AUTOMATED COMPILER DRIVEN SUPERPAGE ALLOCATION AND ITS APPLICATIONS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Joshua A. Magee, B.A.

San Marcos, Texas
May 2009

For my wife Nele, without whose support and patience this thesis would not be possible.

ACKNOWLEDGMENTS

First and foremost I would like to thank Apan Qasem, my thesis advisor, whose advice and guidance made this research possible. Many thanks go to my committee, Carol Hazlewood and Xiao Chen, whose patience and support is greatly appreciated. I am especially indebted to Carol, who put me into contact with Apan while I was floundering in an attempt to find the right faculty member with whom to do research.

Without a doubt I owe a huge debt to my parents, whose support (in more than one aspect) allowed me to pursue my graduate degree.

Last but not least, I would like to thank Jason Cade for many long discussions during the off hours in the tutoring office.

This manuscript was submitted on December 11th, 2008.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF ALGORITHMS

Algorithm Page

ABSTRACT

AUTOMATED COMPILER DRIVEN SUPERPAGE ALLOCATION AND ITS APPLICATIONS

by

Joshua A. Magee, M.S.

Texas State University-San Marcos

May 2009

SUPERVISING PROFESSOR: APAN QASEM

The translation look-aside buffer (TLB) can represent a significant performance bottleneck in modern microprocessor-based systems. The amount of memory available to a system is continuously increasing due to the abun-

dance and affordability of RAM, yet the size of the TLB has grown very little. The increasing ratio of memory page entries to TLB entries has resulted in an increase of TLB misses. Given that TLB misses present a substantial bottleneck to system performance, the need to reduce the pressure placed upon the TLB is well justified. Superpages are one method that aims to extend the *reach* of the TLB and therefore reduce the number of misses.

Superpages are supported at both the hardware and software level on most modern microprocessor-based systems. Previous research has studied the usage, management, and implications of superpages from an architectural and operating system perspective, but there has been no research of superpages from the compiler perspective.

This thesis presents a strategy for compiler-driven superpage allocation. Judicious usage of superpages can improve system performance by reducing the number of TLB misses, but indiscriminate superpage allocation can result in page fragmentation and increased application footprint. A significant advantage afforded by a compiler driven superpage allocation strategy is the availability of data-reuse information within an application, a luxury that architectural and operating systems lack. The compiler strategy employs data-locality analysis to estimate the TLB demands of a program and uses this information to allocate superpages only when beneficial. If the compiler determines that it is prudent to use superpages then an optimization is performed that replaces all memory allocation with a custom *malloc* implementation. This *malloc* implementation is superpage-aware and supports both statically and dynamically determined superpage allocation.

In addition to the advantages afforded by the compiler when making judicious use of superpages, superpages also present opportunities for opti-

mization to the compiler. Compiler optimizations attempting to reduce conflict misses, such as array padding, can benefit when used in conjunction with superpages. The fact that superpages allow for a predictable and contiguous allocation of memory allows for the profitability of data-locality optimizations to be increased.

Not only are superpages beneficial to application performance and compiler optimizations but they can also help in benchmarking and empirical tuning. To this end, a method of utilizing superpages to measure certain hardware parameters, such as L2 cache associativity, is presented.

The effectiveness of the strategy is demonstrated on two different platforms with different TLB configurations.

CHAPTER 1

INTRODUCTION

1.1 Motivation

The translation look-aside buffer (TLB) plays a critical role in improving application performance, particularly as application data footprints grow. It has been shown that increased TLB misses can not only degrade performance but may become the principal bottleneck in many data intensive applications [28, 35]. Given the importance of the TLB in performance critical pathways, a significant amount of research has focused on improving TLB behaviour. Superpages have been the most dominant strategy proposed.

Currently most micro-processor based systems support multiple page sizes. For example, the Alpha micro-processor platform provides 8K, 64K, 512K, and 4M pages, the x86 platform supports 4K and 4M page sizes, and the Itanium processor provides page sizes ranging from 4K to 256M [31].

The trade-offs between smaller and larger page sizes is a well known issue covered in most operating system and architecture textbooks. Smaller page sizes can lower internal and external fragmentation but at the cost of increased pressure on the TLB and a decrease in performance. Larger page sizes improve performance but with an increased risk of fragmentation [38].

Superpages are pages larger than the base page size. Furthermore, superpages must be contiguous in both physical and virtual address spaces. The allocated memory must be a multiple of the superpage page size. For example, on the x86 platform each superpage must be aligned to a multiple of 4M. Since each superpage represents only one entry in the TLB, the *reach* of the TLB is effectively extended [31].

While the judicious usage of superpages can improve application performance, indiscriminate usage can lead to an unwarranted increase in data footprint and can lead to internal page fragmentation. Strategies have been proposed to intelligently allocate superpages so as to obviate fragmentation and increased memory footprints. These proposed strategies for smart allocation and management of superpages have either been in the operating system or the architectural domain [31, 34, 15, 6, 3, 25]. Implementing informed superpage allocation strategies at the OS and architectural level is natural since it is at these levels that superpage support is implemented. However, due to its role in program analysis and in setting up the run-time environment, the compiler can make significant contributions to a discerned usage of superpages. The advantages of a compiler-based strategy for intelligent superpage allocation are many:

(i) Developer productivity and code portability The usage of superpages requires that the programmer complete a variety of steps. These steps vary between platforms, but in the worst case scenario require that the superpages are requested from the operating system using low level system

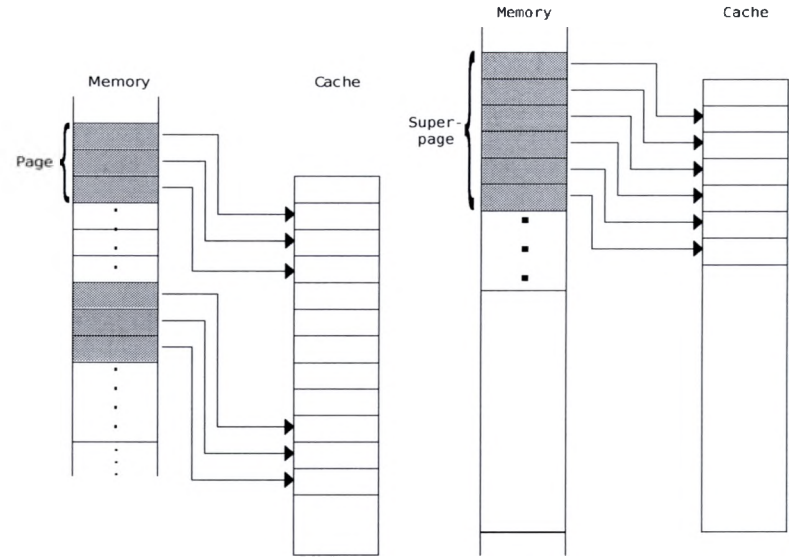
calls. Generally the memory is returned as one large chunk, requiring completely manual memory management. In the best case scenarios the programmer must take special steps to statically or dynamically link against runtime libraries. Regardless of the method used, it is the responsibility of the programmer to perform these actions. Compiler support for automated allocation of superpages relieves the programmer of this responsibility. Furthermore, the interface and usage of superpage varies between platforms and architectures. Applications designed to leverage superpages therefore become less portable and in order to increase said portability the programmer must have knowledge of all the different superpage implementations employed by various systems. By delegating the responsibility of superpage allocation to the compiler, the programmer does not require any knowledge of the underlying platform's mechanisms for superpage support; it is handled transparently by the compiler. This improves the usability, portability, and maintainability of code that employs superpages since the only consideration required by an application developer is whether the proposed compiler framework is available for the platform.

(ii) Enhanced information for allocation decisions One key advantage the compiler possesses over the operating system is the knowledge of the memory access patterns of an application. The operating system, when allocating memory for a process, can only consider the data footprint, or total required memory, of a program. The data footprint does not always correctly indicate the pressure an application places upon the TLB. Certain applica-

tions, such as *gzip*, can have a large footprint but nonetheless exhibit very low pressure upon the TLB, as will be shown in chapter 3. The actual TLB usage depends not solely on the data footprint, but on the data-reuse patterns of a program. In general, the number of distinct pages touched within a *working set* determines the TLB traffic for an application. Allocation decisions made without knowledge of the data reuse patterns are likely to be less effective. Since the *working set* information can only be derived through data-dependence analysis, only the compiler can take advantage of the reuse patterns. A compiler-based heuristic allows for a better tuned allocation strategy.

(iii) Increased effectiveness of memory transformations Many memory hierarchy transformations, such as ones aimed to reduce conflict misses, are most effective when the compiler has knowledge of how the data will be mapped to different cache lines. Since the majority of caches on modern architectures are physically-indexed, memory may not be contiguous and thus the compiler must guess at the most likely mapping to cache lines. The usage of superpages guarantees the contiguity of memory allocations and therefore eliminates the guesswork traditionally performed by the compiler. Figures 1.1(a) and 1.1(b) illustrate how superpages provide a contiguous mapping of memory blocks to cache blocks. This allows for the compiler to use a more effective heuristic in memory hierarchy transformations.

Apart from memory hierarchy transformations, the contiguous mapping of memory that superpages allow can be exploited for use in automatic tuning to identify certain hardware parameters. In particular, certain cache char-



(a) Base pages mapped to non-contigious cache lines (b) Superpages mapped to contigous cache lines

Figure 1.1: Virtual Memory Page Mappings

acteristics such as capacity and associativity can be determined. This information is critical to several different code optimizations but is nonetheless not readily available to the compiler. Methods have been proposed that utilize synthetic benchmarks to identify these parameters, however these methods are often limited by the inability to determine the parameters of physically-indexed caches [43, 46]. The contiguous allocation afforded by superpages can be applied for constructing a benchmark that estimates these hardware parameters with increased accuracy.

1.2 Vernacular

1.2.1 Superpage

Computer science is notorious for the plethora of terms used to describe a topic. Superpages are no exception to this peculiarity. In this thesis the term superpages is used to describe contiguous pages of a large size. Other terms that are often applied are *large pages*, *huge pages*, and *huge TLB*. Other variations may appear in common usage, such as *big pages*.

The terms *huge pages* and *huge TLB* are used in Linux nomenclature to refer to superpages. The term *huge TLB* may superficially appear to be misleading since the size of the pages is changed and not the size of the TLB, however the effect of the increased page size is that the effective size of the TLB is increased. In other words, larger page sizes mean that the *reach* of the TLB is extended. As such the term *huge TLB* speaks more to the advantages of superpage usage than the actual implementation. In this thesis the term *huge TLB* is only used when referring specifically to issues regarding the Linux implementation. Invariantly superpage is used as the general term and any other term is used to refer to an implementation specific detail.

1.2.2 Base page

The term *base page* is used to refer to the standard sized page supported by the underlying architecture. On the x86 platform a base page is 4K. Unless it is specified that superpages are in use, it is safe to assume that the

default page size is the base page size.

1.2.3 TLB Reach

The *reach* of the TLB refers to the amount of memory that is accessible from the TLB. It is defined as the number of TLB entries times the page size. It can also be said that the *reach* of the TLB is the number of pages that the TLB can translate at one time. This is also frequently referred to as the *range* of the TLB.

1.2.4 Acronyms and Abbreviations

Table 1.1 lists the definitions for frequently used acronyms and abbreviations.

1.3 Organization

The organization of this document is as follows: Chapter 2 discusses related work. Chapter 3 presents an overview of how superpages are implemented in hardware and software, discusses the compiler infrastructure used in this research, and presents an implementation for compiler driven superpage allocation. Chapter 4 describes the strategy for smart superpage allocation, presents a heuristic for estimating the TLB demands of an application, and provides an analysis of this heuristic. Chapter 5 discusses leveraging superpages in memory hierarchy optimizations, presents a strategy for array

Table 1.1: Acronyms and Abbreviations

Abbreviation	Full Term	Definition
AMD	Advanced Micro Devices	A manufacturer of microprocessors that are instruction set compatable with Intel Used in the thesis to refer to the architecture platform and not the company
API	Application programming interface	An interface for application programs provided by libraries
GCC	GNU Compiler Collection	A free and open source suite of compilers for various languages
gcc	GNU C Compiler	Used to refer specifically to the C language compiler found in GCC
IR	Intermediate Representation	In terms of compilers, intermediate representation refers to a representation of code that is at a lower level than the source language but at a higher level then the destination format
ISM	Intimate Shared Memory	A Sun Solaris shared memory facility
L2	Level 2	L2 Cache refers to the level 2 cache, the second level of cache on a microprocessor
LLVM	Low Level Virtual Machine	Refers to the Low Level Virtual Machine Compiler Infrastructure, see chapter 3
NP-hard	Nondeterministic Polynomial-time hard	Overly simplistic ly can be viewed as very hard problems See Garey and Johnson [12] for an overview of computational complexity theory
OS	Operating System	See Tanenbaum et al [38] for an in-depth overivew
SPEC	Standard Performance Evaluation Corporation	An organization that provides benchmarks Considered the *de facto* standard for benchmarking
SSA	Static Single Assignment	A form of IR where a variable is assigned exactly one time, A form condusive to optimization
TLB	Translation Look-aside Buffer	A processor cache used to improve the speed of virtual address translation See Hennessy and Patterson [18] for an overview
XCOFF	Extended Common Object File Format	An executable file format used primarily in the AIX operating system

padding, and presents experimental results. Chapter 6 demonstrates the effectiveness of using superpages in estimating hardware parameters. A synthetic benchmark for measuring L2 cache associativity is presented with experimental results. Chapter 7 concludes by outlining the key contributions of this work and discussing future work.

CHAPTER 2

RELATED WORK

2.1 Superpages

There has been significant work dedicated to improving TLB performance with both software and hardware strategies. Hardware approaches have primarily focused on either modifying the TLB organization or extending the existing TLB architecture. Talluri and Hill proposed a TLB organization based upon *partial sub-blocks* that can extend TLB coverage with minimal operating system support [37]. Fang et al. have developed a two level address translation mechanism that allows for multiple smaller pages to be placed into a larger page [11].

A great deal of research has been performed regarding superpages from an Operating Systems perspective. Tanenbaum discussed the trade-offs of page size selection in his seminal textbooks [38]. Gopinath et al. discussed policies for managing page sizes and evaluated algorithms for page size selection. While Gopinath's research focused a great deal on the NP-hard problem of analysing memory reference patterns, it also provided a great deal of insight into alleviating TLB miss penalties through optimal page sizes [14].

Navarro et al. provided an in depth look at superpages and proposed

an effective superpage management system [31]. Their work with superpages laid the groundwork for all future research in the area. Shimizu et al. extended the work of Navarro et al. by providing an implementation of superpages for Linux and analysing its performance. They found their implementation to yield improvements of performance with gains up to 6 times in some cases [34].

Kadayif et al. proposed strategies to reduce the amount of data TLB lookups in code transformations so as to optimize data access. Their research is unique in that it attempted to reduce the number of data TLB lookups via compiler directed address generation [19]. This research provided a nice example of symbiosis between compiler and Operating System research.

Gorman and Healy proposed a policy for allocating superpages to reduce external fragmentation [15, 6, 3]. Lu et al. proposed modifications to the Linux kernel for mapping application text regions to superpages for enterprise workloads [25].

All of the previous research for improving TLB performance with superpages have been operating system centric. There has not been any attempt to address the issue of superpage allocation and management from a compiler perspective. Our work proposes a compiler driven strategy and is complementary to any of the operating system based approaches.

2.2 Locality Optimizations

Research in locality optimization is as old as compilers themselves, thus it would take volumes to cover the milestones and achievements of the field.

Presented here is only a small portion of recent and relevant research.

Locality optimizations attempt to reduce the amount and capacity of conflict misses through compiler driven code transformations. Temam et al. demonstrated how cache interference phenomena can degrade cache performance, thus establishing the need to reduce cache misses [39, 40]. Bacon et al. proposed an algorithm for finding optimal padding amounts to eliminate set conflicts and offset conflicts in order to provide a uniform spread of cache misses [2]. Mitchell et al. have looked at improving TLB performance through hierarchical tiling [30]. Lynch et al. discussed cache miss rates with conventional page sizes and explored methods of improving these rates by using page coloring algorithms [26]. Their research explored a complementary approach to addressing many of the same issues addressed by superpages.

Rivera et al. examined the effectiveness of inter-variable and intra-variable padding for eliminating conflict cache misses. They presented several algorithms for data padding and analysed their effectiveness in minimizing cache conflicts [33].

Chatterjee et al. explored non-linear array layouts as a means of improving locality of reference. Their research explored improving locality of reference and performance by using transformations such as loop tiling to reorder computations [7].

Vera et al. discussed padding as a means to reduce conflict misses and provided an effective genetic algorithm to compute the optimal parameters. Their work built upon that of Rivera et al. and improved upon it through the use

of a genetic algorithm to find the optimal padding parameters [42].

None of the presented compiler based methods have attempted to employ the usage of superpages to enhance the effectiveness of their optimizations. One key feature of our compiler based approach is that it allows increased profitability of locality optimizations through superpage exploitation.

CHAPTER 3

IMPLEMENTATION

3.1 Introduction

This chapter opens by providing a brief overview of how superpages are supported at the architectural level as well as the TLB configuration of various micro-processor platforms. The hardware overview is followed by a survey of operating system support for superpages. Since Linux is the target test platform for this research it is explained in greatest detail. Next is an introduction to The Low Level Virtual Machine (LLVM) compiler infrastructure which is used in our compiler-based strategy. The bulk of this chapter details the implementation of our strategy. Finally experimental results are presented.

3.2 Hardware Support for Superpages

Modern micro-processors provide support for virtual memory via page tables that translate between virtual and physical addresses. These mappings are cached in a translation look-aside buffer, or TLB. Over the last decade the size of the TLB has increased at a much slower rate than memory. This is largely due to the fact that memory has drastically declined in price but TLB

hardware has remained comparatively expensive. On many processors the TLB may not cover more than a megabyte of memory, whereas one gigabyte of main memory and several gigabytes of virtual memory is common. Given the discrepancy between the coverage of the TLB and the size of memory, TLB misses are increasing in number and importance [21].

Superpages alleviate the demand on the TLB by increasing the TLB *coverage*. Micro-processors support superpages by providing multiple page sizes. Most modern architectures provide multiple page sizes, for example i386 supports 4K and 4M page sizes and ia64 supports a range of page sizes from 4K to 256M [41]. Historically multiple page sizes are common to micro-processor based architectures [38, 18], but it is a relatively recent discovery that larger page sizes can significantly improve the throughput of memory accesses by reducing the number of TLB misses [37, 14, 31].

Table 3.1 provides a cross-section of supported page sizes in modern micro-processors [31, 27, 29, 41, 37].

Table 3.1: Page sizes in modern architectures

Platform	4K	8K	16K	64K	256K	512K	1M	2M	4M	16M	32M	64M	256M	1G	2G	4G	16G
x86	√								√								
amd64	√							√	√								
ia64	√	√	√	√	√		√		√	√		√	√	√		√	
ppc32	√																
ppc64	√									√							
sparc64		√		√		√			√		√		√		√		√
mips	√		√	√	√		√		√	√							
power4	√									√							
alpha		√		√		√			√								

3.3 Software Support for Superpages

3.3.1 Linux

Superpage support is provided by the Linux operating system in the form of the HugeTLB kernel option. The support is built upon the multiple page size support provided by the underlying architecture, thus the size of the superpages is determined by the architecture and not by the Linux kernel.

The kernel provides a virtual file system, *hugetlbfs*, that provides an interface to access the superpages. All files created on this filesystem are backed by superpages. The primary purpose of this filesystem is to allow superpage-backed files to be *mmap*ped into memory. The utility of using *hugetlbfs* as a virtual ram disk, such as one would use *tmpfs* [36], is limited due to the fact that only read, but not write, system calls are supported. Figure 3.1 demonstrates how *mmap* can be used to manually request superpages.

In addition to the *mmap* interface, the kernel also allows for superpages to be used with shared memory. To request superpage-backed segments of shared memory the function *shmget* is called with a special flag *SHM_HUGETLB*. Using shared memory with superpages does not require that the *hugetlb* file system is mounted, but the user of the application must be in a group that has privileges to use superpages with shared memory. Figure 3.2 shows a simple program that uses shared memory with superpages.

The parameters of the *hugetlb* module in the Linux kernel is configured through *procfs*. *procfs* is a pseudo file system provided by Linux to access ker-

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>

#define HUGE_PATH "/mnt/huge/mem"
#define PROTECTION (PROT_READ | PROT_WRITE)
#define PAGE_SIZE (4*1024*1024)
#define NO_PAGES (5)
#define MEM_SIZE (size_t)(PAGE_SIZE*NO_PAGES)

#ifdef __ia64__
#define ADDR (void *)(0x8000000000000000UL)
#define FLAGS (MAP_SHARED | MAP_FIXED)
#else
#define ADDR (void *)(0x0UL)
#define FLAGS (MAP_SHARED)
#endif

int main(int argc, char **argv) {
        void *mem,
        int i,
        int super_fd,
        double *a,

        super_fd = open(HUGE_PATH, O_CREAT | O_RDWR, 07555),
        if(super_fd < 0) {
                fprintf(stderr, "Could not open HugeTLB\n"),
                exit(1),
        }

        mem = mmap(ADDR, MEM_SIZE, PROTECTION, FLAGS, super_fd, 0),
        if(mem == MAP_FAILED) {
                fprintf(stderr, "Mapping failed\n"),
                perror("mmap"),
                close(super_fd),
                unlink(HUGE_PATH),
                exit(1),
        }

        a = (double*)mem,
        for(i = 0, i < MEM_SIZE/sizeof(double), i++)
                a[i] = rand() * (1.0/rand()),

        /* this will generate a _lot_ of output */
        for(i = 0, i < MEM_SIZE/sizeof(double), i++)
                printf("a[%d] = %lf\n", i, a[i]),

        munmap(mem, MEM_SIZE),
        close(super_fd),
        unlink(HUGE_PATH),
        return 0,
}
```

Figure 3.1: Allocating superpages with *mmap*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>

#ifndef SHM_HUGETLB
#define SHM_HUGETLB 04000
#endif

#define PAGE_SIZE (4*1024*1024)
#define NO_PAGES (5)
#define MEM_SIZE (size_t)(PAGE_SIZE*NO_PAGES)

#ifdef __ia64__
#define ADDR (void *)(0x8000000000000000UL)
#define FLAGS (SHM_RND)
#else
#define ADDR (void *)(0x0UL)
#define FLAGS (0)
#endif

int main(int argc, char **argv) {
        void *mem,
        int i,
        double *a,
        int shared_id,

        if((shared_id = shmget(2, MEM_SIZE, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W)) < 0) {
                fprintf(stderr, "Error getting shared memory.\n"),
                exit(1),
        }

        if((mem = shmat(shared_id, ADDR, FLAGS)) == (void*)-1) {
                fprintf(stderr, "Error attaching shared memory.\n"),
                shmctl(shared_id, IPC_RMID, NULL),
                exit(1),
        }

        a = (double*)mem,
        for(i = 0, i < MEM_SIZE/sizeof(double), i++)
                a[i] = rand() * (1.0/rand()),

        /* this will generate a lot of output */
        for(i = 0, i < MEM_SIZE/sizeof(double), i++)
                printf("a[%d] = %lf\n", i, a[i]),

        (void)shmdt(mem),
        shmctl(shared_id, IPC_RMID, NULL),

        return 0,
}
```

Figure 3.2: Allocating superpages with Sys V shared memory

nel information and control configurable kernel options at runtime. Table 3.2 shows the various "tunable" parameters available in *procfs* related to superpages.

Table 3.2: *procfs* superpage attributes

emphprocfs entry	Access	Purpose
/proc/meminfo HugePages_Total	RO	Number of total superpages
/proc/meminfo HugePages_Free	RO	Number of free superpages
/proc/meminfo HugePage_Rsvd	RO	Number of superpages scheduled to allocate (reserved) but that have not yet been allocated
/proc/meminfo HugePages_Surp	RO	The number of "extra" superpages allocated (overcommited superpages)
/proc/meminfo Hugepagesize	RO	The size of a superpage
/proc/sys/vm/nr_hugepages	RW	The number of OS allocated superpages
/proc/sys/vm/nr_overcommit_hugepages	RW	The number of surplus superpages that can be reserved once the value in /proc/sys/vm/nr_hugepages is exceeded
/proc/sys/vm/hugetlb_shm_group	RW	The group ids (GID) of groups allowed to use shared memory with superpages
/proc/sys/vm/hugepages_treat_as_movable	RW	Superpages are not moveable, but setting this paramater will force superpages to be treated as moveable This can be used to obtain superpage from the *ZONE_MOVEABLE* pool

Linux must allocate superpages for the system before they can be requested by an application. The memory is preallocated and reserved for superpages, thus the memory reserved for superpages can only be used for superpages. Generally superpages should be preallocated during or shortly after boot, since it may be difficult to obtain sufficiently large contiguous chunks of memory on a long running system [41]. Figure 3.3 shows the *init* script used to set-up the usage of superpages on our test system.

```
#!/bin/sh
# chkconfig  2345 30 80
# description  Start and Stop superpage (HugeTLB) allocation
#
NUMBER_PAGES=64
MNT_POINT="/mnt/huge"
MODE=1777
PIDFILE=/var/run/hugetlb run
  /etc/rc d/init d/functions
start() {
        if [ -f ${PIDFILE} ], then
                echo "Allready running",
        else
                echo -n "Loading HugeTLB"
                [ -z "${MNT_POINT}" ] && MNT_POINT="/mnt/huge"
                [ -z "${NUMBER_PAGES}" ] && NUMBER_PAGES=0
                [ -z "${MODE}" ] && MODE=1755
                mkdir -p ${MNT_POINT}
                echo ${NUMBER_PAGES} > /proc/sys/vm/nr_hugepages
                mount -t hugetlbfs -o mode=${MODE} nodev ${MNT_POINT}
                [ -z $? ] && echo " failed." || echo "."
                echo 1 > ${PIDFILE}
        fi
}
stop() {
        if [ -f ${PIDFILE} ], then
                echo -n "Unloading HugeTLB"
                [ -z "${MNT_POINT}" ] && MNT_POINT="/mnt/huge"
                umount ${MNT_POINT}
                echo 0 > /proc/sys/vm/nr_hugepages
                [ -z $? ] && echo " failed." || echo ". "
                rm ${PIDFILE}
        else
                echo "Not Running"
        fi
}
case "$1" in
        start)
                start
                ''
        stop)
                stop
                ''
        restart|reload)
                stop
                start
                ''
        status)
                [ -f ${PIDFILE} ] && echo "Loaded" || echo "Not Loaded"
                ''
        *)
                echo $"Usage: $0 {start|stop|status|restart|reload}"
                exit 1
                ''
esac
exit 0
```

Figure 3.3: Linux init script for allocating superpages

libhugetlbfs

libhugetlbfs is a library, by David Gibson, Adam Litke, and others, to facilitate easy access to superpages [13]. *libhugetlbfs* provides library utilities and the ability to remap data segments to superpages, but the primary contribution of the library is a superpage-backed *morecore*. Application developers can utilize the library by either linking directly to it or by setting environmental variables to enable superpage allocation at runtime. It achieves this by overriding *malloc*'s standard *morecore* with one that provides chunks of memory backed by superpages.

One advantage of using *libhugetlbfs* is that it allows superpages to be used without any changes to the source code. While the library is definitely easier to use than using *mmap* or shared memory to access superpages, it nonetheless involves a procedure that may be undesirable to many programmers since it requires either custom linking or setting up a specialized runtime environment (*id est* with environmental variables). Furthermore it makes it difficult for developers to distribute binary executables with built-in superpage support [1].

Our strategy overlaps with some of the support provided by *libhugetlbfs*, however there are key differences:

libhugetlbfs is a convenience library. It does not provide any mechanism to determine when it is profitable to use superpages. Our compiler-driven strategy provides easy access to superpages, compile time heuristic analysis, and

[1] Note the converse is also true One nice feature of *libhugetlbfs* is that it can be used to add superpage support to any program installed on the system with extreme ease

run-time heuristic analysis.

libhugetlbfs is independent of the compiler. Given the overlap it seems logical to employ the functionality provided by *libhugetlbfs* to implement compiler driven superpage allocation. Unfortunately one of the key features of the library also made it ill-suited for this purpose: *libhugetlbfs* provides superpage support for *malloc* solely at runtime. Even when an application is linked against the library, the decision to use the custom version of *morecore* is made at runtime based upon the value of *HUGETLB_MORECORE*. It is possible for the compiler, based upon its data-reuse analysis, to inform both the linker and runtime environment to use *HUGETLB_MORECORE*, but there is no guarantee that superpages will actually be used. It is a one way street for the compiler: it can still help to judiciously orchestrate superpage allocation, but it cannot safely exploit superpages in compiler optimization since the usage of superpages, and thus the contiguity of memory, is not guaranteed.

Ultimately *libhugetlbfs* is an excellent utility for utilizing superpages. It was considered for use in our compiler-based strategy but ultimately did not meet all of the requirements. Nonetheless it was beneficial source of reference.

Linux implementation of Navarro [31]'s superpage prototype

A recent project opened on sourceforge.net in October of 2008 proposes an implementation of *transparent* superpage support for Linux. The implementation would be a port of the prototype developed by Navarro et al.

for FreeBSD [31]. At the time of writing, however, this project has not released any code or documentation apart from a statement of purpose [44].

3.3.2 FreeBSD

The FreeBSD Operating systems is scheduled to include *transparent* superpage support in version 8, scheduled to be released in early 2009. Navarro et al. [31] provided a prototype implementation of *transparent* superpage for FreeBSD running on alpha and ia64 architectures. This prototype is being extended to include other architectures and will be integrated into the FreeBSD kernel [9]. Transparent support for superpages is a pure operating system centric approach. At the time of writing it is unknown whether *explicit* support, such as one that could be leveraged in a compiler-based strategy, will be provided, but it is expected that at some point explicit usage of superpage will be supported in FreeBSD.

3.3.3 Solaris

The Solaris operating system supports superpages from version 9 and up on both UltraSPARC and x86 platforms. Solaris support for superpages is quite sophisticated and provides a great deal of flexibility to developers. Older versions of Solaris (version 8 and earlier) allowed the usage of superpages only through a shared memory system termed *intimate shared memory (ISM)*. Version 9 and onward retain the ability to request superpages with ISM, but introduces two new approaches. Solaris 9 allows for superpage-backed memory

to be used in mapping with */dev/zero/*. This allows for superpages to be allocated using *mmap*. Solaris 9 also introduces the *MPSS* library which allows for the usage of superpages to be specified through the command *ppgsz*. The Sun Forte compiler provides the ability to request superpage allocation with the usage of the compiler flag *-xpagesize=n* where n is the size of the page to use. The compiler will attempt to allocate memory using the specified page size but makes no guarantee that the request will be honoured [29]. The Forte compiler provides a similar functionality to our compiler-based approach, but unlike our approach it does not:

- Use a heuristic to intelligently allocate superpages. The page size must be explicitly requested by the programmer at compile time.
- Exploit the usage of superpages for compiler optimizations.
- Run on operating systems besides Solaris. The design of our compiler-based strategy can be ported to many operating systems and architectures, Sun Forte only runs with Solaris on UltraSPARC or x86 platforms.

3.3.4 AIX

The AIX operating system supports superpages for IBM's *power4* processor line. The *power4* platform provides two page sizes: 4K base pages and 16M superpages. Superpages can be requested in AIX by modifying the *XCOFF* header in an executable to specify that heap and data segments should be backed by superpages, by setting up specific environmental vari-

ables that override the *XCOFF* header, or by using shared memory [27].

3.3.5 Microsoft Windows

Recent versions of Microsoft Windows, such as Windows Server 2003, Windows Vista, and Windows Server 2008, support superpages. Superpages are obtained using an approach similar to that of *mmap*. The function *CreateFileMapping* is called with a flag of *SEC_LARGE_PAGES* to request a mapping of superpage backed memory [8]. Given that the approach employed by Windows is similar to that used on Linux, the proposed compiler-based strategy would be able to support recent versions of Microsoft Windows operating systems.

3.3.6 Overview

Table 3.3 shows the current superpage support status of a variety of operating systems. It also marks entries that currently implement and/or are capable of supporting our compiler strategy.

Table 3.3: Overview of software support for superpages

OS	Superpage support	Currently Supported	Can be supported
Linux	Yes	Yes	N/A
FreeBSD	Scheduled in next release	No	Unknown
Solaris	Yes	No	Yes
AIX	Yes	No	Yes
Microsoft Windows	Yes	No	Yes
Darwin (Mac OS X)	Unknown	No	Unknown

3.4 LLVM

The Low Level Virtual Machine Compiler Infrastructure is a system that provides a framework for language independent analysis and optimization, inter-procedural analysis, front-end development, and compile, run, and link-time optimizations. LLVM provides a low level virtual machine that supports a virtual instruction set, a compilation strategy, and a compiler infrastructure [16, 23].

The C and C++ front-end is based upon that of the GNU Compiler Collection (GCC). LLVM can generate native code, portable C code, LLVM bitcode, and Microsoft Intermediate Language (MSIL) code. A Just-In-Time compiler is provided for the emitted LLVM bitcode, allowing for extensive run-time analysis and optimization [23].

3.4.1 LLVM Compilation Strategy

LLVM uses *gcc* as the compiler frontend. For C based languages *gcc* performs preprocessing, lexical analysis, parsing, semantic analysis, and all optimization that does not occur at the machine level. Most *gcc* optimization flags, including *-O[123]* are supported. The *gcc* version that LLVM provides is patched to output LLVM IR (intermediate representation) bit-code or assembly listings.

The intermediate representation is the heart of LLVM as it used in all phases throughout the LLVM compilation strategy. The IR code is a static single assignment (SSA) representation with low level operations that also is ca-

pable of representing high-level language constructs such as type safety, selection, iteration, and functions. The ability to retain high-level representations is something most assembly languages lack and is the very feature that makes LLVM IR suitable for intermediate level analysis. LLVM IR can be used in three different forms: an in-memory compiler IR, on disk bit-code suitable for usage in the LLVM just-in-time compiler, or a human parseable assembly listing.

The IR representation that is generated by the *gcc* frontend can be input for any subsystem of the LLVM compiler infrastructure such as optimization passes, analysis passes, alias analysis, and code generation. Passes are run using *opt*, the LLVM optimizer, which accepts as input the IR bit-code and a list of requested optimizations and analyses, and outputs the (possibly) transformed IR code. After any and all passes the resulting IR code can be run via two different mechanisms. The LLVM static compiler (*llc*) can be used to translate IR bit-code into native assembly for the specified architecture. The resulting assembly language output can subsequently be run through the system assembler and linker to provide a native executable. The IR bit-code can also be executed using the LLVM just-in-time compiler (*lli*). Figure 3.4 shows the various stages of compilation.

To facilitate the use of LLVM, a compiler driver tool is provided. *llvmc* and its successor *llvmc2* provide a one-stop tool for running all the necessary commands to transform high-level source code into a native executable or fully optimized IR bit-code suitable for execution with *lli*.

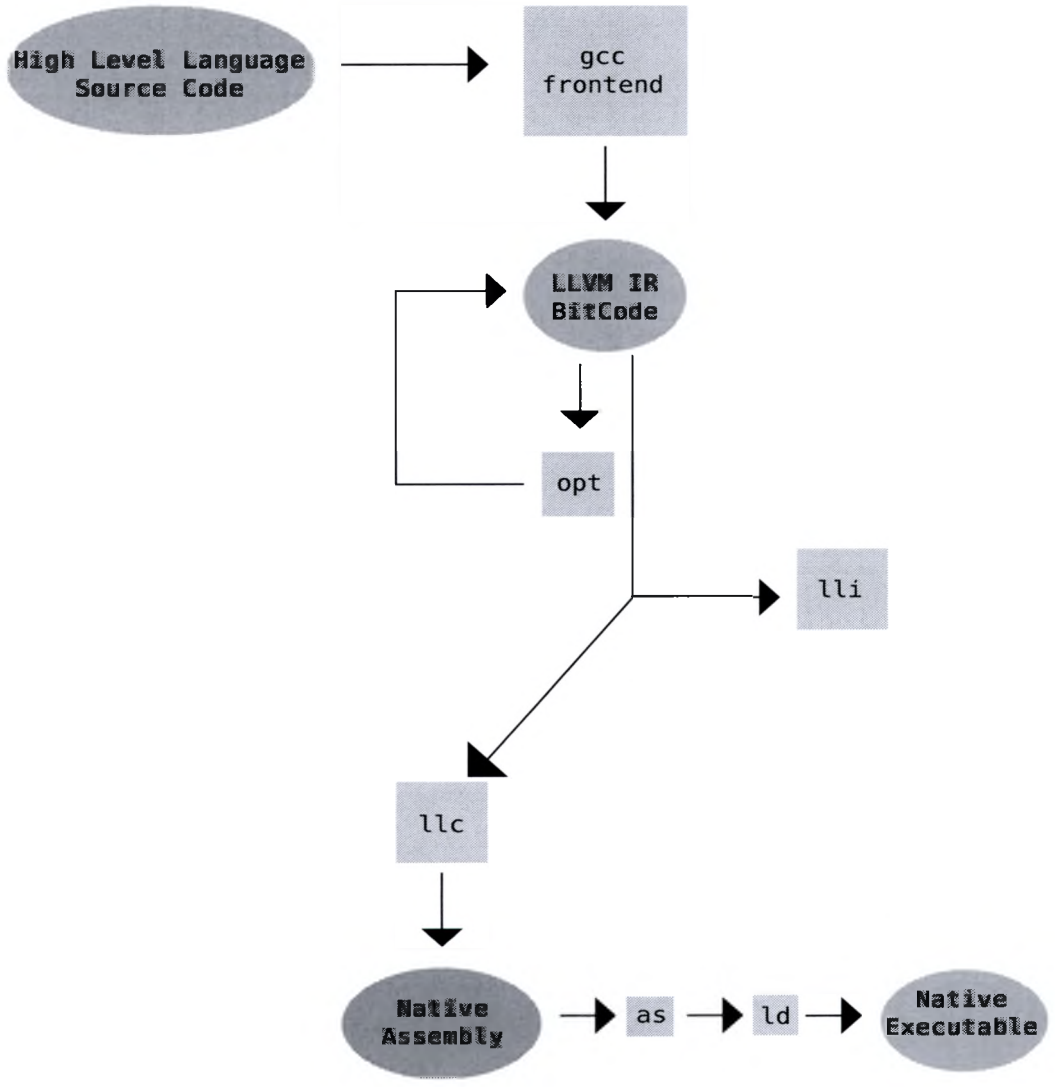


Figure 3.4: LLVM Compilation Strategy

Our research focuses primarily on C language code, however LLVM provides support for a variety of other languages including C++, Objective-C, Ada, and Fortran.

3.4.2 The Pass Framework

There are two approaches to implementing a pass in LLVM. A pass could be written for the *gcc* frontend. This approach does not necessitate the usage of LLVM; one would simply implement a pass in mainline *gcc*. The second approach takes advantage of the convenience, flexibility, and vigor of the LLVM compilation strategy by using the LLVM pass framework. Some of the key advantages the LLVM pass optimizer has over implementing a *gcc* pass are:

- The LLVM pass framework provides a high-level, well documented, and reusable pass environment written in C++.
- The LLVM IR representation is well defined and capable of representing high-level constructs.
- *gcc* provides an in-memory IR. LLVM's external IR allows for the IR assembly listing to be viewed before and after an individual pass.
- The LLVM framework is focused on providing a clean environment for compiler research and therefore expects that many of its users will be writing passes, performing analysis, and developing backends. *gcc*, on

the other hand, is focused on providing a production grade compiler with internals that the majority of the user base will never touch.

The LLVM pass framework is written in C++ and provides a number of classes that can be employed when writing a pass. Presented are some of the most frequently employed classes:

The ModulePass Class: This class is the most general of all Pass classes. The ModulePass is run on an entire module (or compilation unit) at once. Functions can be added and removed and their bodies can be modified, however there is no set order to how the functions will be traversed. Generally this pass is used when the entire program is considered during the pass but individual functions are not.

The CallGraphSCCPass: This class allows for a program to be traversed from the bottom up on the call graph. This pass is provided for when the optimization or analysis requires a bottom up traversal where *callees* are encountered before *callers*.

The FunctionPass: The FunctionPass class executes on each function in a program independently from the others. Function passes can only modify the currently processed function and cannot add or remove functions and globals in the module.

The LoopPass: This class runs on each loop independently. Loop nests are processed from inside-out such that the inner-most loop is processed first and the outer-most loop is processed last. Outer loops are permitted to update inner loops in a loop nest.

The BasicBlockPass: This class is similar to the FunctionPass class except that it executes on each basic block independently. Only the currently processed block can be modified and in general this pass follows the same semantics as the FunctionPass except that it is applied to a block instead of a function.

Figure 3.5 shows a simple FunctionPass that prints the name of each function that it encounters.

```
#define DEBUG_TYPE "printfun"
#include <llvm/Pass.h>
#include <llvm/Function.h>
#include <llvm/ADT/StringExtras.h>
#include <llvm/Support/Streams.h>
#include <llvm/ADT/Statistic.h>
using namespace llvm;

STATISTIC(function_count, "Counts number of functions encountered");

namespace {
        class printfun : public FunctionPass {
                public:
                        static char ID; // Pass identification
                        printfun(void) : FunctionPass((intptr_t)&ID) {}
                        virtual bool runOnFunction(Function &F);
        };

        /* Register the pass */
        char printfun::ID = 0;
        RegisterPass<printfun> X("printfun", "Print Function Pass");

        bool printfun::runOnFunction(Function &F) {
                std::string fname = F.getName();
                EscapeString(fname);
                llvm::cerr << "Function: " << fname << " encountered.\n";
                function_count++;
                return false; // function not modified
        }
}
```

Figure 3.5: LLVM pass that counts and prints functions

3.5 Compiler Driven Superpage Allocation

3.5.1 Superpage Aware Malloc

Background on Malloc

malloc is the C standard library interface for providing dynamic memory to an application. *malloc* is responsible for obtaining memory from the underlying operating system and managing the allocated memory for the application. In the glibc (GNU C Library) implementation of *malloc* memory is obtained with two different techniques.

sbrk() The *sbrk* library function adjusts the size of a program's data segment. On GNU systems *sbrk* is a convenience wrapper around *brk*. Both functions can be used to accomplish the same task. *malloc* uses *sbrk* to adjust the size of the heap.

mmap() The *mmap* library function maps memory from a file or device into memory. Instead of using the data segment of a program, traditionally treated as the heap, memory is mapped using *mmap* to obtain heap memory.

The glibc implementation uses *sbrk* to allocate memory smaller than a set threshold. This threshold, *MMAP_THRESHOLD* defaults to a value of 128K. Once the memory requirements exceed *MMAP_THRESHOLD* then the *malloc* implementation switches to obtaining further memory using *mmap*. The threshold can be adjusted by using the lesser-known *mallopt*, which allows the caller to adjust different parameters of the memory allocator. The parameters that are supported by *mallopt* are implementation dependent, so should be

used with caution. Adjusting *MMAP_THRESHOLD* to zero effectively disables memory allocations with *sbrk* and *malloc* will solely use *mmap*.

There are trade-offs between obtaining memory via *sbrk* and *mmap*. The *sbrk* method allows for a fine granularity over the allocated memory; generally the only limitation imposed upon the allocated units are that they are aligned to an 8-byte boundary. While *sbrk* is well suited for small allocations, large memory allocations can lead to excessive page fragmentation.

Heap allocation using *mmap* restricts chunk allocations to increments of the page size. If only one byte is requested via *malloc* an entire page must nevertheless be allocated. This method can result in wasted memory. On the other hand, while the *mmap* approach can still suffer from page fragmentation, it avoids the worst case behaviour of *sbrk*. One advantage of *mmap*ped memory, in addition to the reduced page fragmentation, is that it is immediately returned to the operating system upon being freed [22].

While the GNU implementation uses both strategies (*sbrk* for small allocations, *mmap* for large), other systems such as OpenBSD solely use the *mmap* method. Table 3.4 outlines the different allocation methods used by various implementations [4, 10, 20, 22].

The glibc implementation of *malloc* employs the function *morecore* to obtain chunks of free memory. It is *morecore*'s responsibility to obtain chunks of free memory via *sbrk*, *mmap*, or a custom supplied mechanism and subsequently extend or shrink the heap for *malloc*.

Table 3.4: *malloc* implementations

Malloc Implementation	*sbrk*	*mmap*
GNU/Linux	√	√
phkmalloc(FreeBSD, Mac OS X)	√	
jemalloc(FreeBSD)	√	√
OpenBSD		√
Hoard		√
smalloc		√

smalloc

smalloc (super malloc) is a superpage-aware implementation of malloc. It is based upon the algorithms presented in Doug Lea's *malloc* implementation which is the basis for the version found in glibc [24]. *smalloc* supports the *core* interface of the C standard, as detailed in table 3.5.

Table 3.5: *malloc* and *smalloc* interfaces

Standard malloc			**smalloc (Super malloc)**		
`void *`	`calloc`	`(size_t nmemb, size_t size)`	`void *`	`scalloc`	`(size_t nmemb, size_t size)`
`void *`	`malloc`	`(size_t size)`	`void *`	`smalloc`	`(size_t size)`
`void`	`free`	`(void *ptr)`	`void`	`sfree`	`(void *ptr)`
`void *`	`realloc`	`(void *ptr, size_t size)`	`void *`	`srealloc`	`(void *ptr, size_t size)`

Memory can be backed by either base pages, superpages, or to a limited extent both. *smalloc* requests memory by calling one of two versions of *morecore*. One version is for base pages and the other is for superpages. Base pages are obtained by *mmap*ping */dev/zero* and superpages are obtained by *mmap*ping a file backed by the *hugeTLBfs* provided by Linux. Each

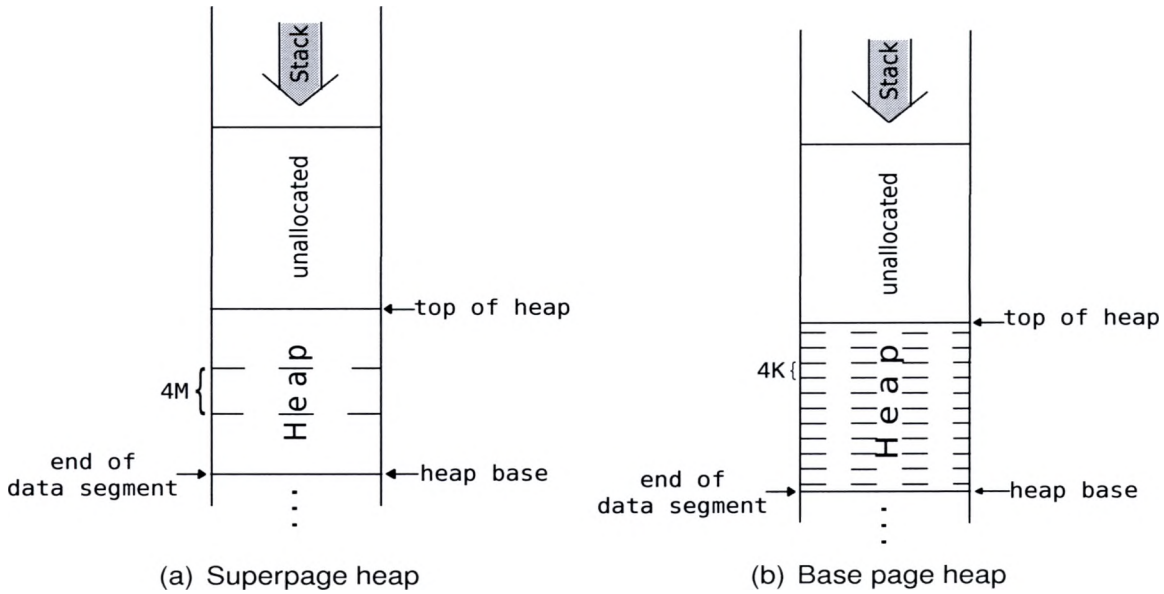


Figure 3.6: Superpage and base page heaps

version will align the base of the heap at the end of the data segment, as shown in figures 3.6(a) and 3.6(b). A problem arises, however, if two different page sizes are used simultaneously. Since chunks of contiguous superpage memory must be aligned upon the size of a superpage, mixing base pages and superpages leads to a *swiss cheese* effect on the heap, since gaps are required to ensure the correct alignment, which effectively disables the utility of the heap. In an effort to maintain a consistent heap, several invariants are defined:

- If superpages are allocated initially then base pages cannot be used.
- If base pages are allocated initially then superpages can later be allocated, but it is required that:

  - The superpage heap is placed at the next superpage alignment after the top of the base page heap.
  - The base page heap is prohibited from growing further.

- Both base pages and superpages can be used simultaneously, with smaller allocations backed by base pages and larger allocations backed by superpages if a fixed cap is applied to the base page heap during initialization, as can be seen in figure 3.7 . This issue is discussed in more detail later in this section.

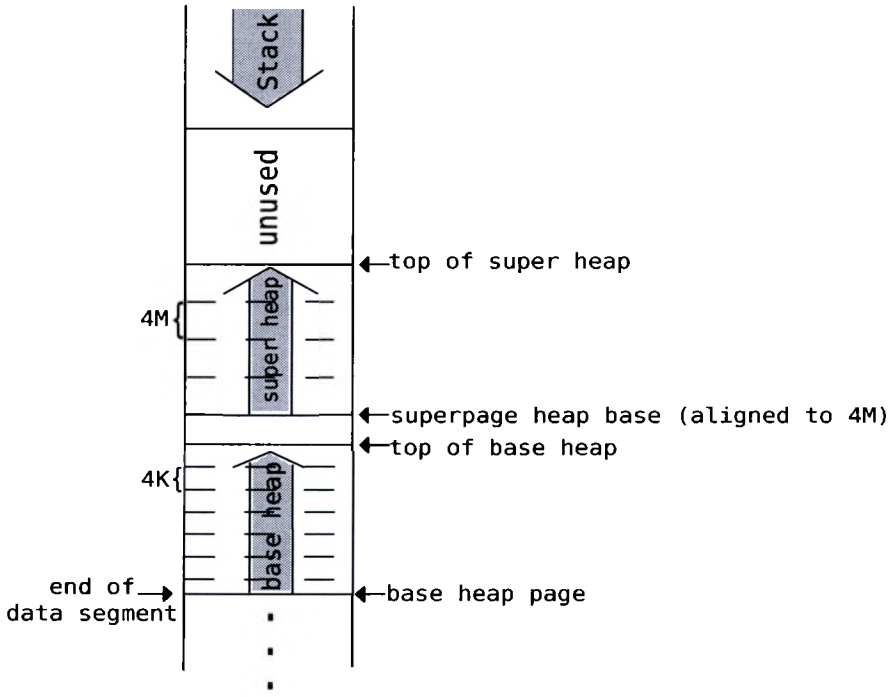


Figure 3.7: Dual page heap

smalloc maintains the sanity of the heap through the use of bookkeep-

ing data and bins of free chunks. Each chunk of memory has 4 machine words of bookkeeping overhead associated with it. A header field contains the status and size of the chunk, followed by pointers to the previous and next free chunks. A footer field is provided for the status and size as well. Maintaining the size of the chunk at both the head and tail allows for adjacent chunks to be efficiently coalesced into one, albeit at an acceptable cost of wasted space. Figure 3.8 illustrates the structure of a chunk.

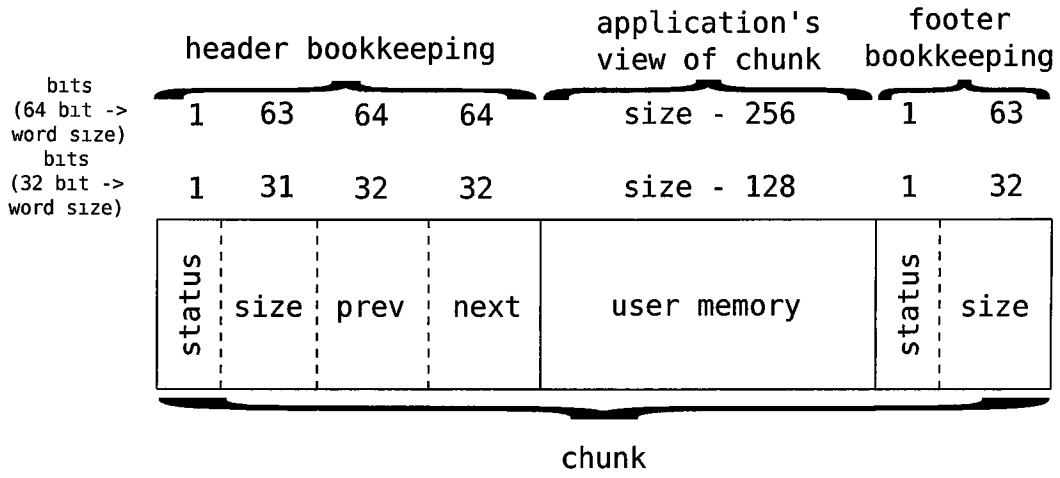


Figure 3.8: A chunk of *smalloc* memory

smalloc uses bins to maintain the list of free chunks. Bins are arranged in increasing size, from small to large. Chunks under 512 bytes are placed into bins spaced 8 bytes apart. Chunks larger than 512 bytes increase in logarithmic intervals. 32-bit platforms have 96 bins while 64-bit platforms have 128 bins. Each bin contains a pointer to a doubly-linked list of free chunks.

Bins are searched in a smallest first, best-fit/first-fit order. The bin index

of a chunk is calculated and if a chunk cannot be found in the bin then the next bin is searched, yielding a best-fit strategy. Best-fit, on the whole, produce less fragmentation than other strategies [45]. Each list in a bin is searched using a first-fit strategy. This binning approach is used to avoid fragmentation while maintaining low time complexity. Figure 3.9 shows the structure of the bins.

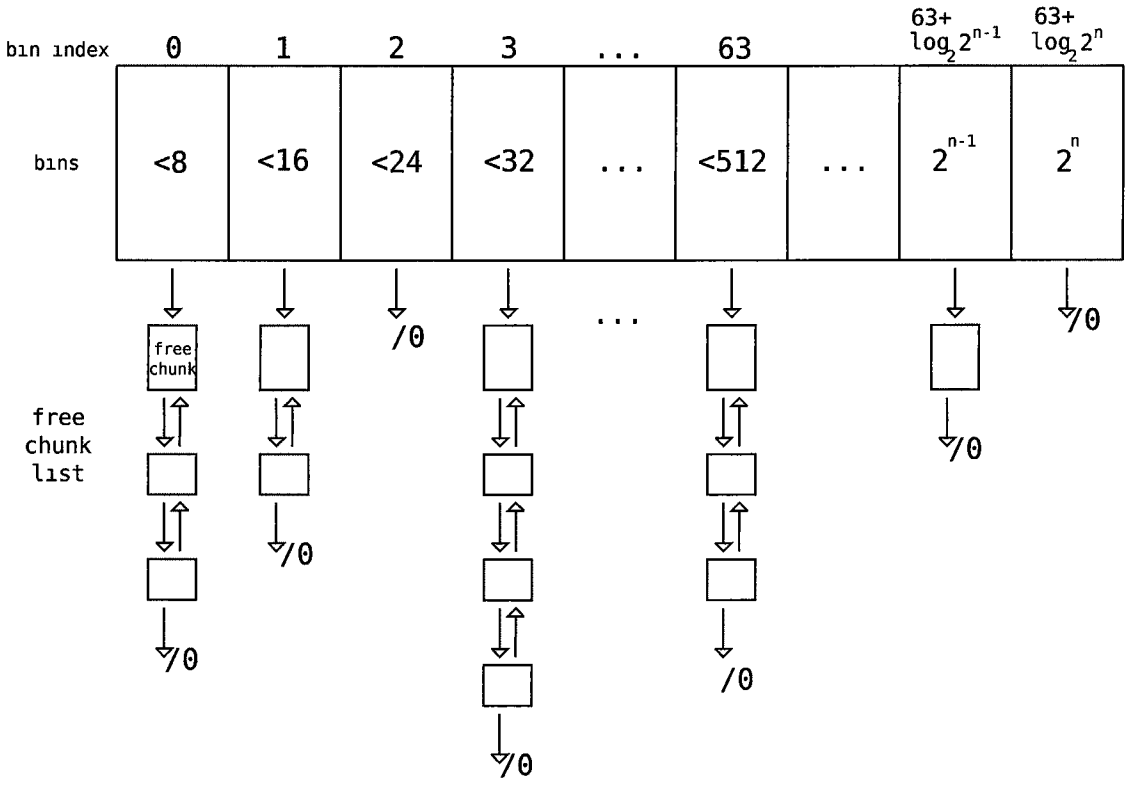


Figure 3.9: Bins of free memory

smalloc operates by first searching for a free chunk large enough to accommodate the user's request. If such a chunk cannot be found then *smalloc* will request more memory using *morecore*. The approach employed by *smalloc* can be seen in algorithm 1.

Algorithm 1 smalloc(size_t size)

Require: size > 0
Ensure: mem points to a valid block size bytes large

```
1  Pick morecore version
2  size ← size + BOOKKEEPING_SIZE
3  if The heap is empty then
4     Allocate memory with morecore
5     Partition heap:
6·    mem ← (heap_base → heap_base + size)
7     free_chunk ← (heap_base + size → heap_base + total_heap_size)
8.    Update bookkeeping information for mem
9     Update bookkeeping information for free_chunk
10    Register free_chunk in bin
11. else
12    Find free_chunk using best fit
13    if free_chunk is found then
14       Remove free_chunk from bin
15·      if free_chunk > size + BOOKKEEPING_SIZE then
16·         Partition Chunk:
17          mem ← (free_chunk → free_chunk + size)
18          free_chunk ← (free_chunk + size → free_chunk +
            total_chunk_size)
19          Update bookkeeping information for mem
20.         Update bookkeeping information for free_chunk
21·         Register free_chunk in bin
22.      else
23          mem ← free_chunk
24.      end if
25.   else
26       Extend heap with morecore
27       Partition Chunk:
28       mem ← (new_chunk → new_chunk + size)
29       free_chunk ← (new_chunk + size → new_chunk + total_chunk_size)
30:      Update bookkeeping information for mem
31·      Update bookkeeping information for free_chunk
32       Register free_chunk in bin
33.   end if
34 end if
35
36 Adjust mem to after header bookkeeping data
37 return mem
```

sfree releases memory obtained by *smalloc*. If the victim chunk neighbours any free chunks then they are joined into one. Finally the chunk is registered in a bin and marked as free. Algorithm 2 outlines *sfree*.

Algorithm 2 sfree(void *ptr)

Require: ptr is not NULL
Ensure: the memory pointed to by ptr is released
1: ptr ← ptr − BOOKKEEPING_OFFSET
2. **if** previous chunk = free **then**
3. Remove previous chunk from bin
4. Merge previous chunk and current chunk
5: Set up bookkeeping for new chunk
6: **end if**
7: **if** next chunk = free **then**
8: Remove next chunk from bin
9 Merge next chunk and current chunk
10: Set up bookkeeping for new chunk
11: **end if**
12. Register free chunk in bin

scalloc is a wrapper around *smalloc* that allocates memory for an array. If successful then the memory is zeroed out (*id est* each element is set to a value of zero). *srealloc* attempts to enlarge a block of memory. The contents of the memory will remain unchanged but any new memory will be uninitialized. If *srealloc* is called with a size of 0, then it functions like *sfree*. Should the pointer be null, then it is equivalent to *smalloc*.

Each function in the *smalloc* memory allocation toolkit is designed to be a drop in replacement for the standard C version. This is achieved by maintaining the same syntax and semantics; only the underlying strategies are different. *smalloc* can be used as a stand-alone library, as can be seen in fig-

ure 3.10, but the primary goal is to allow for the transparent usage of superpage by having the compiler replace all calls to *malloc* functions with calls to *smalloc* functions.

```
#include <stdlib.h>
#include <smalloc.h>
#include <stdio.h>

int main(int argc, char **argv) {
        int size, i;
        double *a;

        if(argc > 1)
                size = atoi(argv[1]);
        else
                size = 100;

        a = scalloc((size_t)size, sizeof(double));
        if(!a) exit(1);

        for(i = 0; i < size; i++)
                a[i] = rand() * (1.0/rand());

        for(i = 0; i < size; i++)
                printf("a[%d] = %lf\n", i, a[i]);

        sfree(a);

        return 0;
}
```

Figure 3.10: Using *smalloc* in a program

Dynamic Superpage Allocation with *smalloc*

One advantage of a custom memory allocator like *smalloc* is that it facilities the ability to have dynamically determined superpage allocation. The two key factors that inform judicious usage of superpages are the data-reuse patterns and the size of the working set. As noted in chapter 2, previous approaches to heuristically determined superpage allocation have solely con-

sidered the size of the working set and therefore fail to take advantage of the data-reuse patterns. Since most of these methods are implemented at the operating system level the reuse patterns are not available. This information is only available to the compiler.

One deficit to compiler-driven superpage allocation is that the size of the working set is usually unknown. The compiler must make an educated guess at the working set size. While this solution is adequate for most applications, it can fail to properly estimate the TLB demands of programs that have incongruous working sets. A custom memory allocator, such as *smalloc*, can augment a compiler-based strategy by providing mechanisms to determine at runtime the memory allocation strategy.

There are two distinct issues that must be considered when implementing dynamically determined superpage allocation. *smalloc* bases its decision to allocate superpage upon a set threshold. If the amount of requested memory exceeds this threshold then superpage allocation is enabled. This raises a question: should a single allocation or the sum of allocations be considered?

Singleton determined allocation makes the decision to allocate memory backed by superpages based on a single call to *smalloc*. Should the requested memory exceed the set threshold then superpages are enabled. The disadvantage of this approach is that it fails to consider data structures that are created across multiple allocations. Two and three dimensional arrays may be allocated in chunks smaller than the threshold even though the total size of the final array exceeds the threshold. Unlike multiple, small, unrelated allocations

that may exhibit no locality of reference, a multi-dimension array or other large data structure is likely to exhibit strong locality of reference and would thus benefit from superpage-backed memory.

Comprehensively determined allocation makes the decision to allocate superpage backed memory by comparing the total amount of allocated memory to the threshold. Once the sum of all memory exceeds the threshold then superpages are enabled. This approach addresses the concern of data spanning multiple allocations, but it may result in unnecessary superpage allocation when an application consists solely of small and unrelated allocations.

The second issue requiring consideration is how the switch from base pages to superpages should be performed. *smalloc* is flexibly designed to allow four different approaches to the issue of *heap switching*.

(i) No heap switching Only one heap is allowed and the memory used to back the heap is determined by the first call to *smalloc*. If the initial requested memory is above the threshold then superpages are used, otherwise base pages are used. The type of pages used does not change for the lifetime of the program regardless of any future allocations. Note this implies the usage of *singleton determined allocation*. This is a "better than nothing" approach with the primary advantage that it is easy to implement. Nonetheless, in cases where the working set is allocated with one initial allocation this solution is optimal.

(ii) Base heap freezing Once the threshold has been reached the size of the base heap is frozen. The superpage heap is allocated at the next

aligned boundary after the base heap and all future allocations are performed with the super heap. The data in the base heap is preserved and can continue to be used throughout the program. This approach works best with *comprehensively determined allocation*, but can also be used with *singleton determined allocation*. This approach allows for a finer granularity in superpage allocation but has several disadvantages. As memory is freed from the base heap after the threshold has been reached is not considered for future allocations. This implies that free memory in the base heap is wasted. Secondly, memory that has been allocated on the base heap cannot take advantage of superpages.

(iii) Base heap migration Once the threshold is reached all memory is copied from the base heap into the new super heap. The base heap is then freed and all further memory allocations are performed with the super heap. This approach has the advantage of avoiding the wasted space associated with *base heap freezing* as well as allowing early allocations to benefit from the usage of superpage. While any determination method for the page size can be used, this approach lends itself to *comprehensively determined allocation*. The critical disadvantage to this strategy is that it incurs a one time penalty of copying the heap. The larger the threshold the more severe the penalty, so a lower threshold should be employed with this method. In many applications the cost of this approach may be too severe, especially if the penalty is incurred during critical sections of the code. In certain scenarios the benefits of using

superpages will offset the cost of migrating the heap, but there are an equal number of scenarios where the cost outweighs any benefits.

(iv) Dual heaps The final approach involves maintaining two heaps: one backed by superpages and the other backed by base pages. Both heaps must be aligned within the same address space thereby requiring that the size of one heap is capped. If both heaps were allowed to grow unbounded then eventually the base heap would overrun the super heap. Since the base heap will be used for small allocations it is given a reasonable upper bound. The super heap, which will be used for larger allocations, will be allowed to grow unbounded (at least to the extent supported by the underlying operating system). Requests for memory below a threshold will be allocated on the base heap and requests above the threshold go on the super heap. This approach implies the usage of *singleton determined allocation* and therefore suffers from the same deficits. Large data structures that span multiple allocations will be mis-categorized. This approach also introduces extra processing overhead since as the base heap approaches its maximum size smaller requests may require a pass through both heaps before finding a free chunk.

Each different strategy for dynamically determining superpage has its own set of advantages and drawbacks. It is difficult for *smalloc* to predict which strategy will be most profitable for an application without external "hints." These "hints" can be in the form of explicit requests from an application, via a mechanism such as *mallopt*, or through recommendations made by the com-

piler during static analysis.

smalloc implements strategy (i), no heap switching with singleton determined allocation, as the default approach to dynamic superpage support. While this approach offers the minimum advantages it also represents the minimum drawbacks. Furthermore dynamic support in *smalloc* is an extension; the primary focus is on compiler driven analysis. *smalloc* refers to dynamically determined superpage allocation as *smart pages*, since the decision of which page type to use is delayed until runtime when more information is available and presumably a "smarter" decision can be made.

smalloc Configuration Overview

Table 3.6 shows the *smalloc* operational parameters and the default settings for the two tested platforms.

Table 3.6: *smalloc* configuration overview

Paramaters	***smalloc* Overview**	**Intel**	**AMD**
Modes	base page, superpage, dynamic	base page, superpage, dynamic	base page, superpage, dynamic
Default Mode	superpage	superpage	superpage
Base Page Size	Determined by System	4K	4K
Super Page Size	Determined by System	4M	2M
Bookkeeping overhead	4 × word size	16 bytes	24 bytes
Free Bins	Determined by System	96	128
Dynamic Threshold	8 × base page size	32K	32K

3.5.2 A Pass in LLVM

One of the primary advantages of having a superpage aware version of *malloc* that follows syntactical invariants is that it allows all dynamic mem-

ory related functions to be replaced. LLVM provides a pass framework which is used to implement a simple source-to-source transformation that converts calls to *malloc* family functions to *smalloc* functions. Refer to table 3.4 for the functions that are replaced. Our implemented pass is termed *superpass* and is a subclass of the FunctionPass.

Since *malloc*, *calloc*, *realloc*, and *free* are all external functions to the compilation unit, *superpass* only needs to replace the references to these functions. It accomplishes this by looking up each function name in the symbol table. If it finds an entry then it changes the entry to refer to one of the functions in the superpage aware *smalloc* library.

While *smalloc* can function without any extra setup, in order for a compiler heuristic to specify the operational parameters of the memory allocator several external variables must be changed. Currently *smalloc* provides two such variables:

__malloc_mode determines which version of *morecore* will be employed:

0 Use base pages

1 Use superpages

2 Dynamically determine the type of page to use at runtime (smart pages)

__smartpage_threshold The size in bytes of the threshold used to determine when superpages are allocated when smart pages are used.

If they are not defined by a program then __malloc_mode defaults to 1 (use superpages) and __smartpage_threshold defaults to 8 times the size

of a base page. The default smart page threshold is decidedly low to accommodate the usage of *singleton determined allocation* (the default). While this value is small with regards to the entire working set of an application, it is quite large for a single allocation. For example, given a 32-bit machine with 4K base pages the smart page threshold would be 32K, which would require an array of 8192 integers to meet the threshold.

The *superpass* sets these variables at the beginning of the *main()* function. The pass registers each variable as an externally weak-linked global variable in the symbol table. Once *main()* is found during a function pass, it inserts two store instructions into the basic block structure of the function. The value of these variables are obtained during the heuristic analysis presented in chapter 4.

3.5.3 Compilation Flags

Table 3.7 shows supported compiler options. The names of the flags may change to better integrate to *llvm* or *gcc* naming conventions, but the functionality will remain the same. Currently only *-super* and *-nosuper* are officially implemented. Chapter 4 presents a heuristic that is employed with the remaining options.

Table 3.7: Compiler Flags

Flag	Description
-nosuper	Disable superpages
-super	Enable and force superpage usage
-super-static	Enable staticaly determined superpage usage
-super-dynamic	Enable dynamically determined superpage usage
-super-full	Enable both statically and dynamically determined superpage usage

3.6 Experimental Results

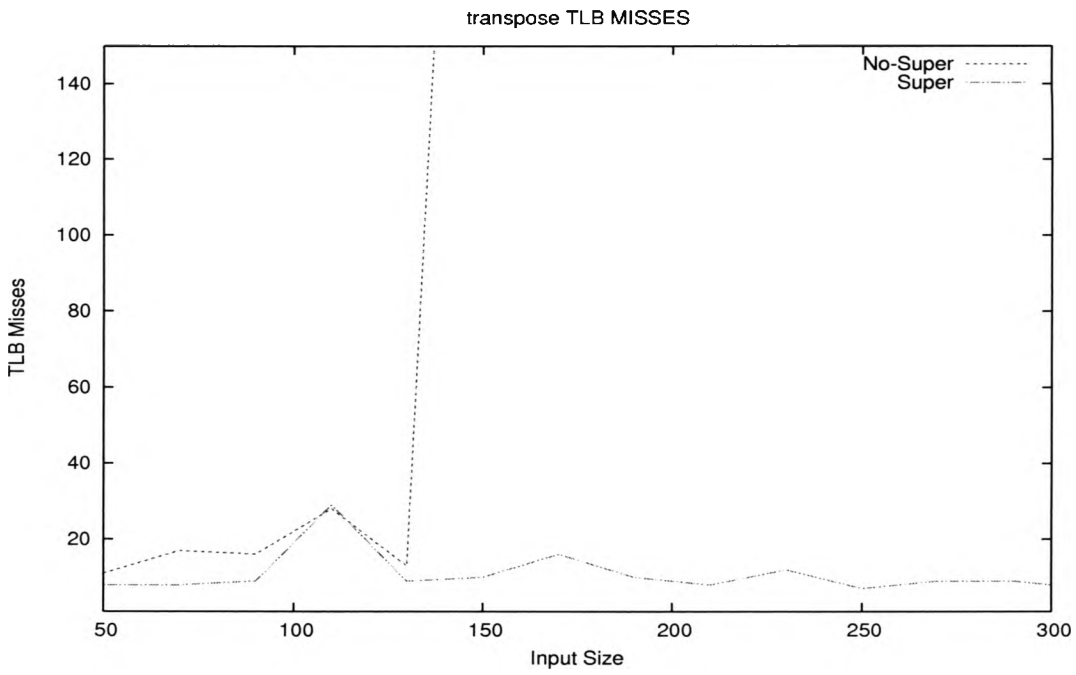


Figure 3.11: *transpose* TLB performance on Intel

3.6.1 Experimental Setup

Benchmarks

The benchmarks used in this research include SPEC benchmarks *164-gzip* and *188-ammp*, in addition to *transpose*, a matrix transposition code. The *ammp* and *transpose* benchmarks were selected as likely beneficiaries of superpages due to their exhibition of high TLB demand. *gzip* was selected as a candidate not benefiting from superpages due to its linear data access pat-

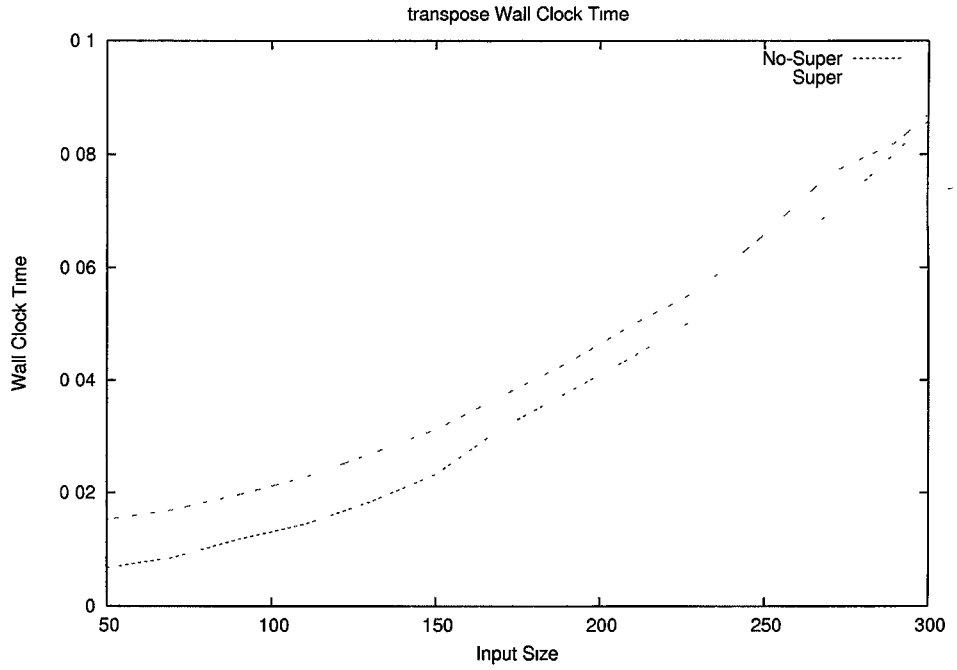


Figure 3.12: *transpose* Wall Clock Time on Intel

terns.

164-gzip is a SPEC benchmark based on the popular compression program *gzip*. *gzip* uses a Lempel-Ziv (LZ77) compression algorithm and performs all compression and decompression entirely in memory so as to remove any unnecessary I/O operations that could taint the benchmark. The benchmark is run with data sets of increasing size, ranging from 1M to 64M.

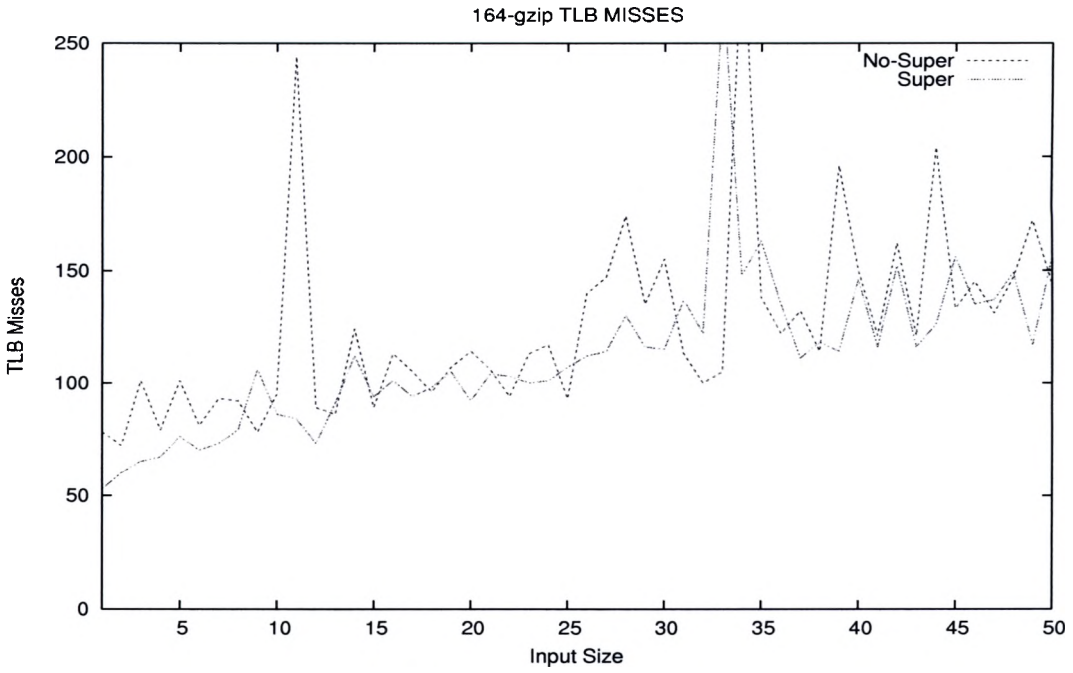


Figure 3.13: *164-gzip* TLB Misses on Intel

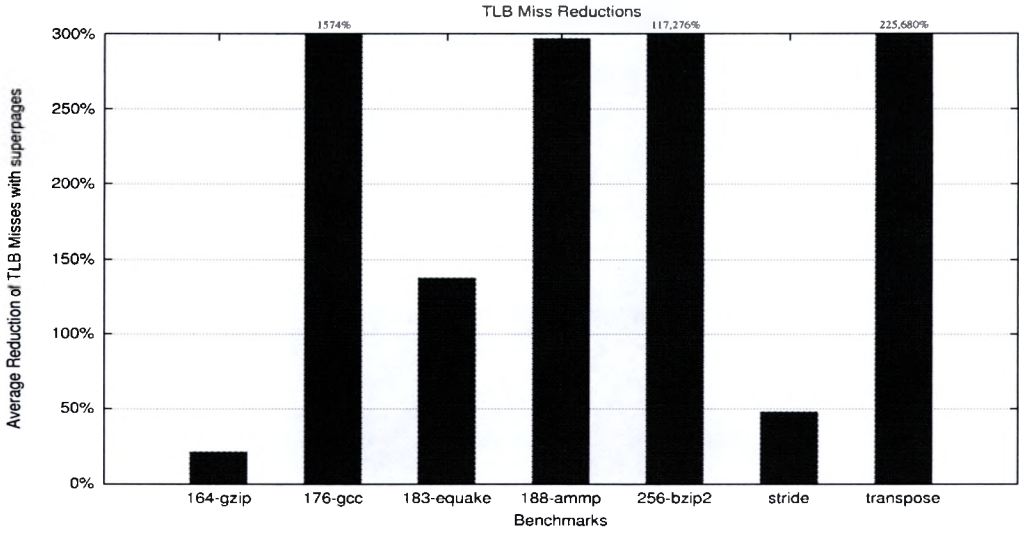


Figure 3.14: TLB Miss Reductions for all Benchmarks on Intel

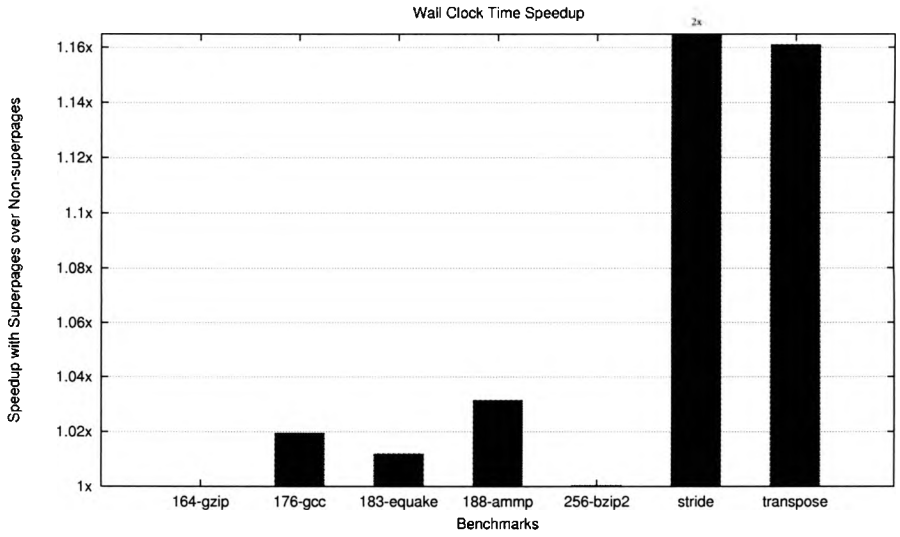


Figure 3.15: Wall Clock Speedup for all Benchmarks on Intel

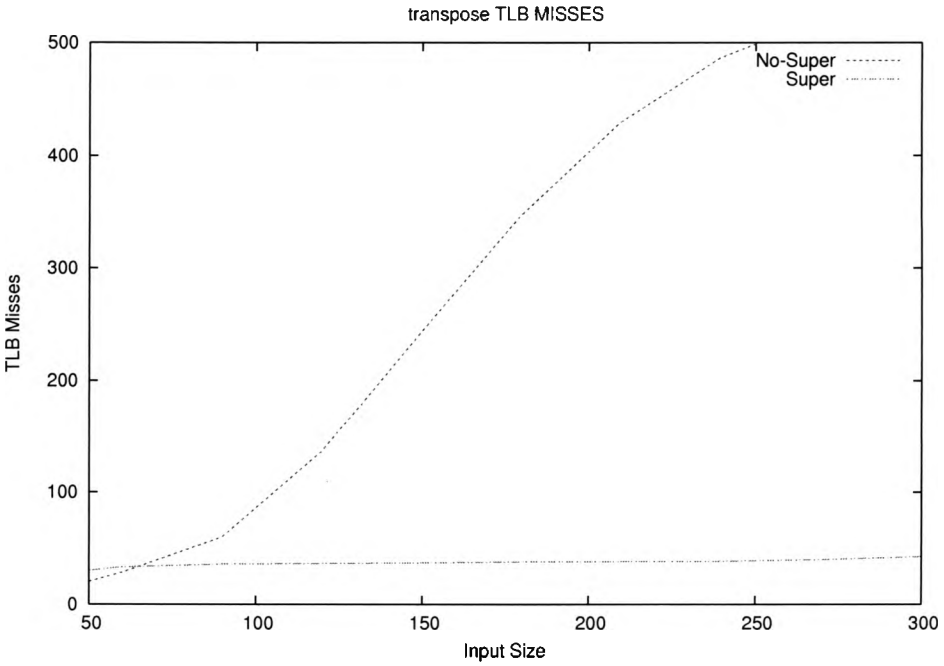


Figure 3.16: *transpose* TLB performance on AMD

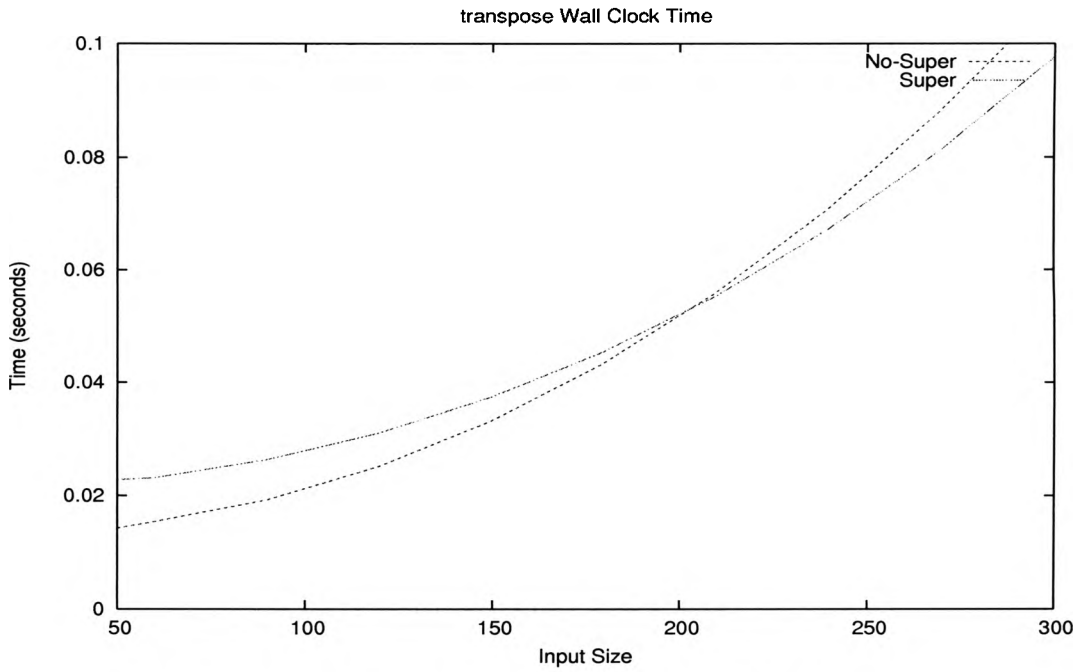


Figure 3.17: *transpose* Wall Clock Time on AMD

188-ammp is a SPEC floating point benchmark that solves Newton's Ordinary Linear Equation for the movement of atoms in a system that is on a protein-inhibitor complex and is embedded in water [17]. This floating point benchmark was selected due to its large data set, consisting of 9582 input parameters (atoms suspended in the water). The benchmark is run with three different data sets of varying size and complexity.

176-gcc is a SPEC integer benchmark that is based on the GNU C compiler. The benchmark is a likely candidate for improving TLB performance due the size and complexity of the data sets.

183-equake is a SPEC floating point benchmark that simulates earthquakes and other seismic activity. Like *188-ammp* and *176-gcc* it is a desirable

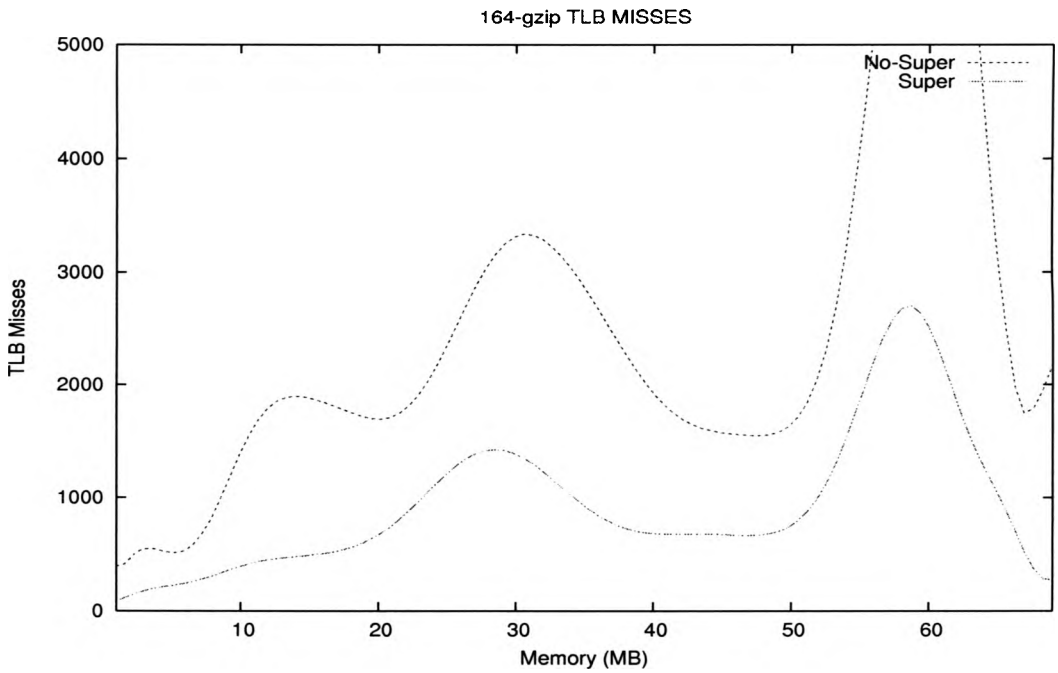


Figure 3.18: *164-gzip* TLB Misses on AMD

benchmark due to the size and complexity of the data.

stride is a synthetic kernel benchmark that strides through a large array with increasing step sizes. The array is 8K large and the stride size increases from 1 to 8192 bytes. When the step size is 8192 then the benchmark will sweep through 64M of memory.

transpose is a kernel benchmark that performs a large number of matrix transpositions. The algorithm can be seen in example 1. The benchmark is run with increasing data sets ranging from matrix dimensions $30 \cdot 30$ to $3000 \cdot 3000$.

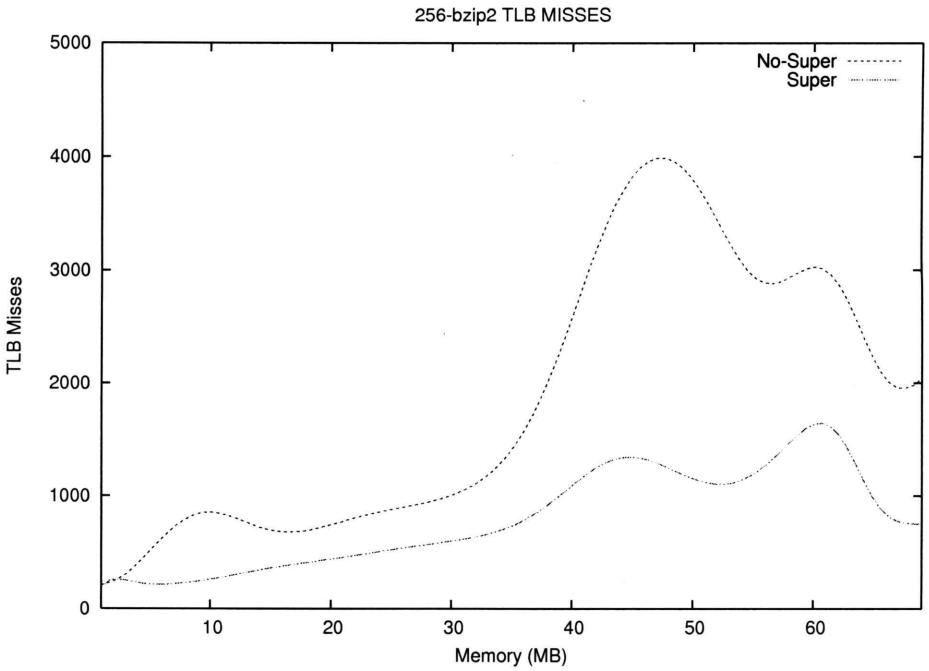


Figure 3.19: *256-bzip2* TLB Misses on AMD

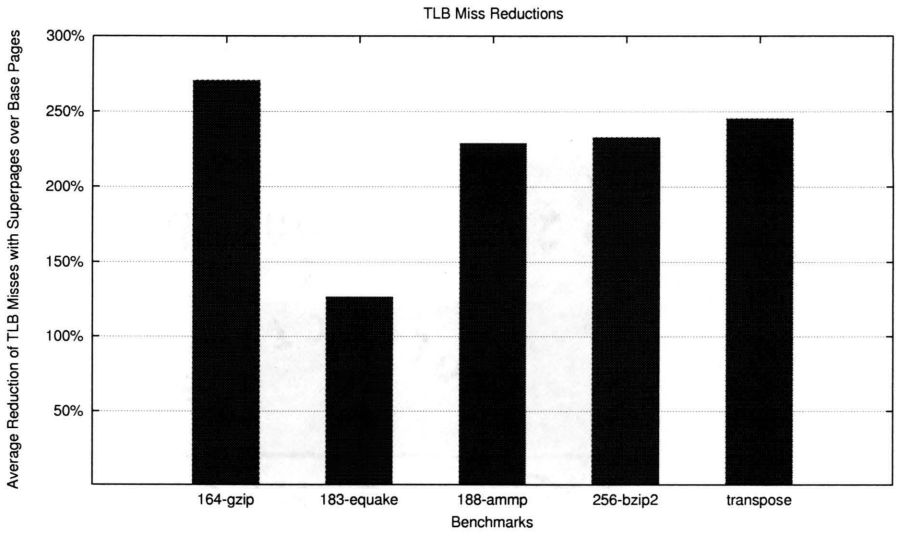


Figure 3.20: TLB Miss Reductions for all Benchmarks on AMD

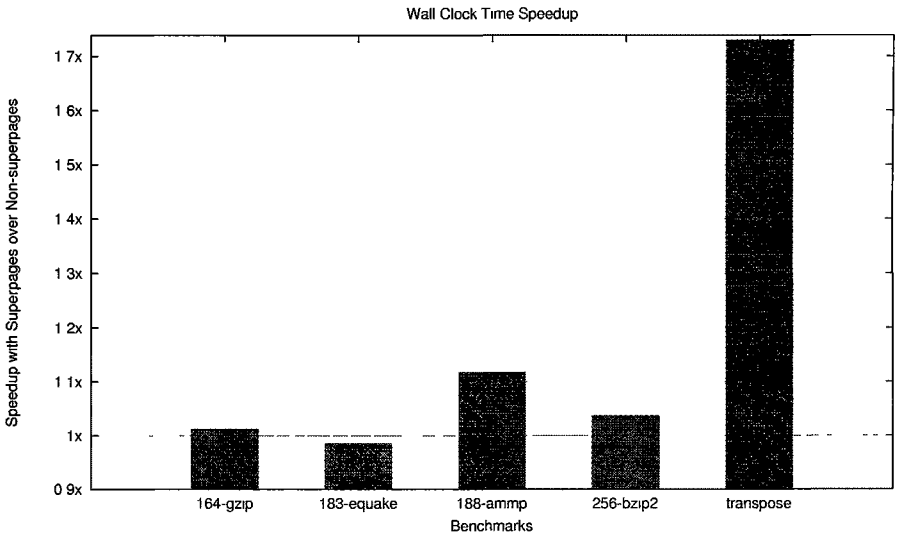


Figure 3.21: Wall Clock Speedup for all Benchmarks on AMD

Example 1 Transpose

Require: Matrix m with dimensions y,x
Ensure: Matrix m_{tmp} is the transposition of m

```
1: for i from 0 to y do
2:     for j from 0 to x do
3:         m_tmp[j][i] ← m[i][j]
4:     end for
5: end for
```

Platforms

Table 3.8 outlines the configurations of the systems used for testing.

Table 3.8: Platform Configurations

Vendor	Intel	AMD
Arch	x86	x86_64
Processor	Core 2 Duo	Athlon 64 X2 Dual Core 4400+
Base Page Size	4KB	4KB
Superpage Size	4096KB	2048 / 4096KB
TLB Associativity	4-way	4-way
TLB Entries	256	512

Tools

The Performance Application Programming Interface (PAPI) was used to collect performance metrics of each benchmark. This API provides access from application programs to the hardware performance counters in the machine via the Linux kernel module *perfctr* [5].

Methodology

Each benchmark trial consisted of two executions of the application with a given data set. One execution was configured to run with standard pages and the other utilized superpages. The results of several metrics from each test were sampled using PAPI. The metrics sampled included TLB misses, L2 cache misses, wall clock time, and total clock cycles, with the number of TLB misses being of foremost importance.

The results of the execution are recorded and pushed to a database and the same benchmark is profiled with the next data set. In such a manner a comprehensive view of the TLB performance of the benchmark can be established over a range of inputs. This facilitates the ability to identify the

point at which the number of TLB misses diverge between superpage and non-superpage data allocations. This data, obtained over a cross section of different benchmarks, provides the basis for defining and refining SPREAD as defined in chapter 4.

The input datasets for the SPEC benchmarks, *164-gzip*, *177-gcc*, *183-equake*, *256-bzip2*, and *188-ammp*, were obtained from the SPEC reference data sets. The input dataset for *transpose* is randomly generated for increasing step sizes of 3600 bytes. The step size was chosen to allow a fine granularity of testing without too much redundancy.

3.6.2 TLB Performance

Intel

Figure 3.11 shows the number of TLB misses for the *transpose* benchmark with and without superpages. The domain is the range of input matrix sizes from $50 \cdot 50$ to $300 \cdot 300$. The variation of TLB misses between the two page sizes is relatively small until input dimension $130 \cdot 130$ at which point the standard sized page execution incurs a marked number of TLB misses. This marked divergence occurs at an input size of approximately 66KB. Figure 3.12 shows the wall clock time for the *transpose* benchmark. Over the entire domain of inputs there is an improvement to the execution time when using superpages.

Figure 3.13 shows the number of TLB misses for the *164-gzip* benchmark. This benchmark is an ideal example of an application that reaps little or

no benefit from the advent of superpages. *164-gzip* is characterized by linear data access patterns, using a floating window as it passes over the data. The cost of fragmentation in this benchmark is not offset by any gains in performance, and therefore superpage allocation should not be recommended.

Figure 3.14 shows the average reduction of TLB misses for each benchmark. For example, the usage of superpages reduced the TLB misses of *188-ammp* by approximately 296%. Figure 3.15 shows the respective speedup in clock time. The number of TLB misses is only a minuscule component of wall clock time, however each benchmark that incurred a substantially reduced number of TLB misses, around 50% or more, also experienced a speedup in time.

AMD

Figure 3.16 shows the number of TLB misses for the *transpose* benchmark with and without superpages. The behaviour of the benchmark is similar to that observed on the Intel platform. While the TLB performance of *transpose* improves at inputs less than $100 \cdot 100$, the wall clock time does not become profitable until an input size of approximately $200 \cdot 200$, as can be seen in figure 3.17. This suggests that the overhead and possible fragmentation introduced by the usage of superpages is not offset until the input is sufficiently large and thus the TLB miss reduction is sufficiently large.

Figure 3.18 presents the number of TLB misses for the *164-gzip* benchmark. Unlike results on the Intel platform, this benchmark shows a remarkable

reduction of TLB misses with the use of superpages. The primary cause of this anomaly is likely the smaller superpage size (2MB compared to 4MB). Benchmarks that experienced a large improvement with 4MB superpages showed slightly more modest improvements with 2MB superpages, thus it is reasonable that benchmarks that incurred a penalty with 4M superpages would experience a less severe penalty with 2MB superpages. In this case *164-gzip* actually demonstrated an improvement with the smaller superpage size, on the same level as the improvements seen with the *256-bzip2* benchmark as seen in figure 3.19

Figure 3.20 shows the average reduction of TLB misses for each benchmark on the AMD platform. It is interesting to note that *164-gzip* showed the *best* improvement. This is primarily a result of particularly poor TLB performance on inputs less than 50 and greater than 60 where the number of misses jumps to 10 times that of the superpage version. Other benchmarks, such as *transpose* and *188-ammp* are more significant in terms of application performance as can be seen in figure 3.21, mostly due to fact the *164-gzip* and *256-bzip2* involve heavy memory I/O traffic and *183-equake* involves a tremendous amount of floating point computation.

CHAPTER 4

A HEURISTIC FOR LOCALITY-CONCIOUS SUPERPAGE ALLOCATION

4.1 Overview

Indiscriminate allocation of superpages can have adverse effects. While superpages can extend the reach of the TLB and reduce TLB misses, they also increase the likelihood of fragmentation and increase the application footprint. In many applications, the benefits of superpages offset the issue of fragmentation, but in some applications, such as those with linear access patterns or small data sets, the benefits are lost and performance can either fail to improve or, in the case of heavy fragmentation, degrade.

To successfully exploit superpages the compiler must be able to estimate the TLB demands of an application and determine if these demands will benefit from the advent of superpages. The primary factors that contribute to the TLB demands are the size of the working set and the locality of reference of the program. The size of the working set may not be known until runtime, but the data-reuse patterns are readily available to the compiler. The compiler can estimate the demands on the TLB and judiciously allocate superpages by utilizing data-locality analysis.

Presented is a heuristic based on established methods for estimating

cache misses as presented by Allen and Kennedy [1]. The heuristic estimates the demands of the TLB and compares this against a configurable threshold to determine when superpage allocation will be most profitable.

An extension to the heuristic demonstrates how it can be used to inform the dynamic usage of superpages in *smalloc*, described in chapter 3. This extension allows for a more accurate determination of the runtime threshold (*__smartpage_threshold*) employed when using a dynamically determined page size. The chapter concludes with an analysis and evaluation of the heuristic that demonstrates its effectiveness in estimating the TLB demands of an application.

4.2 Heuristic

The heuristic bases the decision to allocate superpages upon the relationship between the *spread* of the memory references and the *threshold* of non-local references. The *threshold* represents a conservative estimation of the number of pages needed to incur TLB conflicts. The *threshold* is defined as the number of TLB entries divided by the associativity of the TLB, plus one, and multiplied by the base page size. Mathematically, the *threshold* is calculated as:

$$THRESHOLD = \left(\frac{NO\_OF\_ENTRIES}{ASSOCIATIVITY} + 1\right) \cdot BASE\_PAGE\_SIZE$$

The *spread* of memory references is determined by the following rules:

1. A memory reference that does not depend on the loop induction variable is assigned a *spread* of 1.

2. A memory reference that strides over non-contiguous dimensions is assigned a *spread* of I, where I is the number of the current loop iteration.

3. A memory reference that strides over contiguous dimensions with a step size of s is assigned a *spread* of $\frac{(I\ s)}{BASE\_PAGE\_SIZE}$

4. The *spread* of a reference that varies with the loop index is multiplied by the current iteration count.

5. The *spread* of a reference that does not vary with the loop index is multiplied by 1.

6. The total *SPREAD* is calculated by summing the memory references in the innermost loop.

If the *spread* meets or exceeds the *threshold*, then the compiler will choose to allocate superpages. This heuristic algorithm is presented as pseudo code in algorithm 3.

4.3 Dynamic Extension

smalloc, described in chapter 3, provides a *smartpage mode* where the usage of superpages is determined by the size of the working set. By default *smalloc* uses a general default *smartpage threshold*, *id est* the point at which superpages are likely to be profitable. The compiler heuristic presented in the

Algorithm 3 Compiler heuristic for allocating superpages

1. $THRESHOLD \leftarrow (\frac{NO\_OF\_ENTRIES}{ASSOCIATIVITY} + 1) \cdot BASE\_PAGE\_SIZE$
2. **for all** loop-nests s in procedure **do**
3. **for all** memory references r in s **do**
4. $I \leftarrow$ number of current loop iteration
5. **if** r depends on loop induction variable **then**
6. $spread_r \leftarrow 1$
7. **else if** r strides over non-contiguous dimensions **then**
8. $spread_r \leftarrow I$
9. **else if** r strides over contiguous dimensions **then**
10. $s \leftarrow$ step size
11. $spread_r \leftarrow \frac{(I\ s)}{BASE\_PAGE\_SIZE}$
12. **end if**
13.
14. **if** r varies with the loop index **then**
15. $spread_r \leftarrow spread_r \cdot I$
16. **else**
17. $spread_r \leftarrow spread_r \cdot 1$
18. **end if**
19. **end for**
20. **end for**
21.
22. $SPREAD \leftarrow 0$
23. **for all** memory references r **do**
24. $SPREAD \leftarrow SPREAD + spread_r$
25. **end for**
26.
27. **if** $SPREAD \geq THRESHOLD$ **then**
28. Use superpages
29. **else**
30. Use base pages
31. **end if**

previous section uses static analysis to determine if the data-reuse patterns are likely to benefit from the usage of superpages, but it can also be improved to estimate the *smartpage threshold*.

The heuristic estimates the *smartpage threshold* by running multiple passes of static analysis, in monotonically increasing increments. This yields the following approach:

1. A value n is chosen such that for any loop nest s, n is sufficiently small that when used as the upper boundary of each array, static analysis will return false (use base pages).

2. An increment is chosen, N, such that N is both positive and sufficiently large. A finer value of N will yield a more accurate *smartpage threshold* at the cost of increased number of passes through each loop nest. In general a course value of N is sufficient.

3. Beginning with loop sizes of n, perform heuristic analysis in increments of N until the heuristic returns true (use superpages). Calculate $n^S \cdot w$, where S is the total number of loops in the nest and w is the width of the underlying data type to be allocated, and assign it to the *smartpage threshold*. If the heuristic never returns true then use base pages.

The heuristic algorithm with dynamic threshold analysis is presented in algorithm 4.

Algorithm 4 Compiler heuristic for allocating superpages and estimating a dynamic working set threshold

```
1  n ← sufficently small value
2  N ← sufficently large value
3. upper_bound ← maximal possible upper boundary of a loop
4  for n → upper_bound do
5    THRESHOLD ← (NO_OF_ENTRIES / ASSOCIATIVITY + 1) · BASE_PAGE_SIZE
6.   for all loop-nests s in procedure do
7·     for all memory references r in s do
8        I ← number of current loop iteration
9        if r depends on loop induction variable then
10.        spread_r ← 1
11       else if r strides over non-contiguous dimensions then
12         spread_r ← I
13       else if r strides over contiguous dimensions then
14.        s ← step size
15         spread_r ← (I s) / BASE_PAGE_SIZE
16       end if
17
18       if r varies with the loop index then
19         spread_r ← spread_r · I
20·      else
21         spread_r ← spread_r · 1
22       end if
23.    end for
24.  end for
25
26   SPREAD ← 0
27   for all memory references r do
28     SPREAD ← SPREAD + spread_r
29·  end for
30
31   if SPREAD ≥ THRESHOLD then
32     smartpage_threshold ← n
33:    Use superpages
34     return
35   end if
36   n ← n + N
37. end for
38 Use base pages
```

4.4 Analysis and Evaluation

4.4.1 The heuristic applied to a loop-nest that exhibits high TLB pressure

Given a configuration featuring 4KB pages and a 4-way associative TLB with 256 entries, the heuristic is applied to example 1.

Example 1 Loop Nest 1

Require: Matrix m with dimensions 150,150

```
1 for i from 0 to 150 do
2   for j from 0 to 150 do
3     m_tmp[j][i] ← m[i][j]
4   end for
5: end for
```

Reference $m[i][j]$ depends on the loop induction variable and strides over contiguous dimensions with a step size of one, thus the *spread* can be calculated as:

$$\begin{aligned} spread[m_{ij}] &= 150 \sum_{i=0}^{150} \frac{i}{4096} \\ &= 414.73 \end{aligned}$$

Reference $m_{tmp,ji}$ depends on the loop induction variable and strides over non-contiguous dimensions, thus the *spread* can be calculated as:

$$\begin{aligned} spread[m_{tmp,ji}] &= 150 \sum_{i=0}^{150} i \\ &= 1698750 \end{aligned}$$

The total *spread* can be calculated as:

$$\begin{aligned} SPREAD &= spread[m_{ij}] + spread[m_{tmp,ji}] \\ &= 414.73 + 1698750 \\ &= 1699164.73 \end{aligned}$$

The *threshold* is defined as:

$$\begin{aligned} THRESHOLD &= \left(\frac{NO\_OF\_ENTRIES}{ASSOCIATIVITY} + 1\right) \cdot BASE\_PAGE\_SIZE \\ &= \left(\frac{256}{4} + 1\right) \cdot 4096 \\ &= 266240 \end{aligned}$$

Since $1699164.73 > 266240$, the heuristic condition $SPREAD \geq THRESHOLD$ holds true and therefore superpages are allocated.

4.4.2 The heuristic applied to a loop-nest that exhibits low TLB pressure

Given a configuration featuring 4KB pages and a 4-way associative TLB with 256 entries, the heuristic is applied to example 2.

Example 2 Loop Nest 2

Require: Matrix m with dimensions 300,300

```
1 for i from 1 to 300 do
2:   for j from 1 to 300 do
3      m[i − 1][j − 1] ← m[i − 1][j − 1] + m[i][j]
4    end for
5 end for
```

Reference $m[i][j]$ depends on the loop induction variable and strides over contiguous dimensions with a step size of one, thus the *spread* can be calculated as:

$$\begin{aligned} spread[m_{ij}] &= 350 \sum_{i=0}^{300} \frac{i}{4096} \\ &= 3306.88 \end{aligned}$$

Reference $m[i-1][j-1]$ depends on the loop induction variable and strides over contiguous dimensions, thus the *spread* is the same as that for reference $m[i][j]$. While there are two references to $m[i-1][j-1]$, they are counted as 1 reference group, since they are accessing the same memory location and should incur no additional penalty.

The heuristic condition is evaluated as false and therefore superpages are not allocated.

$$\begin{aligned} SPREAD &< THRESHOLD \\ 3306.88 + 3306.88 &< \left(\frac{256}{4} + 1\right) \cdot 4096 \\ 6613.76 &< 266240 \end{aligned}$$

4.4.3 The extended heuristic applied to a loop-nest that exhibits high TLB pressure

Given a configuration featuring 4KB pages and a 4-way associative TLB with 256 entries, the extended heuristic is applied to example 3.

Example 3 Loop Nest 3

Require: Matrix m with dimensions n,n

```
1· for i from 0 to n do
2:    for j from 0 to n do
3·       m_tmp[j][i] ← m[i][j]
4·    end for
5· end for
```

The analysis of example 1 demonstrates that a n of 150 is large enough to warrant the usage of superpages. Given this information, we can select our parameters as $n = 30$ and $N = 30$. In general the heuristic will not have any information to assist in the selection of n and N, but for analytical purposes using artificially selected values reduces the number of iterations.

$$\begin{aligned} SPREAD[n = 30] &= 30 \sum_{i=0}^{30} \frac{i}{4096} + i \\ &= 13953.4058 \\ &< 266240 \end{aligned}$$

$$\begin{aligned} SPREAD[n = 60] &= 60 \sum_{i=0}^{60} \frac{i}{4096} + i \\ &= 109826.8066 \\ &< 266240 \end{aligned}$$

$$\begin{aligned} SPREAD[n = 90] &= 90 \sum_{i=0}^{90} \frac{i}{4096} + i \\ &= 368639.9780 \\ &> 266240 \end{aligned}$$

Since the *SPREAD* is over the *THRESHOLD* then superpages can be allocated and the *smartpage threshold* is $90^2 \cdot w$ or 32400 if the data type is a 32-bit integer.

4.4.4 The extended heuristic applied to a loop-nest that exhibits low TLB pressure

Given a configuration featuring 4KB pages and a 4-way associative TLB with 256 entries, the extended heuristic is applied to example 4.

Example 4 Loop Nest 4

Require: Matrix m with dimensions n,n

```
1: for i from 1 to n do
2:    for j from 1 to n do
3       m[i − 1][j − 1] ← m[i − 1][j − 1] + m[i][j]
4    end for
5 end for
```

The values of n and N used in this analysis are both 60.

$$\begin{aligned} SPREAD[n = 60] &= 120 \sum_{i=0}^{60} \frac{i}{4096} \\ &= 53.6133 \\ &< 266240 \end{aligned}$$

$$\begin{aligned} SPREAD[n = 120] &= 240 \sum_{i=0}^{120} \frac{i}{4096} \\ &= 425.3906 \\ &< 266240 \end{aligned}$$

$$\ldots$$

$$\begin{aligned} SPREAD[n = 1020] &= 2040 \sum_{i=0}^{1020} \frac{i}{4096} \\ &= 259337.9883 \\ &< 266240 \end{aligned}$$

$$\begin{aligned} SPREAD[n = 1080] &= 2160 \sum_{i=0}^{1080} \frac{i}{4096} \\ &= 307831.6406 \\ &> 266240 \end{aligned}$$

Since the *SPREAD* is over the *THRESHOLD* then superpages can be allocated and the *smartpage threshold* is $1080^2 \cdot w$ or 4665600 if the data type is a 32-bit integer.

4.4.5 Evaluation

Figure 4.1 shows the number of TLB misses, at a logarithmic interval, in relation to the array dimensions of loop nest 1 and 3. The values used in the static analysis of loop nest 1 and the *smartpage threshold* derived in the analysis of loop nest 3 are marked respectively at x-intercepts 90 and 150. At the static analysis threshold the number of TLB misses incurred with superpages is relatively constant while the base page version increased linearly. The heuristic decision to allocate is justified and verified by the graph: At an array size of 150 by 150 there is already a noticeable divergence between base and superpages, a trend which is consistent in the long term behaviour

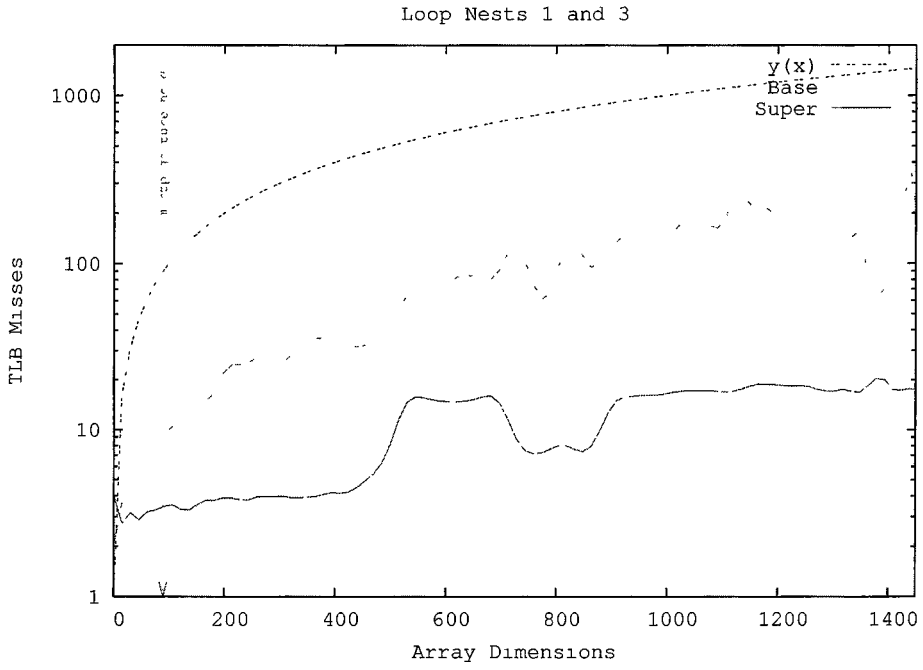


Figure 4.1: Nest 1 TLB performance

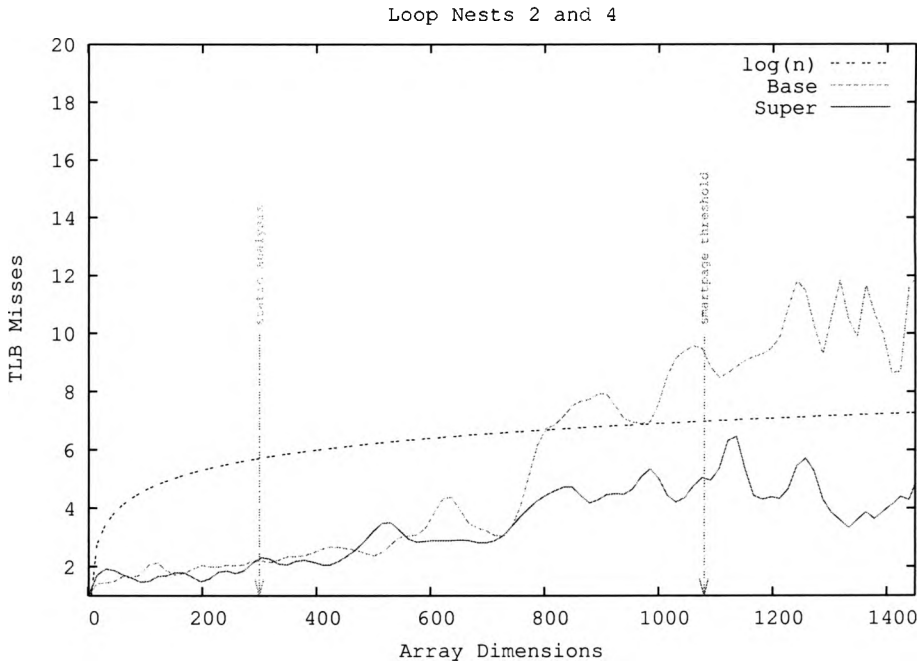


Figure 4.2: Nest 2 TLB performance

of the loop nest. Dynamic analysis selects a value of 90 for the *smartpage threshold* which also is in congruence with the presented results. The threshold is chosen at a point where both superpages are profitable and the long term divergence between base and superpage is reasonably established.

Figure 4.2 shows the results of executing loop nests 2 and 4. Note this graph, unlike the previous, is not presented on a logarithmic scale. The long term TLB complexity of both versions, base pages and superpages, is $\theta \log(n)$. The static analysis threshold is at a point where the number of TLB misses incurred by both types of pages is the same, thus the decision *not* to allocate superpages is well justified. The dynamic heuristic analysis chooses a value of 1080 as the *smartpage threshold*. While this value could reasonably be ad-

justed ± 100, the chosen threshold is a reasonable point at which superpages have become profitable, albeit only slightly.

Analysis of the heuristic shows that it effectively estimates the TLB demands of an application. The chosen *threshold* is conservative and could be adjust to allow less enthusiastic allocation of superpages, however it is successful in avoiding the worse case of superpages degrading performance. Since the *threshold* can easily be adjusted the heuristic can be fine tuned to meet the demands of any application or system.

CHAPTER 5

LEVERAGING SUPERPAGES FOR COMPILER OPTIMIZATIONS

5.1 Overview

The primary candidates for compiler optimizations that may benefit from the usage of superpages are memory hierarchy optimizations. As discussed in chapters 1 and 3, one requirement imposed upon superpages are that they are allocated contiguously in memory. Since the majority of caches on modern architecture are physically indexed the memory is not guaranteed to be contiguous. This limits the effectiveness of many memory hierarchy optimizations since the compiler must either guess at the most likely mapping to cache lines or *pretend* that memory is allocated contiguously.

Most memory hierarchy optimizations fall into the category of locality optimizations which attempt to improve a programs locality of reference. One such optimization is *array padding*. Array padding is an ideal beneficiary of superpage allocation and is the focus of this chapter. A description of array padding, a padding strategy, and experimental results are presented in the following sections.

5.2 Array Padding

Array padding is a data layout transformation that aims to reduce the number of cache conflict misses. A conflict miss is a cache miss that results from a request for a recently evicted entry. In the most pathological case, termed *thrashing*, the same set of entries may be repeatedly cached and evicted. There is a point at which conflict misses cannot be avoided, however in most cases they can be reduced by ensuring that sequentially accessed memory references with poor locality are mapped to different cache lines. If two arrays are accessed sequentially in a loop, such as in figure 5.1(a), and they are both mapped to the same cache line then a substantial number of conflict misses may be incurred. Furthermore there will be little or no data reuse since elements of each array will have to be re-fetched after eviction.

Array padding aims to force different arrays to be mapped to different cache lines so that they can reside contemporaneously in cache. Roughly speaking array padding can be subdivided into two categories: *inter-array* padding and *intra-array* padding [2, 33, 42].

5.2.1 Inter-array Padding

Inter-array padding addresses the problem of *cross interference* between array references. This occurs when two or more arrays are mapped to the same cache line. Each reference from an array forces the eviction of an element from another array. This is the type of interference experienced in 5.1(a). Inter-array padding aims to ameliorate *cross interference* by forcing

```
int a[1024],
int b[1024],
int c[1024],
int d[1024],



for(i = 0, i < 1024, i++)
        a[i] = a[i] + b[i] * c[i] - d[i],
```

(a) Array Cross Interference

```
int a[1024],
int pad[x],
int b[1024],
int pad[x],
int c[1024],
int pad[x],
int d[1024],

for(i = 0, i < 1024, i++)
        a[i] = a[i] + b[i] * c[i] - d[i],
```

(b) Inter-array Padding

Figure 5.1: Inter-array padding applied to combat cross interference

each array to a different cache line. This is accomplished through the use of a *pad*, or dummy variables that deliberately spaces apart the arrays so that they are mapped to different cache lines. The amount of padding introduced can vary but generally depends on the size of the cache, the number of cache lines, the set associativity of the cache, and the size and access patterns of application memory.

5.2.2 Intra-array Padding

```
int a[1024][1024],

for(i = 0, i < 1024, i++)
  for(j = 0, j < 1024, j++)
    for(k = 0, k < 1024, k++)
      a[i][j] = a[i][j] + a[j][k] - a[k][i],
```

(a) Array Self Interference

```
int a[1024][1024+PADDING],

for(i = 0, i < 1024, i++)
  for(j = 0, j < 1024, j++)
    for(k = 0, k < 1024, k++)
      a[i][j] = a[i][j] + a[j][k] - a[k][i],
```

(b) Intra-array Padding

Figure 5.2: Intra-array padding applied to combat self interference

Intra-array padding addresses the concern of *self interference* between array references. Instead of multiple arrays mapping to the same cache line, multiple dimensions of a multi-dimensional array map to the same line. From the perspective of the hardware inter-array padding and intra-array padding are identical, however from the perspective of a high level language they are semantically different. Figures 5.2(a) and 5.2(b) show how intra-array padding can be employed to combat *self interference*.

Avoiding *self interference* is accomplished by adding a *pad*, or in this case dummy array variables, to the leading dimension of an array. In C and C-like languages this is the right-most dimension but in Fortran it is the left-most. Similarly to inter-array padding, the amount of padding depends on a variety of factors.

5.2.3 Superpage-aware Array Padding

Algorithm 5 Superpage-aware Array Padding

1 $offset \leftarrow \left(\frac{capacity}{associativity} + x : \frac{capacity}{asociativity} + x \equiv line\ size \pmod{line\ size}\right)$
2· $total\_mem \leftarrow 0$
3. **for all** arrays a in program **do**
4· Select padding value for each array:
5 $\text{padding}_a \leftarrow offset - (len(a) \bmod (offset + 1))$
6· $total\_mem \leftarrow total\_mem + (len(a) \cdot sizeof(element_a) + padding_a)$
7. **end for**
8· Allocate $total\_mem$ bytes with *smalloc*

In general array padding is effective at reducing cache conflicts, however the effectiveness can be limited and in some cases array padding may

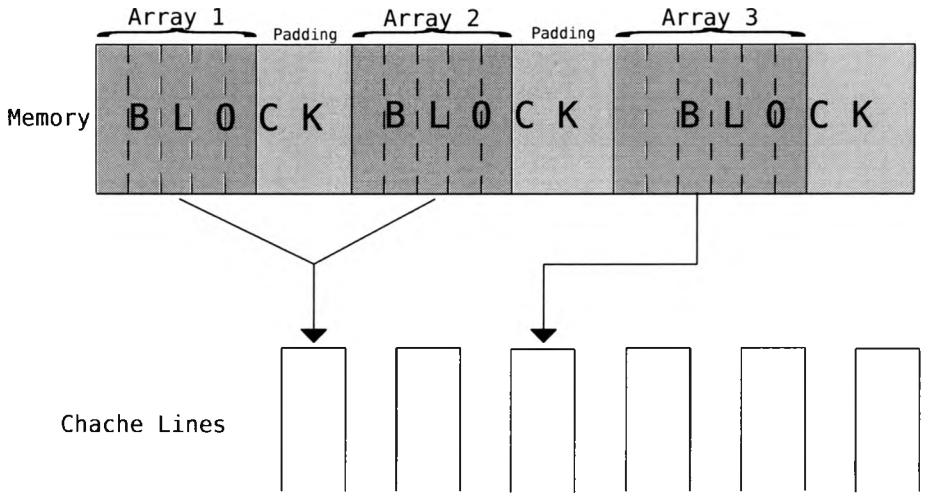


Figure 5.3: Array padding with base pages resulting in conflict

even *increase* the amount of cache conflict. Undoubtedly using randomly selected amounts of padding will result in random and unpredictable results, however on physically indexed cache architectures even intelligently selected padding values can result in undesirable behaviour. Since the contiguity of memory is not guaranteed with base pages, the compiler must either guess at the most likely mapping or assume contiguity. In some cases the use of padding may *encourage* conflict misses since the mapping of memory to cache is unknown at compile time. Figure 5.3 demonstrates how the use of inter-array padding to force alignment on page boundaries can result in the memory references being mapped to the same cache line.

Superpages can increase the effectiveness of array padding and ensure the profitability of padding (*id est* to completely eschew the chance of padding increasing conflict). Since superpages are contiguous in memory, the mapping

of pages to cache is guaranteed to be predictable and the guess-work that traditionally must be employed by the compiler is eliminated. The compiler, with knowledge of superpage allocation, can select the optimal amount of padding such that it forces arrays to be mapped to different cache lines.

Algorithm 5 describes a superpage-aware approach to array padding. The algorithm considers only *inter-array padding* since *intra-array padding* can be viewed as a derivative case of the former. Furthermore, since dynamically allocated multi-dimensional arrays are generally allocated in many small increments array padding is unlikely to be effective due to the unlikelihood that the dynamic memory allocator will place the increments of memory contiguously in memory. In order for intra-array padding to be effectively applied to *dynamically* allocated memory the space for the data structure must be allocated as one large chunk, an approach identical to the presented strategy of inter-array padding.

The array padding algorithm estimates the ideal amount of padding based upon the set associativity, capacity, and line size of the L2 cache.

$$\begin{aligned} offset &= \left(\frac{capacity}{associativity} + x : \frac{capacity}{asociativity} + x \equiv line\ size \pmod{line\ size} \right) \\ padding &= offset - (array\ size \bmod (offset + 1)) \end{aligned}$$

Intel Given an associativity of 8, a capacity of 2MB, a line size of 64

bytes, and 64K arrays, the offset is calculated as:

$$\begin{aligned} offset &= \left(\frac{capacity}{associativity} + x : \frac{capacity}{asociativity} + x \equiv line\ size \pmod{line\ size}\right) \\ &= \left(\frac{2048}{8} + x : \frac{2048}{8} + x \equiv 64 \pmod{64}\right) \\ &= (256 + 0 : 256 + 0 \equiv 64 \pmod{64}) \\ &= 256 \end{aligned}$$

The padding value is:

$$\begin{aligned} padding &= offset - (array\ size \bmod (offset + 1)) \\ &= 256 - (65536 \bmod 257)) \\ &= 256 - 1 \\ &= 255 \end{aligned}$$

AMD Given an associativity of 16, a capacity of 512KB, a line size of 64 bytes, and 64K arrays, the offset and padding is calculated as:

$$\begin{aligned} offset &= \left(\frac{capacity}{associativity} + x : \frac{capacity}{asociativity} + x \equiv line\ size \pmod{line\ size}\right) \\ &= \left(\frac{512}{16} + x : \frac{512}{16} + x \equiv 64 \pmod{64}\right) \\ &= (32 + 32 : 32 + 32 \equiv 64 \pmod{64}) \\ &= 64 \end{aligned}$$

$$padding = offset - (array\ size \bmod (offset + 1))$$

$$= 64 - (65536 \bmod 65))$$

$$= 64 - 16$$

$$= 48$$

Calculating the ideal padding is not a trivial task and there has been a great deal of research in determining the ideal amount of padding needed to reduce conflict misses [2]. Some research has even employed *genetic* or other artificial intelligence based techniques methods in determining padding values [42]. Qasem demonstrates the importance of *global array padding* over *local array padding* in reducing cache conflict [32]. Superpage-aware array padding is no exception to these concerns, and as such the proposed approach to selecting a padding value is simplistic when compared to other approaches. However, as can be seen in section 5.3, the proposed approach to superpage-aware padding is consistent in reducing the amount of cache conflict. Even if a sub-optimal padding value is selected superpage-aware array padding is still effective. Ultimately the contiguity of superpage allocated memory not only increases the effectiveness of array padding but also simplifies the work of the compiler.

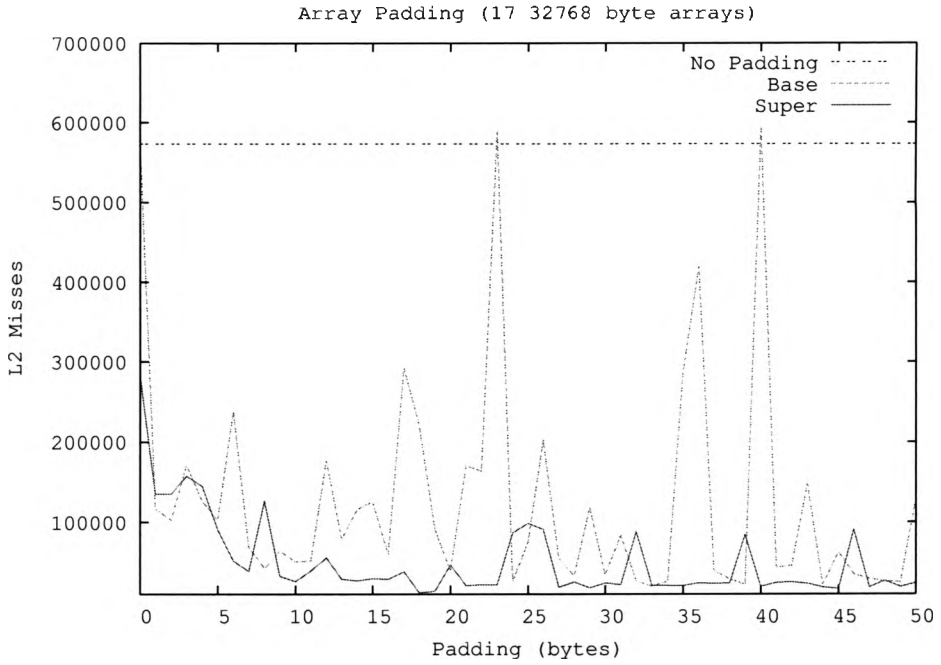


Figure 5.4: Padding without superpages performing worse than no padding

5.3 Experimental Results

Figure 5.4 shows the performance of array padding on 17 32K arrays using variable amounts of padding with both superpages and base pages. The superpage version exhibits moderate fluctuation with different padding values, however the highest number of conflict misses are incurred without padding. The base page version, on the other hand, experiences increased conflict misses on padding values of 23 and 40.

Figure 5.5 shows the performance of array padding on 17 64K arrays with both base and superpages on the Intel platform. The amount of padding ranges from none to 512 bytes. On all amounts of padding the superpage ver-

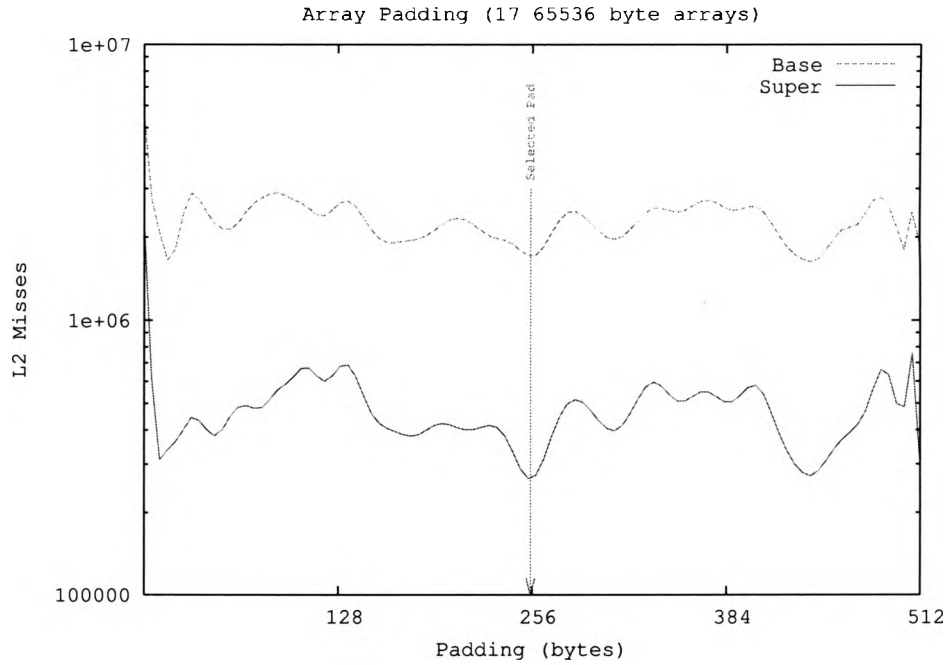


Figure 5.5: Array padding with and without superpages on Intel

sion incurs an order of magnitude less conflict misses. The complete implication of these results are difficult to quantify due to the presence of foreign factors that can effect the number of conflict misses. Other optimizations, such as prefetching, can contribute to the reduction of conflict misses and in general all of the incurred misses cannot be attributed to array reference conflicts. Nonetheless it is safe to conclude that superpages substantially increase the effectiveness of array padding.

Figure 5.6 shows padding results on the AMD platform. The improvement exhibited with the usage of superpages is still significant, but not as large as the improvements observed on the Intel platform. Again, there are a variety of factors that contribute to these results, however the difference between plat-

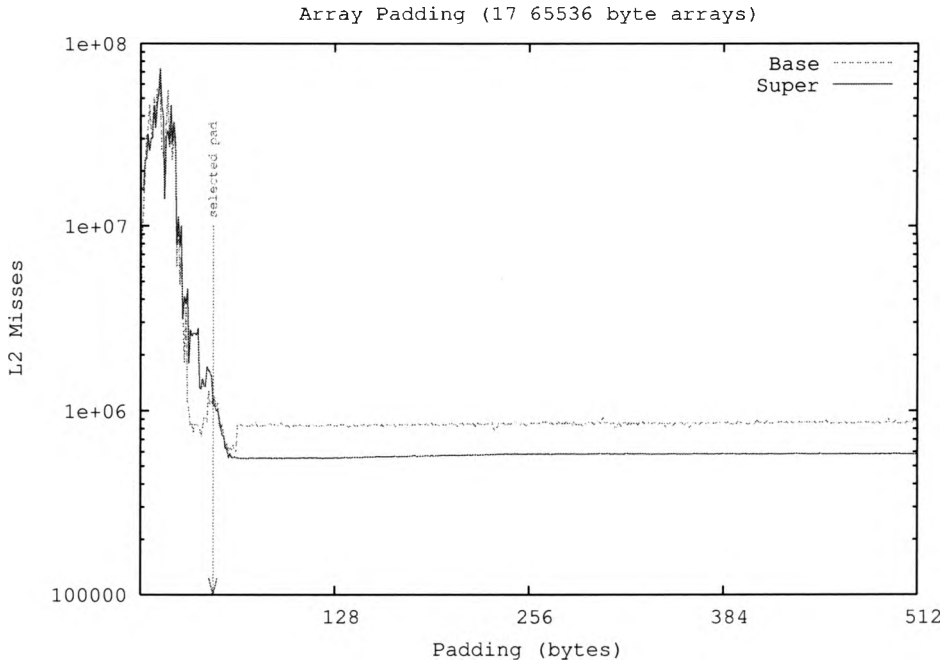


Figure 5.6: Array padding with and without superpages on AMD

forms can primarily be attributed to the varying cache configurations. Note that the difference in superpage size is unlikely to play a significant role in these results. While a larger superpage may contribute to a general improvement in conflict reduction, the size of a superpage does not have an important impact on the effectiveness of array padding. Array padding exploits the contiguous allocation of superpages and not the size.

CHAPTER 6

UTILIZING SUPERPAGES TO ESTIMATE HARDWARE PARAMETERS

Apart from reducing TLB conflicts and improving optimizations, superpages also present an important application in the field of aumatic tuning. This chapter presents a tool for estimating cache paramters by exploiting the allocation of contigious physical memory provided by superpages.

6.1 A Tool for estimating L2 Cache Parameters

Algorithm 6 Measuring L2 Cache Parameters

1. Pick initial offset s
2: Pick maximum offset e
3 **for** i from $s \rightarrow e$ **do**
4 Pick initial number of sweep regions t
5. Pick maximum number of sweep regions f
6: **for** j from $t \rightarrow f$ **do**
7. Sweep through j regions at an offset of i simulantiously
8 Record cache misses $m[i][j]$
9 **end for**
10. **end for**
11· Identify exceptional values in $m[i][j]$
12 Offset $o \leftarrow i : (\forall m : i_m = i)$
13· Region $n \leftarrow (j-1) : (\forall m : j_m = j)$
14 Cache Associativity $\leftarrow n$
15· Cache Capacity $\leftarrow n \cdot o$

Measuring L2 cache parameters, in addition to other hardware parameters, is particularly useful for self-optimizing tools used in automatic tuning. These tools require detailed information about hardware parameters to adapt themselves to different architectures. On many platforms this information may not be readily available to the tool and therefore a heuristic for estimating the parameters must be employed. However, most heuristics are generally not effective in the presence of physically indexed caches because of the uncertainty regarding contiguous allocation of memory [46]. The usage of superpages addresses this problem and a tool for estimating L2 cache parameters is provided as an example.

The tool operates by generating a series of micro-benchmarks that simultaneously sweep through multiple contiguous regions in the virtual address space. The *sweep regions* are selected so that they are non-overlapping. The number of *sweep regions* for each micro-benchmark is varied with different starting locations and access strides so as to regulate the memory access patterns. The micro-benchmarks are executed and searched for a set of sweeps that all map to the same cache line. A set of *sweep regions* that map to the same cache line can be identified by an excessive increase in the number of conflict misses. The minimum set of *sweep regions* that map to the same cache line and subsequently *thrash the cache* determine the associativity of the cache. Furthermore, once the associativity is determined the size of the cache can be determined by multiplying the associativity with the size of the offset used to control the access stride. Algorithm 6 demonstrates how the tool

estimates the L2 cache parameters. Table 6.1 outlines how the size and associativity can be derived. In order for this strategy to correctly estimate the L2 parameters all physical memory for the program must be contiguous.

Table 6.1: L2 Cache Paramater Derivations

L2 Cache Parameter	**Derivation**
Associativity	Minimum number of *sweep regions*
Size	Associativity · Offset Size

6.2 Experimental Results

Table 6.2: L2 Cache Paramaters

System	**L2 Cache Associativity**	**L2 Cache Capacity**
Intel Core 2 Duo *Turing*	16	4MB
Intel Core 2 Duo *Forkbomb*	8	2MB
AMD 64	16	512KB

6.2.1 Intel

Figure 6.1 shows the L2 cache misses for a set of micro-benchmarks generated by the tool. Benchmark *base 8* shows no observable change in the number of L2 misses for any offset. Similarly *super 8* shows little change in the number of L2 misses. At an offset of 256 *super 8* increases slightly but is too small of a change to justify any conclusions. However *super 9* shows a

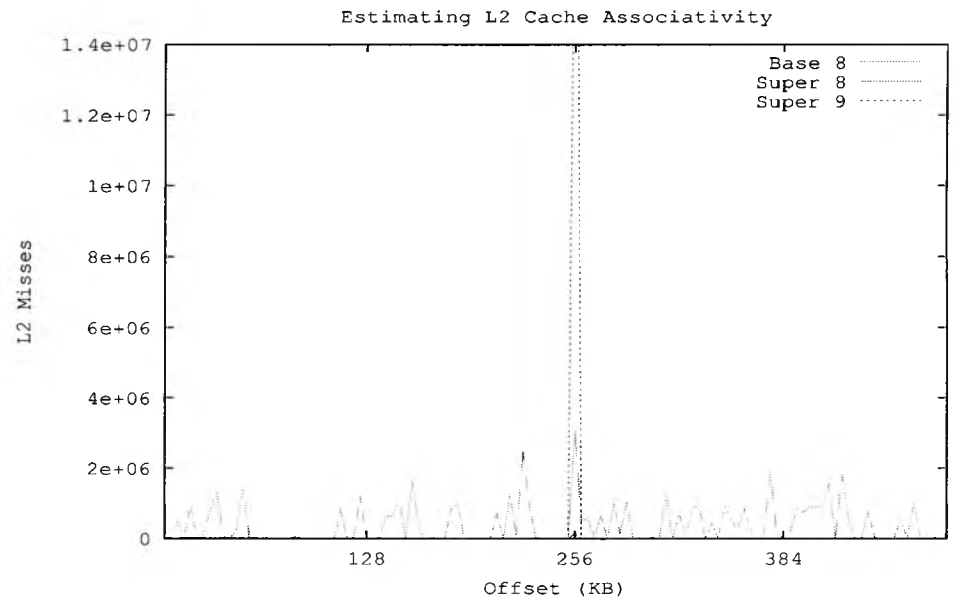


Figure 6.1: Estimating L2 Parameters on Intel Core 2 Duo *Forkbomb* using Superpages

staggering increase in L2 misses at an offset of 256. This increase suggests that memory locations that are 256KB apart will land upon the same L2 cache line. Since the number of *sweep regions* is 9 it is safe to conclude that the set associativity of the L2 cache is 8. In general the associativity of the cache will be one less then the number of *sweep regions* required to elicit a *cache thrash* since it follows from the *pigeon hole principle* that if x mappings are supported then $x + 1$ mappings will result in conflict. Given the offset and the associativity the capacity of the L2 cache can be derived to be 2MB.

Figure 6.2 shows the L2 cache misses for a second Intel 2 Core Duo machine. The cache configuration is different: The offset at which cache misses occur is at 256KB intervals like before, however the miss increase does not

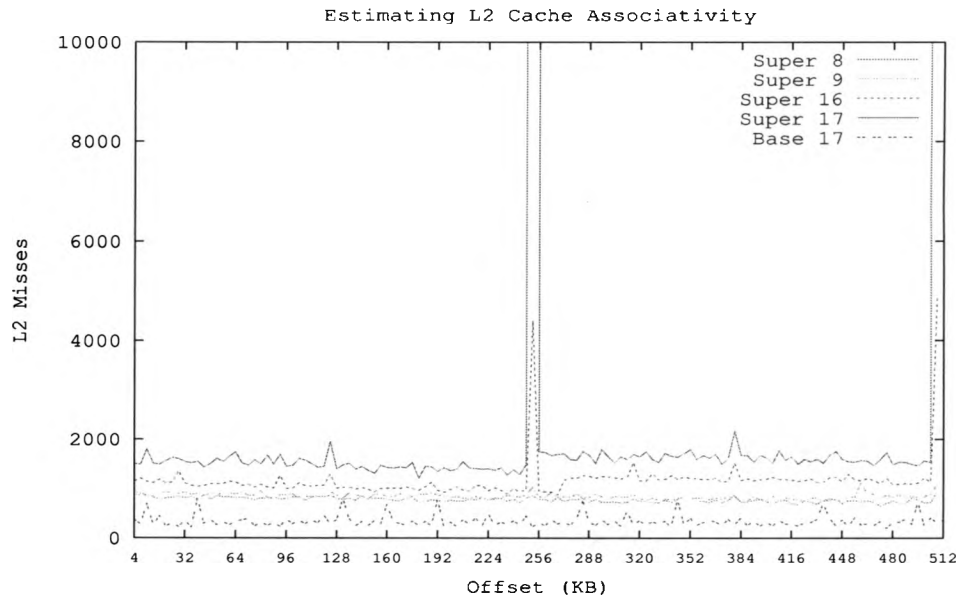


Figure 6.2: Estimating L2 Parameters on Intel Core 2 Duo *Turing* Using Superpages

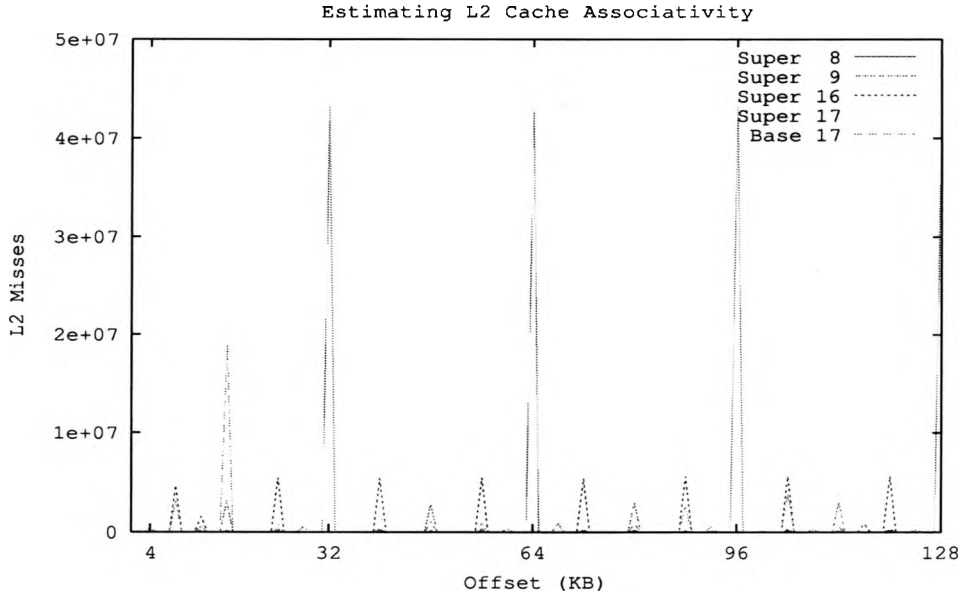


Figure 6.3: Estimating L2 Parameters on an AMD 64 using Superpages

occur until 17 simultaneous sweeps are performed. This indicates that the L2 cache associativity is 16 and the capacity is 4MB. Note that the L2 misses for a benchmark with 17 sweep regions that does not use superpages does not exhibit the expected behaviour and thus does not provide any information relevant for estimating the cache parameters. Only memory allocated contiguously, such as with superpages, can be used for estimating cache parameters on physically indexed caches.

6.2.2 AMD

Figure 6.3 shows the L2 cache misses for a set of micro-benchmarks on the AMD 64 platform. The *super 17* benchmarks shows a marked increased in cache misses at offsets of 32KB, therefore the associativity of the L2 cache is 16. Given an associativity of 16 and an offset of 32KB, the size of the L2 cache can be derived as $16 \times 32KB = 512KB$. Compared to the results from the Intel platforms, the AMD graphs generated by the tool are not as acutely apparent. Since the size of the AMD's L2 cache is relatively small compared to that of the Intel platforms the micro-benchmarks exhibit more irregular cache misses. Nonetheless, *super 17* incurs the most extreme spike in terms of number of misses.

6.2.3 Summary

The presented tool for estimating L2 cache parameters using a suite of micro-benchmarks was tested and verified on three different machines with dif-

ferent L2 cache configurations. Table 6.2 provides an overview of these cache parameters. As previously noted, the ability to accurately estimate hardware parameters is essential for automatic tuning. The usage of superpages allows for hardware parameters that cannot be measured without a contiguous allocation of memory, such as L2 cache, to be measured. This proves to be a useful extension to self optimizing tools such as X-Ray [47].

CHAPTER 7

CONCLUSIONS

This thesis presented an overview of superpages from the perspective of the compiler. After a suvey of related work (chapter 2), an in-depth overview of the implementation of superpages was presented along with our own superpage-aware compilation strategy. The implementation details of the strategy, including *smalloc* and the LLVM optimization *superpass* were discussed in depth and experimental results were provided to validate the positive impact of superpages upon application performance in chapter 3.

Chapter 4 presented a heuristic for smart superpage allocation. This heuristic was analysed by hand and correlated with experimental results. It was concluded that the heuristic effectively estimated the TLB demands exhibited by a program.

Chapter 5 discussed leveraging superpages in compiler allocations. An overview of how memory hierarchy optimizations can benefit from superpage allocated memory was presented and array padding was selected as a case study and explored in depth. Chapter 6 discussed how superpages can be used to estimate hardware parameters and demonstrated how L2 cache parameters can be obtained.

This chapter will outline the contributions of this thesis and discuss directions for future work.

7.1 Contributions

(i) Compiler Driven Superpage Allocation The primary contribution of this research is the advent of compiler driven superpage allocation. Previously there has been no attempt to direct the allocation of superpages with the compiler. While hardware and operating systems approaches logically follow from the fact that superpages are implemented at a hardware and operating system level, the compiler has access to the data-reuse patterns of the working set which allows for more judicious superpage allocation. The primary contributions are:

- *smalloc*, a superpage-aware memory allocator.
- *superpass*, an LLVM optimization pass that transforms applications so as to take advantage of superpages.
- A heuristic for superpage allocation.
- Dynamically and statically determined superpage allocation.

(ii) Improved Compiler Optimizations The compiler has a lot to offer to superpage allocation, but equally as important is that which superpages offer to the compiler. The contiguity of superpage allocated memory allows

optimizations designed to reduce conflict misses to be more effective. A strategy allowing for array padding, a memory hierarchy optimization aiming to reduce cache conflict, to exploit superpage is presented. The effectiveness of this strategy is demonstrated and evaluated. While array padding is the only presented optimization, this research establishes the importance and validity of leveraging superpages in code optimization. This research lays the ground work for studying the interactions between superpages and the compiler and how the compiler can profitably utilize awareness of superpages.

(iii) Estimated Hardware Parameters The application of superpages to estimate hardware parameters is of particular importance to the field of automatic tuning. New and innovative microprocessor-based architectures are constantly being developed and with each new platform the necessary knowledge required by software developers to port and optimize their software increases. Automatic tuning alleviates the pressure placed upon developers by automating the process of determining the optimal parameters for a system. Some hardware parameters, such as L2 cache parameters, are difficult to correctly estimate on machines with physically-indexed caches. We have demonstrated how a suite of micro-benchmarks utilizing superpages can correctly estimate L2 cache parameters on three different platforms.

7.2 Future Work

(i) Improvement to the memory manager One advantage of providing a custom memory manager for use with superpages is that the underly-

ing algorithms can be fine-tuned to minimize internal page fragmentation, the primary disadvantage to superpages. Currently *smalloc* employs algorithms to reduce fragmentation, however more advanced approaches could be applied. In general there is room for many improvements to the memory manager which are necessary to classify it as a *production grade* allocator which are not in the scope of this research and thus were not addressed.

(ii) Implementation of the proposed heuristic for compiler driven superpage allocation The effectiveness of the proposed heuristic was demonstrated in theory, however a working implementation is highly desirable. Providing a working implementation will likely be a component of the author's future research.

(iii) Implementation of compiler-driven superpage-aware array padding The effectiveness of array padding with superpages was demonstrated through a series of experiments using padding applied *by hand*. Given that it was determined that superpage-aware array padding is more effective than traditional padding it is recommended that the described approach be implemented as a compiler optimization.

(iv) Research in further code optimization that may benefit from superpages Array padding is just one memory hierarchy optimization. Other optimizations that aim to reduce cache conflicts, such as array merging, loop fusion, or tiling, may benefit from the superpages. In general, any aspect of the compiler may be a candidate for study with superpages.

(v) Research in estimating hardware parameters and utilizing these

methods in automatic tuning Estimating L2 cache parameters is just one example of how superpages can be employed in measuring hardware parameters. There are likely other architectural parameters that could be more accurately measured with superpages. Additionally, this research only demonstrates the effectiveness of superpages in this field; further research is needed to determine how best to apply the usage of superpage to existing automatic tuning strategies.

BIBLOGRAPHY

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[2] D. F. Bacon, J.-H. Chow, D. ching R. Jug, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. *Proceedings CASCON '94*, 1994.

[3] S. Bahadur, V. Kalyanakrishnan, and J. Westall. An empirical study of the effects of careful page placement in linux. In *ACM-SE 36: Proceedings of the 36th annual Southeast regional conference*, pages 241–250, New York, NY, USA, 1998. ACM.

[4] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2000.

[5] S. Browne, J. Dongarra, N. Garner, G. HO, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3), 2000.

[6] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. *SIGOPS Oper. Syst. Rev.*, 30(5):244–255, 1996.

[7] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. *ICS '99*, 1999.

[8] M. Corporation. Creating a file mapping using large pages. http://msdn.microsoft.com/en-us/library/aa366543(VS.85).aspx.

[9] A. Cox and O. Crameri. Transparent support for superpages in the freebsd kernel. Quarterly Report 2, The FreeBSD Project, 2005.

[10] J. Evans. A scalable concurrent malloc(3) implementation for freebsd, 2006.

[11] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. Mckee. Reevaluating online superpage promotion with hardware support. In *In Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 63–72, 2001.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[13] D. Gibson, A. Litke, et al. libhugetlbfs. http://libhugetlbfs.ozlabs.org.

[14] K. Gopinath and A. P. Bhutkar. Program analysis for page size selection. *IEEE*, 1996.

[15] M. Gorman and P. Healy. Supporting superpage allocation without additional hardware support. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 41–50, New York, NY, USA, 2008. ACM.

[16] L. R. Group. The llvm compiler infrastructure project, June 2009.

[17] R. Harrison and I. Weber. Molecular mechanics analysis of drug resistant mutants of hiv protease. *Protein Engineering*, 12, 1999.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufmann, 2003.

[19] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for reducing dtlb power. *IEEE*, 2004.

[20] P.-H. Kamp. Malloc(3) revisited. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, New Orleans, Louisana, USA, 1998. USENIX Association.

[21] G. Kandiraju and A. Sivasubramaniam. Characterizing the d-tlb behavoir of spec cpu2000 benchmarks. *Proceedings of the ACM SGIMETRICS Conference on Measurement and Modeling of Computer Systems*, 2002.

[22] M. Kerrisk and Contributors. Linux programmer's manual: malloc(3), Mar 2008.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[24] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 2000.

[25] H. J. Lu, K. Doshi, R. Seth, and J. Tran. Using hugetlbfs for mapping application text regions. In *Proceeding of the Ottawa Linux Symposium*, 2006.

[26] W. L. Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[27] M. Mall. Aix support for large pages. Technical report, Sun BluePrints Online, April 2002.

[28] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *In Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, 2008.

[29] R. McDougall. Supporting multiple page sizes in the solaris operating system. Technical report, Sun BluePrints Online, March 2004.

[30] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(5), 1998.

[31] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[32] A. Qasem. *Automatic Tuning of Scientific Applications*. PhD thesis, Dept. of Computer Science, Rice University, July 2007.

[33] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[34] N. Shimizu and K. Takatori. A transparent linux super page kernel for alpha, sparc 64 and ia32 - reducing tlb misses of applications.

[35] A. J. Smith and R. H. Saavedra. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, 1995.

[36] P. Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users£ Group Conference*, pages 241–248, 1990.

[37] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 171–182, New York, NY, USA, 1994. ACM.

[38] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Pearson Prentice Hall, 2006.

[39] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on numerical dense loop nests. *Proceedings of the IEEE*, 81(8), 1993.

[40] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. 1994.

[41] L. Torvalds, W. Irwin, et al. hugetlbpage.txt. Linux Kernel in-tree documentation, version 2.6.25.

[42] X. Vera, J. Llosa, and A. González. Near-optimal padding for removing conflict misses. 2002.

[43] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.

[44] T. Wildman. Linux superpages. http://sourceforge.net/projects/linuxsuperpages, October 2008.

[45] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, September 1995.

[46] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *In Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems*, 2005.

[47] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. 2005.

VITA

Josh Magee was born in Stavanger, Norway on 1980 August 18th. He received his Bachelor of Arts from The University of Texas at Austin in May of 2003.

Permanent Address: 431 Britni Loop

Kyle, Texas 78640

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab. The macros were adapted for use at Texas State University-San Marcos by Josh Magee