisms (including those used for caches), we briefly go over these to refresh the reader and to point out the exact implementation that is used later on in the evaluations. We also present our new prefetching mechanism - DP - in this section. It is to be noted that for uniformity in this adaptation, all these mechanisms initiate prefetches only by looking at the miss stream from the TLB, that is done in the earlier proposed RP mechanism [97] for TLB prefetching. All these mechanisms bring the prefetched entry into a "prefetch buffer" that is concurrently looked up with the TLB, and the entry is moved over to the TLB only on an actual reference to that entry

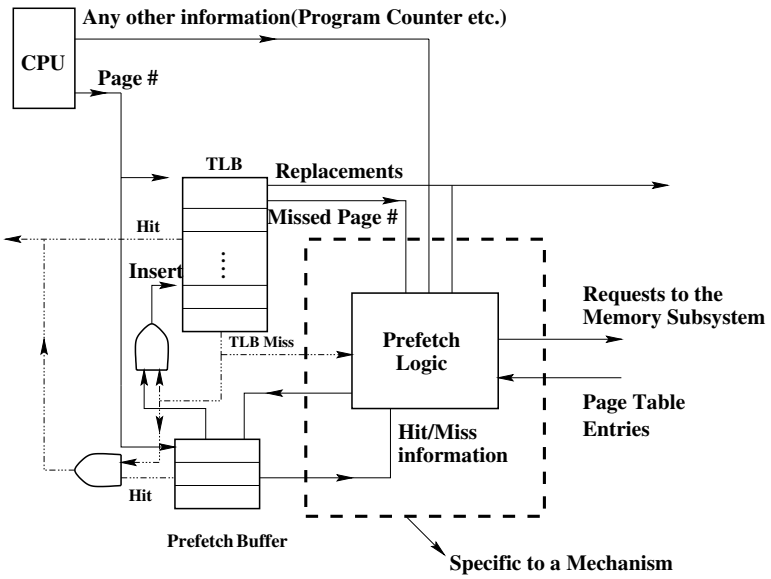


Fig. 2.7. Schematic of Hardware for Prefetching in all the Considered Mechanisms

from the application. Prefetching can thus not increase the miss rates of the original TLB. There is, however, the issue of additional memory traffic that is induced by prefetching, and that is discussed later on. In addition to a brief description of each mechanism, we also give a block diagram showing the hardware needs/functioning of each mechanism. In ASP, MP and DP, the prefetching engine uses a prediction table that has a given number of rows (r). MP and DP allow aggressive predictions, and each row of the table can have s slots. In ASP, each row contains only one slot as defined in [42] since this mechanism makes at most one prediction on a given reference. The indexing of the rows and what goes into each slot is specific to a scheme. The slots essentially determine what entries to prefetch, and thus s puts a bound on number of entries that can be prefetched on a given miss. The prefetch buffer size b is the same across all the mechanisms. A schematic of the overall prefetching hardware implementation is given in Figure 2.7.

2.5.1 Sequential Prefetching (SP)

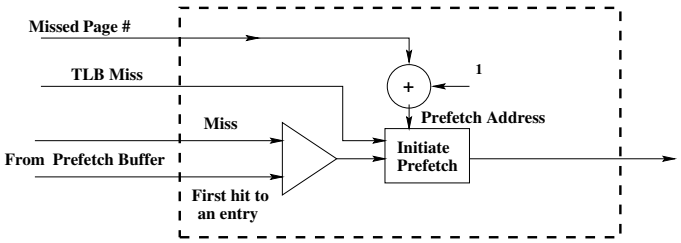


Fig. 2.8. Schematic of Hardware for SP

This mechanism tries to exploit the sequentiality of references, and prefetches the next sequential unit (page table entry) based on the current reference. Several variations have been proposed, that are discussed by Vanderwiel and Lilja [108]. They point out that of all the schemes, *tagged* sequential prefetching - where a prefetch is initiated on every demand fetch and on every first hit to a prefetched unit, is very effective. Another variation proposed by Dahlgren and Stenstrom [48] dynamically varies the number of units to prefetch based on the success rate. However, simulations have shown only slight differences between these schemes [108, 48]. Consequently, we limit ourselves to the tagged version of SP in this thesis. On a TLB miss, if the translation also misses in the prefetch buffer, it is demand fetched and a prefetch is initiated for the next virtual page translation (stride = 1) from the page table. The CPU resumes as soon as the demand page translation arrives. In case of a prefetch buffer hit, CPU is given back the translation (and resumes), the entry is moved to the TLB, and a prefetch is initiated for the next translation in the background. A simplified hardware block diagram implementing SP is given in Figure 2.8.

2.5.2 Arbitrary Stride Prefetching (ASP)

SP captures only spatial proximity, but there are several applications that have regular strided reference patterns. Prefetching mechanisms to address this have been proposed by Chen and Baer [42], Patel and Fu [57] and several others. It has been pointed out [108] that the scheme proposed by Chen and Baer is the most aggressive of these. We use this scheme, referred to as Arbitrary Stride Prefetching (ASP) in this thesis, for comparisons. ASP uses the program counter (PC) to index a table (referred to in [42] as Reference Prediction Table (RPT)). Each row has one slot which stores a tuple containing (i) the address that was referenced the last time the PC

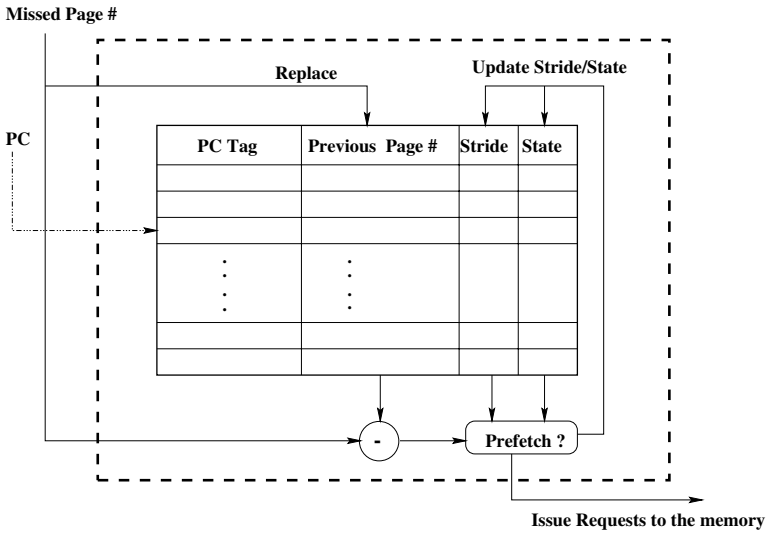


Fig. 2.9. Schematic of Hardware for ASP

came to this instruction, (ii) the corresponding stride, and (iii) a state (PC tag may also need to be maintained for indexing). The address field needs to be updated each time the PC comes to this instruction, and the prefetch is initiated only when there is no change in the stride for more than two references by that instruction (the state is used to keep track of this information). Such a safeguard tries to avoid spurious changes in strides. This is the mechanism that is evaluated in this thesis, though there are several variations proposed [42]. A simplified hardware block diagram implementing ASP is given in Figure 2.9.

2.5.3 Markov Prefetching (MP)

The previous two are representative of schemes that attempt to detect regularity of accesses (by observing sequentiality or strides), and fail if there is no such regularity in the differences between successive address references. However, it is possible that history repeats itself,

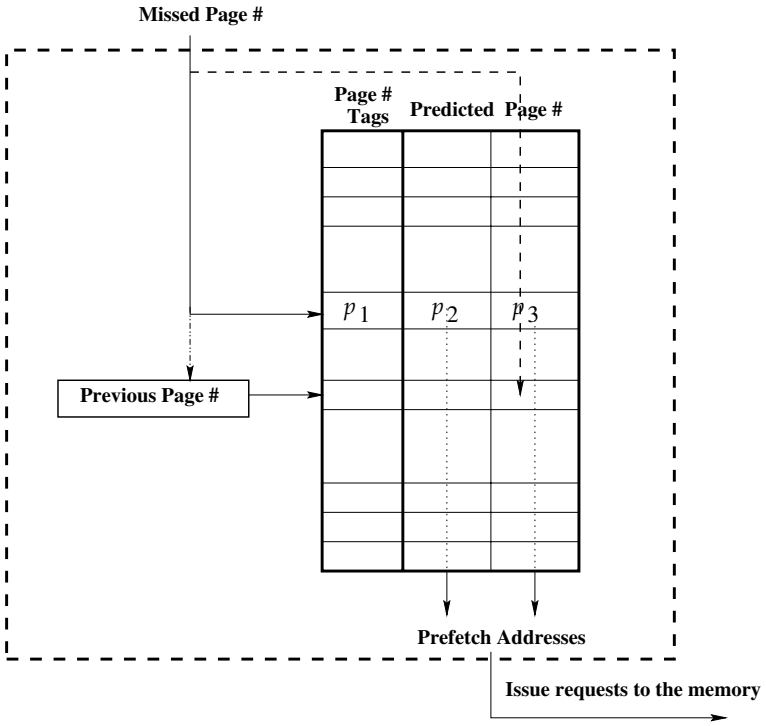


Fig. 2.10. Schematic of Hardware for MP with $s = 2$.

even without any regularity in strides, and MP tries to address that angle. MP attempts to dynamically build a Markov state transition diagram with states denoting the referenced unit (pages in this context) and transition arcs denoting probability of needing the next page table entry when the current page is accessed. The probabilities are tracked from prior references to that unit, and a table is used to approximate this state diagram. This scheme was initially proposed for caches [68], and we have extended this to work with TLBs as discussed below.

The prediction table for MP is indexed by the virtual page address that misses. Each row of the table has s slots, with each slot containing a virtual page address that is initially empty (they correspond to entries to be prefetched when this address misses the next time). On a miss, this table is indexed based on the address that misses. If not found, then this entry is added, and the s slots for this entry are kept empty. In addition, we also go to the entry of the previous page that missed, and add the current miss address into one of its s slots (whichever is free). If all the slots are occupied, then we evict one based on LRU policy. As a result, the s slots for each entry correspond to different virtual pages that also missed immediately after this page. If a missed address hits in the table, then a prefetch is initiated for the corresponding s slots of this address. Since the table has limited entries, an entry (row) could itself be replaced because of conflicts. A simplified hardware block diagram of MP is given in Figure 2.10.

2.5.4 Recency Based Prefetching (RP)

While all the previous mechanisms have been proposed for caches, Recency Prefetching is the first mechanism, to our knowledge, that has been proposed solely for TLBs. This mechanism works on the principle that pages referenced at around the same time in the past will also be referenced at around the same time in the future. It builds an LRU stack of page table entries

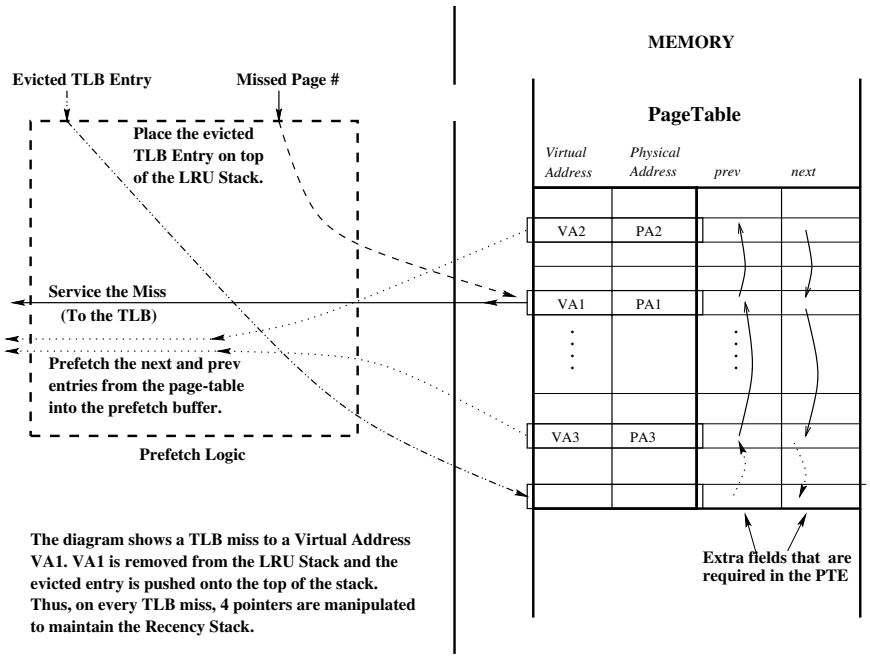


Fig. 2.11. Schematic of Hardware for RP

to achieve this. Specifically, when an entry is evicted from the TLB it is put on top of the stack, its next pointer is set to the previous entry that was evicted (whose previous pointer is set to this entry). As a result, each entry has two pointers, which are actually stored in the page table (space is thus one of the issues with this mechanism, almost tripling the page table size). When an entry is loaded on a miss, the prefetch mechanism fetches the entries corresponding to the next and previous pointers into the prefetch buffer in the hope that they will be needed as well (this is the mechanism that is implemented and evaluated here, though there is a variation in [97] with regard to prefetching some more entries). RP, thus, keeps its prediction information in the page table itself (in memory) and does not have additional storage costs on-chip. This comes at the cost of a significant increase in page table size. Further details can be found in [97] and a hardware schematic of this mechanism is given in Figure 2.11.

2.5.5 Distance Prefetching (DP)

This is an entirely new mechanism that we propose, and could be used at any level of the memory hierarchy (i.e. TLBs, caches or maybe even I/O). In this thesis, we illustrate its benefit for TLBs.

The advantage with SP and ASP is that they take very little space to detect patterns and initiate actions accordingly, while MP and RP can take considerably more space because they can detect more patterns than the restricted patterns that SP and ASP can detect. They also take a while to learn a pattern, since only repetitions in addresses can effect a prefetch for RP and MP (not first time references). Our DP mechanism can be viewed as trying to detect many of the patterns that RP or MP can accommodate (and maybe some that even they cannot), while benefiting from the regularity/strided behavior of an execution. In fact, if there is so much

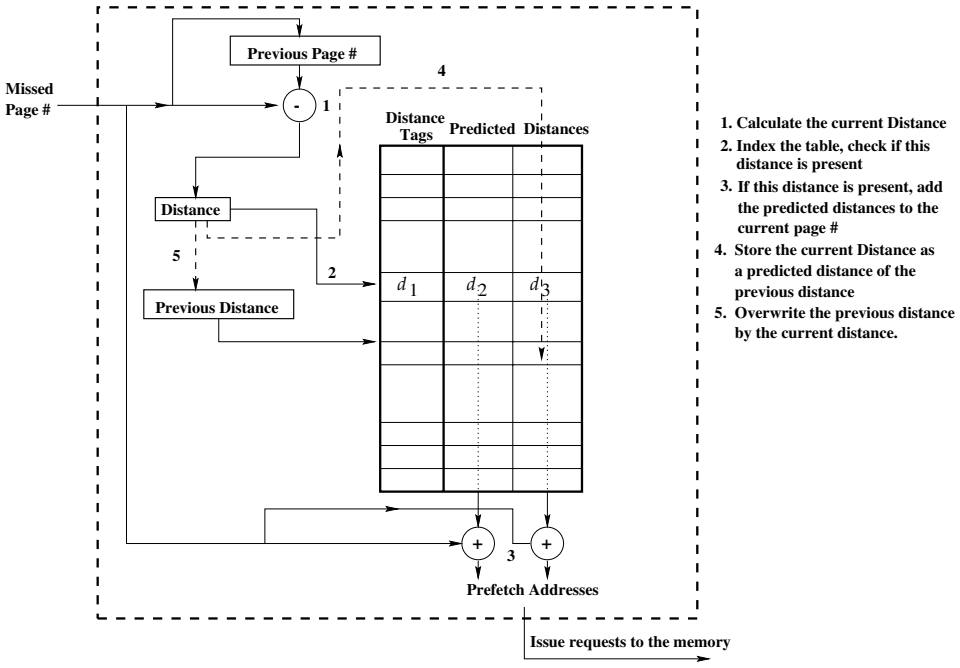


Fig. 2.12. Schematic of Hardware for DP with $s = 2$.

regularity that SP and ASP can do very well in a reference pattern, then DP should automatically take only as much space as these two. Remember that MP and RP need considerable space even to capture sequential scans while SP and ASP can do this in little space.

DP works on the hypothesis that if we could keep track of differences between successive addresses (spatial separation or stride, which we call as *distance* for this mechanism) then we could make more predictions in a smaller space. For instance, let us say that the reference string is 1, 2, 4, 5, 7, and 8. Then, if we just keep track of the fact that a distance of "1" is followed by a (predicted) distance of "2" and vice versa, then we would need only a 2 entry table to make a prediction as opposed to the markov mechanism where an entry is needed for each page (6 entries in this example). This is exactly what our distance prefetching mechanism does. A reference string touching all pages sequentially (that SP optimizes) can be captured by DP using an entry saying distance of "1" is followed by a (predicted) distance of "1".

The hardware implementation (as is shown in Figure 2.12) for DP requires that the table be indexed by the current distance (difference of current address and previous address). Each entry has a certain number of slots (maintained in LRU order) corresponding to the next few distances that are likely to miss when the current distance is encountered (similar to how MP keeps the next few addresses based on the current address). Pages corresponding to the distances in these slots are prefetched when this virtual address misses. One could, perhaps, envision indexing this table using the PC value together with the distance, or using a set of consecutive distances. These are issues that could be investigated in future research, and are not discussed in this thesis.

	ASP	**MP**	**RP**	**DP**
How many rows?	r	r	No. of PTEs	r
What are the contents of a row ?	PC Tag, Page #, Stride and State	Page # Tag 2 Prediction Page #s	*next, prev* pointers	Distance Tag, 2 Prediction Distances
Where is the Table maintained ?	On-Chip	On-Chip	In Memory	On-Chip
How is the table indexed ?	PC	Page #	Page #	Distance
How many memory operations are necessary per miss ? (*excluding prefetching*)	0	0	4 (at least)	0
How many prefetches can be initiated ?	1	2	1-3	2

Table 2.5. Comparing the Hardware Issues of the Schemes at a glance. s is assumed to be 2 for MP and DP. PC Tag, Page # Tag, and Distance Tag for ASP, MP and DP respectively are needed for tag comparison when indexing/looking up the table.

2.5.6 Review of Hardware Requirements

Table 2.5 gives a quick review of the above description by comparing the schemes in terms of the hardware requirements and functionality. ASP usually subsumes SP, and we do not show SP separately here or in the experimental results. For the ASP, MP and DP mechanisms, we uniformly use a parameter r to study its effect on the resulting performance as mentioned earlier. The previously proposed RP mechanism, keeps information (2 pointers) in each entry of the page table. Since the number of virtual pages is usually quite large, the space taken by RP considerably dominates over the much smaller r (32 to 1024 rows) that we consider for ASP, MP and DP. The only benefit for RP in this regard is that the storage is in main memory, while the other three require on-chip real-estate. These two pointers for RP refer to the previous and next pointers of the LRU stack. Both MP and RP, index the information based on the page number that misses in the TLB, and DP indexes using the current distance (stride). ASP, on the other

hand, indexes using the PC value. In ASP, MP and DP, the corresponding tag information (of the indexing field) needs to be maintained to ensure the corresponding match since more than one entry can map on to a single row. There is, thus, not a significant difference in storage requirements across the schemes for a single row.

ASP, MP and DP, have all the necessary information to initiate a prefetch on-chip, and thus need not incur any additional memory references. On the other hand, removing the page table entry that is currently required and pushing the evicted entry on top of LRU stack requires manipulating four pointers in RP. This can become an issue in increasing memory traffic, thus interfering not only with other prefetch actions but also with normal data traffic.

The maximum number of prefetches that can be initiated on a miss for MP and DP depend on the chosen s values. This is, typically, quite small (around 2-4) that is not only shown to be a good operating point later in this thesis, but has also been pointed out by [68] for MP. ASP, as defined in [42], prefetches the address incremented by the corresponding stride. RP prefetches entries on either side of the LRU stack upon a miss, and there is also a version discussed in [97] that prefetches three entries. It should be noted, that the number of prefetches that are initiated is not necessarily indicative of the performance of the scheme. Eventually, the prefetches are put in the (small) prefetch buffer, and a more aggressive scheme can end up evicting entries before they are used.

2.6 Qualitative Comparison of Schemes

Having given a quick overview of the five prefetching mechanisms, we next try to provide a better understanding of their pros and cons using some example reference strings. It should be noted that for the discussion in this section, we are not commending or putting down

any mechanism based on its performance with these strings. These strings are not necessarily realistic of actual application references. Further, one could argue that with a different set of parameters than what is chosen to illustrate the point, a mechanism may actually outperform the others for the given string. On the other hand, our intention here is only to get a general idea of how a scheme works, and to gain a better understanding of what kind of reference behavior is better suited for a given mechanism. Uniformly, let us assume prefetch buffer size (b) is 1 entry, MP, RP and DP fetch only 1 entry (i.e. $s = 1$ for MP and DP, and we follow only the next pointer in LRU stack for prefetching in RP and do not prefetch the previous pointer's entry). Let us also assume that the reference string is generated by the same PC value, which becomes important for ASP. The TLB is assumed to be 4 entry fully associative for the following discussion.

We consider four reference strings here, that are shown in Table 2.6 (each number denotes the virtual page number being referenced) together with the number of accurate predictions by each scheme. *Note that the miss string from the TLB would be the same as the reference strings for all of them.*

Reference Strings		**Correct Predictions**				
		SP	ASP	MP	RP	DP
Liner Scan(RS_1)	[1 2 3 4 5 6 7 8 9 10 ... 100]	99	97	0	0	97
Deterministic Iterative(RS_2)	[1 2 3 4 5 6 7 8 9 10] $\times$ 10	90	79	89	89	78
Alternating Pattern(RS_3)	[1 2 3 4 .... 20] [1 3 5 7 .. 19 2 4 6 8 .. 20] [1 2 3 4 .... 20] [1 3 5 7 .. 19 2 4 6 8 .. 20] [1 2 3 4 .... 20]	57	81	4	23	80
Alternating Stride(RS_4)	[1 3 4 6 7 9 10 12 13 15] $\times$ 10	40	0	89	89	77

Table 2.6. Example Reference Strings and No. of Correct Predictions for Each Mechanism.

RS_1 corresponds to a sequential access of 100 contiguous virtual pages in an application, that is perhaps representative of the miss behavior in the initialization phase of an application (incurring cold misses). SP will fetch all but the first page, thus predicting the next 99 page references correctly. ASP [3] needs 3 references before it can stabilize (it needs to observe two constant strides before making a prediction), and will then predict the rest (97) correctly. MP and RP will not have any prior history for each referenced page, and will thus need all 100 references to get warmed up, and will not be able to make any predictions in this case. DP will need two distances (3 references) to warm up (and create an entry in the table), and will be able to accurately predict all other references (97).

RS_2 shows 'deterministic iterative' behavior described in [97]. Pages 1-10 are traversed 10 times (represented by $\times$ 10 in Table 2.6). A repeated scan through an array could produce such a miss string. As before, SP will start predicting correctly after the first page for each scan of the array. There will be 10 mispredictions in all (each time when we come to the end of the array), making 90 accurate predictions totally. ASP will need the first 3 page references to warm up in the first traversal. In the next traversal, note that ASP does not update its stride to -9 when it encounters page 1 (since it needs at-least two mispredictions to update the stride), and will not predict only the first two references. Consequently, we have 79 (i.e. 7 + 9*8) references predicted correctly by ASP in all. RP will not be able to predict any of the first 11 references since the LRU stack needs to build up, and will then accurately predict all the rest (since history repeats itself exactly). Similarly, an MP with a 10-entry Markov table can capture all the 89 subsequent references after the first 11 references required to build the state transition diagram. DP will take 3 references to warm up the table with a prediction distance of 1 for a

[3] ASP is assumed to be in steady state with some other stride to begin with.

current distance of 1. The subsequent 7 references will then be correctly predicted for the first traversal. The subsequent page 1 reference is not going to be predicted, nor are the following two references (the predictions for distances of -9 and 1 are not going to be set right for those). Consequently, only 7 references in the second traversal will be predicted correctly. In each of the subsequent 8 traversals, 8 references will be predicted correctly (since the prediction for a distance of -9 will be correctly set to 1). DP thus gives a total of 78 correct predictions for this string.

In RS_3, we have chosen a string that has five substrings obtained by repeating two patterns (each of length 20) alternately. SP will predict the latter 19 references of the first sequential pattern whenever it occurs, and will thus make a total of 57 correct predictions. ASP will not be able to predict the first 3 references of each occurrence of a pattern. In addition, ASP will not be able to predict the transition from 19 to 2 and the subsequent 2 to 4 (it will however predict the following 4 to 6). Thus, ASP will give a total of 81 correct predictions. MP will not be able to predict any of the references correctly except the transitions from 20 to 1 after the first such occurrence. So there will only be 3 correct predictions for MP. RP analysis is a little more complicated, and we do not cover all the details here. It is best understood by drawing an LRU stack, pulling out entries (letting the entries above fall down) and putting it on top. If we do such an analysis, one would get 23 correct predictions for RP (an intuitive reason why it does better than MP here is that when for instance entry for page 3 is pulled out of the stack, the connection is automatically made between pages 2 and 4 in the stack by this algorithm, thus helping some of the patterns). Finally with DP, the incorrect predictions are in the first 3 references of each

substring. In addition, it will not predict the 11th, 12th and 13th references of the second substring, and the 11th and 13th references of the fourth substring. Overall, DP will thus predict 80 references correctly.

RS_4 is a repetition of a string of alternating strides (2 and 1). SP will predict only the strides of 1, and will thus have 4 correct predictions in each repetition (40 overall). ASP does not do even as well as SP, since it needs to observe two successive references of the same stride, which does not happen in this string. Consequently, it will not be able to make any correct predictions. MP and RP will build the history information for the first 11 references, and the subsequent 89 references will be predicted accurately. DP will not predict the first 4 references of the first repetition, By then, the entries in the table would indicate - a distance of "2" is followed by a distance of "1", and a distance of "1" is followed by a distance of "2". All the references until page 15 would then be predicted correctly. At this time, page 1 will not be predicted nor will page 3 or page 4 since a distance of "-14" is not set and the distance of "2" will now have a prediction of a distance of "-14". In the subsequent repetitions, pages 1 and 4 will still not be predicted but page 3 will be since the prediction for distance "-14" is set correctly. Consequently, DP will totally predict 77 (4 + 3 + 2*8 incorrect predictions) references correctly.

Discussion: As mentioned earlier, the above exercise is only intended to understand the suitability of the prefetching mechanisms for different patterns in reference strings. It is indeed possible for a mechanism to do better than what is mentioned above in a more realistic implementation of the scheme (prefetching more than one entry, or using PC information to differentiate the strides at different points in the program). Still, this exercise has given us good insight to make some overall remarks:

- If there is good sequentiality in the accesses (constant strides such as RS_1 and RS_2), SP and ASP are able to quickly detect this and do a good job of prediction, compared to MP or RP. The latter two need the history to build up at an address granularity (not at a stride granularity), and are thus not very good at predictions for first time references to a page. DP, on the other hand, does as well as SP and ASP in quickly making use of such constant strides even for (cold) first time references. The storage that it needs in these cases to make predictions is also comparable to that for SP and ASP (and much lower than that needed for MP or RP).

- RS_4 represents very rapidly changing strides (in fact, alternating). In such cases, schemes that stabilize on stride non-variability such as SP or ASP are not as good as the history-based schemes (MP and RP) that can keep a lot more information about prior accesses. In this case, we find that DP automatically trends towards the better prediction behavior of MP and RP.

- RS_3 falls between these extremes where there are some periods of constant stride, with the stride length changing after some duration. Further, in this string, these changes are effected at the same addresses as the previous duration. In such cases, history based schemes (MP and RP) falter, and SP is not good either. ASP is able to detect and adapt to such changes. Even for this string, we find DP making as accurate predictions as the best of the other schemes.

In these examples, the miss string turns out to be the same as the reference string because of the TLB size, and thus there is no loss/filtering of information that is available to the mechanisms. In such cases, we find that RP which can use the TLB implicitly to extract LRU

information based on the replacements, does not get significant gains over the others (except in RS_3 where it gains a little over MP because of this feature). We would like to mention that we have conducted similar exercises with more complicated strings and with strings from real applications with regular reference behavior (such as FFT and Matrix Multiplication), where the miss string can be quite different from the reference string, and we found similar trends.

An indepth study of application behavior is needed with more realistic TLB configurations and prefetching mechanism implementations, for a better comparison of the pros and cons, which is undertaken in the rest of this thesis.

2.7 Performance Evaluation

2.7.1 Experimental Setup

We have conducted an extensive evaluation of the prefetching mechanisms for a wide variety of applications spanning several benchmark suites. Our evaluations use all 26 applications from SPEC CPU2000 [45], 20 applications from MediaBench [76], 5 applications (`bcc`, `mpegply`, `msvc`, `perl4`, and `winword`) from the Etch traces [107] and 5 applications (`anagram`, `bc`, `ft`, `ks` and `yacr2`) from the Pointer Intensive Benchmark suite [102]. In all, we have considered 56 applications that we hope are representative enough of realistic scenarios. The MediaBench applications are characteristic of those in embedded and media processing systems, and the Etch applications are characteristic of desktop/PC applications. The Pointer Intensive suite helps us evaluate the mechanisms for non-array based reference behavior, which can be more irregular. The SPEC 2000 applications are really long running codes and it is extremely difficult to simulate all of them completely, as has been pointed out by others [37, 87]. To our knowledge,

only a recent cache study [37] quantifying miss rates has been done to completion using simulation. In this thesis, we fast forward (skip) the first two billion instructions of their execution, and present results for the subsequent one billion instructions. The simulations have been conducted using SimpleScalar [34], using the default configuration parameters (4 way issue). Most of the simulations are conducted using sim-cache since we are mainly interested in the memory system references, and the prediction accuracies of the schemes. We also present one set of execution cycle results for one billion instructions with five of the applications with high TLB miss rates to compare DP and RP using sim-outorder (as can be imagined, these experiments take an excessively long time). The MediaBench, Etch and Pointer Intensive suite were simulated using Shade [44]. Though it is also important to consider the effect of the OS, the evaluations are only for application behavior in these results as in the earlier study [97].

We consider different TLB configurations - 64, 128 and 256 entries that are 2-way, 4-way and fully associative, and different values for prefetch buffer size (16, 32 and 64 entries). We have also varied the s and r values for the prediction table configurations of the mechanisms. We present representative results using 128 entry fully associative TLB and 16 entry prefetch buffer. We, however, present the impact of these parameters for our DP mechanism in isolation.

2.7.2 Comparing the Schemes

In our first set of evaluations, we compare RP, MP, DP and ASP (compared qualitatively until now) with the 56 workloads in Figures 2.13 and 2.14. Since MP, DP and ASP predictions depend largely on the size of the prediction table that is allowed, we have varied r (the number of entries) as 32, 64, 128, 256, 512 and 1024. Further, we have allowed the corresponding tables to be indexed as direct-mapped (D), set associative (2 and 4 way) and fully associative

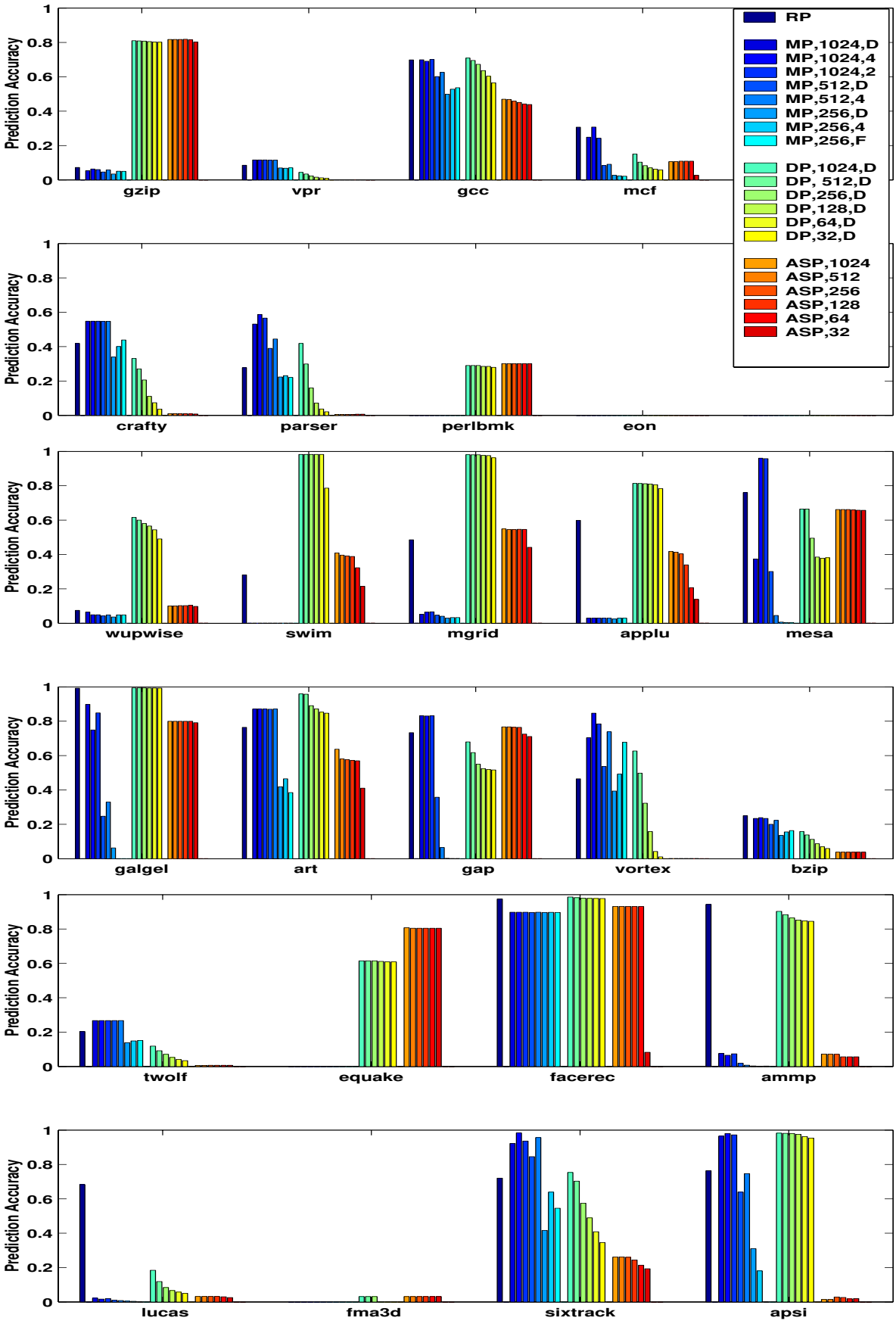


Fig. 2.13. Prediction Accuracy of different Prefetching mechanisms for all the SPEC CPU2000 Applications

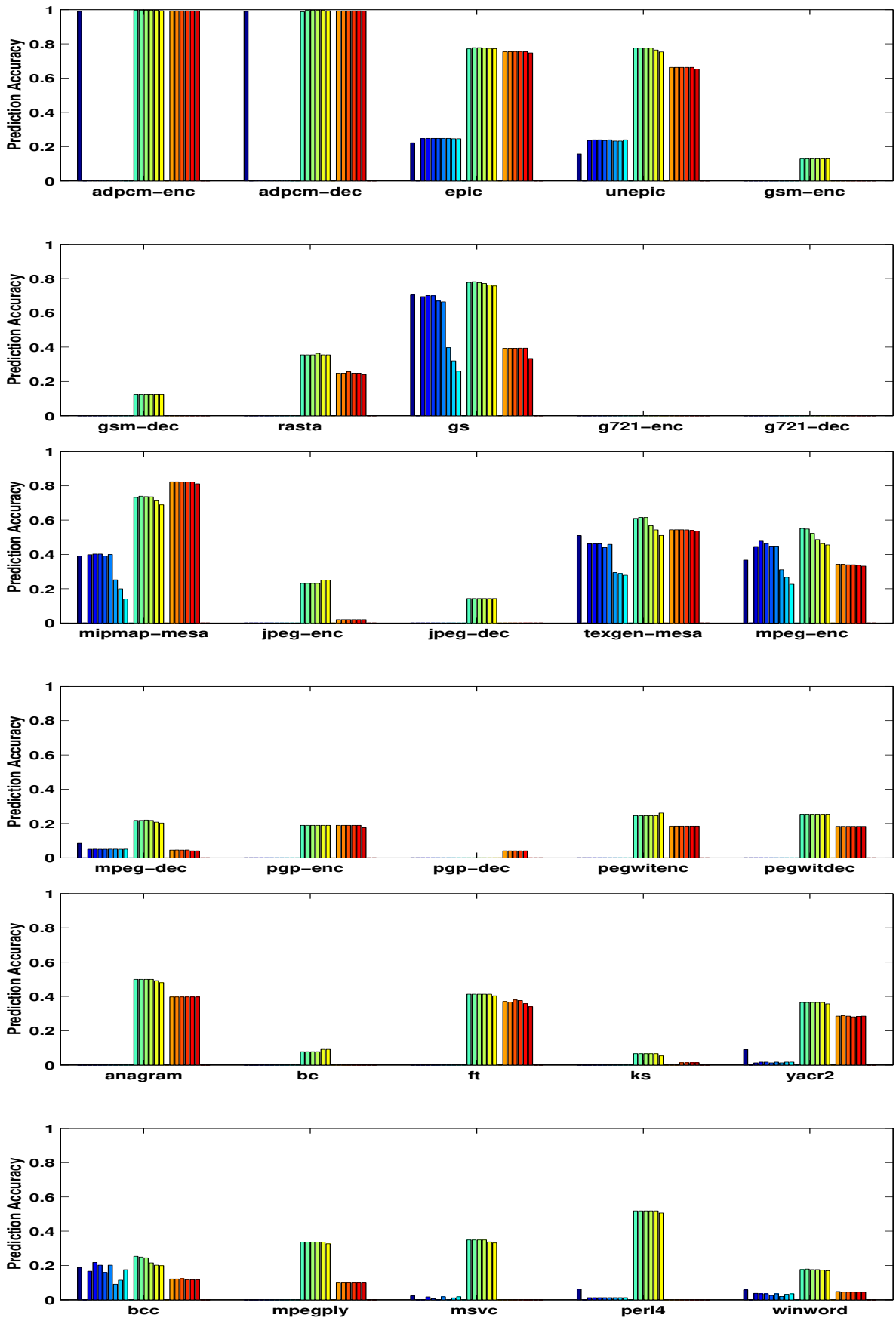


Fig. 2.14. Prediction Accuracy of different Prefetching mechanisms for Mediabench, Etch and Pointer Intensive benchmark Suites. Legends are same as in Figure 2.13.

(F). Since the graph becomes very difficult to read, we show results for DP and ASP only with direct-mapped (D) configurations. We show F, 2 and 4 way associativity influence only for MP. We would like to point out that the indexing mechanism for the prediction table (F, 2 or 4 way) has very little influence on the prediction accuracy in most cases (as one would infer from the bars for MP, and in the bars for DP later in section 2.7.3.1). In these graphs, the left-most bar for each application is for RP, following which is a gap and then the bars for MP, a gap, bars for DP, a gap and finally the bars for ASP. In some cases, the bars are either completely or partially absent because the prediction accuracy is close to 0.

These mechanisms are compared in terms of their prediction accuracy, which has been the metric used in earlier research [97] to argue the capabilities and potential of TLB prefetching. Prediction accuracy is defined as the percentage of TLB misses that hit in the prefetch buffer at the time of the reference. Accuracy is an important concern since it has a direct bearing on the amount of stall time incurred by the CPU during a TLB miss. Uniformly, a prefetch buffer of size $b = 16$ entries is used in all these experiments. Remember, that *a mechanism which fetches more aggressively can evict entries from this buffer before they are actually used for the translation* (and will consequently have an effect on the prefetch accuracy). One can in fact observe this effect with ASP, when the prediction accuracy decreases for a more aggressive $r = 1024$ entry table (compared to smaller prediction table sizes) in some applications like `apsi`, `ft` and `wupwise`.

There are applications such as `facerec`, `galgel`, `art`, `gap`, and `mesa` where nearly all mechanisms give quite good prediction accuracies. In these applications, there are regular strided accesses that repeatedly go over the items already accessed in the same regular fashion. Consequently, both stride-based predictions (ASP) and history-based predictions (RP and MP)

do a fairly good job of predicting the future. The only exception is that in some cases (such as `galgel`, `art`, `mesa`) MP performs poorly with small r. Since these are quite large datasets, keeping the history for all the references needs considerably more space, and small tables are not adequate for this purpose. RP, on the other hand, builds the history in memory and is not limited by on-chip storage as in MP. We find that our DP mechanism gives good prediction accuracies, being able to capture the strided patterns, without requiring the higher space requirements of MP to maintain history. Even a $r = 32$ predictor table for DP, gives very good predictions. In the following discussion, we go over each mechanism pointing out where it does the best and when it does not do as well.

Apart from the above five where all mechanisms give good performance, we find RP giving the best, or close to the best performance for applications such as `gcc`, `crafty`, `ammp`, `lucas`, `sixtrack`, `apsi`, `adpcm-enc/dec`, `gs`, and `texgen`. These applications have good repetition of history, i.e. the next reference after a given address is very likely to remain the same the next time we come to this address again. RP does a very good job of capturing this pattern.

MP gives the best or close to best performance for many of the applications that RP does very well. However, as was pointed out a little earlier, sometimes the history information that needs to be maintained can get quite long, and this can lead to poor predictions for small tables (such as $s = 32$). In some applications, where past history is a good indication of the future (i.e. RP does very well) such as in `adpcm-enc/dec`, MP performs very poorly for this very reason. RP is able to track history for all addresses since it keeps the information in memory, but MP does not have that luxury and may have to keep evicting its table entries from the on-chip storage. There are some applications such as `parser` and `vortex` where MP does better than

even RP despite this downside. The possible reason is that RP can look at only what happened at this address the previous time the program came to it, while MP can possibly keep track of what happened the last few times (depending on the s value of its table). In these applications, it is possible that there is alternation (i.e. a sequence such as 1, 2, 3, 4, 1, 5, 2, 6, 3, 7, 4, 8, 1, 2, 3, 4, ... would do better with MP than RP for $s = 2$) in history that is leading to this behavior (this is also the reason ASP does not do well for these applications).

ASP does very well in many of the applications that are suited to RP and MP such as `facerec`, `galgel`, `art`, `gap` and `mesa`, and also in some where RP does better than MP (`adpcm-enc/dec` and `texgen`). The regularity in strides in these applications help this mechanism provide good accuracy. This regularity also helps ASP capture many of the first time reference predictions that history based mechanisms are not very well suited to, as in `gzip`, `perlbmk`, `equake`, `epic/unepic`, `mipmap`, `pgp-enc/dec`, `anagram`, and `yacr2`. The working sets are much smaller in some of the non-SPEC 2000 applications, and cold misses do become prominent for these. On the other hand, there are applications such as `crafty` and `parser` where the accesses are not strided enough for ASP to perform well, but historical indications can give a much better perspective of future behavior for RP and MP.

Moving on to DP, we find that it gives very good prediction accuracies in several cases. DP comes very close to RP or MP in several applications where history-based predictions do the best such as `gcc`, `mesa`, `galgel`, `gap`, `parser`, and `ammp`. On the other hand, if history is not a good indication (or has not established) but strides are more determining (as in `gzip`, `adpcm-enc/dec`, `mipmap`, and `perlbmk` where ASP does very well), DP is able to deliver as good accuracies as ASP. Beyond coming close to the better of history or stride based schemes, there are several applications such as `wupwise`, `swim`, `mgrid`, `applu`, `mpeg-dec`,

bc, mpegply, msvc, and perl4 where DP does much better than the others. In fact, for gsm-enc/dec, jpeg-enc/dec, ks, msvc and bc, DP is the only mechanism which makes any noticeable predictions (even if the accuracy does not exceed 20%).

We would like to point out, that there are a few applications such as eon, fma3d, g721-enc/dec and pgp-dec where none of the mechanisms are able to make any significant predictions. Many of these applications (eon, g721-enc/dec, pgp-dec, bc, ks) have so few TLB misses that a significant history does not build up nor does a strided pattern (and TLB prefetching is not as important for them anyway). In fma3d, the irregularity makes it very difficult for any mechanism to do well, and this motivates the need for further future work on prefetching mechanism.

Prefetching Scheme	**Average** $(\Sigma\ p_i)/n(=56)$	**Weighted Average** $\Sigma\ (m_i \times p_i)/(\Sigma\ m_i)$
DP	0.43	0.82
RP	0.29	0.86
ASP	0.28	0.73
MP	0.11	0.04

Table 2.7. Table showing the average and weighted average of prediction accuracy for the prefetching schemes which was calculated using the miss rates(m_i) and prediction accuracies(p_i) over all the 56 applications. $s = 2$ and $r = 256$ for DP, MP and ASP.

In summary, we would like to point out that DP gives very good predictions for many of the applications. In fact, it provides the best or within 10% of the best prediction accuracy in 39 (and best in 36) of the 56 applications considered (the others are less than half this number). DP does well for regular and irregular applications, and applications that have strided and/or

history-based access patterns. Another important point to note is that *DP can provide such good predictions with just a(s =)32-256 entry prediction table*, compared to the others (MP and ASP) which may need many more entries, nor requiring the considerable storage and memory bandwidth taken by RP. Examining only the miss stream from the TLB, and not the actual reference stream (which to a certain extent can be viewed as a case in favor of RP because there is an implicit LRU tracking within the TLB) does not seem to penalize DP in any significant way.

DP also turns out to be the best in terms of the average prediction accuracy that was calculated over all the benchmarks(($\Sigma\ p_i$)/n) for each scheme. From the second column in Table 2.7, we can see that DP and RP take the first and second places respectively. One could argue, that it is important to not just provide good accuracies for all applications, but to those where it really matters (i.e. the higher TLB miss rate incurring applications). To capture this effect we present the weighted average ($\Sigma\ (m_i \times p_i)$/($\Sigma\ m_i$)) of the prediction accuracy (i.e. the accuracy p_i for each benchmark is weighted by the corresponding TLB miss rate m_i) for the schemes in the third column of Table 2.7. As we can see, RP comes out a little in front (around 5% better) of DP in this case because a long history helps a select set of applications with very high miss rates (even though DP does better in a majority of applications). However, this comes at a higher storage cost in memory, as well as the higher memory traffic. Consequently, the rest of this subsection gets into greater detail comparing DP with RP, in terms of performance implications of these prediction accuracies, particularly for the applications with higher TLB miss rates.

Comparing DP with RP in greater Detail: Having compared the prediction capability of the mechanisms using all the applications and all the different configurations, we specifically focus on 8 applications (`galgel`, `adpcm-encoder`, `ammp`, `mcf`, `vpr`, `twolf`, `lucas`, `apsi`)

which have the highest TLB miss rates (0.228, 0.192, 0.0113, 0.090, 0.016, 0.013, 0.016, and 0.018 respectively) for a 128 entry fully associative TLB amongst all these applications. Of these 8 chosen applications, RP provides better accuracy than DP for 5 applications - `vpr`, `mcf`, `twolf`, `ammp` and `lucas`. Further, RP is the only other prefetching mechanism explored for TLBs, and we would like to show some of the trade-offs that DP provides over RP despite slightly lower prediction accuracies in these 5 applications (which is what tilted the balance in favor of RP in Table 2.7).

RP requires as many as 6 possible memory references upon a TLB miss. While the CPU resumes computation as soon as the miss is serviced, there are other memory references needed to maintain the LRU stack. If the item was in the middle of the stack, then it needs to be removed (taking 2 references), and the evicted item needs to be put on top (taking 2 references). After this, the actual prefetching can proceed (since it prefetches on either side of the removed item, this takes 2 more references). On the other hand, DP references memory only to bring in the s (which is 2 here) predicted entries, i.e. DP does not need to update any state information in memory similar to MP or ASP.

To briefly study the impact of the additional memory traffic imposed by RP and DP, we conduct a simple experiment using SimpleScalar, wherein we use its memory system model to account for the overheads associated with the prefetch operations. It should be noted that in this examination, the prefetch memory traffic does not contend with the normal data traffic, but only with other prefetch traffic (this in fact, *is a more biased model that favors RP over DP*). These applications are run using sim-outorder (with a 4 issue width) to account for actual CPU cycles. When the CPU incurs a TLB miss, and does not find the data in the prefetch buffer, but the prefetch for that entry has already been issued, it is made to stall until the entry arrives. Further,

if a prefetch needs to be issued on a TLB miss, this memory loading operation will be impacted by any prior issued prefetch memory transactions (such as the pointer manipulations for RP, or the actual prefetching of entries for DP and RP). One other issue where we give the benefit of doubt for RP in its implementation is that, if there is a TLB miss soon after the previous one (and not for the same entry) and the prefetching initiated earlier is not complete, we only wait for the LRU stack to get updated and do not prefetch those items at that time (this is as though there was a wrong prediction, but we are not going to incur the corresponding memory traffic in fetching the nearby entries at that time). In this case, there would be only 4 memory transactions instead of 6.

We present the results from this experiment in terms of both the number of additional memory transactions (traffic) beyond that taken in servicing misses for the two schemes (normalized with respect to that taken by DP in Figure 2.15), as well as in terms of the cycles taken for execution of the programs (the billion instructions considered) that is shown in Table 2.8. The results are presented for the five benchmarks where RP has better accuracy over DP.

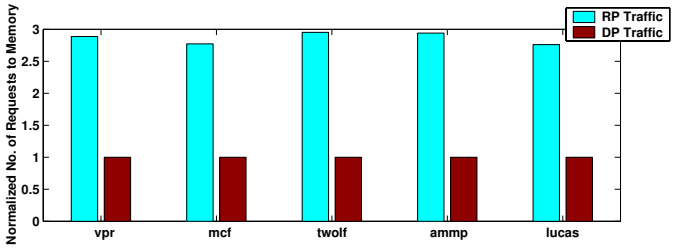


Fig. 2.15. Normalized Memory Traffic (in terms of requests) generated by RP and DP with respect to DP. In each application, the left bar is the traffic for RP, and the right bar is for DP. The bar for DP is with direct-mapped table of $r = 256$ and $s = 2$.

Application	(1 billion)		(100 million)	
	RP	DP	RP	DP
vpr	0.995	0.989	0.99	0.98
mcf	1.09	0.984	1.09	0.95
twolf	0.98	0.985	0.98	0.99
ammp	0.968	0.866	0.97	0.86
lucas	1.002	0.99	1.002	0.99
galgel	0.16	1.89	0.16	1.90
apsi	1.08	0.93	1.15	0.88

Table 2.8. Comparing DP with RP: Normalized execution cycles(w.r.t. no prefetching) for RP and DP for 1 billion instructions after the first 2 billion instructions. $s = 2$ and $r = 256$ for DP.

We find that despite the slightly higher prediction accuracy that RP provides for these applications, DP still comes out in front when considering execution cycles. This is because RP generates substantially larger volume of memory traffic as can be seen in Figure 2.15 ranging from anywhere between 2-3 times that for DP with $r = 256$. As was pointed out, DP gives fairly good predictions even with $r = 32$ which incurs even lower traffic. It should be remembered that in this simulation, we are in fact more biased towards RP, since the prefetch traffic does not interfere with the normal data traffic, and consequently a more realistic model would favor DP further.

2.7.3 Fine-tuning Distance Prefetching

Having demonstrated the potential of DP over the other schemes, we focus on this mechanism in the rest of this thesis and the eight applications with the highest TLB miss rates.

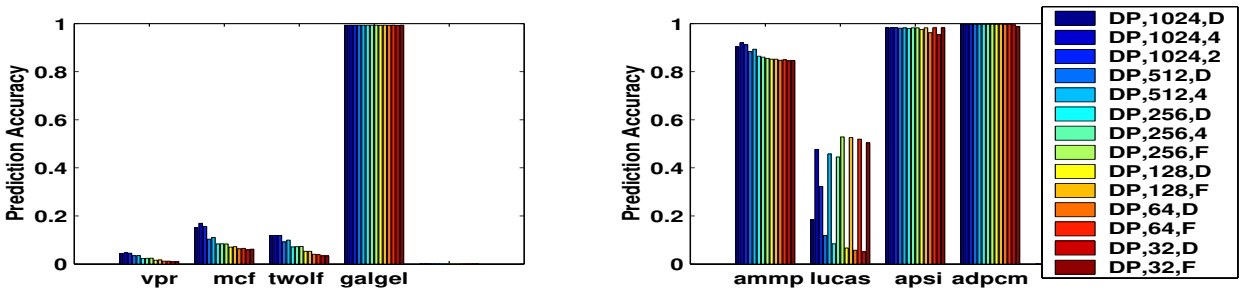


Fig. 2.16. Influence of Prediction Table Parameters on DP accuracy

2.7.3.1 Number of Rows and Associativity of the Prediction Table

Figure 2.16 shows the prediction accuracies of DP for the eight applications varying the table sizes (r =32 through 1024 entries) and associativity for indexing this table (D, 2, 4 and F). As we can observe, except for `lucas`, the prediction is not significantly affected for most table configurations. DP is able to capture the reference patterns (of the current working set) in a very small space for `galgel`, `ammp`, `apsi` and `adpcm`. In the others, there is either that much irregularity, or a lot of history needs to be maintained, that changing prediction table configurations in the ranges experimented with do not produce significant differences in prediction accuracy. Even in `lucas`, associativity is more important than size (perhaps, due to some conflicting distances). Though not explicitly shown here, we would like to mention that we find similar observations for most of the 56 applications. Consequently, we suggest using a simple direct mapped (or at most 2-way) structure for DP scheme with 32-256 entries, that can perform fast look-ups.

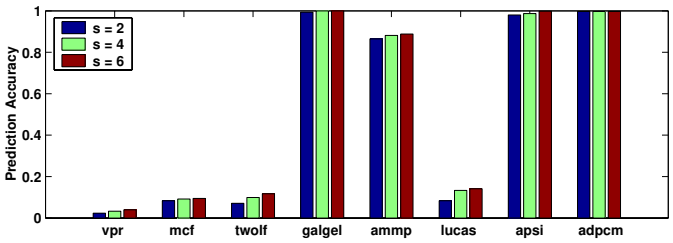


Fig. 2.17. Impact of number of slots(s) on DP accuracy($r = 256$, $b = 16$)

2.7.3.2 Number of prediction entries

Another table parameter that needs to be studied is the number of slots s (predictions) to make for each row. We consider values of 2, 4 and 6 slots in Figure 2.17, and find that the prediction accuracy is quite insensitive to the choice for s. This is because even if we are very aggressive in predicting and bringing in more entries, the prefetch buffer can hold only 16 entries here and the fetched entries may not remain long enough to be useful. Another reason why it may not have much effect is that not all the slots are being utilized (i.e. there are not too many varying strides). Increasing the number of predictions is not only having little effect on applications where we have good accuracy, but also in those where we have poor predictions (`vpr`, `mcf`, `twolf`, `lucas`). Since increasing s can increase memory traffic (and hardware costs), we suggest using $s = 2$ for the prediction table since we are not seeing significant improvements in prediction with higher values of s (a similar observation was made for MP in [68]).

2.7.3.3 Prefetch Buffer Size

As was pointed out in the previous results, the choice of prefetch buffer size can influence the performance of the prefetcher. We consequently study the effect of three different prefetch

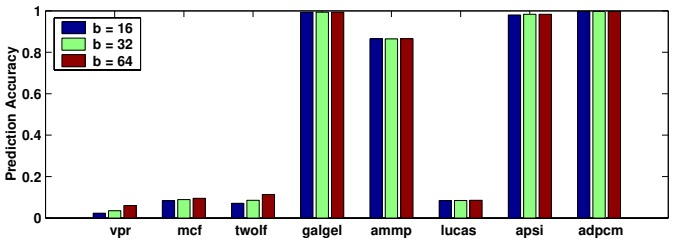


Fig. 2.18. Impact of Prefetch Buffer Size (b) on DP accuracy($r = 256$, $s = 2$)

buffer sizes ($b = 16, 32, 64$) on the prediction accuracy of DP. We find that there are not significant improvements in prediction accuracies with these considered buffer size increments. There are, of course, likely to be points in the working set where one could see large jumps in improvement at some buffer sizes. But we do not see those points for these small prefetch buffers that we consider (we want this to remain small so that it can be associatively looked up in parallel with the TLB as described in [97]). Hence, we advocate a prefetch buffer of 16 or 32 entries for DP.

2.7.3.4 Influence of TLB Size

One could argue that as the TLB gets larger, it can satisfy more of the references, thus filtering the miss string further coming out of it. It is possible that this can result in less regularity in the miss string, making prefetching that much more difficult. To study if DP is still able to perform well with such possible effects, we compare the prediction accuracies for DP with TLB configurations of 64, 128 and 256 entries that are all fully associative in Figure 2.19. We find that DP still does a good job across these TLB sizes, and the changes in accuracy are not very

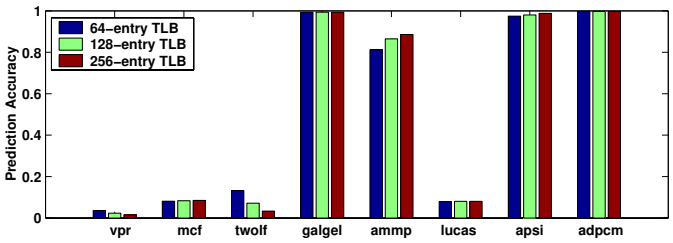


Fig. 2.19. Influence of different TLB sizes on DP Accuracy($r = 256$, $b = 16$, $s = 2$)

significant. We observe that the filtering effect can both decrease (`twolf`) and increase (`ammp`) prediction accuracy.

2.8 Concluding Remarks

Keeping current trends in mind, it is important to optimize for the TLBs. This thesis first of all filled a big hole in the literature for the TLBs - lack of a characterization study. Our thorough characterization study yields several good insights into the application characteristics based on which we develop a novel TLB prefetching technique. Our technique is simple, yet powerful, and it adapts to applications as they execute. Our prefetching mechanism is generic and can be applied to other domains as well(like caches). We applied it to the TLBs and demostrated its potential using a variety of applications.

Chapter 3

Memory Management in the runtime environment

One way to improve the performance of the processor-memory path is by using hardware mechanisms as described in the previous chapter. This can also be done in software. On the software optimizations for caches, which can also have effect on the TLBs, a lot of work has been done on compiler transformations and data reorganization with regard to enhancing locality [40, 98, 72, 52]. Increasing complexity of applications (and hence that of application code) will make compiler optimizations more difficult in future. Also, most of these optimizations target static data and arrays but not dynamic memory, which is managed by the runtime environment.

Dynamic memory allocation is a necessity for many applications today. A lot of research has gone into designing good memory allocators over past 40-50 years and a number of strategies have been proposed and implemented for this purpose [110]. If we look at dynamic memory allocators today, on one side are general-purpose memory allocators that can be used with any application and at the other extreme are application-specific custom allocators. The general purpose allocators are standard allocators that are shipped with the operating systems. They are the default allocators that work with all applications. Consequently, they are not optimized on a per-application basis. Therefore, these allocators are slow. Even a very highly tuned general-purpose allocator could take hundreds of cycles for a single allocation [54]. On the other hand, custom-allocators are written for specific applications (often, on a per-application

basis). These allocators are shipped with the application and 'understand' the application allocation/deallocation patterns. Therefore, they are well-optimized for the application and provide good performance. In general, it is not possible for a general-purpose allocator to beat the performance of a custom-allocator, for a given application.

3.1 Problems with Custom Allocators

Custom allocators may not be easy to write. Often, it may not be possible to know the application allocation/deallocation pattern. In such cases, it becomes even more difficult to write a custom allocator, especially as applications become more complex. Writing custom code to replace general purpose memory allocators may not be a good software engineering practice. If an application is written using custom code, standard memory detection and debugging tools cannot be used to for debugging purposes. Custom code needs to be replaced by calls to general-purpose memory allocators for debugging purposes in such cases since memory allocated by either of the two cannot be managed by the other. The greatest problem with custom allocators is the burden on the application developers. These developers sometimes even run the application for a few inputs to see its allocation/deallocation behavior and then write a custom allocator.

Custom allocators do have some advantages. The main advantage with a custom allocator is performance gain. Apart from this, at times, there can be other advantages. For example, region based allocators simplifies memory management in some applications [54] like `gcc` and `Apache`. At the same time there are some other disadvantages with such approaches [54] - for example, objects allocated within a region cannot be freed until the whole region is freed.

Many allocators today have been optimized for a variety of purposes. For instance, Slab Allocator [31] intends to optimize for incoming requests for a particular size. Hoard [29],

Vmem-Magazines [32] and Nakhimovsky [85] try to address the problems in multi-threaded environment. Barrett [27] tries to optimize based on predicting life-times of the requests. Vmalloc [109] proposes use of multiple regions with different policies etc. In all, custom allocators provide performance but at the cost of software engineering. General-purpose allocators are good to use but do not perform well.

Applications have their own characteristics and any application which does dynamic memory allocation would ideally require its own custom-allocator - i.e., the application behavior and the allocator behavior go hand-in-hand to give good performance. In an attempt to achieve this, traditionally, the solution has been to write custom allocators. In this thesis we take a different approach by proposing an *Adaptive Memory Allocator* that can *observe* the incoming patterns of sizes that an application allocates and tune itself to these patterns to provide better performance. The design of this memory allocator is not only driven by the idea of the adaptivity itself but also by some of the common characteristics which today's applications exhibit. In addition to designing and implementing an adaptive allocator we do a thorough performance evaluation using both applications and synthetic workloads and show where adaptivity can be of use.

We first look at the basic operations of a memory allocator followed by the issues that are important in designing such an allocator. We then look at allocation/deallocation characteristics of few applications and finally present the design and evaluation of our adaptive allocator.

3.2 Basic Operations of a Memory Allocator

This section briefly discusses the basic operations of a memory allocator. Memory allocators typically provide `malloc/free` interface to the application programs. When applications

need dynamic memory, they call the `malloc` function by passing a size argument to it. The memory allocator returns the pointer to the object of that size. Once the application is done with the memory, if frees the object by calling the `free` function and giving back the above returned pointer to the allocator. Thus, from the application view point, `malloc/free` should be of minimal overhead as this will directly impact the execution time. We now discuss various issues that need to be kept in mind when designing such a memory allocator.

3.3 Issues in designing a memory allocator

3.3.1 Issues considered in designing a traditional memory allocator

Traditionally, the following issues are important in the design of a memory allocator.

- *Low Response Time* : When a request reaches the allocator, it should be serviced as soon as possible - response time has a direct implication on the application run time.

- *Low Fragmentation* : Fragmentation is the situation where the total free space in the allocator is greater than or equal to the size of the request but the allocator still cannot satisfy the request because this space is not contiguous. It is important to keep number of such situations as low as possible because fragmentation leads to excessive searching (for free space) and can even lead to an operating system call.

- *High Utilization of Memory* : Memory Allocators typically borrow memory from the operating system (using the `sbrk()` system call) and manage that memory. Ideally, the amount of memory borrowed from the OS must be equal to the memory that is required

by the application. But this is generally not the case, as the allocator needs space for bookkeeping. Utilization is defined as the ratio of the memory required by the application to that borrowed from the OS. It is important to keep it as high as possible.

- *Locality*: If requests can be satisfied using the same cache lines and same TLB entries (without incurring additional cache and TLB misses), it would benefit applications. Locality refers to reducing the number of cache misses and TLB misses by trying to re-use them as much as possible.

The above issues are not only to be considered while designing but they also represent the metrics by which an allocator will be evaluated.

3.3.2 Next-generation memory allocator

While all of the above issues are still important in the design of a memory allocator, allocators for the future applications also need to be *adaptive*. Adaptivity tries to fill in the gap between the application and the allocator by tuning the allocator towards the *ideal allocator* for that application. Developing an adaptive allocator that can tune itself to optimize all of the above metrics is a very challenging problem. In this thesis, we focus on adaptivity which will help to keep the response time low and this is accomplished by observing incoming patterns of sizes and tuning for these. In other words, this allocator might sacrifice in optimizing other metrics like utilization or fragmentation in an attempt to keep the response time low. Response time is one important metric of a memory allocator and increasing virtual address spaces and physical memories further make it more important than other metrics like utilization or fragmentation. Even from an application viewpoint, minimizing response time will have direct impact on reducing

application runtime. Before designing such an allocator, we first of all see if there is any scope for adaptivity.

3.4 Is there any scope for adaptivity ?

It is important to see how today's applications behave, in terms of their dynamic memory allocation, to see if there is any scope for adaptivity at all. An application characterization can not only reveal us of any scope for adaptivity but also give us more hints with regard to the allocator design. A brief application characterization for memory allocation has been done by Dirk Grunwald [112] where characteristics of five heap-intensive applications were brought out. In this paper we consider more applications and also look at more characteristics from an adaptive viewpoint. All these characteristics together with the idea of adaptivity drive the design of this allocator.

Application		Allocations	Deallocations
boxed	Simulation of Polyhedrals	1135111	1125591
cfrac	An implementation of the continued fraction algorithm	10886503	10886503
espresso	Two-level optimization	1675490	1668277
gawk	A search utility	2272645	2272394
gunzip	A decompression utility	32507	32507
make	GNU Make	15302	0
ptc	Pascal-to-C converter	102706	0
twolf	Standard Cell Placement and Global Routing Program (SPEC 2000)	574553	492713

Table 3.1. Applications that were used for malloc study.

The applications which we have chosen for this study are a set of heap-intensive applications (which were also used in the previous study by [112]), a set of normal day-to-day applications which have significant number of allocation/deallocation requests and SPEC 2000 benchmark which has many allocations/deallocations. Table 3.1 shows the applications used along with the number of allocations/deallocations. Number of allocations/deallocations varies a lot. On one hand are the applications like `cfrac` which have large number of allocation/deallocation requests (10 million) where significant part of the application run time is spent in the malloc code. On the other hand there are applications like `make` which have a few mallocs (in thousands) and where memory allocation is up to 5% of the application run time. Applications like `make`, `gunzip` and `gawk` are very commonly used day-to-day utilities.

3.4.1 Application Characterization w.r.t. Adaptivity

3.4.1.1 How many 'Distinct Sizes' do these applications have and what is their frequency ?

Application	Percentage covered by top *n* sizes					
	4	6	8	10	12	Distinct Sizes
boxed	99%	99%	99%	99%	99%	69
cfrac	88%	99%	100%	100%	100%	10
espresso	84%	87%	89%	91%	92%	756
gawk	64%	74%	85%	94%	95%	37
gunzip	94%	98%	100%	100%	100%	8
make	75%	92%	96%	97%	97%	63
ptc	98%	99%	99%	100%	100%	10
twolf	93%	96%	98%	99%	99.9%	74

Table 3.2. Table showing percentage of requests covered by the most frequent sizes

Optimizations have been proposed in the past to reduce allocation/deallocation times of requests of same size appearing again [31]. To see whether such optimizations can be used, we fist start looking at the number of distinct sizes and the frequency distribution of these sizes that the applications have. The frequency distribution of sizes might indicate whether size based adaptivity may be useful for this application. If the application has large number of distinct sizes and the frequency distribution is more or less uniform, adaptivity may not help much since the overhead in implementing size-based optimizations might overshadow the performance gains. On the other hand, large number of requests for fewer sizes is a positive indication. Table 3.2 shows the percentage of requests covered by the top n sizes where n varies. We can see from the table that most of the requests can be captured by top 10 requests. Similar observation has been made by Dirk Grunwald [112]. In fact, if we look at the number of distinct sizes, only `espresso` has significant number of sizes. If we can keep track of these frequently incoming sizes and lower the response time for these, it would benefit the applications. *So, adaptivity based on monitoring frequency could be useful.*

3.4.1.2 What are these sizes ?

It is also important to see what these sizes are as they might effect the optimizations we want to perform with them. For example, if the sizes are small, the allocator could keep a large chunk and peel it of when required. On the other hand, if the sizes are large, then keeping such chunks would become expensive. Also, traditionally, it has been observed that small sizes typically have small lifetimes and large sizes are long-living. Thus, it is necessary to service smaller size requests quickly (compared to requests of larger sizes). Also, if some larger size requests are not frequent, then we can afford to spend more time to service these. Table 3.3

Application	$<$ 256B	$<$ 1K	$<$ 4K	$<$ 16K	$>$ 16K
boxed	99.98%	0.01	0.01	10^{-5}	10^{-5}
cfrac	100%	10^{-7}	-	-	-
espresso	97%	2%	0.68%	0.14%	0.19%
gawk	89.5%	10.4%	0.05%	0.05%	-
gunzip	92%	4.14%	1.53%	2.33%	-
make	99.5%	0.5%	-	-	-
ptc	99.6%	0.4%	-	-	-
twolf	99.6%	0.3%	-	-	-

Table 3.3. Table showing percentage of requests within a specific size

shows the distribution of requests w.r.t. size. As we can see, most of the requests are less than 256 Bytes. *So, from the application view point, it is necessary to optimize for small sizes.*

3.4.1.3 What are the lifetimes for these sizes ?

It has been typically observed that smaller sizes are short-living and larger sizes are long-living. This is probably because the number of operations that need to be performed depends on the size and a smaller size indicates fewer operations on that data structure. To our knowledge no study has substantiated this observation of relation between the size and life till now. In [112], sizes, inter-arrival times and lifetimes are all treated independently and the relationship between size and life (if at all there is any) is not established. Table 3.4 shows the variation of the lifetime of objects (in terms of number of allocation/deallocation requests) with the size i.e., the lifetime of a object is the sum of number of allocations and deallocations between its allocation and deallocation. It can be seen from the table that in general there is no regular behavior, although the values towards the right end of the table are typically high showing that large allocations actually live for long time. One reason for observing no trend in Table 3.4 is that number of

Application	0-256B	256B-1K	1K-4K	4K-16K	> 16K
boxed					
cfrac	2593.31	$1.07x10^7$	-	-	-
espresso	349.82	87.20	273.6	1691.9	45152.4
gawk	24.71	18.0	83.0	$2.27x10^6$	-
gunzip	23.7	1.24	0	41.0	-
make	-	-	-	-	-
ptc	-	-	-	-	-
twolf	14548	2398	2926	77917	149725

Table 3.4. Table showing average lifetime for the requests (in requests)

actual operations between the allocation/deallocation requests is not captured. In order to do this, we show the variation of lifetime with size in terms of the absolute number of seconds in Table 3.5.

From Table 3.5, we can see that there is a general trend of life times increasing as we go towards higher sizes. Thus for these applications, it is important to optimize for small sizes quickly compared to large ones. At the same time it is also possible that there are some other applications which actually allocate and deallocate larger sizes more frequently. So, apart from optimizing smaller size requests (as mentioned in the previous subsection), the system design should make sure that *these optimizations will not become a penalty for some other applications having large sizes*.

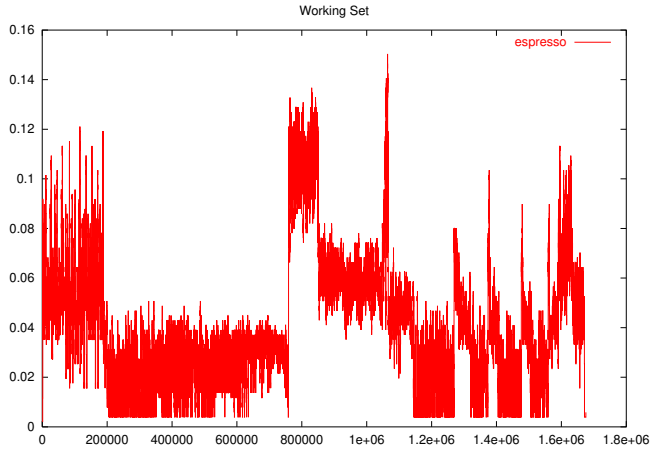
3.4.1.4 Is there a Working-set ?

The discussion above focused on size distribution w.r.t. frequency and lifetimes but it does not capture the temporal behavior of requests. Adaptivity will work well when there is a *Working-set of sizes* for the application that is either a constant or that changes slowly. If

Application	0-256B	256B-1K	1K-4K	4K-16K	> 16K
boxed	0.026	0.08	3.82	x	x
cfrac	0.015	64.2	-	-	-
espresso	0.002	0.000544	0.00176	0.011	2.98
gawk	0.000176	0.00013	0.00064	15.9	-
gunzip	0.0035	0.0059	0.00003	0.005	-
make	-	-	-	-	-
ptc	-	-	-	-	-
twolf	12	3.4	0.17	53	106

Table 3.5. Table showing average lifetime for the requests (in seconds)

gawk



espresso

Fig. 3.1. Number of distinct sizes in a window of 512 for espresso and gawk

adaptivity be applied to a random sequence of sizes, it might increase the run-time instead of decreasing it because of the overhead involved in the adaptivity itself (which will not be useful as the sizes are random). To capture whether these applications have such a working set, we plot the number of distinct sizes in a given window as the window moves across the requests. Figure 3.1 shows this metric for two of the applications (others are similar). *x*-axis is the time (in terms of number of requests) and *y*-axis shows the ratio of the number of distinct sizes to the window size. The graphs were plotted for a window size of 512. The lower the curve, the less the number of distinct sizes in the window and adaptivity might be more useful. For `gawk` the curve is really low which means that the working set is small and more or less a constant. On the other hand, if we look at the curve for `espresso` which has 756 distinct sizes, the working set variation is high. Adaptivity can still be useful in some phases for this application (where the curve is low).

Overall, from the adaptivity viewpoint, these applications show promising characteristics. The number of sizes are quite less and the working set of the sizes is not very large.

3.5 System Design

We now present our allocator design. Our allocator is based on and uses the *Slab Allocator* [31] extensively. Therefore, we first discuss the Slab Allocator and then present our design.

3.5.1 Background : The Slab Allocator

The Slab Allocator [31] optimizes the pattern of incoming requests of a single size by removing the constraint of maintaining an address order. A slab is a chunk of memory created for a given size. The components of a Slab are a large chunk of memory (which is typically

a multiple of requested size) and a stack pointer. On an allocation request, if the stack is not empty, the top element is popped (which is a chunk of requested size) and returned to the user. If the stack is empty, then the larger chunk is peeled off and that element is returned. On a free, the element is simply pushed onto the stack. There is a tag associated with the element returned to the user which directly points to this slab. Thus, freeing is $O(1)$. Allocation is $O(1)$ if we know the slab. Other variations such as coalescing with the larger slice when an element is freed (if they are merge-able) etc. can be implemented. The Slab allocator has typically been used in the kernel of several operating systems. We have used slabs extensively in our allocator design.

3.5.2 Design

Summarizing the observations from the previous sections :

- Optimization to service smaller sizes quickly
- Adapting to frequently incoming sizes - small or large
- It is not essential to service larger-infrequent sizes quickly
- At the same time, it will be good if we can bound the allocation time.

This section describes the three essential components of the memory allocator. These components are built to satisfy the above goals. This section also talks about how these components should be glued together for better performance.

We first start with our initial design which had two components - the *Adaptive Cache* and the Address-Ordered List. We then change our design by introducing more component - the Slab Cache that further improves the performance. The key point of the allocator is to dynamically create slabs for frequently incoming sizes.

3.5.2.1 The Adaptive Cache - for frequently incoming sizes

The main component of the allocator is called the *Adaptive Cache (AC)*. AC is responsible for monitoring frequently incoming patterns of sizes and optimizing for them. AC itself has three components :

L1-Adaptive Cache (aL1)

The aL1 is an array of pointers to various slabs, each for a different size. When a request reaches aL1, the complete array needs to be searched to satisfy the current request. This array is meant to capture the current active pattern of sizes. If the slab for the requested size can be found here, then its an aL1 hit and a slice can be immediately popped/peeled from that slab and returned. If this slab becomes empty after servicing this request, then its becomes an agedOut slab and is thrown out (of the system - information is never maintained for such slabs - explained later). If a slab for a size is not found in aL1 then the request goes to L2-Adaptive Cache.

L2-Adaptive Cache (aL2)

When a request arrives, the size is used to determine an index and the linked list of slabs at this index is searched. If a slab of this size can be found in that list, then a slice is popped/peeled from this slab and this slab is now inserted into the aL1 (if it is still not empty). aL2 contains all the slabs in the allocator, of which the *hot slabs* (which represent the current pattern)are present in aL1. While inserting a slab into aL1, another slab might be evicted from the aL1 (since the size of aL1 is limited) - this is done based on the L1-Priorities which are maintained in aL1. There are also L2-Priorities what are maintained in aL2. As the linked list is traversed in aL2,

the L2-priorities for the slabs which are 'along the path' are decremented. If this priority for a particular slab falls below a threshold, then that particular slab is treated as an agedOut slab and it evicted out of aL2.

Detecting frequent Sizes

In case a size is not found in aL2, the following criteria will determine whether this size is frequently incoming size :

An array of N_1(=16) entries contains the most recent N_1 sizes along with their frequencies and recencies. On every request, the frequency of that size (if it exists) is incremented and the recency for all the other sizes are decremented. If a particular size cannot be found, then the least recency size is replaced with the new size. When the frequency for a size reaches a threshold, then that size is considered as a frequently incoming size and a slab is created for this size. This size is then removed from this array. The newly created slab is inserted into aL2 (is also added to the list of the slabs corresponding to the same aL2 index and this slab will be the first in that list). This slab is also inserted into aL1. Thus, the criteria to create new slabs is based both on frequency and recency of the requests.

Usefulness of Adaptive Cache

The adaptive cache is designed to perform well for the applications which have a working-set of sizes. The evictions of slabs from aL1 and promotions from aL2 to aL1 represent the changes in the working set. If the working set is a constant or fits in aL1, interaction between aL1 and aL2 will be very less.

For rarely occurring sizes, adaptivity will not be useful. These sizes (and the first few requests of the frequently occurring sizes until the threshold to create a slab is reached) are directly

served from a different component (Address-Ordered-List of Red-Black-Tree - explained later). But these sizes still incur the overhead involved in monitoring the sizes, and making a decision whether the sizes are frequently incoming or not. If an application does not have a working set at all, then this overhead will be experienced by the requests unnecessarily. Thus, adaptivity need not always improve the application performance. In an attempt to tune the allocator to the application, performance may be lost i.e., the overhead in observing the sizes and maintaining book-keeping information for the sizes might actually overshoot the performance gain. Adaptivity will improve the performance of those applications which have a good working set where the gain obtained from serving frequent sizes quickly is more than the loss incurred due to rarely incoming sizes.

Tolerance in allocation

When a slab is created for a larger size in the adaptive cache, it is being done with the hope that there will be more requests for this size. It there are no more requests for this size, then that memory will not be utilized (until all the requests allocated from that slab are not freed). To reduce this problem (and to increase the hit rate in the Adaptive cache), adaptive cache allocates with some *slack*. In other words, when a request reaches the adaptive cache and the slabs are being checked if they can satisfy the request, a slab need not be of exactly the same size to satisfy that request. Even if it is a little larger size (up to 3% larger), that request is serviced from that slab. By doing this, the overhead of creating a new slab can be avoided but we are sacrificing utilization.

Deallocation

Deallocation of requests serviced from the slabs is $O(1)$ operation. There is a tag associated with

each of the slices given to the user (this tag is stored one word before the user pointer). This tag is nothing but the pointer to the slab. So, when a user frees the memory, the slab can be reached in $O(1)$ operations as there is no search involved. Pushing into the stack is again $O(1)$. So, deallocations are very quick. The tag has a bit which indicates if this request has been serviced from a slab. If not, then this deallocation is routed to other component (discussed later).

Aged-Out Slabs

Aged out slabs are the slabs which either got completely empty or which were evicted from aL2 (due to low priority). There is a status field in the slab which indicates if it is an aged-out slab. As the deallocations occur, the slab is checked to see if it is full (i.e., if all the deallocations are done - remember that the slab is a contiguous piece of chunk and cannot be freed unless all the memory in it has been released). If the slab is full and the status is aged-out, then this slab can be freed to wherever it was allocated from (next section). Thus, aged-out slabs are actually not present in AdaptiveCache. They are not even linked anywhere in the data structures. The only way to reach them is when the user deallocates. The advantage of this is that the allocator does not need to keep any state information for these slabs which anyway cannot service requests.

Address-Ordered-List/Red-Black-Tree

The third component of this framework can be compared to any other traditional memory allocator. The idea here is to service the requests as quickly as possible but with less fragmentation and high utilization (note that in creating slabs, we are actually sacrificing these to an extent). An address-ordered-list (AOL) which serves on a first-fit based policy (and is simple to implement) is a possible data structure that can be used here. In an AOL, the bound to service a request is $O(n)$ when there are n free blocks. On the other hand, balanced tree structures

like red-black trees guarantee a bound of $O(log(n))$. It is difficult to say that one of these data structure performs better than the other - this depends on several factors including the patterns of sizes that are requested, how many requests reach this component etc. If very few requests come to this component, it probably does not matter too much and one might choose to go with simple structures to avoid balancing operations and the complexity (simple structures like a linked-list or a cartesian tree). On the other hand, a large number of requests could imply degradation in performance when such simple structures are used. Section 3.6 brings out these differences quantitatively. The requests that reach this component include the rare large size requests from the application and requests from the AdaptiveCache to allocate and deallocate slabs.

3.6 Performance Evaluation

3.6.1 Experimental Setup

All the experiments were conducted on an IBM Regatta machine that has 1.1 GHz Power4 processor and 2 GB Memory. The applications were compiled using IBM's xlc compiler with aggressive optimizations. The applications were instrumented using hardware counters to measure the total instructions and cycles spent over the complete application run as well as in the malloc code. We are using the HPM performance monitoring tools developed at IBM for this purpose [8]. The time spent in the malloc code was further split into time spent at different components. Numerous statistics with regard to hit and miss rates at different components, number of allocations and deallocations of slabs, number of average traversals during allocations/frees etc. were also collected to get further insight. Though we collected both cycles and instructions for all the experiments, we primarily deal with instructions when we talk about response time

since the variance in nummber of cycles was quite high. We compared our allocator mainly with the traditional AIX Allocator that is available on the AIX platform.

This section presents the quantitative comparison of various components followed by performance evaluation with applications.

3.6.2 Quantitative comparison of the components

The allocator infrastructure comprises of all components discussed in the previous section. These components can be glued to a specific *architecture* (this can be done simply by using an environment variable in our implementation). We first examine each of these components quantitatively and determine the conditions under which one would do better than the other. Such an understanding will not only give further insight into the performance results but can also be used by the user to select an architecture that would benefit applications (this can be done by setting the environment variable mentioned). We make observations (which are valid for this OS/Hardware) that can help users to make decisions regarding selecting an architecture for the allocator. If the OS/Hardware platforms are different, then the user can run similar experiments and make decisions based on those results.

3.6.2.1 Adaptive Cache

As mentioned earlier, Adaptive Cache (AC) is very useful when there is a working-set of sizes. Now, since the resources in the AC are limited, there is a limit on the working set for which the AC will perform well - i.e. AC will not be able to capture very large working sets (in such cases, AC can be disabled using the environment variable). To see this limit on the working-set size, we constructed a synthetic workload which generates requests randomly from a working

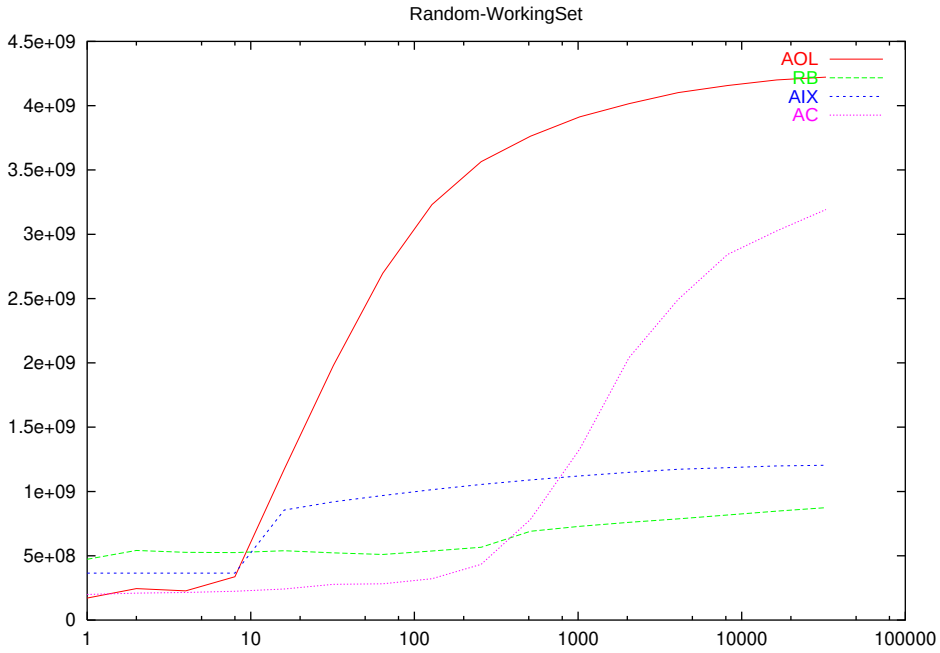
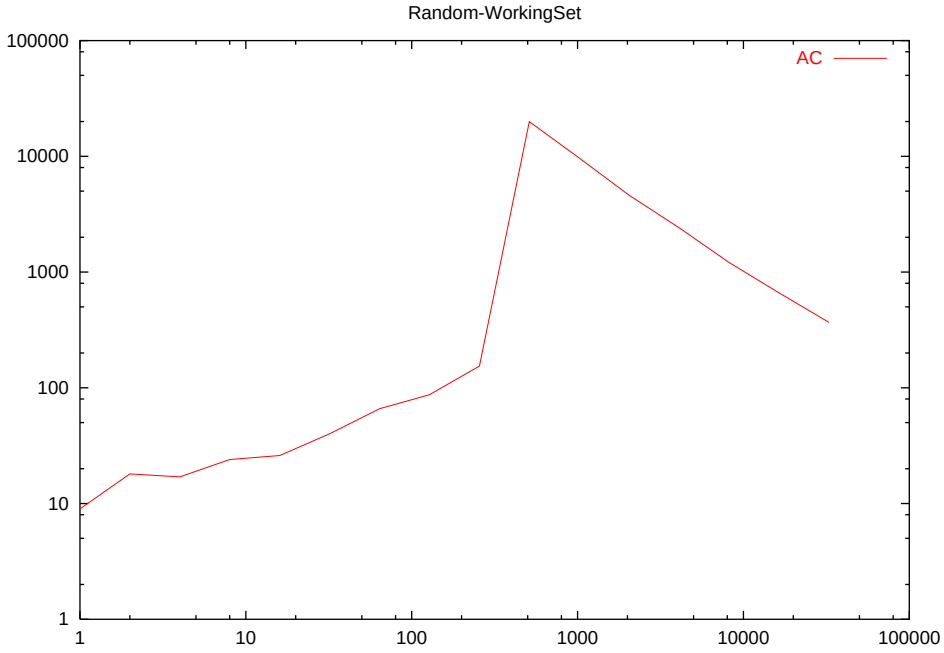
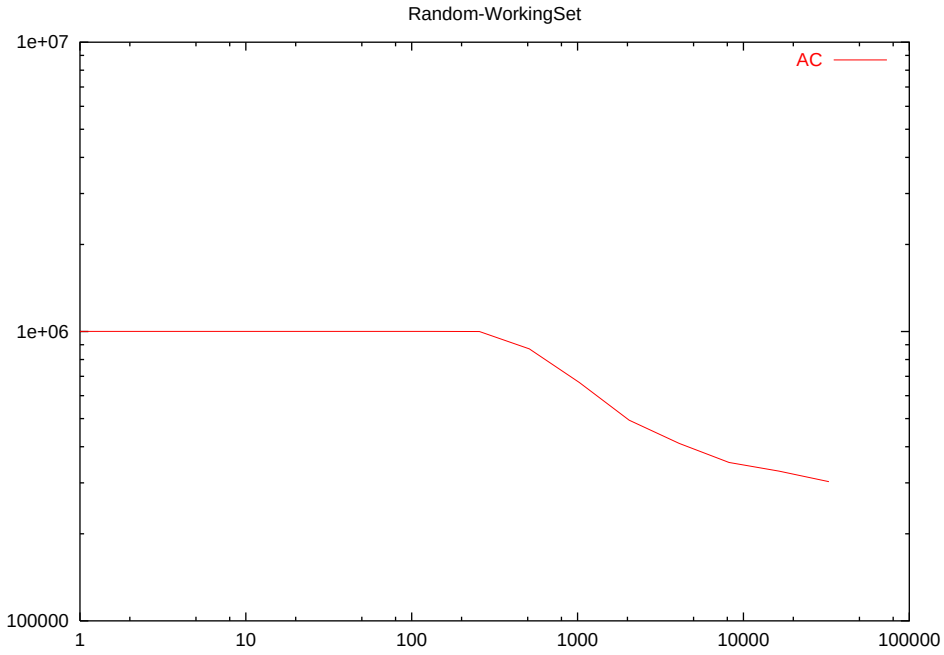


Fig. 3.2. Performance of various components for the varying working- sets



(a) Slabs created in AC with varying working sets



(b) aL1 hits(for million requests) with varying working sets

Fig. 3.3.

set of sizes. We then plotted a graph (Figure 3.2) varying the working-set, with the size of the working-set on the x-axis and response time on the y-axis for all the components (Note that for comparison, even AIX malloc is being treated as a different component here).

We see that AOL starts breaking down for working sets if size as low as 10 but until then, it performs better than both AIX and RB. This is because AOL requires a search through the linked-list where as AIX and RB are tree structures. The number of operations involved in balancing the trees dominates till the working-set size is 10 after which it is the traversals that matter. AIX does not have as many balancing operations as RB (even if it has the balancing operations, they are only w.r.t. address and not w.r.t size). So, AIX does better than RB but worse than AOL initially. But as the working set increases (beyond 10), RB which balances both based on size and address really starts performing well. In fact, the response time growth in RB is the least of all and RB does very well even for huge working sets also (thousands as can be seen from the graph). Coming to AC, AC is the best as long as the working set is < 128. For smaller working-sets, the number of slabs created will be really small (as we can see from Figure 3.3(a)) and the aL1 hit ratio will be high (Figure 3.3(b) shows the absolute number of hits in aL1 for a million requests). As the working set gets larger, the number of slabs created increases (aL1 hit rate more or less remains the same) and then decreases because the frequency of the same size to appear decreases. This behavior can be seen in the Figure 3.3(b). Note that one important reason why AC keeps its hit rate quite high even after the working set increases is that the slab need not be exactly of the same size to satisfy the request. As mentioned, a slack up to 3% is allowed.

Overall, we quantitatively determined that AC can be useful if the working set is small ($\leq$ 128). AOL and AC, both do a good job for very low working sets ($\leq$ 10). From an asymptotic view point, RB is good candidate.

3.6.2.2 AOL/RB

Theoretically, RB is a better choice since it gives a bound of $O(log(n))$ where as with an AOL, the bound is $O(n)$. These bounds hold true as n grows large. When n is small, multiplying constants make a difference. Therefore, a user simply cannot assume that RB will always perform better than AOL. For smaller values of n, the constants associated with these bounds also come into picture and it is necessary to investigate and determine the value of n from where (as n increases) RB really performs better than AOL. To determine the value of n, we did a simple experiment where a list of n free blocks is always maintained and the allocations and -allocations are always done in such a manner that the list is not disturbed (i.e. the free blocks in the list will not be able to satisfy the requests and each allocation/deallocation will traverse n blocks in the AOL). A similar experiment is done with RB where the blocks will be arranged in form of a tree. We then vary n and plot the response time of AOL and RB as shown in Figure 3.4

AOL initially performs better than RB because when the list is small since the the overhead of traversing the list is less than that caused due to balancing operations. At n = 25, the curves intersect after which RB is a better choice. So, if the user has some basic idea of the allocation/deallocation patterns, he can select the AOL configuration (using the environment variable) instead of RB. The graph also shows the curve for AIX. AOL is better than AIX for n $\leq$ 15.

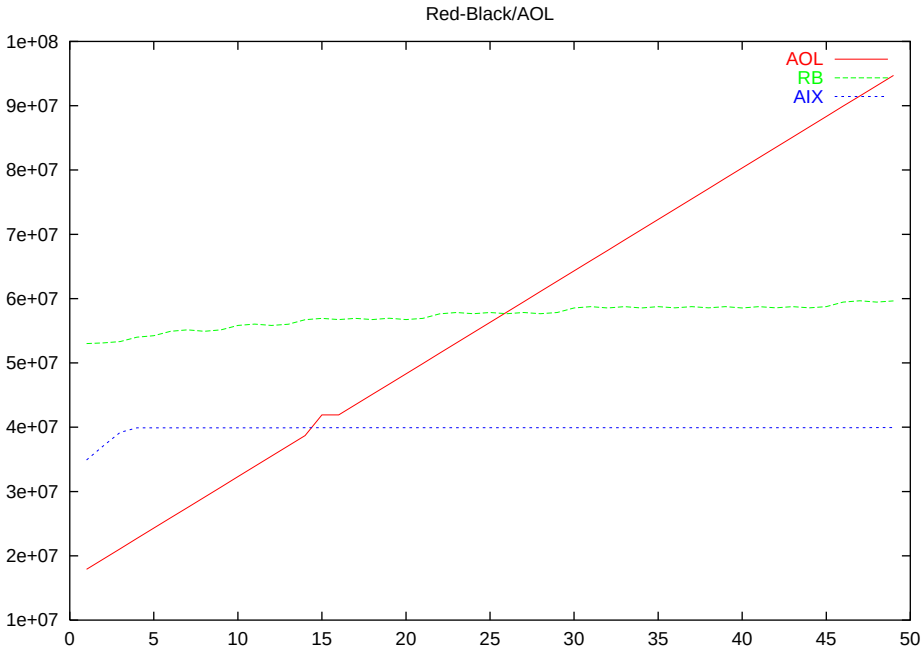


Fig. 3.4. Graph showing the response time of RB and AOL - AOL is a better than AIX for $n \leq 15$ and RB for $n \leq 25$

3.6.3 Performance Evaluation using applications

3.6.3.1 The Adaptive Cache

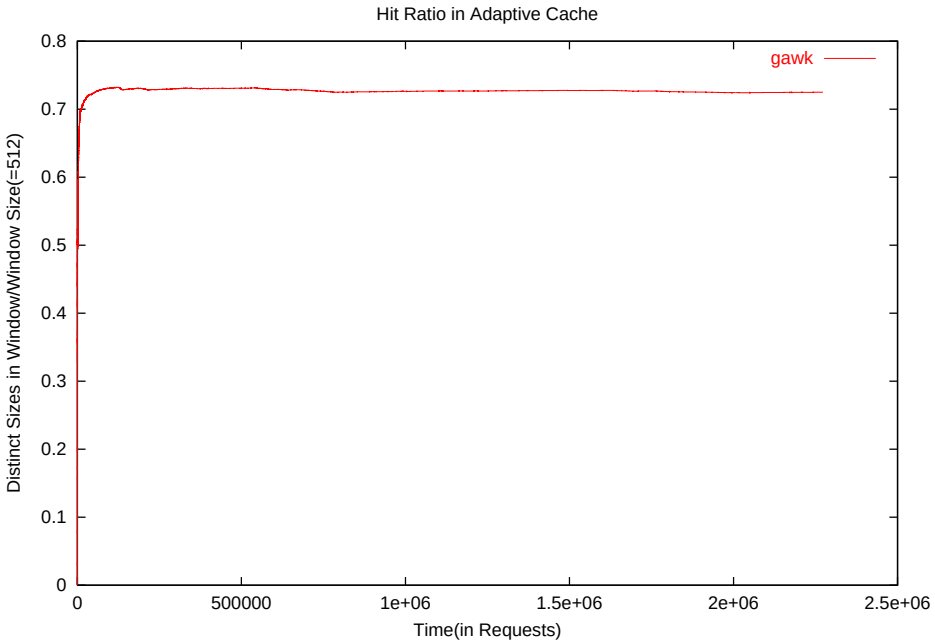
The performance of the adaptive cache entirely depends on the application. Since most of these applications have small working sets (top 10 sizes capture more than 90% of the requests as seen in Section 3.4), we expect the adaptive cache to do well.

In order to capture the dependency of the performance of the adaptive cache on the working set, we first plot a graph which shows the hit rate in the adaptive cache (Figure 3.5) and compare it with the working set graph shown in Figure 3.1. Comparing these two graphs, we see that hit rate increases as the working set decreases and vice versa.

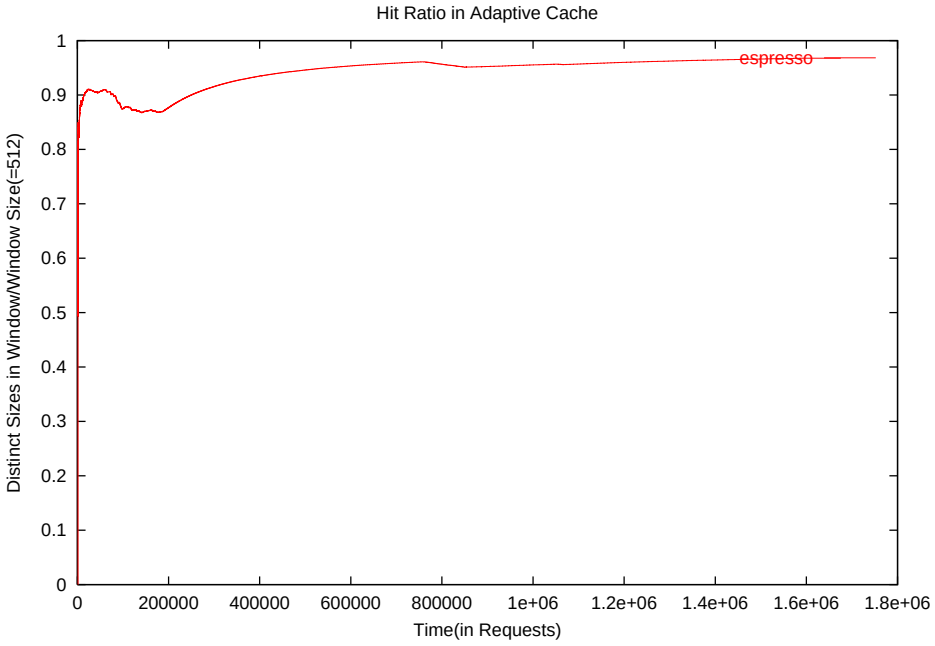
Application	AC/AOL	Requests	Average Searches		Hits		Slabs
			aL1	aL2	aL1	aL2	Created
cfrac	0.50	10886503	2.65	1.1	99%		31
espresso	0.81	1675490	11.1	3.9	97%	9 req	7162
gawk	1.12	2272645	14.4	0	72%	28 req.	43
gunzip	0.56	32507	1.62	0	99%	8 req	19
make	1.52	15200	9.05	0.19	80%	32 req	123
ptc	1.15	102706	2.67	2.72	99%	10 req	77
twolf	2.0	574553	8.5	0.02	98%	20 req	128
synthetic1	0.44	108000	2.50	2.16	99%	4 req	52

Table 3.6. Adaptive Cache Statistics

Table 3.6 shows the performance of the Adaptive Cache for all the applications. The first column is the normalized runtime for all the applications w.r.t to the AIX malloc. AC/AOL stands for an Adaptive Cache/Address-Ordered-List hierarchy where slabs are dynamically created and maintained in the two levels of the adaptive cache. Large request sizes and infrequent sizes fall



gawk



espresso

Fig. 3.5. Hit Rate in the Adaptive Cache

back on the AOL. Also seen in the table is *synthetic1* - an artificial workload of working set of 16 sizes which we have created.

Whether AC performs better than AIX depends on the application. Two main factors contribute to this - working set and the total number of requests. If the total number of requests is very low, even if the working set is small, the overhead in *adapting* (creating slabs in this case) will not be useful. Small working set with large number of requests is the ideal condition for adaptivity. As we can see, AC performs better than AIX in a few applications and worse than AIX in others. For applications like `cfrac` which have a large number of allocation requests, AC does a good job. The main reason for AC not to do well in other applications is the number of operations required to find a slab. Since AC creates and maintains slabs dynamically (as said earlier), when a request comes, AC will first search in the aL1 level to find a slab for that request. If the slab is found, then allocations is request is satisfied. From Table 3.6, we see that the number of operations spent in aL1 to an attempt to service requests is quite high. This is the main factor for AC to not perform as well as AIX. In fact, though 90% of the requests hit in aL1 in many applications, the overhead in searching is still high. A good thing to note in the table is that search in aL2 does not seem to be a big overhead. Thus overall, AC by itself is on par with the AIX malloc. Below, we show further optimizations that can be made which will improve the performance of the allocator.

So, AC is better than AIX for some of the applications. We also conducted experiments to see how the other configurations (AOL and RB) perform for these applications and compare then with AIX and AC. Table 3.7 shows these results. The first three columns show the normalized time spent in the malloc for the AOL, RB and AC respectively. In the absence of AC, all the requests directly go to AOL/RB - this number is shown in the fourth column followed by the

Application	AOL	RB	AC/AOL	Without Adaptive Cache			With Adaptive Cache		
				Requests	AOL-Tr	RB-Tr	Requests	AOL-Tr	RB-Tr
cfrac	5.32	1.16	0.50	10886503	143.9	2.2	55	1.5	2.2
espresso	0.44	1.03	0.81	1675490	8.5	5.2	44053	130	4.4
gawk	0.36	1.24	1.12	2272645	1.6	3.3	624974	0.34	3.9
gunzip	0.43	1.66	0.56	32507	0.48	1.5	54	1.0	0.6
make (no frees)	0.54	1.53	1.52	15200	0.014	0.004	2675	0.008	0.004
ptc (no frees)	0.57	1.63	1.15	102706	10^{-4}	10^{-4}	109	0.013	0.13
twolf	125	3.5	2.0	574553	1500	7	10265	3.0	3.2
synthetic1	93.3	1.51	0.44	108000	3500	9	73	5.0	1.25

Table 3.7. AOL and RB operations and timings, AC timings - comparison

average number of traversals in AOL/RB (AOL in fifth column and RB in the sixth). In the presence of AC, lot of requests get satisfied in AC itself. Some infrequent requests and requests for Slabs will be sent to AOL/RB - this number is shown in the next column followed by the average number of traversals for these requests.

The first observation to make is that AOL is doing better than AIX malloc in many cases. This is because, if the average length of the AOL is < 15, AOL is a better choice (from previous section) and for these applications, the average number of traversals (listed in the fifth column) are less than 25 (except for `cfrac`, `twolf` and `synthetic1` and AOL is not doing good in these cases). So, if the user is aware beforehand or has some hint about the allocation/deallocation patterns, he can make use of AOL. The second observation is that RB is consistently bad compared to AIX and this happens because of the same reason (average traversals are > 25 only for `cfrac`, `twolf` and `synthetic1`). The third observation is that AC is doing much better than RB. We can see from the table that most of the requests get satisfied in the adaptive cache (by comparing columns 4 and 7). Still AC itself is performing worse than the

AIX malloc in a few cases and it is necessary to optimize further. For this we propose the slab cache.

3.6.4 Slab Cache

3.6.4.1 The Slab Cache - servicing smaller sizes quickly

SlabCache is an array of slabs that are created for smaller sizes. Its purpose is to service the smaller sizes very quickly without search. When an allocation request for a size $s(< S$, where S is the maximum size the SlabCache can serve) is generated, an array of slab pointers is indexed using s. If the array slot is not empty, then the request can be serviced immediately by popping/peeling out a slice from that slab. If that slot is empty, then a new slab is created for this size, the request is serviced and a pointer to that is stored in that slot. This new slab is obtained by making a call to a different component (discussed later).

Thus, servicing a request through SlabCache requires no search. If the slab is found, the request can be serviced in $O(1)$ operations. The size of the Slab that needs to be created depends on the demand for this size. The slabs which are created initially are smaller ones which can service a few requests. If a Slab becomes completely empty due to the demand for that particular size, then that slab is called an *agedOut* Slab (agedOut slabs are discussed later). Each time an agedOut slab is replaced by a new slab, the new slab will have double the size of the original slab. Therefore, servicing smaller sizes very efficiently is accomplished with SlabCache.

The conditions under which a slab cache would be very useful are obvious. If the slabs can be created up to size S, applications would significantly benefit from slab cache if most of the requests are $\leq S$. Even if many of the requests are $> S$, application run time can still be improved as long as there are some sizes ($\leq S$)that repeat - this being at an expense of utilization.

Nevertheless, since most of the short size requests are short-living, it is important not to spend too much time in allocating/deallocating these and SlabCache is a good idea to speed up such request allocations.

Application	AOL	RB	SC/AOL	SC/RB	SC/AC/AOL	SC/AC/RB
cfrac	5.32	1.16	0.32	0.32	0.32	0.32
espresso	0.44	1.03	0.22	0.23	0.23	0.23
gawk	0.36	1.24	0.30	0.30	0.30	0.30
gunzip	0.43	1.66	0.40	0.42	0.40	0.40
make	0.54	1.53	0.58	0.59	0.61	0.62
ptc	0.57	1.63	0.62	0.62	0.62	0.62
twolf	125	3.49	0.78	0.78	0.78	0.78
synthetic1	103	1.01	0.33	0.23	0.79	0.43

Table 3.8. Effect of the Adaptive Cache

Finally, we evaluate the effect of having a SlabCache on the applications. The fact that more than 90% of the requests are less than 256 Bytes is an indication that the SlabCache will improve the performance. Table 3.8 shows the performance of SC with and without AC. The first column is the normalized time spent in malloc for AOL configuration (w.r.t. AIX), the second column is for RB, third is for SC/AOL and so on (as listed).

Introducing an SC improves the performance drastically. In `espresso`, improvement is about 5 times (compared to AIX malloc). In no application does the performance worsen since most of the requests are for smaller sizes. Thus, it seems to be a very good idea to have SC. The other important observation is that SC/AC configurations (with AOL or RB underneath AC) perform almost the same as SC configurations (without AC). In other words, using adaptivity below SC is at least not hurting these applications. Since most of the sizes are small, SC is able

to satisfy them but if we have an application which would have patterns of larger sizes, then the effect of AC would come into picture.

Having seen that SC is an important component, the next question to address it - what size of SC do we use ? In order to answer this, we ran experiments varying the size of SC from 32 bytes to 4K (by saying that size of SC is S, we mean that all the requests of size $\leq S$ would be satisfied in the SC). Other requests will be sent to AOL underneath it. Figure 3.6 shows the ratio of the time spent in malloc to the ratio of time spent in AIX malloc for all the applications as the size of the SlabCache varies. From the figure, we can see that after a SlabCache size of 128, the improvement is negligible. We recommend an SC of size 256/512 entries to get good performance.

3.6.5 Summary

Overall, we observe the following from this comparison :

- Having the Slab Cache enabled is a good idea
- For working set sizes < 128, adaptivity might help a lot.
- If the working set is < 10, AC and AOL are more or less the same
- If the application does not have a working set but if there is an indication (obtained by looking code structure and how mallocs and frees are being made) that the AOL length will be ≤ 15, AOL is better than AIX. If AOL length ≤ 25, AOL is better than RB.
- Asymptotically RB is the best configuration.

Overall, SC/AC/RB seems to be a good organization - it has the ability to serve the smaller sizes quickly, adapt to larger sizes which are frequent by creating slabs only for those

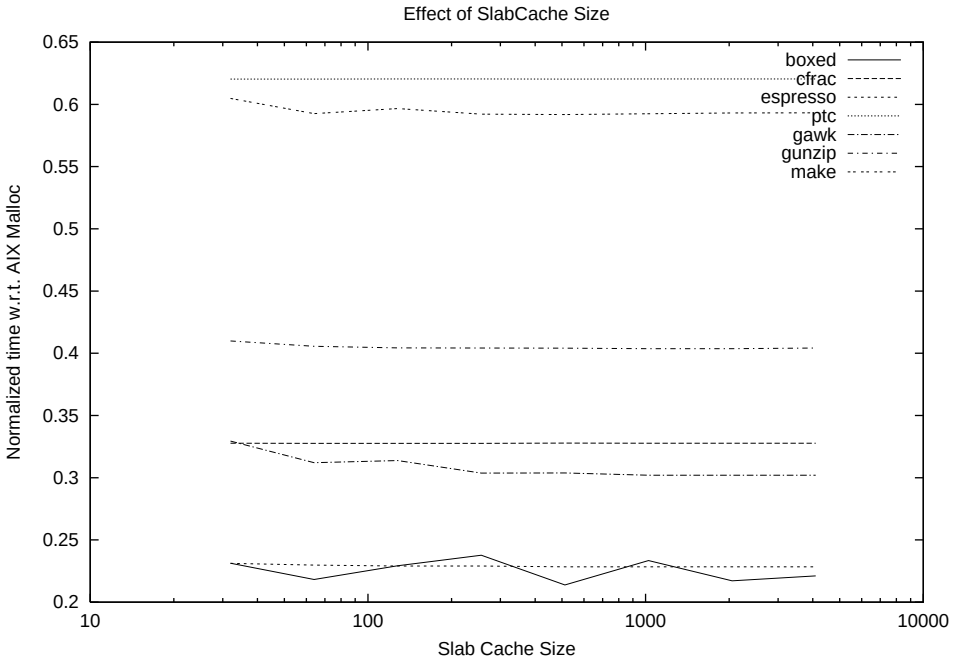


Fig. 3.6. Effect of Slab Cache size on the performance

sizes which are really frequent and finally, serving rare large sizes using a red-black-tree in a best fit manner in a bounded time. We finally recommend SC/AC/RB configuration.

3.7 Conclusion

Dynamic memory allocation is a necessity of many application programs today. For good performance of heap-intensive applications, programmers typically write custom allocators. These are not flexible and have high software engineering overhead. On the other hand, general purpose allocators are slow. In this thesis, we take a different approach by proposing an adaptive memory allocator that tunes itself to the application programs.

Adaptive allocator performs well compared to other traditional allocators like AIX allocator. Adaptivity does not come for free, it has its overheads as well. After looking at the application characteristics we find that most of the objects allocated are usually small. Therefore, a small optimization performs much better. Nevertheless, adaptivity will be useful when there is a working-set of large sizes.

Chapter 4

Memory Management in the Operating System

Applications which fit in the main memory will benefit from the hardware/software optimizations done in the processor-memory path discussed in the previous two chapters. Many applications today do not fit into the memory. For these applications, it is important to minimize the disk access. Disk access times are orders of magnitude higher than the memory access times. Therefore, optimizations in the operating system to prevent disk access will have a good impact on the performance of such applications.

The component of an Operating System (OS) that is responsible for maintaining important data in the memory and minimize disk access is called the Virtual Memory Manager (VMM). The behavior of VMM plays a significant role in determining the performance of many applications. While other system components like processor architecture, cache design, memory design etc. are important and do have an impact on the performance of an application, the large difference between access times of memory and disk make VMM more important, especially for those applications that do not fit into the memory.

The performance of an Operating System VMM mainly depends on two factors: the algorithm of the VMM (the page-replacement algorithm) and the parameters associated with this algorithm. Over the years, much work has been done on designing, simulating, implementing and evaluating page-replacement algorithms [39, 71, 96, 92, 38, 79, 78, 106, 99]. Models have been proposed to capture and use program behavior [51, 50, 81] to improve the performance of VMM.

Compiler based approaches have also been proposed to improve the performance of VMM [79]. Other runtime approaches to improve VMM performance include approaches like compression of virtual memory [74, 3]. On the other hand, improving performance of VMM by changing the parameters in the VMM has received much less attention in the past. The parameters of VMM play an important role in many key decisions as to what pages to keep in the memory and what to throw out. Setting these parameters to the right values can improve the performance of applications drastically. In fact, it is already well-known to the community/OS developers that the values of these parameters do have significant impact on the application performance [11]. To our knowledge, no one has attempted to study and understand these parameters in great detail. In order to get a good hand on them, it is important to study their functionality, classify them, qualitatively reason out their influence on different kinds of applications and finally quantitatively support these arguments. One main impediment to do this, we think, has been increasing complexity of operating systems. It is extremely time consuming and one might have to spend months to browse through thousands of lines of code and make inferences out of it. The only work in this context, to our knowledge, can be found on some of the open source mailing lists [11] where people exchange ideas about improving performance of applications by changing these parameters. A significant amount of discussion and experimentation goes into setting each of such parameters. This is also true with commodity operating systems which are not open source. In such cases, when end-users start complaining about performance, the OS developers start tuning these parameters by massive experimentation. Due to large number of parameters in a typical OS (including various other components like networking, scheduling, I/O etc.), it may not possible to tune every parameter. As a result, tuning is typically need-based i.e., only when a user/customer is not satisfied or starts experiencing performance degradation in a particular

subsystem. Also, tuning in such cases is limited to improving the performance of specific applications. Consequently, there is a lack of thorough understanding of such parameters and their influence on the application performance.

Methods to self-tune these parameters dynamically would help enterprises as well as open source communities. It is not only economical in terms of time and cost, but can also buy more performance. While attempts are being made to develop such techniques [24], it is first necessary to have a thorough understanding of behavior of such parameters before developing techniques to dynamically tune them. Specifically, with in the scope of VMM, we would like to gain insights on the following questions:

- First of all, what are all the important VMM parameters in a typical operating system?
- What are the functions of each of these parameters and what role do they play in the VMM? Are all these parameters known (exposed) to the users?
- Can we make qualitative arguments about effects of these parameters on various applications?
- Using a variety of applications from different classes, can we support these qualitative arguments with quantitative results obtained by varying all these parameters across reasonable ranges?
- Do all the parameters have significant effect on the performance of the applications? If not, what are the important ones? Are there optimal values for these across various applications? Are the optimal ranges same for different applications?

- Finally, can we correlate the effect of these parameters to pure application characteristics (based on application memory references)? or Can we correlate them to application characteristics that can be observed with in an operating system (application-OS characteristics obtained by observing page-faults alone) ?

Answering some of the above questions would not only provide good insight into characteristics of the parameters but also would help us to propose techniques to make the system self-tune the parameters dynamically.

In an attempt to answer some of these questions, this thesis makes contributions by studying all the parameters in the Linux [12] Operating System's Virtual Memory Manager, classifying them both based on their functionality as well as their influence on applications, qualitatively and quantitatively analyzing their effect on applications from different classes, relating these effects to application characteristics and/or application-OS characteristics, providing a few suggestions with regard to exposing these parameters to the open source communities, and finally by providing a good motivation to tune these parameters dynamically.

The rest of this shapter is organized as follows: Section 4.1 presents an overview of the Linux Memory Manager, Section 4.2 describes all the parameters and their effect (qualitatively), Section 4.3 presents a quantitative analysis of all the parameters with a discussion and finally, Section 4.4 concludes.

4.1 Overview of the Linux VMM

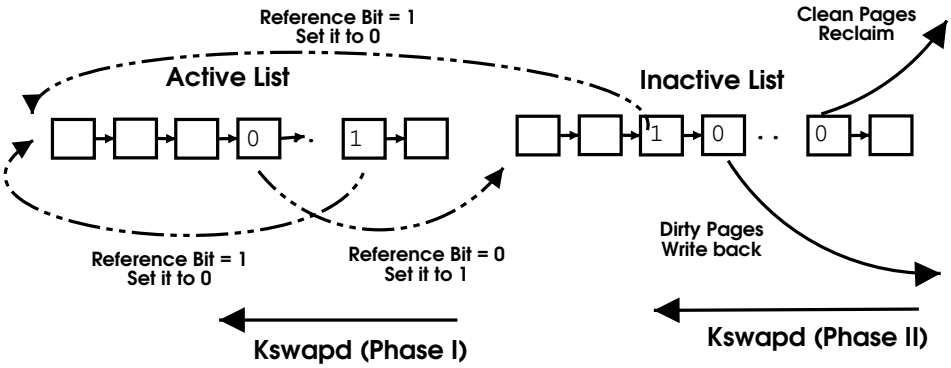


Fig. 4.1. Linux Page Management Policy

This section presents a brief overview of the Linux Virtual Memory Manager. As we describe the algorithm, we also refer and relate to the parameters of the algorithm, and try to bring out how both together determine the behavior of the Linux VMM. Since the Linux Kernel is continuously evolving, we chose a particular version of it for our study. The following discussion is based on Linux Kernel version 2.5.74.

Linux VMM keeps all the pages in the physical memory in two lists: The Active List and the Inactive List (actually a few, relatively small number of pages are locked for the kernel code and data structures - we ignore this for now). These lists are maintained using a FIFO-like policy. The pages in the Active list represent 'hot' pages that have been recently accessed and the pages in the Inactive List are 'cold' pages that have been accessed sometime ago and could potentially be candidates for replacement.

As the application keeps executing, requiring more memory for its execution, memory is allocated to it from a *free-pool* where pages are managed according to buddy system [91]. When the number of free pages in this free-pool falls below a threshold called *LowWatermark*, the system is said to be under memory pressure. At this point, a pager-daemon called *kswapd* is activated to reclaim pages (explained later). When the system is under memory pressure, applications are still allocated memory from the free pool until the number of free pages falls to the next threshold called *MinWatermark*. If the number of free pages falls below MinWatermark, VMM stops allocating pages to the applications. Only kernel allocations go through. This is to prevent applications from clogging all the memory and blocking the kernel from performing its other activities. Thus, the parameter MinWatermark determines the critical point when applications are denied memory.

As mentioned above, kswapd gets activated when the number of free pages falls below LowWatermark. The purpose of kswapd is to free pages i.e., to reclaim clean pages and write back dirty pages. Kswapd functions in two phases (Figure 4.1). In the first phase, it parses the active list, starting from its tail. As it scans, it checks the pages for *accessed* bit in the Page-table Entry (PTE) of the page (the bit is set in the TLB when an application generates a reference - hardware takes care of propagating the bit into the PTE). Pages that have been accessed (that have the accessed bit set) are moved to the head of the active list and those that have not been accessed are moved to the head of the inactive list. There is a limit on number of pages that are scanned in the active list. This limit is determined by a parameter called *ActiveInactiveRatio*. This parameter indirectly determines the ratio of the active list to the inactive list. Before moving pages from active list to the inactive list, it is also determined whether mapped pages (pages

that not pure file pages) should also be candidates for the move. This is determined using the parameter *VMSwappiness*.

The first phase of kswapd, therefore, is the movement of pages from the tail of the active list to the heads of the active and inactive lists. After this phase, kswapd enters the second phase where it starts scanning from the tail of the inactive list. All the pages that are accessed (note that pages in the inactive list also have their page-table mappings and accessed bit gets set even for these pages when they are accessed) are moved to the head of the active list. Now, among the pages that are not accessed, clean pages are reclaimed and sent to the free-pool and dirty pages are written back. The number of pages that should to be reclaimed in the inactive list is determined by another parameter called *HighWatermark*. When kswapd gets activated, it fixes itself a goal of reclaiming $target(= [HighWaterMark - NumFreePages])$ pages i.e., after kswapd is done with its job, the system will approximately have HighWatermark free pages. Kswapd goes back to sleep as soon as it reclaims its target number of pages. There is also a limit on the number of pages that kswapd scans in the inactive list. If kswapd is unable to reclaim target number of pages in one scan, then it increases this limit and scans more number of pages in the next pass. Thus, kswapd, in a single activation keeps increasing this limit and repeatedly scans more number of pages in the inactive list until it reclaims target number of pages. The rate at which this limit of scanning increases is determined by the parameter *PriorityIncrement*. After scanning the inactive list, kswapd is done with its job and goes back to sleep to be woken up again when the number of free pages falls below the LowWatermark and then the whole process repeats.

Page-reclamation also occurs through a direct-reclaim path. When the number of pages in the free-pool reaches below MinWatermark, applications, unable to allocate, start doing the

work of kswapd by themselves (in their context). They reclaim a handful of pages and proceed with their computation instead of waiting for kswapd complete its work. The number of pages which are reclaimed this way in the context of applications is the parameter *SwapClusterMax*.

Overall, the Linux VMM algorithm can be called as an approximate-LRU algorithm. A page that is placed at the head of the active list slowly trickles down to the end of the active list. If it is accessed in between it goes back to the head of the active list. Otherwise, it trickles down the inactive list and is finally reclaimed. It is to be noted that while this discussion is almost exact, some minor details are omitted for the sake of clarity and brevity. The above discussion briefly mentioned the parameters and their usage in the algorithm. Next section describes the parameters and their function at a greater detail.

4.2 Description and Qualitative Analysis of Parameters

In this section, we give a qualitative description of the parameters and their effects, classify them based on their functionality, summarize them and briefly describe which all parameters are exposed to the users. Note that while some of these parameters are easy to find since they are exposed through specific interfaces (such as `/proc`), others are buried in the code. In fact, some of them just appear as constants at a few places in the source code. In this thesis, we give names to all such parameters (based on their function and context) for the ease of discussion. We now go through all the VMM parameters, describing the function and possible influence on application performance for each parameter. We start with the first parameter, LowWatermark.

1. **LowWatermark:** LowWatermark determines the activation of kswapd. For our evaluation, LowWatermark is described using notation l. kswapd is activated when the amount

of free memory in the system falls below M/l where the ratio M/l is the LowWatermark. In the Linux Kernel, the number of pages that are to be reclaimed when number of free pages falls below LowWatermark is also a function of the LowWatermark. In other words, the ratio between the HighWatermark (which determines the number of pages to be reclaimed) and LowWatermark is fixed. Thus, changing LowWatermark not only affects the kswapd activation but also the number of pages that will be reclaimed on each activation. As the value of l increases, the LowWatermark (M/l) decreases, effectively giving more pages to the applications. Therefore, as we increase l, we can expect increase in the performance. However, on further increase, there would be too few free pages in the system and this would not only increase the number of times kswapd gets woken up but also the I/O time of the applications as they have to wait for pages to be written back to disk to get free pages (otherwise, this would happen in the background). Thus, increasing l can benefit applications to a certain extent but increasing it aggressively might hurt. The default value for l is 64.

2. **MinWatermark:** MinWatermark determines the critical condition when applications are denied memory and only the kernel has privilege to allocate physical memory. MinWatermark is always less than LowWatermark and we parameterize it as a percentage of LowWatermark. m denotes the percentage of LowWatermark which the MinWatermark is set to. As m increases, MinWatermark increases thereby reducing the scope for applications to allocate memory (under memory pressure). Therefore, a higher value of m would probably hurt applications (since it occurs only at extreme memory pressure).

3. **ActiveInactiveRatio:** ActiveInactiveRatio determines the number of pages that need to be moved from the active list to the inactive list on each activation of kswapd. Indirectly, this parameter determines the ratio of the length of the active list to the inactive list, expressed as percentage. The exact number of pages that move from the active list to the inactive list is given by the $T \times \frac{A*100}{I*a}$ where T is the target number of pages that need to be reclaimed by kswapd, A is the length of the active list, I is the length of the inactive list and a is the ActiveInactiveRatio parameter. The default value of this parameter is 200, which indirectly means that the active list is twice the inactive list. Note that pages in the inactive list are the only pages for direct reclaim. kswapd never directly reclaims pages from the active list. Therefore, having a larger active list (larger value of a) would make the pages stay for a longer time in the memory. If the application size fits in the physical memory but not in the active list, then increasing a would help such an application. At the same time, extreme values of a could leave very few pages in the inactive list (for reclamation) causing kswapd to get activated too often. This might increase kswapd overhead and worsen the performance of the application. On the other hand, if an application does not fit into the memory at all, then increasing a might hurt the application since the pages are unnecessarily kept in the active list.

4. **HighWatermark:** HighWatermark determines the number of pages that need to be reclaimed when kswapd gets activated. Pages will be reclaimed until the number of free pages reaches HighWatermark. HighWatermark is described in terms of h which denotes the value of HighWatermark as a percentage of LowWatermark. A value of 150 would mean that HighWatermark is one-and-half times LowWatermark. The default value in the

Linux Kernel is 150. A very low value of h can mean repeated activation of kswapd since it will not reclaim enough number of pages each time, thereby increasing the overhead of the system. A very high value of h can be too aggressive to throw out many useful pages at once, hurting application performance.

5. **Priority Increment:** Priority Increment is the rate at which the scan length of the inactive list is increased in case kswapd is unable to reclaim enough number of pages (it will reclaim more by scanning more). It also determines the number of times inactive list will be scanned. In other words, this parameter determines how the aggressiveness of kswapd increases each time it gets activated. We think that this parameter may not have too much impact on the performance of the applications. Either kswapd wakes up a few number of times, each time making more number of scans through the inactive list, or it wakes up very frequently, each time scanning fewer times. We think that ultimately, the total number of scans through the inactive list will even out. We refer to this parameter as i and its default value is 1.

6. **SwapClusterMax:** SwapClusterMax is the number of pages to be reclaimed in the direct reclaim path (when the system is under extreme pressure). Behavior of this parameter can be expected to be like HighWatermark. A high value would be too aggressive in reclaiming pages and a very low value will not serve its purpose. We refer to this parameter as s and its default value is 32.

7. **VMSwappiness:** VMSwappiness determines how aggressively mapped pages (pages that are not pure file pages) should be moved to the inactive list . If this parameter is 100, then all the mapped pages are candidates for swap and are moved to the inactive list. A low

value of 0-10 will try to keep in mapped pages as much as possible. The default value of this parameter is 60. A pure virtual memory intensive application might need a low value and an application which is more file based would prefer a high value for this parameter. This parameter is denoted by v.

8. **PageCluster:** While all of the above parameters come into effect during the page-reclaim process, there is also a parameter which plays role during the page-fault process. This is called *PageCluster* and represents the number of pages that need to be prefetched from the swap area (on the disk) during a page-fault. These extra pages can be prefetched without much extra overhead since there is no seek time involved in prefetching these pages. If an application exhibits good amount of sequentiality (if pages are accessed in sequence), then a high value for this parameter might benefit the application. On the other hand, irregular applications might be hurt with a high value for this parameter. We refer to this parameter as p and its default value is 8.

Classifying the Parameters

The parameters described above can be briefly classified into several classes based on their functionality.

Page-reclamation process can be viewed as three-step process - (i) Activation of kswapd (ii) Scanning the Active and Inactive Lists and (iii) Reclaiming the Pages (actually reclamation happens while scanning - we logically separate these two operations). Parameters come into play in each of these logical steps.

Table 4.1 shows the three categories of parameters. It can easily be seen, based on the discussion in the previous Section, as to which parameters fall into which category. Table 4.1

Class	Parameters
Activation-related Parameters	LowWatermark, MinWatermark
Scan-related Parameters	ActiveInactiveRatio, PriorityIncrement
Reclaim-related Parameters	HighWatermark, SwapCluster, VMSwappiness
Page-fault related Parameters	PageCluster

Table 4.1. Table showing a classification of VMM parameters based on their functionality

also shows another category called Page-fault related parameters. While the first three classes refer to the parameters that play active role in page-reclamation process (throwing out pages), PageCluster is used during a page-fault (while bringing in pages).

Table 4.2 summarizes all the parameters in the Linux VMM. The first column shows the parameter and the second column gives a brief functional description of that parameter. The default values of the parameters are shown in third column. Note that sometimes, for convenience, the notation used to denote the parameters does not directly denote the parameter but indirectly reflects its value. For instance, we use l to denote LowWatermark parameter, but the actual LowWatermark is M/l. This is just for convenience when we present quantitative results. Column 4 in the table shows this notation and finally, the last column shows whether that parameter is exposed to the users today. As we can see, the first five parameters are not exposed to the users. Such decisions as to what to expose are purely based on the intuition of the developers. We shall revisit this issue after the quantitative analysis section.

Parameter	**Brief Description**	**Default Value**	**Notation**	**Exposed**
LowWatermark	Determines Kswapd Activation	64	l	No
MinWatermark	Determines Critical Condition of few free pages	50	m	No
HighWatermark	Determines the number of pages to reclaim from the inactive list	150	h	No
ActiveInactiveRatio	Decides the ratio of the active list to the inactive list	200	a	No
PriorityIncrement	Decides the rate at which the scan length of inactive list is increased	1	i	No
VMSwappiness	Priority in reclaiming anonymous pages [100 $\Rightarrow$ aggressive]	60	v	Yes
SwapClusterMax	Number of pages to be reclaimed on the direct page-reclaim path	32	s	Yes
PageCluster	Number of pages to prefetch on a page-fault [disk-contiguous]	8	p	Yes

Fig. 4.2. Summary of various parameters in the Linux VMM

4.3 Quantitative Analysis of Parameters

4.3.1 Experimental Setup

Application	Brief Description	Physical Memory Used
dbt3	An Open Source Database application running TPC-H like Queries [17] (we executed 20 Queries in each run)	163MB
gzip	A common compression utility (also part of SPEC 2000) [7, 19]	95MB
apsi	A Meteorology application (Pollutant Distribution) (SPEC 2000) [19]	192MB
mummer	A Genome Sequence analyzer [14]	159MB
lucas	A number theory application used for primality testing (SPEC 2000) [19]	155MB

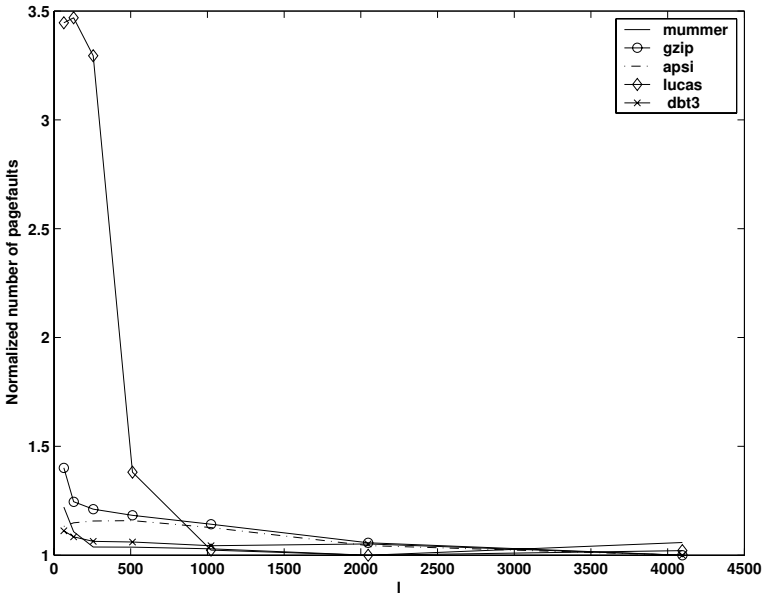
Table 4.2. Applications used

In this section, we present quantitative analysis of all the VMM parameters. All the experiments were done on a 2.8GHz Pentium 4 machine running Linux 2.5.74 Kernel. Since all the parameters were not exposed to the users by the default kernel, we provided hooks using the `/proc` filesystem to set the parameters dynamically as the system is running. We have chosen 5 applications from *different* domains for this study so that we do not make any biased inferences about effects of varying these parameters. Table 4.2 shows all the applications that were used. All the applications were executed onto completion. Table 4.2 also shows the amount of physical memory that was used with each application. Note that providing more physical memory is always a solution to improve performance. At the same time, there are always applications that do not fit into the memory. The goal here is to to observe the effect of these parameters when

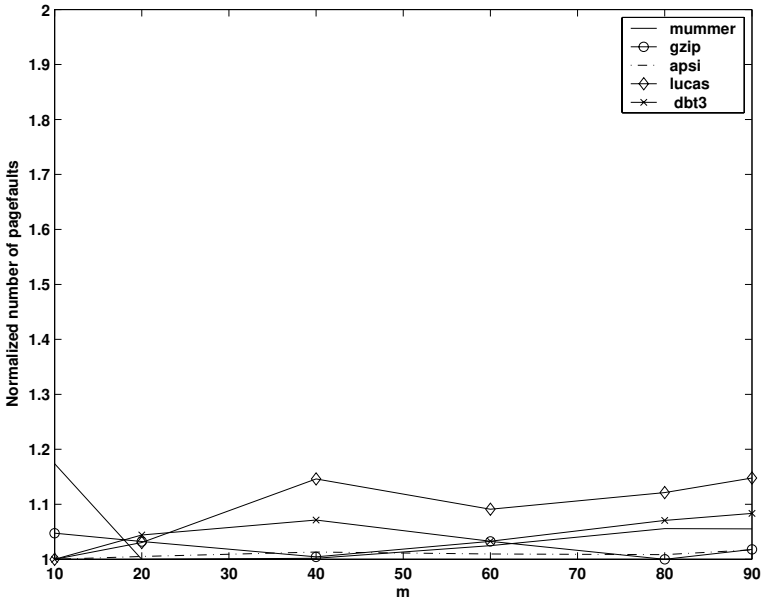
memory is a constraint. If applications fit well into the memory, no parameters of an OS will have effect on the application performance. At the same time, if the memory is too small so that applications incur large number of page-faults and become outrageously slow, again parameters of OS will not matter. It is the range in between where there is some amount of disk access and where applications proceed at a reasonable rate, that we feel is interesting. For this study, we observed the amount of memory that applications need and chose slightly lesser physical memory sizes so that there would be reasonable amount of disk access.

4.3.2 Quantitative analysis

In this section, we evaluate the effect of varying all the parameters on the applications. The main metric of comparison is the normalized number of page-faults across the range each parameter was varied. We have chosen number of page-faults (Major faults in Linux Kernel) as the main metric as opposed to the application run time because we think that number of page-faults is a metric that is more immune to noise in the system. runtime. In any case, for most of the runs of all the applications, we observed a direct correlation between the number of page-faults and the application runtime. So, our conclusions would not be any different even if we had compared application run time. While varying each parameter, all the other parameters are fixed (to their default value in the Linux Kernel) so that we can observe the effects of parameters individually. For each parameter, we chose a reasonable range that also covers the current default value in the Linux Kernel. We begin with Activation-related Parameters, starting with LowWatermark.



(a) Effect of LowWatermark



(b) Effect of MinWatermark

Fig. 4.3. Figure showing the effect of LowWatermark and MinWatermark

LowWatermark

To quantitatively see the effect of LowWatermark, we varied l from 64 to 4096, in powers of 2. Figure 4.3(a) shows the effect of varying l on all the applications. In the figure, y-axis is the normalized number of page-faults w.r.t. the minimum value that is attained over the x-range i.e., maximum performance is the point with y-value equal to 1. As we can see from the figure, increasing l (decreasing LowWatermark) helps all the applications. This is not just because more pages remain in the memory but that they also remain in for a longer time (applications effectively get more free memory). Increasing l to values greater than 1000 hardly benefits applications. This can be explained using the LRU stack distance table (Table 4.3) which was derived from application memory references that were collected using the Valgrind [20] tool. Valgrind instruments application binaries and generates memory references on the fly. We pass this stream of references to an LRU stack. Table 4.3 shows the frequency of references at LRU stack depth measured in MB i.e., a reference that was at a stack distance of 1024 (pages) would be treated as a reference at a depth 4 MB. This way, it is easy to see how many references will hit within the memory as parameters (like LowWatermark) are changed. Now, remember that the Linux VMM algorithm is approximate-LRU (as described in Section 4.2). It is for this reason that we can approximately explain the behavior of these applications using the LRU stack depth table.

For example, if we consider `apsi` which was executed with a physical memory of 192MB (with some memory being reserved for kernel code, data structures, DMA etc.), as the value of l increases beyond 1000, the amount of memory in the active list becomes close to 180MB. If we look at the columns for `apsi` in Table 4.3, we see that there are hardly any

references that are accessed from LRU stack depth of 180MB to 192MB. Also, as the value of l increases, the rate at which more pages are available to the application decreases. It is for these reasons that `apsi` does not gain much in performance on further increasing l. Similar explanations can be given for other applications as well using the LRU stack depth table. We show in Table 4.3, LRU stack depth information for three applications. It can be expected, on the other hand, that as l increases more, there would be performance degradation (as said in qualitative discussion of LowWatermark, Section 4.2). Though this does happen, it happens at extreme values of LowWatermark (= 10-15 pages) at which point the system becomes unbearably slow. We do not consider such values in our experimentation since we think they are unreasonable.

For `lucas`, the performance improvement is significant. `lucas` is an application that accesses addresses quite iteratively all over the address space. Access pattern graphs for `lucas` are shown in Figure 2.6. As we increase the LowWatermark, `lucas` fits more and more into memory thereby performing well. Iterative nature is not this high for other applications. As a result, they do not benefit much with increasing LowWatermark.

Overall, LowWatermark is an important parameter since it can effect performance of applications significantly. There exists optimal range for it which is good across all the applications. Fixing the values of such parameters (like LowWatermark) based on one time experimentation might be good enough.

MinWatermark

We now move on to the next parameter, MinWatermark. As said in Section 4.2, as m increases, MinWatermark increases thereby reducing the scope for applications to allocate memory under memory pressure. Therefore, a higher value of m would probably hurt applications.

mummer		gzip		apsi	
LRU Stack Depth	Frequency	LRU Stack Depth	Frequency	LRU Stack Depth	Frequency
120 - 124 MB	1355382	80 - 84 MB	105343	160 - 164 MB	130964
124 - 128 MB	1286964	84 - 88 MB	63317	164 - 168 MB	67121
128 - 132 MB	1208009	88 - 120MB	0	168 - 172 MB	71630
132 - 136 MB	1112525	$>$ 120MB	70913	172 - 176 MB	77352
136 - 140 MB	817316			176 - 180 MB	772758
140 - 144 MB	10312			180 - 192 MB	0
144 - 148 MB	3721			192 - 196 MB	13679
148 - 152 MB	2373			$>$ 196 MB	48918
152 - 200 MB	0				
$>$ 200 MB	37564				

Table 4.3. Table showing LRU stack depth frequency for some applications. This is a pure application characteristic that was derived from application memory references.

Figure 4.3(b) shows the effect of m on all the applications. We varied m from 10 to 90 (default value is 50). First observation from the figure is that MinWatermark is not as influential a parameter as LowWatermark. The maximum performance difference that MinWatermark can make is around 20% (for `lucas`). MinWatermark is not so influential because, once the number of free pages fall below LowWatermark, kswapd is activated and it starts freeing the pages in the background. Thus, its not too often that the number of free pages falls below MinWatermark for the application performance to be hurt significantly. We can see from the figure that the performance of applications is quite random as we vary this parameter. It is important to note that the system under consideration is quite complex. There are a number of events happening inside the kernel and it is not possible to monitor all of these. We attribute such minor variances (up to 10%) in performance to 'noise' in the system.

In all, although MinWatermark is an interesting parameter, it is 'shielded' from the above by the LowWatermark. Therefore, its not as influential as LowWatermark. From the view point of applications, setting this parameter to a non-extreme value seems to suffice.

ActiveInactiveRatio

We now move on to the next class of parameters summarized in Table 4.1, Scan-related parameters. ActiveInactiveRatio determines the number of pages that need to be moved from the active list to the inactive list. As discussed in Section 4.2, a very high value of this parameter might lead to a large kswapd activation overhead and a very low value might allow useful pages to be reclaimed.

Figure 4.4 shows the effect of a on all the applications. We varied a from 10 to 10000 (default value is 200). From the figure, we can make several observations. First, a is effecting all the applications significantly. There is at least a difference of 25% in performance for all the applications across different values of a. In fact, `lucas` suffers performance degradation of as much as 200% if the value of a is not chosen appropriately. Second, a more interesting observation, different applications have different optimal ranges for a. For `gzip` and `dbt3`, maximum performance is achieved at a value around 2000 where as values around 100 seem to be good for `apsi` and `mummer`. For `lucas`, 10 is the best value. The reason for this behavior, again, can be related to characteristics of the applications. In fact, the LRU stack distance Table (Table 4.3) can be used to explain the influence of this parameter as well. If increasing a captures higher frequencies in the Table 4.3, then the application will benefit. Otherwise, its performance will not change much or worsen. For example, if we take `mummer`, a value of a=1000 corresponds to LRU stack depth of 140MB and we can see from Table 4.3 that `mummer` does not have too many references after 140MB. Therefore, its performance drops on increasing a to values greater than 1000. Another interesting observation from Figure 4.4 is the performance of `lucas` which drops consistently. This is because the working set of `lucas` does not fit into the

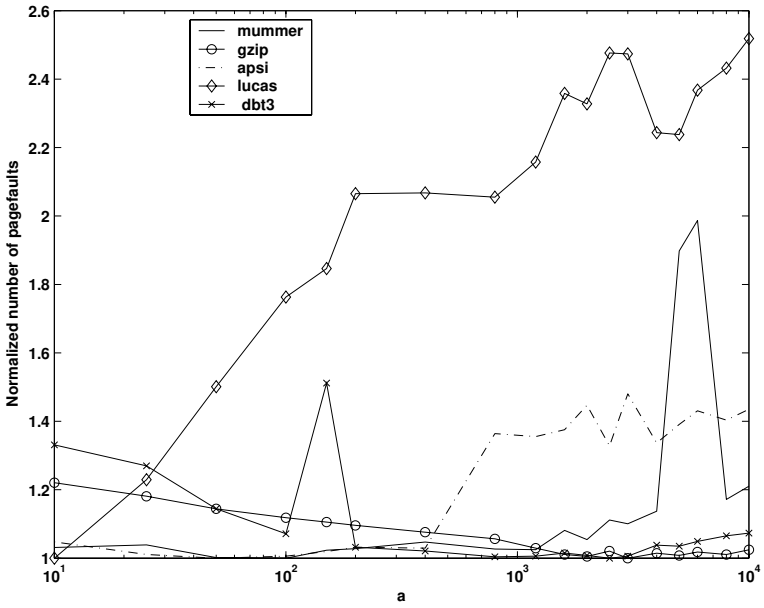


Fig. 4.4. Figure showing the effect of ActiveInactiveRatio on applications

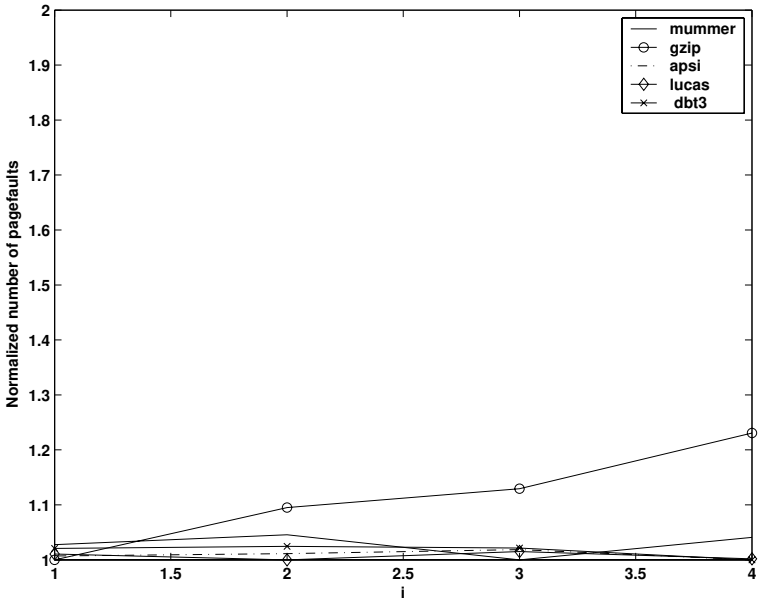
memory. Increasing a keeps pages in memory for longer time, only not to be used by `lucas` thereby dropping its performance.

Note that setting this parameter to a value based on one application would hurt others. At the same time, if all the applications are considered, there is no common optimal value. In fact, from the figure, we see that a value of 200 seems to be reasonable for 4 of the applications though `lucas` would still suffer. Interestingly, this is the default value in the Linux Kernel.

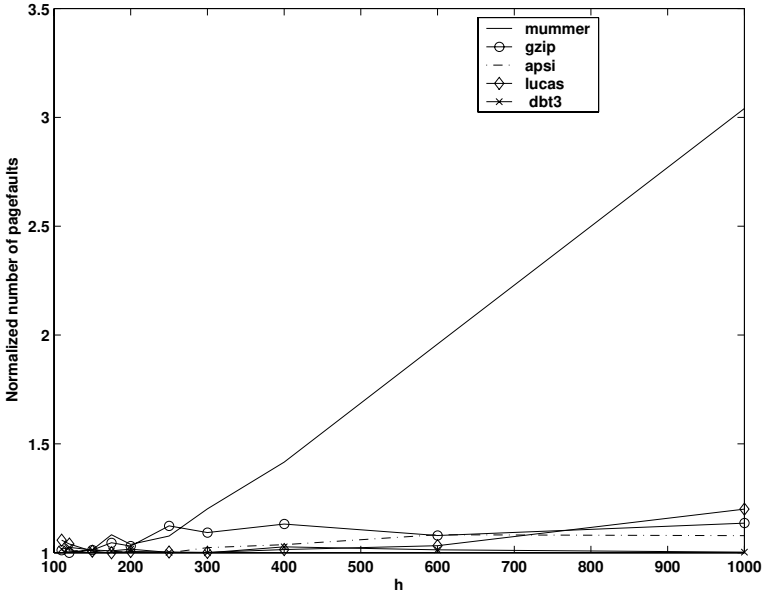
We think that a parameter like ActiveInactiveRatio, that has different optimal ranges across different applications is very interesting. It gives us motivation to set such parameters dynamically as the applications execute, by monitoring their characteristics. On the other hand, a tougher problem is that it may not be possible to monitor characteristics. Note that the LRU stack depth table was obtained from pure application memory references collected from the Valgrind tool. Such information will not be available in the operating system. Setting these parameters dynamically, by using only those characteristics that can be observed in the OS, we think is a challenging problem. We move on to our next Scan-related parameter, Priority Increment.

Priority Increment

Priority Increment determines the rate at which the scan length of the inactive list is increased and also the number of times the inactive list will be scanned. We already said in Section 4.2 that this parameter may not matter much. Figure 4.5 (a) shows the impact of this parameter on the application performance. Most of the applications get effected only up to 5% or so. It is only for `gzip` that this parameter makes any difference. The reason this parameter was not expected to have much impact was already pointed out in Section 4.2. Also, value of 1, which is the default value seems to be good for all the applications.



(a) Effect of PriorityIncrement



(b) Effect of HighWatermark

Fig. 4.5. Figure showing the effect of PriorityIncrement and HighWatermark

HighWatermark

We now move on to the next class of parameters, Reclaim-related parameters (Table 4.1). We start with HighWatermark. Figure 4.5(b) shows the effect of varying h. Except for `mummer`, h hardly has any effect on other applications. HighWatermark also has a range that is optimal across all the applications. Values around 100-200 seem to be the best. Incidentally, the default value of this parameter is 150.

The behavior of HighWatermark to an extent is also determined by other parameters. For example, VMSwappiness, which determines whether mapped pages should be reclaimed also affects number of pages that will be reclaimed. Setting a high value for HighWatermark and not reclaiming mapped pages will effect the applications in a certain way where as reclaiming mapped pages might affect in the opposite way (depending on the application). This raises an interesting issue of influence of multiple parameters simultaneously. These effects, though important, are not in the scope of this thesis.

VMSwappiness

VMSwappiness determines the aggressiveness to move mapped pages (pages that are not pure file pages) to the inactive list. As discussed in Section 4.2, a high value for v will imply that mapped pages will also be candidates to be moved to the inactive list. Once moved to the inactive list, the probability to reclaim these pages will increase. Most of these applications are virtual memory intensive rather than file intensive. Therefore, increasing v will only hurt the performance. Figure 4.6 shows the effect of changing v on the applications. As can be expected, almost all the applications suffer with increasing v. `mummer` is the only application that does not degrade much in performance because of some file activity (about 20MB of file activity).

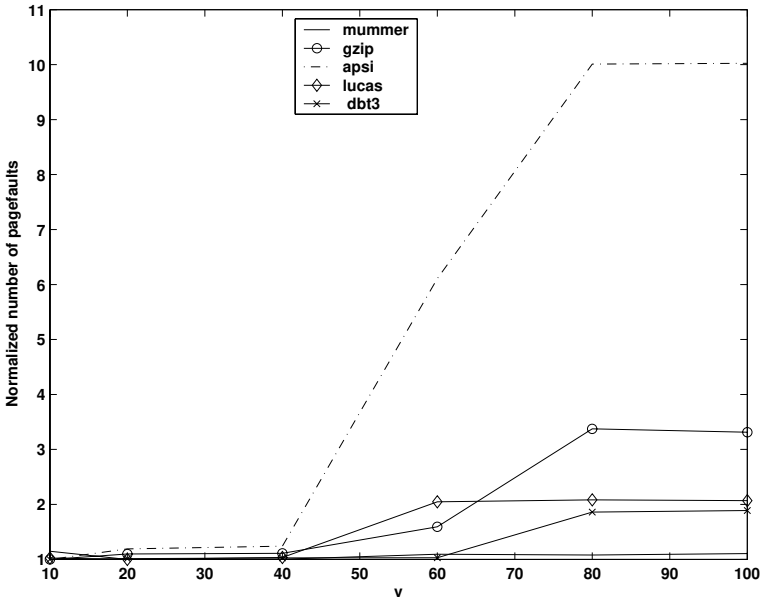


Fig. 4.6. Figure showing the effect of VMSwappiness

Though not seen in the graph clearly because of the scale, performance of `mummer` actually improves a little (actually, a small rising edge can be seen in the graph on the left, near y-axis) as v is increased to 30. This is because, file pages get some priority to stay back in the memory. But further increase degrades the performance slightly since anonymous pages become candidates to be reclaimed.

Overall, VMSwappiness is an important parameter that can effect application performance as much as 10 times! All the applications seem to have an optimal value range around 20-40. The default value of this parameter is 60. While one could say that setting this value after some experimentation to an optimal value would be good enough, the risk of severe performance loss, if the value is not optimal for some other application, becomes a big concern. Unlike LowWatermark (which also has same optimal range across all the applications) for which setting a wrong value does not affect the application performance significantly, setting a non-optimal value for VMSwappiness could kill the application performance. One should be more careful in dealing with such parameters. We will revisit the observations made here towards the end of this Section.

SwapClusterMax

SwapClusterMax denotes the number of pages that are to be reclaimed in the direct reclaim path, which happens when the number of free pages falls below MinWatermark (which signifies extreme memory pressure). This parameter is also used at some other places in the kernel but those cases are not relevant for this discussion. A very high value of SwapClusterMax could unnecessarily reclaim more pages from the inactive list. A very low value will not serve its purpose. Note that this parameter is quite important in extreme critical condition. Figure

4.7 shows the influence of this parameter. As expected, applications benefit initially but suffer on increasing s to very high values. Also note that performance degradation is around 25% for some applications. Values around 32-50 seem to be optimal. In fact, default value for this parameter is 32. If we observe closely, different applications do have different optimal values which motivates us to set this parameter dynamically by self-tuning mechanisms.

Page Cluster

The last parameter which we move onto is PageCluster. While all the other parameters come into effect in the page-reclamation path, PageCluster is used during a page-fault. This represents the number of pages that will be prefetched from the disk during a page-fault. Note that pages which are prefetched are those that are contiguous on the disk (in the swap area). These pages could belong to different processes or different areas of same process. Since we are dealing with single applications here, physical contiguity on the disk also corresponds to virtual address space contiguity to a large extent. As said earlier, a high value for this parameter could hurt applications that do not exhibit good sequentiality.

Figure 4.8 shows the effect of PageCluster (p) on all the applications. There are several observations that can be made from the figure. First, PageCluster is yet another important parameter that influences application performance significantly. Performance difference for applications is as much as 4 times. Second, increasing the value of p initially benefits all the applications but starts hurting some of the applications later on. We explain the reason for such influence of page-cluster using the page-fault characteristics of the applications.

PF Distance	1	2	≤ 4	≤ 8	≤ 16	≤ 32	≤ 64	≤ 128	≤ 256	≤ 512	> 512	Negative
mummer	133	52	641	157	157	0	6	5	15882	1484	20796	24700
apsi	33	3	344420	1696	93	0	3	231	612263	11688	503289	91786

Table 4.4. Table showing page-fault distances and their frequency for `apsi` and `mummer`

The characteristic of page-faults using which we explain the influence of PageCluster is called 'Page-fault Distance' (PF Distance). PF Distance is the difference between two consecutive page-faults in number of pages (i.e., (Y-X) if page-fault for page X is followed by page-fault for page Y). PF Distance can give an indication as to whether prefetching contiguous pages can be useful. Table 4.4 shows PF Distance for 2 applications `apsi` and `mummer` categorized into several bins. This data is derived from the page-fault data which we collected for the applications. If we look at `mummer` in Figure 4.8, performance of `mummer` does not get affected until p reaches a value of 64. This is because, although extra pages are prefetched on page-faults (prefetch is only initiated, need not necessarily complete), `mummer` does not have too much to gain, as can be seen from the Table 4.4 where the number of PF Distances which are less than 64 are quite less. At p = 128, mummer starts suffering drastically because, while pages are being prefetched aggressively, those pages are not at all useful as can be seen from Table 4.4, where only 5 new page-faults add up when the PF Distance is increased to 128. On further increasing p, `mummer` really starts benefiting. This can again be seen from the Table which shows that significant number of PF Distances are present for values > 128. Similar explanation can be given for other applications as well.

Note that though overall optimal range for p seems to be around 512, it is important to remember that all these experiments have been done in the context of a single application.

In multiple application scenario, increasing the value to such large values might degrade the performance of other applications.

Summary and Discussion

Parameter	Influential?	Exposed?	Same Optimal Range?
MinWatermark	No	No	Yes
PriorityIncrement	No	No	Yes
LowWatermark	Yes	No	Yes
HighWatermark	Yes	No	Yes
VMSwappiness	Yes	Yes	Yes
SwapClusterMax	Yes	Yes	No
PageCluster	Yes	Yes	No
ActiveInactiveRatio	Yes	No	No

Table 4.5. Summary of influence of parameters

Section 4.2 presented a classification of parameters based on their functionality. From an application performance view point, as well as from an OS developers view point, a classification based on the influence of these parameters would be more interesting. We classify the parameters into three categories based on their influence. Table 4.5 presents this summary. The first column of the table lists the parameters. The second column says if this parameter has significant influence ($> 20\%$) on the performance of at least one application. The third column shows if the parameter is exposed to the users today (otherwise, it is buried as a constant in the code!) and the last column shows if the parameter has same optimal value range across all the applications.

The first category of parameters that we classify are those that do not have significant impact on the performance of the applications. MinWatermark and PriorityIncrement fall under this category (first two rows of the table). Note that both the parameters have same optimal range for different applications. This means that it is not really necessary to expose them. In fact, Table 4.5 shows that both of them are indeed not exposed to the users. The second class of parameters are those that do have significant influence on the application performance but have same optimal ranges across various applications. LowWatermark, HighWatermark and VMSwappiness come under this category. Only VMSwappiness is exposed to the users. The other two parameters are constants which cannot be changed - happy news is that at least the default values of these constants in the Linux Kernel fall into the optimal range for all the applications. Finally, what we think are more interesting are the third class of parameters, which not only have significant effect on the application performance but also that different applications have different optimal ranges for these parameters. It is interesting to note that some of these important parameters are not exposed to the users. For instance, ActiveInactiveRatio is such a parameter that has not been exposed for tuning (we think it should have been). We are not blaming the OS developers for this but our point is that finding such parameters and exposing the necessary ones requires a lot of effort and experimentation, and it is not economical in terms of time and cost. While one could think that exposing all the parameters is a solution, it would only increase the state space (for tuning) for the users and confuse them even more. Exposing all the parameters is not a solution. At the same time, it does not mean that OS developers should spend months in determining which parameters to expose. This is not a solution too.

We think that the solution to this problem lies in developing and integrating algorithms into the system which will self-tune the parameters. Developing algorithms that observe application-OS characteristics (application characteristics that can be observed in the operating system) and translate these observations into policies that can self-tune these parameters, we think, is a challenging problem. Self-tuning parameters will not only relieve users of the burden of tuning but also allow OS developers to actually focus on the real developments in the system.

Pronouncements

Overall, on the basis of our qualitative and quantitative analysis, we make several statements with regard to the parameters:

- **Its not just the algorithm!** First of all, our study showed that its not just the algorithm that matters but also the parameters associated with the algorithm. Parameters affect the performance of applications by as much as 10 times in a few cases. Therefore choosing the right values for the parameters is extremely important for good system performance.

- **Correlation can only be approximate.** In our study, we have tried our best to explain the influence of parameters by correlating them to the properties of application references or application page-faults. Behavior at times cannot be explained using these characteristics - we attribute this to the noise component in the system. We think that in a such a complex system where there are a number of external events on which the system is dependent on, (for example, time interrupts, network packets, cache effects on the processor etc.), correlation can only be approximate.

- **Expose the right parameters!** Our study clearly shows the importance of exposing the right parameters. We understand that this is a non-trivial task (as we discussed above) but we see this as yet another impediment in improving system performance.

- **Dynamically Self-tuning has scope.** Parameters like PageCluster and ActiveInactiveRatio, which have application dependent optimal value ranges, are a solid motivation to develop techniques for self-tuning the parameters. Self-tuning not only has scope in terms of improving application performance but as said, will save a huge investment, both for users and OS developers.

- **Application characteristics can be useful.** If application chacacteristics can be used to explain the influence of the parameters, then they probably can also be used to tune the parameters. It should be noted that in our study, application characteristics were used to explain the behavior of about half of the parameters and that those half were very influential parameters. This gives us motivation to use these characteristics to dynamically set the parameters.

4.4 Conclusion and Future Work

As application sizes increase, memory management, whose goal is to provide efficient data access becomes all the more important. Memory management policies in an operating system have significant effect on performance of many applications. Approaches taken in the past have focused on improving the performance of VMM by developing new algorithms. In this thesis, we take a different approach and, through a study of VMM parameters, quantify that its not only the algorithm that matters, but also the parameters of the algorithm. The contributions

here are in identifying VMM parameters, classifying them based on their function as well as on their influence, analyzing them qualitatively and quantitatively, relating the influence of these parameters to application/application-OS characteristics, and finally providing solid motivation to dynamically set these parameters. We also believe this thesis contains suggestions and material that could be valuable to the open source community.

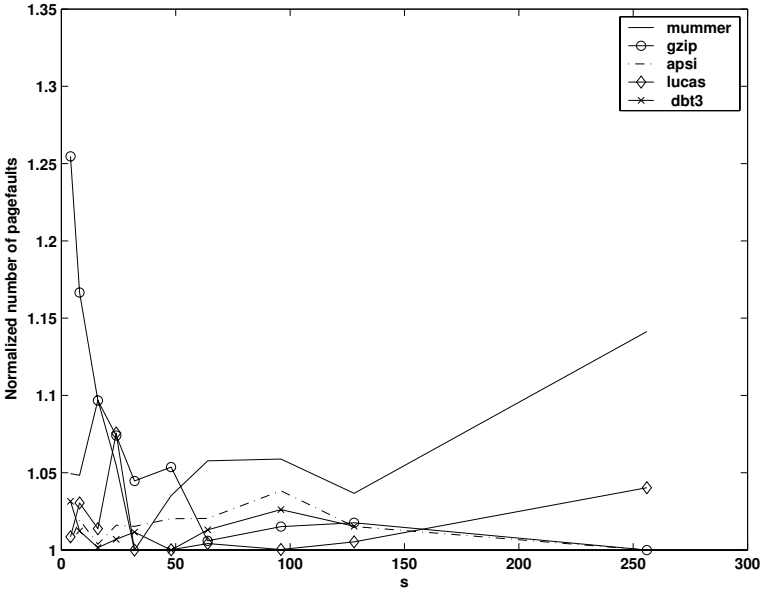


Fig. 4.7. Figure showing the effect of Swap Cluster

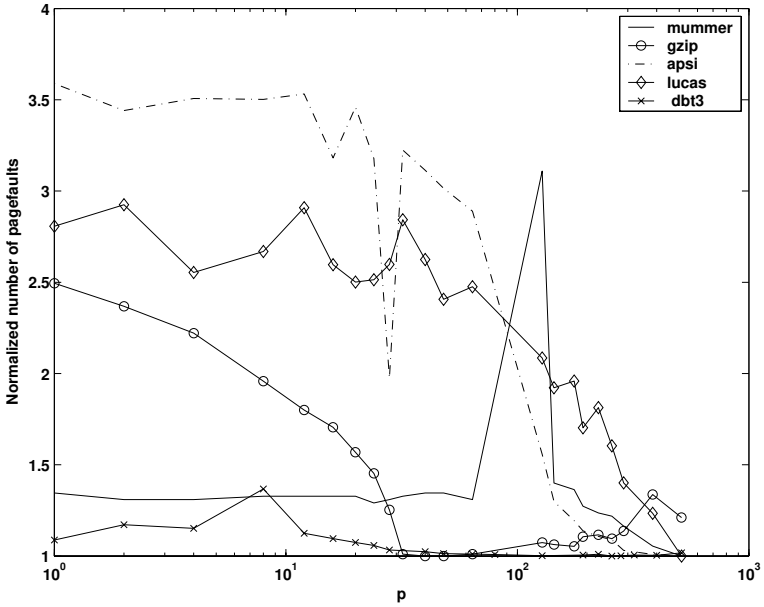


Fig. 4.8. Figure showing the effect of Page Cluster

Chapter 5

Conclusions

Increasing application sizes re-iterate the importance of effective and efficient memory management. Increasing application and system complexities stress the need for self-management. At the same time, diverse requirements of different applications ask for memory management optimizations to be provided at all the levels. The research presented in this thesis takes a step towards making memory management self-optimizing by proposing and evaluating memory management optimizations at all the levels.

Summary of Contributions

- **Hardware Enhancements for the TLB:** On the hardware front, this thesis first presents a thorough characterization for the TLBs. Using these characteristics as a solid base, it proposes a novel prefetching mechanism called Distance Prefetching that dynamically adapts to the application behavior and finally demonstrates its ability to outperform existing mechanisms.

- **Adaptive Dynamic Memory Allocation:** On the software side, this thesis proposes and evaluates a novel adaptive dynamic memory allocator that can tune itself to applications by observing their characteristics.

- **Towards a self-optimizing VMM:** Finally, on the OS front, this thesis undertakes a thorough study of several VMM parameters and their effect on the application performance.

It shows that a few parameters have significant impact on the performance of the applications, qualitatively and quantitatively analyzes their effect on applications, relates these effects to application/application-OS characteristics, and finally provides a solid motivation to self-tune these parameters dynamically.

In all, this thesis presents optimizations at different levels of memory management. However, the problem of making memory management self-optimizing is by no means solved. The remainder of this chapter presents some future directions in this area.

5.1 Future Research

Research presented in this thesis introduces several interesting problems. In particular, on the hardware side, applicability of Distance Prefetching (DP) to other domains remains an important question. While researchers have already started evaluating DP for caches [86], DP could also be applied in operating systems for predicting I/O requests etc. A very attractive feature of DP is that it is simple. Developing more sophisticated prefetching mechanisms using DP as basis, we think, is a challenging problem by itself. Distance Prefetching is a simple and powerful mechanism. Hardware resources required by DP are much less than those required by other prefetching mechanisms. A detailed evaluation of Distance Prefetching for caches using cycle-level simulations, investigation of trade-offs between hardware costs and performance benefits, and finally working with a product development team to incorporate DP into a processor is a possible direction for future research. The TLB work in this thesis focused on single applications with user-level memory references. Though it might be expected that multi-programming may not effect the performance of DP significantly (typically, the granularity of time given to

each application between context switching is large enough to cause thousands of TLB misses which are sufficient for DP to recreate its history and predict), an investigation of the effects of multi-programming and that of operating system references (that might disturb the application reference patterns) on DP is necessary before incorporating it into a real design.

The user-level adaptive memory allocator presented in this thesis can be extended to kernel. Many operating system kernels today do explicit slab management for good performance. A set of limited sizes and their frequent allocation makes kernel an important candidate for optimization using the adaptive allocator. Incorporating such an adaptive allocator into an operating system and evaluating it is a non-trivial task. Nevertheless, as operating systems get more complex, from a software engineering view point, this would be really beneficial to the operating system developers.

On the OS front, this thesis looked at stand-alone effect of each parameter of the Virtual Memory Manager of an operating system. Developing techniques to set these parameters dynamically is a very challenging problem and we think this requires significant amount of theoretical research as well. Influence of multiple parameters varying simultaneously would further complicate matters. In spite of our best efforts to keep the noise component minimal by isolating systems to the maximum, some cases of unexplainable behavior only re-iterate the complexity of such a system. In such a complex system, what we have presented are our observations that stand out at a coarse granularity. Dealing with such a system at a finer granularity and isolating the noise component will be very useful information to the community. This is one area where we hope to work in future. Moving to multi-programming environment would be a challenge in itself. Even if we come up with techniques to set these parameters dynamically in the context

of single applications, demonstrating their applicability in the presence of multiple applications, we think, is tough.

In this thesis, we have looked at adaptive optimizations at different levels in the memory management system. As systems get larger and complex, manually managing them is not only difficult and cumbersome but also not economical in terms of time and cost. As the world moves into a different era of developing self-managing systems, we believe that this thesis took a small step towards self-optimizing memory management.

References

[1] Apache webserver. http://www.apache.org/.

[2] Bugzilla: The Bug Tracing System. http://www.bugzilla.org/.

[3] Compressed Caching - Linux Virtual Memory. http://linuxcompressed.sourceforge.net/.

[4] Gnu awk. http://www.gnu.org/software/gawk/gawk.html.

[5] Gnu c compiler. http://gcc.gnu.org.

[6] Gnu make. http://www.gnu.org/software/make/make.html.

[7] The gzip home page. http://gzip.org.

[8] Hardware performance monitor (hpm) toolkit. http://www.sdsc.edu/SciApps/IBMtools/hpm.html.

[9] IBM Websphere Software Platform. http://www.ibm.com/websphere.

[10] K Desktop Environment. http://www.kde.org.

[11] Linux kernel mailing list. http://lkml.org.

[12] The linux operating system. http://linux.org/.

[13] Mozilla. http://www.mozilla.org.

[14] MUMmer: fast alignment of large-scale DNA and protein sequences. http://www.tigr.org/software/mummer/.

[15] Mysql database server. http://www.mysql.com/.

[16] Netscape Navigator. http://www.netscape.com.

[17] Open Source Development Lab Benchmarks. http://www.osdl.org/.

[18] Standard performance evaluation corporation. http://www.spec.org.

[19] Standard Performance Evaluation Corporation. http://www.spec.org/benchmarks.html.

[20] Valgrind - a tool for debugging and profiling x86-linux programs. http://valgrind.kde.org/.

[21] The Bluegene Project, 2001. http://www.research.ibm.com/bluegene.

[22] A. Agarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. *ACM SIGARCH Computer Architecture News*, 21(2):179–190, 1993.

[23] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, California, April 1991.

[24] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001.

[25] T. M. Austin and G. S. Sohi. High Bandwidth Address Translation for Multiple Issue Processors. 1996.

[26] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. pages 243–253, 1994.

[27] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *ACM SIGPLAN Notices*, 28(6):187–196, 1993.

[28] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory system characterization of commercial workloads. In *International Symposium on Computer Architecture*, pages 3–14, 1998.

[29] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[30] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, November 2002.

[31] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[32] J. Bonwick and J. Adams. Magazines and vmem extending the slab allocator to many cpus and arbitrary resources. In *Usenix Annual Technical Conference*, pages 15–33, 2001.

[33] J. Borkenhagen, G. Handlogten, J. Irish, and S. Levenstein. AS/400 64-bit PowerPC compatible processor implementation. In *Proceedings of the International Conference on Computer Design*, 1994.

[34] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.org.

[35] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[36] J. F. Canti and M. D. Hill. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.

[37] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. October 2001. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.

[38] Richard W. Carr and John L. Hennessy. Wsclock-a simple and effective algorithm for virtual memory management. In *Symposium on Operating Systems Principles (SOSP)*, pages 87–95, 1981.

[39] Donald D. Chamberlin, Samuel H. Fulier, and Leonard Y. Liu. A page allocation strategy for multiprogramming systems. In *Symposium on Operating Systems Principles (SOSP)*, pages 66–72, 1973.

[40] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 195–205, January 2000.

[41] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. pages 114–123, 1992.

[42] T. Chen and J. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[43] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. 3(1), 1985.

[44] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[45] Standard Performance Evaluation Corporation. http://www.spec.org.

[46] D. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kauffman Publishers, 1998.

[47] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[48] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. *International Conference on Parallel Processing*, pages 56–63, August 1993.

[49] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

[50] Peter J. Denning and Kevin C. Kahn. A study of program locality and lifetime functions. In *Proceedings of the fifth ACM symposium on Operating system principles*, 1975.

[51] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. In *Proceedings of the third ACM symposium on Operating system principles*, 1971.

[52] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, 1999.

[53] C. Dougan, P. Mackeras, and V. Yodaiken. Optimizing the idle task and other MMU tricks. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, pages 229–237, 1999.

[54] B. Zorn E. Berger and K. McKinley. Composing high-performance memory allocators. In *Proceedings of Programming Language Design and Implementation*, June 2001.

[55] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. Re-evaluating Online Superpage Promotion with Hardware Support. pages 63–72, January 2001.

[56] Z. Fang, L. Zhang, J.B. Carter, S.A. McKee, and W.C. Hsieh. Online superpage promotion revisited. In *Proceedings of the ACM SIGMETRICS 2000 Conference on Measurement and Modeling of Computer Systems*, 2000.

[57] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th MICRO*, pages 102–110, 1992.

[58] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, Volume 42, No 1, 2003.

[59] L. Gwennap. Hal reveals multichip SPARC processor. *Microprocessor Report*, 19(3):1–11, March 1995.

[60] P. J. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. R. Lebeck, and E. Parker. The combinatorics of cache misses during matrix multiplication. In *Journal of Computer and System Sciences*, August 2000.

[61] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[62] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001. http://www.research.ibm.com/autonomic/manifesto/.

[63] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. pages 39–50, May 1993.

[64] Intel Itanium. http://developer.intel.com/design/ia-64/downloads/24532002s.htm.

[65] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. pages 60–75, July-August 1998.

[66] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating System*, pages 295–306, 1998.

[67] L. K. John and A. M. Maynard. *Workload Characterization for Computer System design*. Kluwer Academic Publishers, 2000.

[68] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. 48(2):121–133, 1999.

[69] N. Jouppi. Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, 1990.

[70] AMD K-7. http://www.amd.com.

[71] Richard Y. Kain. How to evaluate page replacement algorithms. In *Symposium on Operating Systems Principles (SOSP)*, pages 1–5, 1975.

[72] M. T. Kandemir, J. Ramanujam, and A. N. Choudhary. Improving cache locality by a combination of loop and data transformation. *IEEE Transactions on Computers*, 48(2):159–167, 1999.

[73] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: An application driven study. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[74] Scott Kaplan. Compressed caching and modern virtual memory simulation, ph.d. thesis, university of texas at austin, december 1999., 1999. http://www.cs.amherst.edu/ sfkaplan/papers/index.html.

[75] D. Koppelman. Neighborhood prefetching on multiprocessors using instruction history. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[76] C. Lee, M. Potkonjak, and W. Magione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997. http://www.cs.ucla.edu/ leec/mediabench/.

[77] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, 1997. ACM Press.

[78] Mohammad Malkawi, Deborah Knox, and Mahmoud Abaza. Page replacement in distributed virtual memory systems. In *IEEE Symposium on Parallel and Distributed Processing*, pages 394–401, 1992.

[79] Mohammad Malkawi and Janek Patel. Compiler directed memory management policy for numerical programs. In *Symposium on Operating Systems Principles (SOSP)*, pages 97–106, 1985.

[80] E. P. Markatos. Visualizing Working Sets. In *Operating Systems Review*, May 1997.

[81] Takashi Masuda. Effect of program localities on memory management strategies. In *Proceedings of the sixth ACM symposium on Operating system principles*, 1977.

[82] G. McFarland. *"CMOS Technology Scaling and Its Impact on Cache Delay"*. PhD thesis, Computer Science Department, Stanford University, 1997.

[83] S. Mirapuri, M. Woodacre, and N. Vasseghi. The MIPS R4000 Processor. 12(4), April 1992.

[84] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design Tradeoffs for Software Managed TLBs. pages 27–38, 1993.

[85] G. Nakhimovsky. Improving the scalability of multithreaded dynamic memory allocation, July 2001. www.ddj.com.

[86] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA)*, 2004.

[87] A. K. Osowski, J. Flynn, N. Meares, and D. J. Lilja. *Adapting the SPEC2000 Benchmark Suite for Simulation-based Computer Architecture Research*. Kluwer-Academic Publishers, 2000. (papers from Workshop on Workload Characterization).

[88] C. H. Park, J. Chung, B. H. Seong, and D. Park Y. Roh. Boosting superpage utilization with the shadow memory and the partial-subblock TLB. In *Proceedings of the 2000 international conference on Supercomputing*, pages 187–195, 2000.

[89] J. S. Park and G. S. Ahn. A Software-controlled Prefetching Mechanism for Software-managed TLBs. *Microprocessors and Microprogramming*, 41(2):121–136, May 1995.

[90] J. Peir, Y. Lee, and W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Architectural Support for Programming Languages and Operating Systems*, pages 240–250, 1998.

[91] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[92] Dominique Potier. Analysis of demand paging policies with swapped working sets. In *Symposium on Operating Systems Principles (SOSP)*, pages 125–131, 1977.

[93] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187.

[94] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[95] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[96] E. Sadeh. An analysis of the performance of the page fault frequency (pff) replacement algorithm. In *Symposium on Operating Systems Principles (SOSP)*, pages 6–13, 1975.

[97] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. pages 117–127, June 2000.

[98] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. In *Proceedings of the International Conference on Computational Science (ICCS)*, May 2001.

[99] Yannis Smaragdakis, Scott Kaplan, and Paul R. Wilson. EELRU: Simple and effective adaptive page replacement. In *Measurement and Modeling of Computer Systems*, pages 122–133, 1999.

[100] E. Smirni and D. A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. In *Performance Evaluation*, pages 27–44, 1998.

[101] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. pages 24–35, 1993.

[102] Pointer-Intensive Benchmark Suite. http://www.cs.wisc.edu/ austin/ptr-dist.html.

[103] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using Superpages backed by Shadow Memory. pages 204–213, 1998.

[104] M. Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy* . PhD thesis, Dept. of CS, Univ. of Wisconsin at Madison, 1995.

[105] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. pages 171–182, 1994.

[106] Y. Tian, E. Sha, C. Chantrapornchai, and P. Kogge. Optimizing page replacement for multiple-level memory hierarchy, 1998.

[107] Etch Traces. http://memsys.cs.washington.edu/memsys/html/traces.html.

[108] S. VanderWiel and D. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 1999.

[109] K. Vo. Vmalloc: A general and efficient memory allocator, 1996.

[110] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[111] K. W. Cameron Y. Luo and O. M. Lubeck. Instruction-level performance characterization of computational physics and multimedia applications using performance counters. In *Second Workshop On Computer Architecture Evaluation Using Commercial Workloads*, January 1999.

[112] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs, 1992.

Vita

Gokul Bhargava Kandiraju received his B.Tech degree in Computer Science and Engineering from the Indian Institute of Technology - Madras, India in the year 1999. He is expecting his Ph.D degree in May, 2004 from the Department of Computer Science and Engineering at the Pennsylvania State University. He spent the summers of 2001, 2002 and 2003 at the IBM T. J. Watson Research Center, New York. His research interests mainly include Computer Architecture, Operating Systems, Memory Mangement, Modeling and Self-Tuning. He is a student member of ACM and an active member of the Art of Living Foundation.