UNIVERSITY OF CINCINNATI

Date: 13-May-2010

I, Edward C Herrmann ,

hereby submit this original work as part of the requirements for the degree of:

Master of Science

in Computer Engineering

It is entitled:

Threaded Dynamic Memory Management in Many-Core Processors

Student Signature: Edward C Herrmann

This work and its defense approved by:

Committee Chair: Philip Wilsey, PhD
Philip Wilsey, PhD

Threaded Dynamic Memory Management in Many-Core Processors

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering
of the College of Engineering

May 13, 2010

by

Edward C. Herrmann
BSCE, University of Cincinnati, 2008

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey

Abstract

In recent years the number of simultaneous threads supported by desktop processors has increased dramatically. As the number of cores in processors continue to increase, it may prove difficult for modern software to fully exploit the amount of parallelism these processors can provide. It is essential that these extra cores are utilized to the fullest extent. To address this issue, this thesis explores the construction of a new dynamic memory library that transparently forks a new thread to service dynamic memory calls in standard applications. The goal is to offload a majority of the processing involved in `malloc` and `free` calls to a separate thread that runs on a different core than the main application thread. By placing the new memory management library in the dynamic link path ahead of the standard memory management library, the system will utilize the new dynamic memory management library in (dynamically linked) programs without requiring recompilation of the application source code.

The implementation of the threaded memory management library evolved through three different approaches that successively improved performance and reduced shared memory access costs. In the final (lock-free) implementation, profiling shows a 50% reduction in average instructions executed for the `malloc` and `free` calls of the main application thread. However, atomic instruction costs and cache effects negatively affect the runtime savings. Thus, instead of achieving a 50% reduction in dynamic memory costs, the threaded memory management library currently provides only a 25% reduction in dynamic memory costs. Experimental results with running programs show total runtime performance speedups of 2-3% in the memory-intensive SPEC CPU2006 benchmarks and speedups of 3-4% when compiling the SPEC tests using the `gcc` and `llvm` compilers. Of course, the speedups achieved are directly related to the total memory management costs of the application programs. In the above cases, the total dynamic memory costs are just over 10% of the total runtime. Applications with higher dynamic memory costs will show higher total performance improvements.

The performance of threaded programs with shared data structures can be strongly impacted by hardware implementations and operating system core scheduling policies. Therefore, a final set of experiments with different hardware platforms and operating systems was performed. In particular, the threaded memory management library was tested on AMD64 X2, Intel Core 2 Duo, and Intel Core i7 hardware and also with the Linux and Solaris operating systems. The hardware impacts were dramatic with only the i7 delivering consistent performance improvements. While not as consistent as the hardware results, the operating system results generally show better performance with Solaris over Linux.

Acknowledgments

There are a large number of people I must recognize for making contributions to this work and to my personal and professional development over the past years. I would like to thank Dr. Philip Wilsey for his gracious support and advice over the past two years. I want to thank Karthik Vadambacheri Manian, Prudhvi Janga, and Ryan Miller for their software and research support during the project. Special thanks goes out to the organizers and committee members of the MuCoCoS workshop and the 2010 CISIS conference which generously supported and published my work. I also would like to acknowledge the support provided for my work from the National Science Foundation under grant CNS–0915337 and Sun Microsystems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Trends in desktop microprocessors over the past several years show that the shift from multi-core to many-core is on the horizon. Reviewing the roadmaps of the major processor providers (Intel, AMD, Sun, and IBM) will clearly show the progression. Intel initiated their parallel processing architectures with the Core Duo and Core 2 Duo dual- and quad-core processors. The current Intel Core i7 processor supports up to eight simultaneous threads with four cores supporting 2-way simultaneous multithreading. Recently Intel announced a new version of the i7, the i7-980X that will have support for 12 threads using 6 cores each with 2-way simultaneous multithreading. IBM announced that their next generation Power7 product is expected to support up to 32 threads per chip [1]. Sun Microsystems has the T2 UltraSparc processor supporting up to 64 threads on a processor that contains up to 8 cores each supporting 8-way simultaneous multithreading. Intel has announced their experimental "single-chip cloud computer," which contains 48 cores [2] and will be available for research use in the summer of 2010. In 2007, Intel also announced a prototype for a processor that contains 80 cores [3]. Following these patterns, it is clear that desktop processors will soon contain hardware support providing the capability for hundreds of simultaneously executing threads.

While having these emerging many-core processors provides a large amount of processing power, it remains to be seen whether the full extent of this parallel form of processing power can be fully utilized on the desktop. Operating systems and many common applications can be reprogrammed with additional thread-level parallelism, and many programs have already accomplished this. However, there are limits to the amount of parallelism that can be extracted from common applications. A majority of home PC applications in use today cannot scale effectively beyond 32 threads [4]. Software such as web browsers, word processors, and multimedia applications can benefit from having a handful of threads, but cannot scale well enough to take advantage of hundreds of cores/hardware supported threads. In addition, many users often retain single-threaded legacy applications that are not programmed to take advantage of multiple cores. Due to these issues there is a need to research how to transparently extract additional parallelism out of applications. This thesis documents an exploration into this problem through the design, implementation, and testing of a custom multithreaded dynamic memory library that targets many-core processors. This library transparently migrates a majority of the processing needed for the `malloc` and `free` dynamic memory operations to a concurrent thread that pre-allocates dynamic memory blocks for rapid handoff back to the main processing thread.

Throughout the remainder of this thesis, the term core will be used to refer to any hardware processing unit that can execute a single thread. Although some processors use methods such as simultaneous multi-threading to allow multiple concurrent threads to run on a single processor core, this research will use the term *core* to represent a single hardware thread processing unit.

1.1 Hypothesis and Technical Approach

The primary goal of this research is to find techniques to help existing programs better utilize the increasing parallelism available on emerging many-core processors. The emphasis of this work is to find mechanisms that can achieve speedup with existing program binaries without changing, recompiling, or re-analyzing the original application program. That is, the techniques should be transparent to the original application program.

This thesis studies optimizations to the dynamic memory management functions of an application program. The principle hypothesis of this thesis is that *enough independent processing exists in the management of dynamic memory operations that it is possible to achieve performance gains by offloading this processing to another processor core*. Much of the processing that occurs when managing dynamic memory is not directly related to the handling of individual memory requests. Therefore this research postulates that this processing can be done in parallel and that the potential for parallelism exists in the dynamic memory library.

In order to test the hypothesis, several steps need to be taken. First, it must be proven that a majority of the processing carried out in the dynamic memory functions is in fact independent of the main application thread. This is tested by creating distributed algorithms for the dynamic memory functions where as much processing as possible is separated from the main thread and processed independently. If an algorithm is found that can reduce the processing time spent in the main thread significantly, then it will prove that a portion of the processing involved in managing the dynamic memory system can in fact be independent from the main program thread. Finding an algorithm to accomplish this task is the first goal of this thesis.

Second, it must be shown that the overhead involved with managing a separate thread and communicating data is low enough to allow a multithreaded algorithm to provide speedup. In many fine-grained multithreaded applications, communication overhead is often the main bottleneck preventing parallel algorithms from obtaining the full speedup that they are capable of achieving. If the dynamic memory library is to benefit from multiple threads, the amount of communication and other threading overhead must not offset the computation savings provided by the separate thread. The library must aim to be as lightweight as possible and reduce all associated overheads. The second goal of this thesis is to prove that the overheads costs can be minimized to the point that speedup can be obtained in dynamic memory-intensive applications.

To create a new dynamic memory management system, a series of replacement functions will be created for the standard memory allocation and deallocation functions. In particular, this work studies the development of a new library for the `malloc` and `free` functions. This new library will parallelize the housekeeping that goes on behind the scenes of `malloc`/`free` function calls by creating a separate thread to carry out these computations. This thread is spawned upon the first call to `malloc` or `free`. Any subsequent call to the dynamic memory functions communicate with this manager thread to carry out the memory requests.

Both threads communicate dynamic memory system information through a group of data structures stored in shared memory. Because these data structures are shared, any modifications must be made atomically. The main application thread uses this shared memory to grab pre-allocated blocks of memory and store addresses of recently freed blocks. The manager thread monitors the allocation patterns of the program through the shared data structures and predicts the future memory needs of the program. It then pre-allocates fixed size blocks based on these predictions and makes them available for future requests by storing information about these new blocks in the shared memory. All memory block management including allocating, freeing, and rearranging of blocks is handled by the memory manager thread, while the main application thread simultaneously executes the remaining program code.

Finally, to use this memory library in standard PC applications, the dynamic linking capabilities of modern operating systems is exploited. The new threaded memory management library can be inserted into the dynamic link path, allowing any application with high dynamic memory costs to easily use the library by simply modifying the search path used for resolving dynamically linked libraries. All calls to `malloc` and `free` will execute using the threaded library versions, allowing these programs to achieve speedup on many-core processors without requiring modifications to the source code (assuming the original program is dynamically linked).

1.2 Experimental Plan of Development and Analysis

In order to carry out this research, the first step was to learn how to create functions to override existing library functions and discover how to manipulate the dynamic link environment to allow new library code to be loaded. Once the ability to insert new function code was obtained, the development of the threaded library progressed with an iterative refinement of the implementation pursued as major improvements in the implementation strategy were identified. In the end there were three different implementations of the library created, each version using a refined approach based on the lessons learned from creating and testing the previous implementations. Analysis of preliminary results and review of the code determined that the final implementation was optimized and refined enough to begin application testing.

Of course in order for the library to provide speedup, it must be applied to applications that spend a significant amount of processing time in dynamic memory functions. The more execution time that is spent in the dynamic memory functions, the more processing can be potentially overlapped with the main thread. Therefore as the next step, the dynamic memory costs of various applications were measured to identify which ones are dynamic memory intensive. The testing focused primarily on the collection of benchmarks used in the SPEC CPU2006 suite [5].

After profiling the benchmarks, the performance of the threaded memory library had to be measured. All benchmarks in the SPEC suite were run with and without the threaded library to determine whether it provided any performance gains. The tests were repeated on a variety of hardware systems with different processors, including an Intel Core i7, AMD64 x2, and Core 2 Duo. The performance of each of these platforms was analyzed, along with the performance of the i7 using both the Linux and OpenSolaris operating systems. The data from each of these tests were compared and contrasted to determine how different hardware and software affected the performance of the threaded library.

This thesis will document the development of the proposed threaded dynamic memory library, provide an in-depth analysis of the testing results, and explain how this threading approach could be applied to other library functions.

1.3 Overview of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 provides relevant background information needed to explain this research. It provides a brief summary of the dynamic memory functions and explains why they are ideal candidates for threaded optimization.

Chapter 3 discusses previous research into dynamic memory management. A brief overview is given of modern dynamic memory implementations, including custom `malloc` libraries and garbage collection. Also discussed is the concept of parallelizing applications using computing futures.

Chapter 4 explains the methods used for profiling applications. Two profiling tools are compared and their methods of collecting data are explained. This chapter concludes by explaining the decision of which profiler was ultimately chosen and the reasons behind the choice.

Chapter 5 provides application profiling results of various test applications. Several groups of applications where heavy dynamic memory use was expected are identified and profiling results from each group are shown and discussed.

Chapter 6 describes the development and implementation of a threaded dynamic memory library. Three separate approaches towards creating a threaded library are explored. Each method is explained in detail and the progression of the library through each phase is documented.

Chapter 7 discusses the results of the SPEC benchmark tests. The results of the memory-intensive benchmarks are examined in detail and differences in expected and potential speedup are explained. Also discussed are results from compiler comparison tests.

Chapter 8 gives an overview of the various hardware platforms that were tested with the library and compares how the architectures affected the results. Systems tested include an Intel Core i7, Intel Core 2 Duo, and AMD64 x2 processor.

Chapter 9 investigates how software running on the system affects the library results. In particular, the effects of different compilers and operating systems and their effects on library performance are explored. Compilers tested include `gcc`, `llvm`, and `Sun Studio`. Linux and OpenSolaris are the two operating systems compared.

Chapter 10 overviews further optimizations that could be applied to the library. Improved cache performance, better prediction algorithms, and multithreaded enhancements are outlined.

Chapter 11 discusses how the multithreaded approach could be applied to other libraries. Preconditions for finding good candidate functions are explained. A sample class structure to assist in creating multithreaded libraries is also proposed.

Finally, Chapter 12 concludes the research and summarizes the results. The overall findings of this thesis are laid out and the implications these findings have on future developments is discussed.

Chapter 2

Background

Each year microprocessor manufacturers are continuing to increase the number of processor cores per chip at an steadily increasing rate. With this increase in computing power, the question arises whether the general user will be able fully benefit from this parallelism. One crucial part of many programs that could potentially take advantage of this increased parallel processing power is the dynamic memory management functions. The following section discusses the evolution of many-core computing and explains how the dynamic memory system can take advantage of the increased parallelism in future desktop processors.

2.1 Trends in Computer Architecture

Recent trends in desktop processor design have shifted towards many-core solutions with increased parallelism. Throughout the nineties processor design focused on improving performance by increasing clock speeds and utilizing new methods of instruction-level parallelism in the processor core such as improved pipelining, branch prediction, and dynamic scheduling [6]. However, the point of diminishing returns has been reached in the field of instruction-level parallelism and power and complexity concerns have slowed the performance gains due to transistor speed increases [6]. Due to these issues, computer architects are turning to thread-level parallelism on multi-core systems to provide future performance gains. As the number of supported threads increases with each new generation of processors, it may prove difficult to fully exploit this extra parallelism on commodity desktops.

2.2 Dynamic Memory Management

In a modern Linux/Unix environment, the dynamic memory system is a series of functions that resides in the `glibc` library collection. These functions are designed to allocate and deallocate memory on the fly to applications that request it. The dynamic memory library operates by grabbing memory from a global memory region called the heap and partitioning it to the running application on a per request basis. If the heap runs out of memory, more memory can be requested from the operating system.

The basic request function is `malloc`. The `malloc` function takes a size argument that specifies the minimum size block of memory required for the particular allocation (in bytes). If successful, the `malloc` function returns the address to a new block of memory of the specified size. The `free` function is called to release memory back to the operating system. Its only argument is the memory address of the block to free, and it has no return value. There are many internal data structures used to record where the boundaries are between memory blocks and whether or not each block is currently in use. These implementation details are abstracted from the application programmer; allowing programmers to call the `malloc` and `free` functions without having to worry about managing the internals of the memory management system.

Dynamic memory was developed for modern programming languages for a variety of reasons. Before dynamic memory existed, program memory had to be statically allocated by the compiler, or allocated on the stack as local variables. Statically allocated buffers have a lifetime that extends throughout the duration of the program, while local variables only have function-specific lifetimes. Dynamic memory provides more flexibility by allowing the programmer to explicitly control when blocks of memory are allocated and released. This allows programs to be more efficient in their memory usage. Dynamic memory also provides greater flexibility when programming applications that have varying memory needs, such as applications that require user input or communication buffering. In addition, many modern object-oriented languages use dynamic memory to allocate objects at runtime, allowing for the development of more modular and easier to understand code.

There are several reasons why the dynamic memory library is a good candidate for this research. First, the library is dynamically linked. Therefore the library code exists in an external library and is not usually compiled into the application binaries. The dynamic link path is easily changed to load in a different dynamic memory library, allowing a replacement library to be transparently used in any application without requiring recompilation. Second, dynamic memory is used in a wide variety of application programs. Programs coded in object oriented languages in particular are known to use quite a bit of dynamic memory [7] [8]. Third, the dynamic memory allocation system contains a significant amount of processing that is independent of any specific memory request. This processing can be performed in parallel to the main execution thread. Thus, the dynamic memory system contains a notable amount of processing that can be parallelized.

Chapter 3

Related Work

The area of dynamic memory has been widely studied over the years. Due to scalability issues with the first dynamic memory allocators, many allocation systems have been developed to provide better performance in threaded applications. More recent research has been focusing on automatic memory management in the form of garbage collectors. The following sections will outline previous efforts in the area of dynamic memory and discuss the idea of offloading computation in program futures.

3.1 Traditional Memory Allocators

Many standard memory allocators were designed well before multithreading became commonplace [9]. In order to adapt these older memory allocators for multithreaded applications, many allocators added mutex locks around the critical library functions to ensure memory consistency. These locks add synchronization overhead to the critical paths of all allocations and deallocations [10]. When multiple threads want to access the allocation functions at the same time these mutex locks can create a bottleneck for threaded applications. A multithreaded program will essentially become sequential if threads must wait to enter the atomic sections of code [11]. As a result, traditional dynamic memory allocators tend to scale poorly with multithreaded applications.

3.2 Memory Allocators for Threaded Applications

To overcome this problem, some enhanced allocators have been developed to avoid the locking problem. One method is to create algorithms that avoid locking mechanisms altogether. Dice and Garthwaite [9] and Michael [12] created custom allocators that accomplished this. Hudson *et al* also created the `McRT-Malloc` lock-free allocator [13]. Another method to avoid locking is to separate the heap area by creating localized regions for each thread. The thought behind this method is if each thread has its own memory region to allocate from there will not be any contention, so no locking mechanisms are needed. `ptmalloc` [12] is an enhancement on Doug Lee's original `dlmalloc` [14] that incorporates separate heap regions. Another allocator that is locally conscious is Streamflow [10]. A more recent allocator that incorporates heap regions is the Hoard allocator [15]. Created by Berger *et al*, Hoard creates thread-local heap regions that can grow or shrink by borrowing blocks from a global heap. Hoard also uses a technique to avoid false sharing, which occurs when blocks from two different heap regions share the same cache block.

Many other custom allocators have been developed to help improve the scalability of dynamic memory allocation [16] [17] [18] [19] [20] [21]. The goal of these custom dynamic memory allocators is to help eliminate the problems that traditional allocators have with multithreaded applications. However, none of these allocators aim to extract additional parallelism that potentially exists in the memory allocation subsystem.

3.3 Prototype Threaded Allocators

A research group from North Carolina State University has recently published work on their own prototype threaded memory allocator [22] [23]. Devesh Tiwari *et al* created the MMT custom dynamic memory library with a similar goal of transparently threading off `malloc` and `free` processing. Although their research into threaded dynamic memory management has progressed in parallel with this study, the conclusions they have drawn from their investigation confirm many of the findings discussed in this thesis. In particular, their work emphasizes the importance of reducing communication costs, pre-computing memory blocks, and maintaining cache locality. Using similar techniques to the ones used in this study, the library produced through their research was able to achieve some speedup in a selection of benchmarks which spend around 30% of their computation time in dynamic memory functions. Their work also explored the benefits of threading dynamic memory management in "safe" allocation libraries that perform additional security checks. The MMT library was not designed, however, to support multithreaded applications like the library developed in this thesis does. The positive results produced by the MMT research group along with the findings of this thesis show that exploration into transparent, fine-grained multithreading can provide potential methods to achieve some speedup with existing legacy applications.

3.4 Garbage Collection

In addition to memory allocation libraries, another active area of research in dynamic memory management is automated garbage collection. Garbage collection is a method of assigning and reclaiming heap memory that uses a separate automated entity called a *garbage collector* to monitor and walk through the heap, reclaiming any memory that is no longer in use. Garbage collection falls under the category of automatic memory management. Since the focus of this work is on manual memory management, the remainder of this thesis will discuss research in dynamic memory allocation libraries.

3.5 Program Futures

Another topic related to this work is the concept of *computing futures* (or simply *future*). A future is defined as "a promise to deliver the value of a subexpression at some later time." [24]. Certain parts of programs can be broken down into functional arguments, where futures can be used to evaluate argument expressions in parallel. Each future is assigned to an evaluator process to calculate its value. This allows each future to be calculated independently on separate processors. When the value of a future is needed, if the value is ready it can used immediately. If not, the process must block until the evaluation is complete. Futures use "eager evaluation" where as soon as a value is anticipated to be needed, evaluation will start [24]. This approach introduces the possibility of wasteful calculation if the precomputed future value ends up unused. The initial implementation by Baker and Hewitt dealt with memory management by using futures in a garbage collection system [24].

The threaded dynamic memory approach of this thesis is similar to futures in that it uses eager evaluation by designating an extra processor to pre-allocate memory blocks before they are needed. However, futures are a programming feature that must be supported by the programming language and explicitly coded into software. The threaded memory approach in this thesis uses past allocation patterns to predict future requests and does not rely on the application to explicitly provide information about future processing. In general this work attempts to be independent of the application software so that speedup can be gained transparently without requiring recoding of the original application program.

Chapter 4

Measuring Dynamic Memory Use

To prove that performance improvements obtained by threading dynamic memory are meaningful, it must be shown that the time spent in dynamic memory functions is significant. According to Amdahl's Law the overall performance gained by any optimization is constrained by the amount of time the program spends in the optimized portion of code. Therefore, only programs that spend a reasonable amount of execution time performing dynamic memory tasks will show measurable improvements. For example, if an application spends 10% of its execution time in dynamic memory functions, an improved allocation library will only be able to save at most up to 10% of the processing time.

The cost of dynamic memory management is heavily program dependent. There are many factors that determine the dynamic memory behavior of a process: the frequency of memory allocations (`malloc`s) and deallocations (`free`s), the size of the requested blocks, the pattern in which the blocks are requested, and the order in which the requests occur. All of these attributes can also affect memory fragmentation, which can slow down the search times for future allocations. It is also difficult to obtain consistent measurements because program behavior can change for each run depending on the program inputs, external interrupts, and scheduling of other processes. Because of these issues, the only way to collect accurate data is by measuring actual program runs.

Due to the high frequency at which dynamic memory functions are called, inserting code around the functions to measure time would drastically alter the program execution. In addition, the time measurement functions would not be precise enough to accurately measure the time periods spent inside the library functions. The best method of measuring execution costs of dynamic memory functions is through profiling.

Profiling is a way of obtaining internal data from actual execution runs through the use of program traces. The trace data is examined after the execution completes to determine the amount of time spent in each code portion of the program.

Profiling tools can be broken down into two main categories, namely: *sampling* and *instrumentation*. *Sampling profilers* gather statistical data by periodically recording an application's program counter during runtime to determine what area of code is being executed. Sampling provides an unobtrusive method to observe general program behavior, although the collected data is only an approximation of the overall program execution. In contrast, *instrumentation profilers* modify the application code by adding program probes in each program branch path to obtain a more complete picture of the path a program takes to completion. Instrumentation profiles are more obtrusive as they insert additional instructions into an application and the application typically suffers significant performance degradation as a result.

There are several profiling tools publicly available, but most require the source code of the test applications to be recompiled. Fortunately, some profiling tools are able to operate on program binaries. For this work, two packages were tested for profiling binary programs, namely: *Valgrind* [25] and *Oprofile* [26].

Oprofile is a sampling profiler that uses hardware performance counters inside the CPU to collect data about running programs. In particular, oprofile reads out the values of the program counters to learn what code region in memory is currently executing. A kernel driver and daemon are used in tandem to collect sample data from all of the processors. Because it collects data at the instruction level the entire system is profiled, including user applications, the kernel, interrupts, and libraries. Oprofile contains several analysis tools to interpret the collected data and summarize the number of samples found in each section of code.

Valgrind is a runtime instrumentation profiler that executes applications inside its own virtual machine, simulating program execution on a per-instruction basis [27]. This allows Valgrind to monitor the state of the machine during each instruction and trace the path the program follows to completion. Valgrind comes packaged with several tools that utilize this extra information to collect useful debugging and profiling data. The tool of choice for this research was the Callgrind tool.

Callgrind is a profiling tool that takes data provided by Valgrind and creates a complete callgraph tree, tracing program execution from start to finish. Callgrind monitors the program stack inside the Valgrind virtual machine and records a trace of all function calls, which includes the total number of instructions executed inside each function. It can also make use of the simulated cache hierarchy built into Valgrind to obtain a more accurate measurement of the time spent inside each function. A separate data analysis tool called KCachegrind is used to read the trace files from Callgrind and display the function call hierarchy and then calculate the relative execution costs of each function.

After using both profilers, it was found that Valgrind/Callgrind produces results that are more accurate and more relevant to the needs of this study. Most of the differences between the two profilers stem from the way each collects data. Since Valgrind runs the program code through its own instrumented virtual machine, it is capable of closely monitoring the behavior of a program. Callgrind can use this data to construct complete callgraphs, allowing it to count the exact number of function calls and the number of instructions executed inside each function. Due to the sampling nature of oprofile instructions are only monitored at fixed intervals, leading to an incomplete picture of the overall execution path of the program. While sampling can provide an unobtrusive way to get a general idea of where execution time is spent, the collected data may not be representative of the actual path of program execution. In addition, oprofile collects data from all running processes including the kernel, making it impossible to separate which samples came from which process when examining shared code such as dynamic libraries. Even though Valgrind profiling can slow a program down to 1/20th of its normal speed, the data collected presents a more complete picture of where the execution time is spent in a single running process.

While Valgrind was the profiler of choice for this study, it is important to note some of its limitations. As mentioned before, Valgrind only profiles application code and libraries; it does not collect data from kernel mode. As a result, any code that executes a system call that traps to the kernel will not be represented in the profiling results. In addition, Valgrind is only capable of simulating a single core architecture. In general this limitation does not affect the number of instructions a program executes. A multithreaded application typically executes the same number of instructions on a single-core machine as it would on a multi-core machine. However, hardware interactions between multiple cores such as atomic memory operations and cache coherency are not captured. Despite these limitations, Valgrind still provides a fairly accurate representation of the execution path of a program.

Chapter 5

Dynamic Memory Execution Costs

Before discussing the profiling results it is important to clarify exactly what type of programs this research is targeting. It can be easy to confuse a program that is dynamic memory intensive with one that uses a lot of dynamic memory. Just because an application uses a large amount of dynamically allocated memory does not mean that a large amount of time was spent in allocating and deallocating that memory. Programs that stand to benefit from multithreaded memory management are programs that spend a large amount of processing power in executing calls to the dynamic memory functions. Any application that frequently allocates and deallocates memory most likely spends significant time in the `malloc` and `free` functions, making it a good candidate for optimization with the threaded memory library.

When developing the library, testing primarily focused on the SPEC CPU2006 [5] benchmark suite for performance analysis. The SPEC benchmarks are processor intensive and consist of representative programs from a diverse range of applications. The benchmarks are widely used by the computer architecture community to study design tradeoffs. They also come complete with batch execution scripts to facilitate their use in testing and profiling system performance. The entire SPEC benchmark suite was profiled with Valgrind to analyze the amount of time spent in the dynamic memory functions. The results of these tests are shown in Figure 5.1.

The results show that dynamic memory costs vary from benchmark to benchmark. The majority of the tests use dynamic memory sparingly and spend insignificant amounts of time in the standard memory functions. There are however three benchmarks that spend more than 10% of total processing time dealing with dynamic memory management, namely: `tonto`, `gcc`, and `omnetpp`. `tonto` [28] [29] is a quantum

fp_tests	**percent of total computation**				
	malloc	**realloc**	**calloc**	**free**	**total**
bwaves	0.00	0.00	0.00	0.00	0.00
gamess	0.00	0.00	0.00	0.00	0.00
milc	0.01	0.00	0.00	0.00	0.01
zeusmp	n/a	n/a	n/a	n/a	n/a
gromacs	0.00	0.00	0.00	0.00	0.00
cactusADM	0.00	0.00	0.00	0.00	0.00
leslie3d	0.00	0.00	0.00	0.00	0.00
namd	0.00	0.00	0.00	0.00	0.00
dealll	n/a	n/a	n/a	n/a	n/a
soplex	0.01	0.00	0.00	0.00	0.01
povray	0.03	0.00	0.00	0.02	0.05
calculix	0.02	0.00	0.00	0.01	0.03
GemsFDTD	0.01	0.00	0.00	0.00	0.01
tonto	7.52	0.00	0.00	5.96	13.48
ibm	0.00	0.00	0.00	0.00	0.00
wrf	0.17	0.00	0.00	0.09	0.26
sphinx3	0.00	0.00	0.23	0.06	0.29

int_tests	**percent of total computation**				
	malloc	**realloc**	**calloc**	**free**	**total**
perlbench	0.96	0.16	0.00	0.57	1.69
bzip2	0.00	0.00	0.00	0.00	0.00
gcc	0.19	0.00	9.68	0.28	10.15
mcf	0.00	0.00	0.00	0.00	0.00
gobmk	0.01	0.00	0.00	0.00	0.01
hmmer	0.02	0.02	0.01	0.02	0.07
sjeng	0.00	0.00	0.00	0.00	0.00
libquantum	0.00	0.00	0.00	0.00	0.00
h264ref	0.00	0.00	0.00	0.00	0.00
omnetpp	5.86	0.00	0.00	4.24	10.10
astar	0.21	0.00	0.00	0.12	0.33
xlancbmk	2.26	0.00	0.00	1.24	3.50

$^{n/a}$Data for these SPEC benchmark programs were not gathered due to their extremely long callgrind runtimes.

Table 5.1: SPEC CPU2006 callgrind profiling results

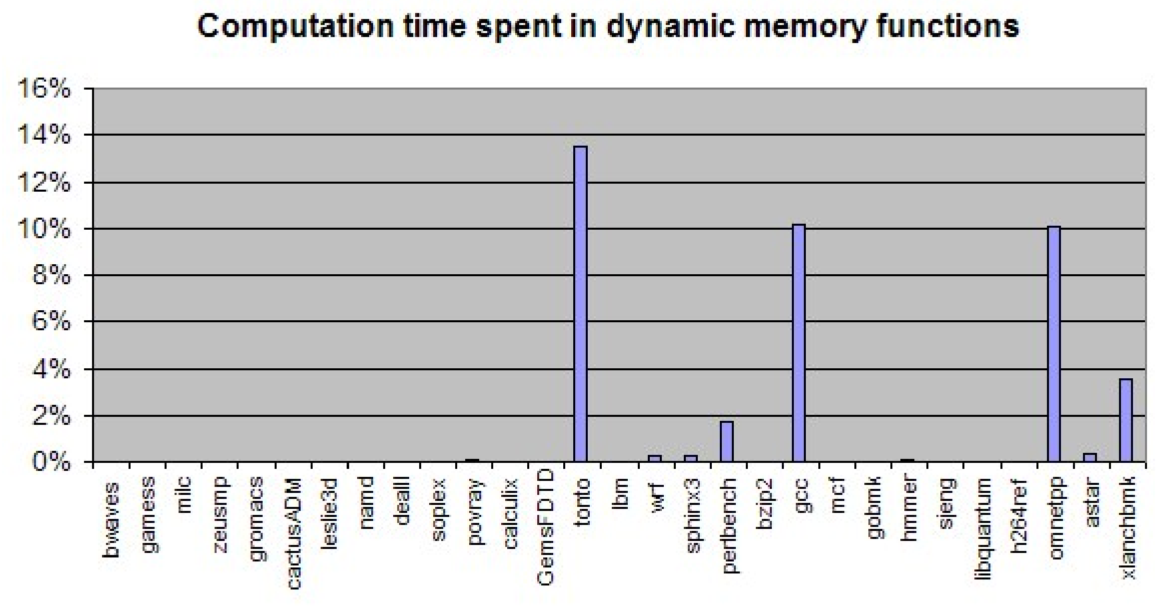

Figure 5.1: Profiling results for SPEC CPU2006 benchmarks

chemistry package, `omnetpp` [30] [31] is a network simulator, and `gcc` [32] [33] is the standard GNU C compiler. Testing for the threaded library dynamic memory focused primarily on the performance of these three memory-intensive benchmarks.

A wide range of other applications were also profiled with Valgrind. First, a small collection of standard, fairly CPU-intensive, Linux applications were run through Valgrind to gather information on how much dynamic memory processing occurs in common everyday programs. The profiling results for these applications are given in Table 5.2. Following these tests, a few groups of applications where it was presumed dynamic memory played an important role were chosen for further investigation.

There were several types of applications where frequent dynamic memory use was expected but not found. One of these groups was computer games. Games tend to be fairly processor intensive and most tend to create and destroy objects frequently. The profiling results of a number of open source Linux games (Table 5.3) showed that most games contained fairly little dynamic memory computation. Many games perform a lot of allocations upon initialization, but allocate less frequently afterwards. As it turns out, some older games either pre-allocate memory during loading, or they manage blocks of memory internally, cutting the system dynamic memory allocator out of the equation. The reasoning behind these methods is that traditional dynamic memory allocators tended to be a performance bottleneck for games. Many

		percent of total computation				
Test	**Description**	**malloc**	**realloc**	**calloc**	**free**	**total**
firefox	load top 10 most visited websites	1.08	4.88	2.13	1.62	9.71
vlc	2 minutes of 420p MPEG4 video	1.86	1.18	0.40	1.29	4.73
gcc	compiling the new malloc library	26.37	0.10	0.24	0.10	26.81

Table 5.2: Linux applications callgrind profiling results

		percent of total computation				
Game	**Description**	**malloc**	**realloc**	**calloc**	**free**	**total**
warzone2100	3D RTS	0.42	0.00	0.05	0.27	0.74
wolfenstein: ET	3D FPS	0.03	0.00	0.01	0.03	0.07
torcs	3D racing sim	0.02	0.03	0.01	0.00	0.06
openarena	3D FPS	0.04	0.01	0.02	0.04	0.11
tuxsupercart	3D racing	0.78	0.01	0.03	0.38	1.20
racer	3D racing	0.60	0.14	0.84	0.89	2.47
spring	3D RTS	1.62	0.36	0.02	1.17	3.17
Quake 4 demo	3D FPS	0.08	0.00	0.01	0.05	0.14
Americas Army	3D FPS	0.74	0.27	0.01	0.60	1.62
tremulous	3D FPS	0.03	0.00	0.01	0.02	0.06

Table 5.3: Linux games callgrind profiling results

developers preferred internal memory management believing that it was more efficient than system-level allocation (although studies have shown this is no longer the case for newer allocators [34]) Since dynamic memory allocation calls can perform varying levels of processing (especially if a system call is needed to grab more memory from the operating system), there is no guarantee that an allocation request will be fulfilled within a fixed amount of time. In games, calculations need to be done quickly in order for the program to be responsive, so many game developers tended to avoid using dynamic memory extensively during game operation.

Another area where more dynamic memory use was expected was scripting languages. Scripting languages tend to be object-oriented, so a decent amount of dynamic memory processing was expected in the

		percent of total computation				
Test	**Description**	**malloc**	**realloc**	**calloc**	**free**	**total**
sunspider	javascript benchmark	1.89	1.80	0.32	1.29	5.30
scimark	java benchmark	0.11	0.00	0.01	0.07	0.19
pybench	python benchmark	1.55	0.11	0.00	1.27	2.93

Table 5.4: Scripting language benchmark callgrind profiling results

		percent of total computation				
Simulator	**Description**	**malloc**	**realloc**	**calloc**	**free**	**total**
google earth	map simulator	2.85	0.31	0.04	2.19	5.39
step	physics simulator	5.95	2.96	0.14	4.61	13.66
ns2	network simulator	0.66	0.03	0.00	0.64	1.32

Table 5.5: Simulator callgrind profiling results

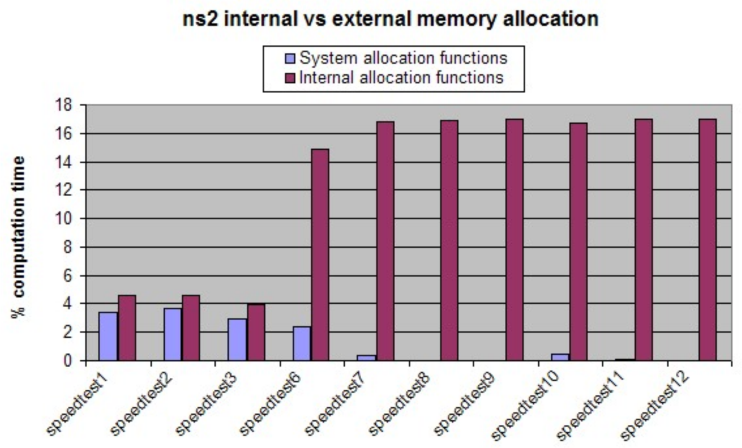

Figure 5.2: Internal allocation processing vs system allocation processing for the ns2 simulator

various scripting benchmark tests that were run. The results from these benchmarks (Table 5.4) showed some time spent in dynamic memory, but not the large amounts expected. The callgrind results show that several scripting languages such as java and python contain their own internal garbage collection system, which reduces the amount of calls made to the system `malloc` and `free`. Since these applications manage memory internally, the system dynamic memory functions are used rarely, preventing the threaded library from having a large effect on performance.

The last genre of programs tested were simulators. Simulations, especially discrete event simulations, should create and destroy objects fairly frequently, which requires frequent dynamic memory usage. The results from the group of simulators tested (Table 5.5) show a decent amount of dynamic memory usage. The ns2 network simulator was expected to have the largest amount of dynamic memory, but again internal packet allocation and deallocation functions prevented the system dynamic memory allocator from getting getting used frequently. Figure 5.2 shows the percent of time spent in the internal packet allocator functions versus the system `malloc` and `free`.

Overall, the results of this research show that simulators and compilers are key areas where the system dynamic memory functions are used frequently. Unfortunately, several applications that were found to be memory-intensive could not be used for performance testing. Programs that rely on user-input to function, such as Google Earth, Firefox, and the Step simulator, cannot be easily scripted for testing purposes. Other programs such as games contain a variable amount of processing for each run, so identical comparisons cannot be made. In addition, certain applications could not be timed because they involve non-continuous processing. For example a two minute video clip played in VLC will always take two minutes to complete, so there is no way to compare the performance with a new library. Due to these limitations, the best programs to make comparisons with are benchmarks and constant fixed-processing applications. Therefore when testing the library, this study focused primarily on the performance of the memory-intensive SPEC benchmarks and various collections of compiler tests.

Chapter 6

Threaded Library Development

When the basic `malloc` and `free` functions are invoked, the processing that is carried out can be split into two categories: *request processing* and *system processing*. Request processing is the minimal processing required to fulfill the current request. For a call to `malloc`, this includes finding a block of free memory of the adequate size and returning the address. For calls to `free`, the request-specific processing involves marking the block at the specified address as unused. A memory allocation system cannot function with just the minimal processing. Additional work needs to be done to make sure the system runs smoothly and is prepared for future allocation requests. This can be referred to as system processing. System processing includes coalescing adjacent free blocks, maintaining sorted linked lists of available blocks, and requesting new chunks of memory from the operating system. The work done by the system processing maintains efficiency by keeping the level of internal and external memory fragmentation low and by performing required maintenance on internal data structures.

Both types of processing are essential to the successful and efficient operation of a dynamic memory management system. Current implementations of dynamic memory management systems distribute system processing across all the function calls. Requests for more memory from the O/S are made by `malloc` when no more free blocks are available. The `free` function combines newly freed blocks with adjacent free blocks as they arrive. Both functions maintain internal linked lists and other data structures that allow for quick indexing of freed blocks. In single processor systems this method of distributing the system processing alongside the request processing works very well. The main program has to be interrupted at some point to handle the system processing, so dividing the processing up evenly among the calls allows the system to carry out each request in a reasonable amount of time. In multiprocessor environments, the main program does not have to be interrupted to handle system processing. If properly implemented the system processing can be done in parallel with the main program, performing background cleanup and maintenance operations on the dynamic allocation system while the main program continues. The goal of this research is to create a plug-and-play dynamically linked library that will take advantage of this parallelism by having a separate thread take care of the system processing. Ideally the threaded library developed in this study will try to overlap the system and request processing as much as possible without causing negative side effects in the main program thread.

The first step was to override the default system `free` and `malloc` commands. Since Linux was used as the development environment, the `LD_PRELOAD` environment variable allowed easy overriding of the default library search path. By creating a new dynamic library that redefined the dynamic memory functions and placing it earlier in the dynamic link chain, any dynamically linked program will use the replacement `malloc` and `free` functions. For this work, completely recreating the entire dynamic memory system was out of the question. Therefore, the memory management system was built on top of the original memory functions by allowing the library to act as a wrapper. The `dlsym` function recurses through the dynamically linked library structure to locate the next occurrence of a particular function. Using this command the custom library was able to create new chunks of memory by calling the system `malloc` command, and similarly was able to free memory using the system `free`. Finally, the custom library had to spawn a separate thread upon initialization to handle the dynamic memory requests. An initialization flag was used to detect the first call to any dynamic memory function. Upon the first call, the initialization flag was set and a separate thread was spawned. This thesis will refer to this thread as the “manager thread,” while the thread carrying out the rest of the program operation will be referred to as the “program thread” or “main thread.”

6.1 Pass-Through Approach

The library development went through several iterations before a suitable configuration was discovered. The first implementation attempted to pass each call to `malloc` and `free` directly from the program thread to the memory manager thread. This implementation will be referred to as the *pass-through* (or *PT*) implementation. Figure 6.1 provides a visual representation of how the program thread signals and communicates with the manager thread to service the `malloc` and `free` function calls. Using this approach, the `malloc`

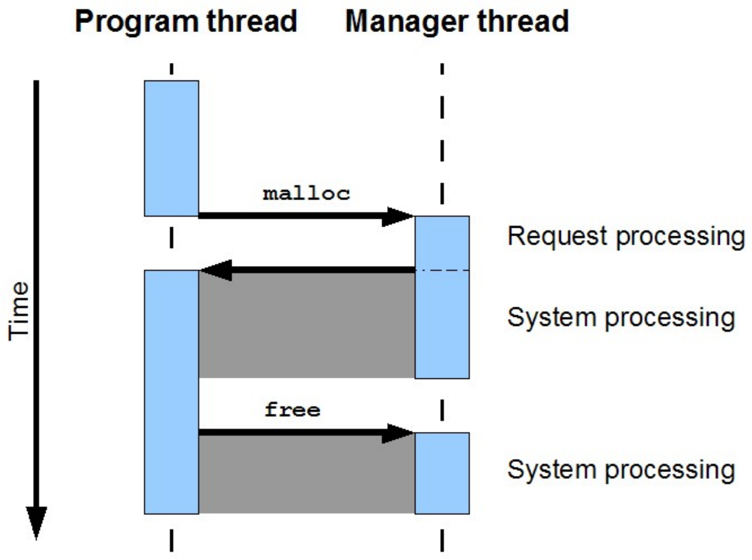

Figure 6.1: Pass-through (PT) algorithm approach

function cannot be fully parallelized because the program thread must wait for a return address. The `free` function on the other hand can be executed entirely in parallel because no return value is expected from the `free` function call.

In the PT implementation the library creates the manager thread upon the first invocation of `malloc` or `free`. The Native POSIX Thread Library (NPTL) was used to create and synchronize the threads. To ensure the fastest possible communication between threads, shared memory was used to transport all inter-thread data. Although shared memory is the quickest form of communication, care must be taken to ensure the threads never accessed memory locations at the same time. To protect from race conditions and ensure thread safety, pthread mutexes were inserted around any code that modified shared structures. Mutexes are synchronization variables that can be locked and unlocked atomically. Only one thread can hold a mutex lock at any particular time. Any thread trying to lock a mutex that is already locked will block until the thread holding the mutex unlocks it. In addition to mutexes, the pthread library also provides thread synchronization mechanisms known as condition variables. Whenever a thread required communication from the other thread, it would sleep on a particular condition variable using the `pthread_cond_wait` function. The other thread would then wake up the sleeping thread by signaling on that same condition variable using the `pthread_cond_signal` function. Whenever a thread was waiting for a response, it would sleep waiting on a particular condition variable until the other thread signaled on that condition variable.

```
struct message
{
  short int type;
  int intparameter;
  void *addrparameter;
  void *returnvalue;
}
```

Figure 6.2: Pass-through (PT) implementation shared data structure

The system uses a basic message structure (Figure 6.2) to pass requests between the program thread and the memory manager thread. The `type` variable specifies whether the request is a `malloc` or `free` request. The `intparameter` variable is used to pass the size argument for allocations and the `addrparameter` variable is used for identifying the block address on free requests. Return block addresses are passed back to the program thread through the `returnvalue` variable.

Because the message passing structure was shared between threads, a data mutex lock had to surround any code that modified any data in the message. Furthermore, since a single message structure was used to communicate between threads, an additional request mutex lock had to surround any `malloc` or `free` call to ensure that future requests did not disturb the data before the current request completed. Since the manager thread processed only one request at a time, other requests that arrive while the manager thread is busy had to wait until the manager thread completed the current request. Therefore, any memory requests that occurred close to each other were often serialized.

The basic pseudo-code for the `malloc` and `free` functions is shown in Figures 6.3 and 6.4. Pseudocode for the manager thread is show in Figure 6.5. Note that the manager thread performs the request processing immediately and delays the system processing until after it signals the main thread that the request is complete. The program thread must wait while the request processing is done and cannot proceed until the manager thread signals back. The manager thread was programmed to do the minimum amount necessary to service the request and then signal back. Any processing that could be delayed was placed after the signal so that it could be done in parallel with the main thread.

The pthread signaling functions required each condition variable to be linked to a particular mutex. This is required to ensure proper order of operation when handing off control between threads. With regards to the pseudocode, this means that the message mutex was released upon signaling and then re-acquired when the thread was subsequently woken up.

```
malloc
{
  lock request mutex
    lock data mutex
      place request
      signal manager thread
      wait for finished signal
      store return address
    release data mutex
  release request mutex
  return address
}
```

Figure 6.3: Pass-through (PT) implementation malloc pseudocode

Since a return address is not available until the system `malloc` call completes, `malloc` requests contain only request processing meaning memory allocations are not performed in parallel. The calls to `free` on the other hand can be executed completely in parallel since the calls do not expect a return value. When a call to `free` arrives, the main thread copies the address into the message structure and signals the manager thread that a request is ready. Once the manager thread stores the address and signals back, the program is free to continue while the manager thread handles the `free` request in the system processing section.

After initial testing, however, it was clear that any performance gains obtained by parallelizing the `frees` are negated by the overhead incurred through the pthread mutex locking and condition signaling functions. The time to signal another thread and have it signal back was measured to be roughly 5-10 times longer than the time it takes to call the regular system `malloc`. Due to the high amount of overhead involved with thread signaling and operating system scheduling, it was clear that the communication costs were too high for the pass-through method to be effective.

```
free
{
  lock request mutex
    lock data mutex
      place request
      signal manager thread
      wait for finished signal
    release data mutex
  release request mutex
}
```

Figure 6.4: Pass-through (PT) implementation free pseudocode

```
manager thread
{
  loop start
    wait for request signal
      read message
      determine request type
      //perform request processing
      if malloc request
        call system malloc
        store address in message
      else if free request
        store address to free
      end
    send finished signal
      //perform system processing
      if free request
        call system free on stored address
      end
  loop end
}
```

Figure 6.5: Pass-through (PT) implementation manager thread pseudocode

6.2 Locked Bins Approach

The second implementation (called *locked bins* or *LB*) took a new strategy aimed at minimizing the communication required between the threads. In many parallel systems communication is the major bottleneck, so programming approaches that distribute the workload to minimize required communication generally provide the most opportunity for parallelism. In order for the library to reduce the amount of inter-thread communication required, the strategy of servicing individual requests had to be thrown out. Instead blocks of memory had to be prepared in advance. These memory blocks were created by the manager thread and made available to the program thread (through shared memory structures) so they could be used for future allocation calls. By pre-allocating free blocks of predetermined sizes, the locked bins approach allowed `malloc` requests to be satisfied without directly signaling the manager thread. Figure 6.6 outlines how the pre-allocated memory blocks created a buffer to prevent the need for direct communication between threads.

Using this approach, pre-allocated memory blocks (referred to as “blocks”) are sized according to powers of two and stored in structures called “bins.” Each bin comprised of a linked list of blocks of a particular size along with supporting information, such as the number of blocks currently stored (bin count) and the

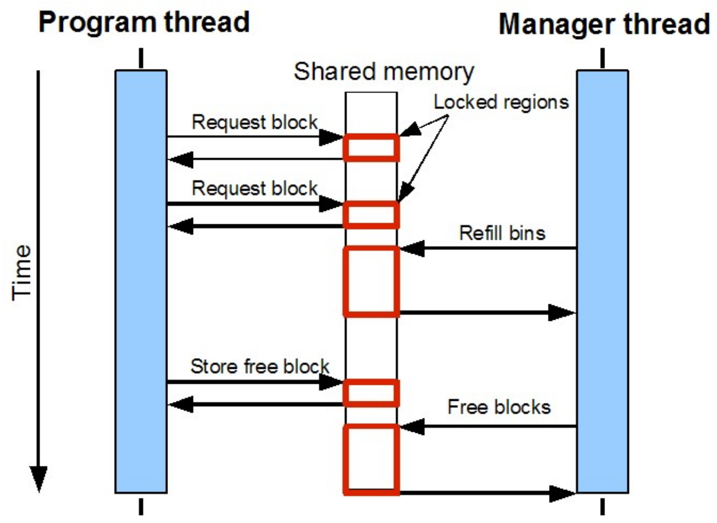

Figure 6.6: Locked bins approach

maximum size of the linked list (bin size). Because the pre-allocated blocks contain currently unallocated memory, pointers to the next block are stored in the first several bytes of the current block. Therefore each memory block has to be at least large enough to store a pointer (4 bytes on 32 bit systems and 8 bytes on 64 bit systems). The linked list will be referred to as the “bin queue” since the blocks are removed from the list in a FIFO fashion by the program thread.

On allocation requests, the requested size is rounded up to the next highest power of two in order to find the appropriate block size to obtain. Concurrent with the main program thread, the dynamic memory manager thread works in the background to “refill” any bins that are running low. Refilling consists of adding additional blocks to a bin queue until there are at least half the maximum number of blocks allowed. Using this method, an allocation request requires no communication between threads if a block of the requested size is available at the time of the request.

As mentioned before, each bin contains a “bin size” variable that determines how many blocks can be stored in the bin queue. Since the frequency of memory allocation sizes is program dependent, each bin size starts off at zero and increases dynamically throughout the program based on the demand for blocks of that particular size. Whenever an allocation request arrives for a bin that is empty (called a bin miss), the bin size is increased by one. This prevents more allocation misses in the future and allows block sizes that are used more frequently to have larger buffers. The bin size grows until enough blocks are queued up so that bin misses no longer occur. This means that upon initial startup, bin misses will occur until the bins reach their optimal size. Since different programs exhibit different allocation patterns, this method of slowly increasing the bin sizes allows the bin queues to dynamically adapt to the request patterns of the program. Block sizes that are requested more frequently will have more blocks buffered in the bin queue, whereas block sizes that are used infrequently will have small queue sizes. In an effort to minimize the number of bin misses and simplify the complexity of the algorithm, bin sizes are not allowed to decrease. Each bin also contains a variable used to keep track of how many blocks are currently stored in the bin. This variable is used to quickly obtain the number of available blocks without having to traverse the entire linked list.

On bin refreshes, the manager thread cycles through all the bins and locates any that are under half full and refills them with new blocks. New blocks are allocated until the number of blocks stored equals half the bin size. The bins are only filled to half their capacity to allow room for blocks to be added from other sources (such as recently freed blocks, the implementation of which is described below). The manager thread refills bins in two cases. First, the manager thread is signaled by the program thread at regular intervals to refill the bins. The optimal refresh time was determined to be after every 1000 `malloc`/`free` calls. This periodic refilling ensures that new blocks are constantly buffered so that future allocation requests can be satisfied. Secondly, in the case that a bin is empty when an allocation request arrives, the main thread signals the manager thread for an immediate refresh and then waits until a new block is available. After initial startup, this case should happen only rarely since the periodic refreshes will typically keep enough blocks buffered so that bins should never run out.

The `free` function works in a similar distributed fashion. When a free request arrives, the address is stored in a “free array” to be freed later by the manager thread. The manager thread periodically parses through the array (after every 100 `free`s), grabbing the addresses and freeing their corresponding blocks. This array acts as a circular queue, meaning addresses beyond the end of the array wrap around and are placed in the beginning slot. In the rare event that the free array is full during a call to `free`, the main thread signals the manager thread to clear the array and waits until a spot becomes available.

Because the bins are accessed by both the main program thread and the manager thread, mutexes are used to protect the bin data. Mutexes are required whenever blocks are added or removed from a bin or when changing the size of a bin. However, each bin has its own mutex which allows the manager thread and main thread to access different bins at the same time. A separate mutex is used to protect access to the free address array. Using separate mutexes per bin eliminates a major obstacle of the PT implementation

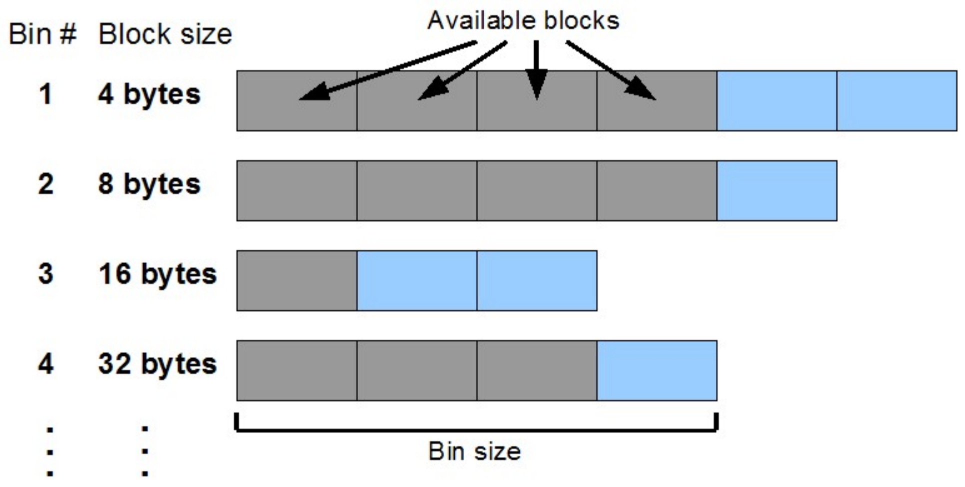

Figure 6.7: Bin system visualization

by allowing multiple `malloc` and `free` requests to be handled at the same time. As long as the manager thread does not access a bin or the free array at the same time as the main thread, neither thread should have to wait on mutexes. This combined with the removal of per-request synchronization allowed the locked bins approach to drastically reduce dependence between the manager and program threads.

Figure 6.7 below illustrates the operation of the memory bins and free queue. Revised pseudocode for the locked bins implementation is also provided. Note that for this approach, all of the request processing (the minimum required processing per request) is performed in the program thread. The minimum amount of processing for a `malloc` call is: (i) acquire the mutex lock, (ii) remove a block from the appropriate bin, and (iii) release the mutex lock. For `free`, the minimum processing is to: (i) acquire the mutex lock, (ii) store the address in the free array, and (iii) release the mutex lock. All of this is done in the main thread and neither method requires direct communication with the manager thread (in the common case). However, communication with the manager thread does occur on underflow of a bin queue or overflow of the free array.

Results show that the LB approach performs much better than the PT implementation, but, in many cases, it still lags behind the original single-threaded performance. Further investigation showed that the overhead involved with locking and unlocking the mutexes (even without lock contention) was too high to achieve speedup. Callgrind results showed that an average of 90 assembly instructions are executed in a lock and unlock pair, whereas the average number of instructions in the system `malloc` and `free` are 158 and 138, respectively. This means that the overhead associated with a single mutex lock and unlock costs about

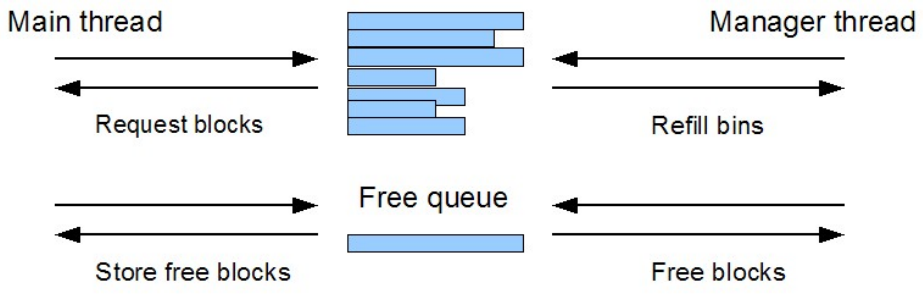

Figure 6.8: Bin and free queue system operation

```
malloc
{
  find bin for requested size
  loop start
    if block is available
      lock bin
        numblocks--
        grab block address from queue
      unlock bin
      return address
    else
      lock bin
        binsize++
      unlock bin
      signal refresh
    end
  loop end
}
```

Figure 6.9: Locked bins implementation malloc pseudocode

```
free
{
    loop start
      if array not full
        lock array
          place address in array
          arrayindex++
        unlock array
      else
        signal refresh
      end
    loop end
}
```

Figure 6.10: Locked bins implementation free pseudocode

50-60% of the instructions it takes to call the regular dynamic memory functions. Clearly a faster method of protecting inter-thread data was required if the threaded library was to outperform the standard dynamic memory library. The third and final approach aims to reduce this overhead by modifying the algorithm so that it does not require mutex locks.

```
manager thread
{
  loop start
    wait for refresh signal
    if malloc refresh signaled
      for each bin
        if numblocks < binsize/2
          malloc new blocks
          lock bin
            place blocks into bin
            update numblocks
          unlock bin
        end
      end
    else if free refresh signaled
      for each array index
        if index is address
          lock array
            remove address
          unlock array
          free address
        end
      end
    end
  loop end
}
```

Figure 6.11: Locked bins implementation manager thread pseudocode

6.3 Lock-Free Bins Approach

The final library version (called *lock-free bins* or *LFB*) uses the same bin queue data structures as the LB approach but replaces the mutex locks with faster atomic operations. Atomic operations are a set of multi-action operations that appear to the system to execute in a single step, meaning no other operation can interrupt them. All modern desktop processors contain support for atomic operations such as test-and-set, compare-and-swap, and atomic increments/decrements. These operations are carried out through hardware support for certain atomic assembly instructions, such as LOCK XCHG, LOCK CMPXCHG, and LOCK XADD. The library algorithm was manipulated for this approach to guarantee thread-safety through careful usage of these atomic operations, eliminating the need for expensive mutex locking. This approach allows the shared memory region to be accessed simultaneously by multiple threads, essentially eliminating lock contention. Figure 6.12 displays a visualization of the LFB approach. Notice how both threads can access the shared data structures simultaneously, even allowing the handling of requests while the manager thread performs maintenance on the same bin.

The first step was to reduce the LB algorithm to allow only a minimal number of variables to be shared between the main thread and the manager thread. Even though atomic operations are much faster than mutex locks, they still are more expensive than normal instructions and should be utilized as infrequently as possible. All variables that can be modified by multiple threads must only be altered using atomic operations.

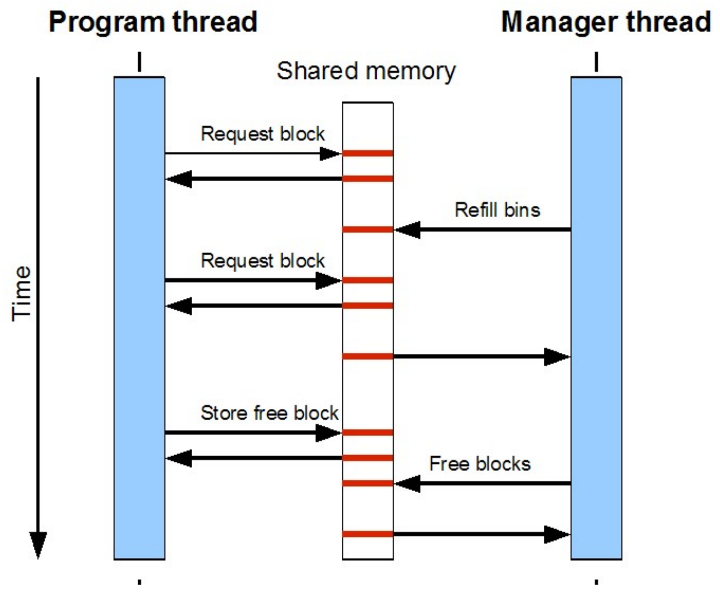

Figure 6.12: Lock-free algorithm approach

Optimizing the algorithm resulted in only three variables per bin that are shared among threads. Therefore, for each bin there are three bin variables that require atomic modifications, namely: (i) the bin size, (ii) the number of current blocks, and (iii) the bin queue head pointer. These three variables need to be atomically modified in the following three conditions. First, the number of free blocks per bin needs to be atomically decremented when removing blocks and atomically incremented when adding blocks. Both the manager thread and program thread modify this variable, so atomic operations are required to protect it. Secondly, bin sizes must be atomically incremented, since different threads of a multithreaded program might try to increment the bin size simultaneously. Finally, the pointer to the first block in a bin must be changed using a compare-and-swap operation whenever a thread removes a block from the bin. Again, this is required to allow multithreaded applications to request memory from multiple threads.

For freeing blocks, the algorithm was reduced so that only one variable was shared between threads, namely the free array index. The array index for the free array must be incremented atomically to ensure that no two program threads store addresses in the same array index. This race condition occurs when the original program is multithreaded, with multiple threads freeing memory at the same time.

As mentioned before, each bin is assigned a "block count" variable that represents the number of free blocks available in the bin. This block count acts as a gatekeeper to ensure blocks are only removed from the bin when free blocks are available. When a `malloc` request comes in, the block count for that particular bin is atomically decremented. If the return value of the decrement is greater than zero, then a block is guaranteed to be available for that allocation request. An atomic compare-and-swap operation is then used to remove a free block from the front of the bin queue. Note that the reservation of a block is separate from the retrieval of the block. It is completely possible that the block at the front of the bin queue at the time of the reservation is gone by the time the request tries to remove a block. However, as long as the reservation is successful at least one block is guaranteed to be waiting in the bin to satisfy the particular request. Therefore once a block is reserved the compare-and-swap can be executed without fear of grabbing a NULL pointer that occurs if the bin is empty.

When the manager thread refills the bins, the same block count variable is atomically decremented to reserve the last block in the bin. Once the last block is reserved, the newly allocated blocks are linked in at the end of the bin queue. The reservation of the last block is required to ensure that no allocation removes the last block while the manager thread modifies that block's memory to link in the new blocks. Then the bin count is atomically increased by the number of blocks added, making the newly added blocks available for future allocation requests. By waiting to update the block count until after the linking is complete, it is ensured that `malloc` requests cannot access the new blocks until the bin refill is complete. If a bin happens to be empty when a refill occurs the new blocks are added to the front of the bin, requiring a compare-and-swap operation to change the bin queue head pointer.

The free array used in the previous implementation was replaced by a linked list of freed blocks. This ensures that the program thread will never block on free requests due to the array being full, and it speeds up the freeing process because the manager thread no longer has to traverse the entire array to look for addresses to free. Again, an atomic compare-and-swap is used when adding new blocks to the free queue to allow multiple threads to call `free` at the same time. Since only the manager thread removes blocks, no atomic operations are required when removing blocks from the queue.

One additional optimization was made to prevent useless re-allocation of blocks. When refilling bins, blocks are taken from the free list if available. This shortcut prevents blocks from being freed to the system, only to be immediately allocated again when refilling the bins. In order to know which bins to put blocks back into, the library needs to keep track of the block sizes. Four additional bytes are reserved at the beginning of each block to store the block size. This block size is checked when refilling bins to determine if blocks in the free array can be placed directly back into the bin queues. While storing the block size takes up additional memory overhead for each allocation, it provides more flexibility for the manager thread when manipulating the blocks. In addition, to preserve temporal locality freed blocks are placed directly back into the front of their respective bins queues during the `free` function (if the bin is not already full). Since recently freed blocks are typically "hot" in cache, placing these blocks in the front of the bins allows the next allocation of that size to grab the same block. Again, the front of the bin is modified using an atomic compare-and-swap operation to preserve data integrity.

Lastly, to reduce wait time in the case the program encounters an empty bin, the manager thread prioritizes refilling bins that allocation requests are waiting on. If the manager thread is triggered by a failed `malloc` request it will refill that particular bin first before checking the other bins. This optimization makes the manager thread more responsive when dealing with the empty bin case, therefore reducing stall times and allowing more of the main program thread to be executed in parallel with the manager thread.

Pseudocode for the LFB approach is shown below. Atomic operations are specified by the "(atomic)"

```
malloc
{
  find bin for requested size
  loop start
    if block is available
      blocksleft = numblocks-- (atomic)
      if blocksleft > 0
        remove block from bin (atomic)
        return block address
      else
        numblocks++ (atomic)
      end
    else
      binsize++ (atomic)
      signal refresh
    end
  loop end
}
```

Figure 6.13: Lock-free implementation malloc pseudocode

keyword at the end of the line.

The LFB approach reduced the communication and locking overhead enough for the library to provide performance gains on memory-intensive applications. Results from the library tests are detailed in the next section.

```
free
{
  find block size
  if bin not full
    add block to front of bin (atomic)
    numblocks++ (atomic)
  else
    add block to free list (atomic)
  end
}
```

Figure 6.14: Lock-free implementation free pseudocode

```
manager thread
{
  loop start
    wait for refresh signal
    if malloc refresh signaled
      for each bin
        if numblocks < binsize/2
          for each free queue element
            if block belongs to same bin
              remove block from free queue
              add block to refill buffer
            end
          end
          malloc new blocks to refill buffer
          blocksleft = numblocks-- (atomic)
            if blocksleft > 0
              place blocks in refill buffer into bin
              update numblocks (atomic)
            else
              place blocks in refill buffer into front of bin (atomic)
              update numblocks (atomic)
            end
         end
      end
    else if free refresh signaled
      for each free queue element
        remove block from queue
        free block address
      end
    end
  loop end
}
```

Figure 6.15: Lock-free implementation manager thread pseudocode

Chapter 7

Test Results

The performance of each implementation of the threaded dynamic memory library was tested using the industry standard SPEC CPU2006 benchmarks. This section discusses these results and takes a closer look into the performance of the LFB implementation using several compiler benchmarks. The LFB library was also profiled with Valgrind to determine how cache effects and other factors affect the overall library performance.

7.1 Overall SPEC Performance Results

The SPEC 2006 benchmarks and the associated run scripts were used to evaluate the various implementations of the threaded dynamic memory management library. The test system hardware consisted of a 2.66GHz Intel Core i7-920 processor with 3GB of RAM. The system `malloc` library used was the Linux glibc `ptmalloc` library. The standard SPEC 2006 benchmark scripts were used to perform the test runs and the reported results are the geometric mean of ten base benchmark runs of the library. The average variance of each test was below .003. The results from the tests are shown in Table 7.1. Listed are the percent of dynamic memory computation for each benchmark followed by the results from the system library and each version of the threaded library.

The first conclusion that can be drawn from these results is that benchmarks that spend little time in dynamic memory functions are generally not affected (positively or negatively) by replacing the dynamic memory library. This result is expected since the lack of dynamic memory execution time essentially prevents any library changes from having any significant impact on performance. In some cases the less in-

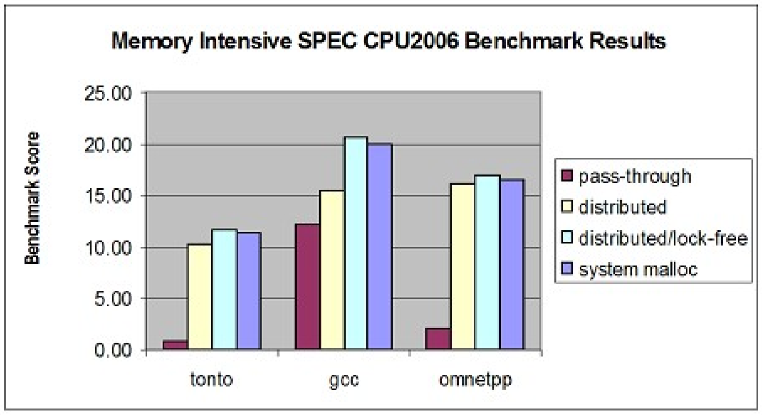

Figure 7.1: Memory-intensive SPEC benchmark results

tensive benchmarks suffered slightly when using the replacement libraries, most likely due to threading overhead and negative cache effects. It is important to note that the threaded memory library can be used on a per-application basis. Therefore since no speedup is expected from programs that do not use dynamic memory extensively, the threaded library does not have to be used with these types of applications.

The dynamic memory intensive benchmarks on the other hand showed improvement over the standard allocation library. The results from the three memory-intensive SPEC benchmarks are shown graphically in Figure 7.1. These benchmarks highlight the progression of the threaded library over the three implementations. The PT library showed significant performance degradation due to the significant communication overhead involved with the frequent calling of dynamic memory functions. The `malloc` and `free` functions were called millions, even billions of times in these three benchmarks. Therefore the thread communication involved with each function call added up to produce significant slowdown with the PT library. The LB library improved on this approach by reducing thread communication to a minimum. The scores from the LB approach were significantly better than the PT library, coming close to the scores of the standard library but not surpassing them. The main bottleneck to the LB approach was the overhead involved with locking and unlocking the shared data structures. The faster synchronization primitives of the LFB library cut back on the synchronization overhead and outperformed the standard library, producing positive speedup. The results show that for each of the three memory-intensive benchmarks, the LFB library implementation provided a 2-3% improvement over the standard Linux allocator.

fp_tests	**% cost**	**sys malloc**	**PT**	**LB**	**LFB**
bwaves	0.00	14.90	14.80	14.20	14.20
gamess	0.00	14.90	14.80	14.80	14.90
milc	0.01	16.40	16.60	n/a	16.70
zeusmp	n/a	12.30	12.20	n/a	12.30
gromacs	0.00	6.56	6.49	6.55	6.56
cactusADM	0.00	8.71	8.63	n/a	8.24
leslie3d	0.00	9.13	9.02	8.61	8.67
namd	0.00	12.10	n/a	n/a	12.10
dealll	n/a	18.90	n/a	n/a	18.20
soplex	0.01	22.20	21.90	21.80	21.90
povray	0.05	15.30	14.40	n/a	15.30
calculix	0.03	4.47	4.28	4.39	4.47
GemsFDTD	0.01	10.50	10.50	9.96	9.95
tonto	13.48	11.10	0.76	10.20	11.50
ibm	0.00	33.80	33.60	32.00	32.00
wrf	0.26	n/a	n/a	n/a	n/a
sphinx3	0.29	27.10	24.70	26.70	27.00

int_tests	**% cost**	**sys malloc**	**PT**	**LB**	**LFB**
perlbench	1.96	19.50	3.13	18.60	19.80
bzip2	0.00	12.40	12.40	12.40	12.40
gcc	10.15	20.20	12.10	15.50	20.60
mcf	0.00	29.20	29.20	29.20	29.30
gobmk	0.01	17.20	16.80	17.20	17.20
hmmer	0.07	8.19	7.87	8.15	8.19
sjeng	0.00	17.10	17.00	17.00	17.10
libquantum	0.00	28.80	28.60	n/a	28.80
h264ref	0.00	21.50	n/a	n/a	21.50
omnetpp	10.10	16.60	2.07	16.10	16.90
astar	0.33	11.80	10.90	11.80	12.00
xlanchbmk	3.50	21.80	n/a	n/a	23.90

PT: pass-through implementation.
LB: Locked bins implementation.
LFB: Lock-free bins implementation.
$^{n/a}$Data for these categories were not gathered due to build errors or errors in the benchmark runs.

Table 7.1: SPEC benchmark result summary

The benchmark scores are automatically calculated by run scripts that are distributed with the SPEC CPU2006 benchmarks. Briefly, these scripts time benchmark runs and compare the runtimes to the those of a fixed reference machine.

	percent of total computation			
	system malloc	**new malloc**	**system free**	**new free**
omnetpp	5.86	2.53	4.24	1.82
tonto	8.76	3.32	6.04	1.66
gcc	0.22	0.16	0.26	0.11

Table 7.2: Comparison of callgrind profiling results of LFB and the standard Linux library

7.1.1 Valgrind Analysis of the LFB Library

To better understand the runtime effectiveness the LFB library, callgrind results were obtained for the three memory-intensive SPEC benchmark programs. Table 7.2 shows the percent of instructions spent inside the dynamic memory functions with the standard library followed by the percent inside the functions with the LFB library. The results show that the average number of assembly instructions executed in `malloc` and `free` were reduced, respectively, by 49% and 62%. Thus, the maximum speedup expected from these benchmarks is between 5-8%, but various conflicts prevented the library from obtaining these results, delivering instead speedups of only 2-3%.

There are several factors that limit the threaded library from reaching its full potential. One of the most important factors that contributes to the reduced speedup is the latency associated with the execution of atomic operations. Two atomic operations are executed per `malloc`/`free` call in the LFB implementation. The processor must take additional steps to insure no other thread can modify values being accessed by an atomic instruction. Each type of processor handles these steps differently, but atomic operations are always more costly than standard operations. Latencies associated with atomic operations are not accounted for in the Valgrind profiling. The profiling results only show number of instructions executed and do not factor in the number of cycles it takes to execute each instruction. Therefore the additional overhead incurred by atomic operations are not factored into the profiling results.

To obtain a more complete picture of the latencies of atomic operations, several tests were run to measure the latencies of atomic operations on the Core i7. First, the “mubench” benchmark program [35] was run on the i7 to measure the latencies associated with the XADD and CMPXCHG instructions. The results showed that the CMPXCHG instruction executed in 5.02 clock cycles and XADD executed in 2.00 clock cycles. Since each of these instructions executes multiple tasks (compare and exchange, exchange and add), the cycle latencies are higher than the basic set of assembly instructions. These numbers do not show the entire picture, however, since the mubench program does not measure the atomic or “locked” versions of these instructions. For those measurements, a custom benchmark program was developed.

The custom benchmark measured the average time it took to execute an atomic addition over 5 runs of 100 million trials each. The results from this test on the i7 show that an atomic addition is approximately 19 times more costly than a regular addition operation. After factoring in this additional atomic latency into the profiling results, the average percent reduction for `malloc` decreases from 49% to 26% and the reduction for `free` drops from 62% down to 36%. These revised profiling numbers more accurately reflect the 2-3% speedup that the threaded library obtained on the memory intensive benchmarks.

To understand why the atomic operations are so intensive compared to normal assembly instructions, it is important to take a look at how the hardware implements atomic support. In assembly, a handful of instructions are designated to support atomic execution. These instructions are prefixed by the "LOCK" keyword when atomic operation is desired. Older architectures ensured the instruction executed without interruption by asserting the LOCK# signal on the bus. Newer Intel architectures have eliminated the need for bus locking in most cases by allowing the cache coherency mechanism to ensure atomicity for data present in the local cache [36]. While this shortcut and other optimizations drastically improves atomic performance on the i7 architecture, atomic instructions still involve memory barriers that serialize the entire pipeline, resulting in stalling of surrounding instructions [37]. This stalling delays future instructions from executing, resulting in longer instruction latencies for atomic operations.

Another factor affecting the results is the effect the library has on the cache behavior of the benchmarks. Because the LFB library assigns dynamic memory blocks differently than the standard memory allocator, the number of cache hits and misses can change drastically. Cache effects can play a large role in determining program performance. For example, a value in L1 cache on the i7 takes about 4 clock cycles to access. If the processor has to go to L3 cache to obtain the value, it can cost around 40 clock cycles. A complete cache miss will force a read from memory which can cost hundreds of cycles. Clearly the use of a different block allocation scheme can introduce quite a bit of variance in overall performance based solely on cache effects.

To attempt to measure how the library changed the cache effects, the built-in cache simulator in Valgrind was used to measure the number of cache misses with both the LFB and system library. The number of cache misses and associated cycle penalties were used to calculate the approximate clock cycles per instruction (CPI). The CPI number can be used to measure how efficient a processor is when executing a particular program. A sample of these results are shown in Table 7.3. The CPI numbers show that

	percent of total computation		
	system library CPI	**new library CPI**	**% difference**
tonto	3.617	3.673	-1.5%
perlbench	3.469	3.480	-0.3%
gcc	3.824	3.825	0.0%
omnetpp	3.849	3.926	-1.0%
astar	3.943	3.946	-0.1%
xlancbmk	3.374	3.357	0.5%

Table 7.3: CPI comparison using callgrind instruction and cache profiling data

the memory intensive benchmarks had slightly worse cache effects with the threaded library than with the standard library. These results further explain why the expected speedup is not fully realized in the memory-intensive benchmarks. In addition to the memory-intensive benchmarks, a less memory-intensive benchmark (specifically `xalancbmk`) was also measured to demonstrate that the library can improve cache performance in particular applications. This result explains why unexpected speedup is seen in a handful of the less memory-intensive benchmarks.

Cache effects are very difficult to measure precisely. The Valgrind cache simulation has several limitations that negatively impact the accuracy of the results. As mentioned before, Valgrind simulates a single-processor system, so only a single cache hierarchy is simulated. Thread switches that normally would not occur on multiple processor systems can introduce additional cache misses due to both threads having to share the same L1 and L2 caches. Additionally, Valgrind can only simulate a 2-level cache so the simulations ignored the effect of the L2 cache and simulated only the L1 and L3 level caches. Valgrind does not simulate any time spent in the kernel, so the execution time spent in system calls and the cache effects of the associated context switches are also not measured. Despite these limitations, the results of the cache tests shed some light on the detrimental (and occasionally helpful) performance implications that the block allocation scheme has on cache hits and misses.

Besides different block allocation, the library also uses more dynamic memory than the system allocation library. The scheme of rounding the size of each request up to the next highest power of two introduces unused space at the end of each allocation block. In the best case an allocation will be exactly a power of two causing no extra memory space to be allocated, but in the worst case the power of two scheme will essentially double the amount of memory used for an allocation request. The library also adds an additional four byte header to the beginning of each allocation, which can add significant overhead for applications that allocate frequently in small blocks. Lastly the buffered blocks that are stored in the bins, even though they are not currently in use by the program, are still counted towards the amount of memory allocated to the application. All of these factors contribute to the increased memory usage of programs while using the threaded library. Results from the SPEC tests show that the threaded library increases the average memory usage of the benchmarks by about 20%. For frequently allocating and deallocating programs that number increases. The three memory intensive benchmarks had an average increase of around 220%. This tripling of memory usage for the intensive benchmarks also contributed to the poorer cache performance they exhibited.

In addition to cache effects and memory size, there are other factors limiting the amount of speedup that can be obtained. Operating system scheduling and thread switching can both introduce additional overhead that cannot be accounted for in the Valgrind simulations. Another factor that affects the results is cache coherency. When a memory location is modified by one processor, any other copies of that cache block in other areas of the cache are invalidated. When the other processors try to access that cache block it essentially acts as a cache miss, meaning the processor will typically have to reach down to the shared L3 cache to retrieve the updated cache block values. Cache coherency is not accounted for in Valgrind since only a single core cache architecture is used, providing another explanation for the discrepancy between the profiling and measured results.

In addition to explaining the disparity in expected to actual speedup on the memory-intensive benchmarks, many of these factors explain why some of the less-intensive benchmarks experience slowdown. Just because an application does not spend a lot of time allocating and deallocating memory, it could still be using a large amount of memory by allocating in large chunks. Therefore the threaded library can still affect the cache miss rates of the benchmarks that do not spend a significant amount of time in the dynamic memory functions. Operating system scheduling, thread switching, and cache coherency can also contribute to the slowdown. On applications that do not spend significant time in allocating dynamic memory, not much performance gain is expected with the threaded library. Obviously, one would not typically want to link in the library for these types of applications. If in doubt an application can always be tested to determine if it experiences any speedup or slowdown.

To summarize the results with the SPEC benchmarks on the i7, the threaded library provided a 2-3% speedup on the memory-intensive benchmarks, and provided similar or slightly worse performance on the benchmarks with moderate use of dynamic memory. Atomic operation latencies prevented the memory-

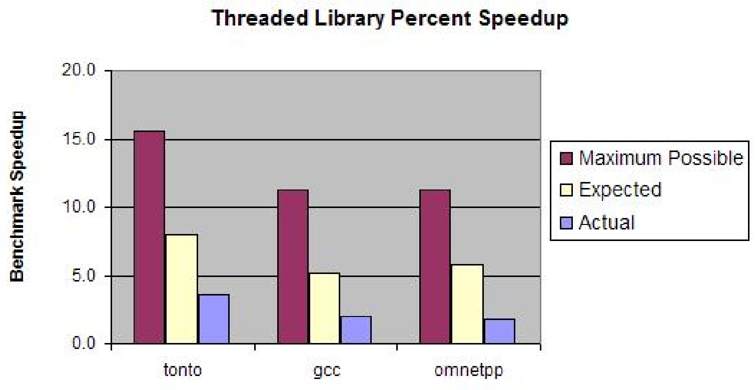

Figure 7.2: Speedup analysis summary for memory-intensive benchmarks

intensive benchmarks from reaching the projected speedup of 5-8%. Some benchmarks outperformed their expected speedup because of increased cache performance, but increased memory usage along with other factors caused the threaded library to have slightly worse cache performance in most other benchmarks. In addition to atomic operation latencies and cache effects, other factors such as operating system scheduling, thread switching overhead, and cache coherency also had negative side effects. Although the library reduced the number of dynamic memory instructions by about half, these other factors limited the amount of speedup that could actually be obtained.

Figure 7.2 provides a performance analysis of the threaded library with respect to what speedup was achievable in the memory intensive SPEC benchmarks. The first column depicts the maximum speedup that could possibly be obtained if the dynamic memory costs were reduced to zero. Of course this is impossible, so this metric serves as the upper limit to the speedup that can be obtained through any dynamic memory optimization. The second column depicts the expected speedup based on the profiling results of the LFB library with the three intensive benchmarks. Since profiling results showed around a 50% reduction in instructions executed, the LFB approach was expected to provide half the maximum possible speedup. The negative side-effects mentioned in the preceding paragraphs prevented the actual benchmark runs from achieving this speedup, as the third column depicts.

gcc			
Trial #	**Regular time(ns)**	**Threaded time(ns)**	**Speedup(%)**
1	620664871896	606460323756	102.3
2	628267056925	603266265277	104.1
3	635297406466	605694212411	104.9
4	630458667011	607401427149	103.8
5	635380815759	606422945462	104.8
	Avg difference:	24164728800	Avg speedup: 104.0
llvm			
Trial #	**Regular time(ns)**	**Threaded time(ns)**	**Speedup(%)**
1	723584918236	691882133455	104.6
2	724780429654	696974112396	104.0
3	714829357879	693343381500	103.1
4	729322211743	702887493433	103.8
5	723332596904	695465632151	104.0
	Avg difference:	27059352296	Avg speedup: 103.9

Table 7.4: Comparison of compile times using the threaded and system libraries

7.2 Compiler Tests

In addition to running the benchmarks, the LFB library was also evaluated against the compilers used to build the SPEC benchmarks. Figure 7.4 shows the timing results from compiling the benchmark suite using both the `gcc` and `llvm` compilers. The average speedup obtained by using the threaded library was between 3-4%. This resulted in a runtime savings of 24 and 27 seconds for the `gcc` and `llvm` compile times, respectively. This speedup was obtained by simply linking in the library (by setting the LD_PRELOAD environment variable) before running the tests. A simple change to realize a 3-4% speedup.

The LFB library was also tested with the Savant VHDL front-end analyzer/compiler [38]. Out of 20 test runs of compiling an example model, the threaded library provided positive speedup in 19 of the 20 test runs and provided an overall average speedup of 2-3%. One aspect of the library that the tests uncovered was the negative effect that the increased memory usage had on performance. When larger models were used the library provided no significant performance gain because of the increased page file usage when the RAM maxed out. Slightly smaller models had to be used in the comparison tests to prevent paging in order to keep the tests running in RAM. Paging and hard drive accesses in general introduce a lot of variance to performance tests due to variable hard drive access times.

Chapter 8

Hardware Architectures

To gather information on how system architecture affects the results, the SPEC benchmark tests were repeated on two additional systems using the LFB library. One system contained a Core 2 Duo T8100 processor operating at 2.1GHZ with 4GB of RAM, and the other system contained an AMD64x2 processor operating at 2.5GHz with 2GB of RAM. The Core 2 Duo has an 800MHz front side bus and contains two separate cores, each with separate 32kb L1 instruction and data caches, and a unified 3MB L2 cache shared between the two cores. The AMD64 x2 processor has a 1GHz front side bus and contains two cores with separate 64KB instruction and data L1 caches, along with separate 512KB L2 caches. The Core i7 has four separate cores each supporting two simultaneous threads, each with 32kb L1 data and instruction caches and a 256kB L2 cache, with a unified 8MB L3 cache shared between all four cores.

Before running the tests, certain processor settings were modified in the bios to turn off particular processor features that could affect the correctness of the results. Many modern processors have down-clocking abilities that allow a processor to reduce its clock speed in order to save power. Some even allow certain cores to be shut off completely when not in use. The i7 and AMD system even had features that could overclock certain cores if other cores were dormant. These settings were turned off to eliminate any latencies and clock speed variances that could be introduced through the use of these features. The overclocking feature of the AMD motherboard could not be completely disabled. As a result, the test results from this system tended to have higher variances than the other systems.

Figure 8.1 shows the ratio of the library benchmark scores to the standard library benchmark scores for each of the three architectures. A ratio greater than one means that the library obtained a higher benchmark

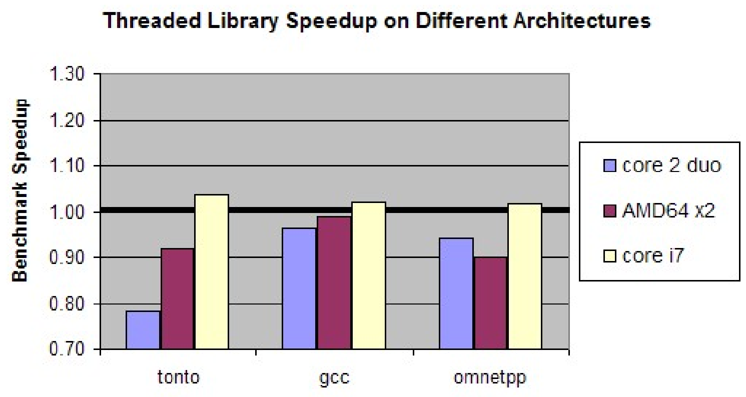

Figure 8.1: Speedup analysis for different architectures

score and provided positive speedup. A ratio under one signifies the library had a negative effect and caused slowdown. The core 2 duo system for the most part showed slowdown across the board, with no speedup gained from the memory-intensive benchmarks. The AMD system had more variation in its results, but again no speedup was obtained on the three allocation intensive benchmarks. The i7 showed consistent performance improvement.

Overall the i7 processor performs the best out of all the tested platforms. Intel claims that the latency of the atomic compare-and-swap operation (LOCK CMPXCHG) for the i7 is 60% less than the latency for the Core 2 [37]. The increased performance of the atomic operations on the i7, along with the shared L3 cache hierarchy between the cores, is one reason why the newer Intel processor gives the best performance with the threaded library. With the number of dynamic memory operations ranging in the millions or billions for the memory-intensive benchmarks, even the slightest improvements with locking and inter-core communication can provide drastic improvements for the threaded memory library. Another difference is that the i7 contains the largest on-chip cache at 8MB. Since the library rounds up block sizes and queues up additional blocks for allocations, it uses more memory than the standard allocation library. Larger cache sizes help minimize the negative effect that larger memory use causes for the benchmarks.

Comparing the two generations of Intel processors, it is clear that the strides taken in the latest Core i7 processor architecture allowed the threaded library to outperform the standard library in the memory-intensive benchmarks. The earlier generation of multi-core processors (Core 2 Duo and AMD64 x2) were released in 2006 when multi-core computing was just hitting the mainstream desktop market. The more streamlined architecture and added features such as hyper-threading and shared L3 cache in the i7 shows how the improved hardware support for multithreading benefits the fine-grained type of threading found in the threaded memory library. Future generations of processors should further improve performance for threaded applications, allowing new programming approaches such as library threading to improve the performance of future applications.

In addition to the AMD and Intel desktop architectures, the library was also tested on the UltraSPARC platform. Sun's UltraSPARC T1 processor is an eight-core processor capable of supporting up to 32 simultaneous threads [39]. The OpenSPARC project is an open-source gate-level verilog design for a SPARC processor [40]. The OpenSPARC RTL code can be loaded onto an FPGA platform to simulate a working UltraSPARC T1 processor. Through the OpenSPARC program and the Xilinx University Program [41] an evaluation FPGA platform was obtained to test the library with.

The evaluation board was a Xlinix Vertex 5 XUPV5-LX110T FPGA platform. The FPGA bitcode for the OpenSPARC processor along with a disk image that included the OpenSolaris operating system was included on a CF card that came with the platform. The OpenSPARC processor loaded onto the FPGA board had four multithreaded cores that ran at a speed of 50MHz. The platform had a 256MB RAM stick which served as the main memory for the system. The board was able to load the processor model, boot into the included OpenSolaris install, and basic commands could be issued. Unfortunately after further investigation too many limitations were found with the platform that ultimately led to abandoning the investigation.

First there was no non-volatile storage on the system. The 158MB disk image loaded from the CF card was mounted to and booted from the RAM module. Therefore any changes in the file system were lost on power down. The CF card was not directly accessible from the operating system, so the only method to load files onto the system was through the Ethernet interface via ftp. However, only 7MB of free space existed on the disk image which was much too small for any of the SPEC benchmarks. The OpenSolaris version installed was stripped down and did not include a compiler or any tool chains. Even if the SPEC benchmarks were transferred onto the system and compiled, there would not be enough RAM on the system to be able to run them. In the end, there were too many differences and limitations with the OpenSPARC FPGA platform that prevented it from accurately representing the performance of a normal UltraSPARC platform.

Chapter 9

Compilers and Operating Systems

In addition to exploring how different hardware affects the library, data was also gathered on how different software components affected the performance of the library. In particular, various tests were set up to observe how compiling the benchmarks on different compilers affected the speedup and how running the benchmarks on different operating systems affected speedup.

Creating application binaries with different compilers can affect the amount of processing that occurs between memory allocations, and can also rearrange the order of allocations as well. Figure 9.1 compares the speedups obtained by running the benchmarks on the Linux i7 system after they were compiled with both the `gcc` and `llvm` compilers. The average speedup difference between both compilers was around 0.004 and the speedup obtained on the memory-intensive benchmarks did not change at all. According to the results of these tests, compiling applications with different compilers does not significantly affect the performance of the library on most applications.

The other software component that plays a factor in the library performance is the operating system on which the application is run. The library was tested with the SPEC benchmarks in both the Linux and Open-Solaris operating systems. The operating system controls the manner in which threads are scheduled and how quickly one thread can communicate to another through synchronization primitives. Solaris mutexes utilize adaptive spin locks whereas Linux utilizes a fast user level mutex called a "futex" [42]. Although the LFB implementation avoids mutexes when protecting data accesses, mutexes are still used when signaling threads.

Another area in which the operating system can affect speedup is from the default memory allocator

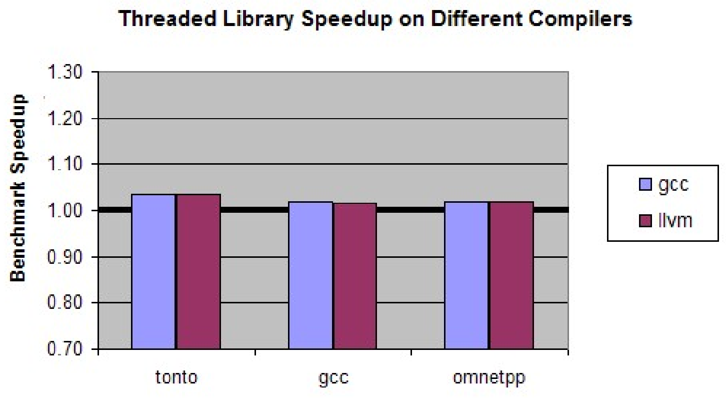

Figure 9.1: Speedup analysis for different compilers

included in the system. Since different allocators use different allocation algorithms, each one will perform different computations when managing the blocks and will have different cache effects based on the effectiveness of the block allocation algorithm. Tests performed by Joseph Attardi and Neelakanth Nadgi show that the standard Solaris allocator outperforms the standard Linux `ptmalloc` implementation on single threaded applications, but does not scale as well with multithreaded applications [43]. The default Solaris allocator also uses more of the heap that the `ptmalloc` library [43]. These tests show that the Solaris `malloc` implementation is not as efficient as the Linux implementation, so the improved allocation algorithm of the threaded library may outperform the standard allocator by a wider margin on Solaris systems.

The tests on the Linux i7 system were compared to benchmark runs on the same system running OpenSolaris. The results are shown in Figure 9.2. The OpenSolaris system compiled the benchmarks using the both the `Sun Studio` and `gcc` compilers, with nearly identical results. This reinforces the results

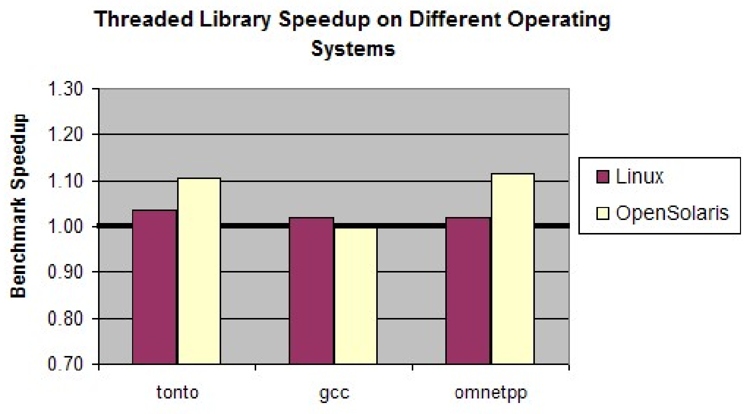

Figure 9.2: Speedup analysis for different operating systems

from the `gcc`/`llvm` comparison that compiler differences play a minimal role in affecting the performance. The results showed minor variations in most benchmarks, but the most surprising results came from the memory-intensive benchmarks. In two of three benchmarks (tonto and omnetpp) the speedup on the Open-Solaris system increased dramatically, providing between 10-11% speedup with the threaded library. It is difficult to pinpoint the exact cause of the difference, but cache efficiency and thread scheduling are key candidates as to why the additional speedup was obtained. As mentioned before the system allocator library is different so the speedup ratio is in comparison to a different reference point. The threaded library may just perform better in comparison to the to Solaris allocator than it does to the Linux allocator. The third memory intensive benchmark, `gcc`, had performance similar to the system library. The `gcc` compiler tends to only free memory at the end of compilation. This means any benefits obtained by reusing blocks would not be seen in this test. To summarize, using the threaded library in Solaris can provide even more performance gains than seen in Linux, but these improvements are not seen across the board. The speedup gained varies on a per-application basis.

Chapter 10

Additional Optimizations

There are a number of optimizations that could be implemented to further increase the performance of the threaded dynamic memory library. A key aspect of efficient dynamic memory allocation is maintaining the locality of the memory blocks. The cache structure in modern processors favor memory accesses to addresses that have recently been used. To increase the temporal locality, the final LFB library implementation replaced recently freed blocks on the front of the bins, allowing recently used blocks to be accessed in a last in/first out order. The SPEC benchmarks and the callgrind tests showed that reusing blocks in this order resulted in significant performance gains. The three memory intensive benchmarks experienced an average speedup of 16% as a result of this optimization. Another type of locality that is not currently addressed is spatial locality. Returning block addresses that are adjacent to each other in physical memory also improves the performance of the caching hardware. Future improvements could allow restructuring the order of the blocks in the bins to group similar addresses next to each other. By keeping the addresses of in-use memory blocks close together in memory the dynamic memory allocator can reduce memory thrashing and improve the overall performance of the cache architecture.

Another outstanding issue with the LFB library is that it does not take memory usage into consideration. The power of two scheme used to determine the bin sizes lends itself to a large amount of internal fragmentation, due to the excess space at the end of each block that does not get used. One method to reduce this fragmentation is to use a finer grained resolution of bin sizes. A common method is to allow three times power of two sizes to be allocated along with the normal power of twos [44]. This method essentially inserts an extra bin size halfway between each power of two. Bin sizes could also be chosen dynamically, depending on the dynamic memory pattern of the current program. Another memory optimization is to dynamically decrease the bin sizes. The LFB library increases the size of each bin based on the demand for blocks of that size. However, programs change their memory access patterns throughout the course of execution, so a bin having many allocations at the beginning of the program may not need as many blocks buffered later on. A simple mechanism could detect when blocks of a particular size are not being used and decrease the bin size accordingly, resulting in less memory wasted due to over buffering. Another issue that contributes to additional memory usage is the added overhead that the system needs due to the fact that it acts as a wrapper to the regular memory allocation calls. Both the system library and the threaded library reserve bytes before each allocation to store block information. Information such as block size is duplicated in the bookkeeping overheads associated with both systems. By removing the calls to the system `malloc` and managing the heap directly, this redundant information would be removed resulting in more efficient usage of memory and increased spatial locality between blocks.

Communication costs between threads should always be kept to a minimum. The LFB implementation aims at decreasing these costs, but there are still areas where improper buffering could result in the main program stalling. Ideally every time a request to `malloc` is made, there should be a block of the requested size waiting in the proper bin. The cost of waiting for bin refreshes can be very costly, so improving the algorithm to more accurately predict future block size requests could go a long way in improving performance. More complex algorithms could analyze the memory allocation behavior of a program over longer periods of time to identify usage patterns, and could respond accordingly by adjusting the bin sizes in preparation for future allocation loads.

Finally, the library implementation is not currently optimized for multithreaded programs. Although the library has been successfully tested with multithreaded applications, most of the tests used for benchmarking were single threaded. There still may be issues with multithreaded programs that have not been discovered yet. Additionally, even though the LFB implementation is lock-free, atomic operations still manage to slow down the system when a large number of threads are requesting memory. Since much work has been done in multithreaded optimization, some of the techniques previously developed could be applied to the library. Per-thread heaps could be created and multiple memory manager threads could be used to manage each private heap. A hybrid approach could also be used where each thread gets a small local buffer to pull blocks from. All threads would share a single large global buffer. When a request arrives, if a block of the required size is not available in the local buffer, a free block could be returned from the global shared bin.

The MMT threaded library [22] discussed in the section 3 offers several optimizations that could also be adapted to the LFB library. The MMT library reduces lock contention by eliminating all mutex locks and atomic operations from the library code. Flagging and time slicing algorithms were used extensively to communicate between threads while avoiding contention. Although a completely atomic operation-free library is most likely an impossibility (due to contention among multiple program threads in multithreaded applications), the current LFB library code could possibly adapt some of these techniques to reduce the number of atomic operations required. Another optimization used by the MMT library to reduce communication costs involves serving memory requests in bulk. The MMT approach transfers "buckets" of memory blocks to the program thread to allow allocations to be taken from the bucket without coordination from the MMT thread. This reduces thread communication from once per allocation to once every n allocations, where n is the number of blocks stored in a bucket. The MMT library only pre-allocates blocks for allocation sizes under 512 bytes, with finer grained block sizes (multiples of 8). Bulk allocation and finer grained bin sizes could be implemented into the LFB library, but the scalability of these approaches for multithreaded programs requires further investigation.

Chapter 11

Further Applications

The concept of multithreading library functions is not limited to dynamic memory management. If a library function satisfies certain criteria, the same method used in the threaded dynamic memory library could be applied to create multithreaded versions of other library code. Although the dynamic memory library functions were ideal candidates, it is definitely feasible that other library functions could also benefit from this approach. In order for a function to be considered a candidate for multithreaded pre-calculation, it has to exhibit certain properties.

First, the arguments passed into the function need to follow a fairly predictable pattern. This is necessary in order for the manager thread to be able to accurately predict future function calls. Pattern matching algorithms can be used to determine if function arguments in a program follow a decipherable pattern. Secondly, the output of the function should only rely only on the arguments that are passed into it. If the result of a function call depends on external state information, it would be impossible to calculate the value ahead of time. If the state variables change after the output was pre-calculated, that prediction result then becomes invalid. Finally the act of executing the function should be side-effect free. Pre-calculating a function call in the manager thread should not in any way affect the operation of the main program.

If a function satisfies these requirements, it would then be possible to create a threaded implementation based on it. However, there are several factors to consider before determining if multithreaded pre-calculation would in fact be beneficial. First, the function in question must take up a meaningful amount of processing time in order to make multithreading worthwhile. Again, Amdahl's law shows that the maximum speedup that can be obtained depends directly on the amount execution time spent inside the function.

```
compare previous values with contexts
find best matching context
if next value exists in context
   predict next value in context
else
   find most frequent value(s) in context
   if only one most frequent value exists
       predict most frequent value
   else
       predict random most frequent value
   end if
end if
```

Figure 11.1: Basic pseudocode for the context predictor

Secondly, the communication overhead must be minimized so that the communication costs do not negate the gains provided by pre-calculation. Misprediction penalties have to be factored into the equation as well. When the input arguments are mispredicted, the function output will have to be processed at the time of the function call, essentially making the function sequential again. If values are mispredicted frequently, the function could take longer to complete because of the added overhead of checking the predicted outputs. Finally, the time between calls to the function have to be spaced out in order to give the manager thread time to perform argument prediction and function pre-calculation. If too many calls are bunched together, the manager thread will not have enough time to get the predicted output ready by the time the next function call comes in, causing the main program thread to perform the function calculation itself.

To determine whether function argument prediction is feasible, it needs to be shown that some functions have arguments that follow a predictable pattern. To test this theory, a library was created to log the arguments passed to the sine and cosine functions. This library was linked in during Linux startup, resulting in a collection of per-process logs from various standard Linux processes. These log files were fed into a modified context predictor algorithm coded in MATLAB. The context predictor learns from previous values to create a series of contexts, which are finite sequences of values, and uses them to predict future values when the same context repeats. These contexts are updated as more values pass through the predictor, providing more accurate pattern recognition for future predictions. Pseudocode for the context prediction algorithm is show in figure 11.1. More details about the prediction algorithm can be found in the appendix.

The results from feeding the process log files into the predictor are shown in Figure 11.2. In general, correct prediction percentages ranged from 40-50%. Some processes had prediction percentages close to

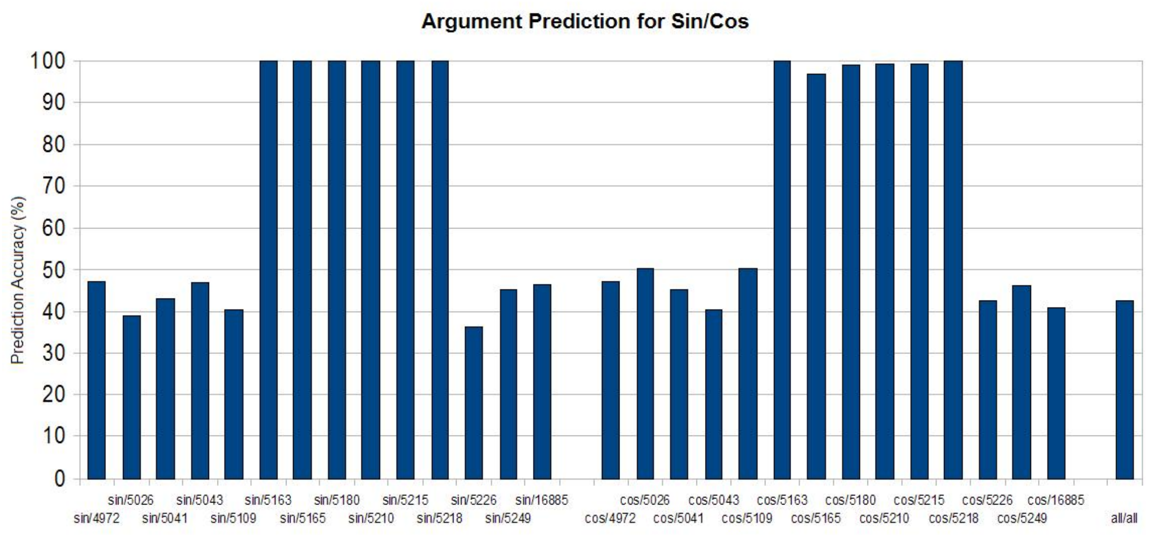

Figure 11.2: Argument prediction accuracy for sine and cosine functions

100%, but these values are ignored since they represent processes that only called the functions once or called the functions with a constant or nearly constant input argument. The prediction percentages of the valid data sets prove that common library functions such as sine and cosine can have reasonably predictable arguments. Using this data, one can assume that other library functions exist that have similar or better argument prediction rates.

Once a function is identified as a proper candidate for a threaded implementation, the next task is figuring out how to implement an effective and efficient prediction system. A software development environment could be created to assist in the integration of pre-calculation into existing function calls. A predictor class object could be coded to serve as an intermediary between the calls to the function and the actual function execution. The class would basically serve as a wrapper that sits in between the program code and the function code. A new instance of the predictor class would be instantiated for each function to be threaded. The normal calls to the function would be replaced by a call to a method of the predictor class. The same arguments would be passed in and the correct output would still be returned. Using this method all prediction, pre-computation, and threading details are abstracted. The application programmer only has to create and set up the predictor object and reroute the function calls to go through it.

When creating the predictor, information about the function is needed to accurately replicate the structure of the function. The number and type of arguments to the function need to be known, as well as the type of the return value. Once this information is know, the method used to replace the function calls can be set to have the exact same interface structure as the real function. In addition, the predictor object needs to be passed the pointer to the original function so that it can use the existing implementation for pre-computation.

The predictor class internally monitors arguments passed into the function and identifies the best prediction algorithm. Once a certain configurable threshold is met, a separate thread is spawned to handle pre-calculation of future function calls. The main thread must communicate the argument patterns to the predictor thread periodically so that it can accurately predict the arguments to future function calls. Similarly, the predictor thread must provide the program thread with predicted future function outputs. These outputs would be stored with the predicted arguments that produced them to allow the main thread to determine if the argument prediction was successful. It may even be beneficial to implement a buffer of future argument-output combinations to increase the chance of a correct prediction. Memoization could also be used to cache a certain number of previous function calls if it is found that calls are periodically duplicated. A happy medium must be reached; the more predicted argument pairs that need to be checked the longer the search overhead becomes. It is possible that the number of future predictions to be buffered and the frequency of thread communication could be adjusted, either statically through a configuration variable or dynamically during program execution (based on current performance). If an argument pair is incorrectly predicted, the function must be computed in-line by the main program thread.

The appropriateness of threaded pre-computation for a particular function may depend on how the function is used inside a particular program. The predictor class could be coded to monitor overall effectiveness and adjust the amount of pre-computation based on the prediction accuracy and processing power availability. It could even switch off the thread if the threading and prediction overhead out weights the benefits of successful pre-computations. Trial and error is the only definitive way to determine if this method would be beneficial for a particular function in a program. If no speedup is found using the predictor class, it does not have to be used.

One benefit of constructing and using a predictor class is that it opens up the possibility of using multiple predictor threads in a single program. Each candidate function would spawn its own predictor thread, allowing a program to have as many predictor threads as it has candidate functions. If properly implemented, a generic predictor class could be used to easily adapt serial programs to many-core systems without putting the burden of parallel programming on the application developer.

Chapter 12

Conclusion

The results of this research have shown that a multithreaded dynamic memory implementation is feasible and that additional parallelism can be extracted from traditional single-threaded applications. The experiments outlined in this thesis have shown a 2-3% performance gain in the three most memory-intensive SPEC CPU2006 benchmarks with the threaded library. A 3-4% speedup in compile time was also observed when the library was used with `gcc` and `llvm` to compile the benchmarks. The Savant VHDL analyzer experienced a 2-3% speedup when used with the threaded library. Although the amount of improvement the library provided varied program to program, in general applications that spent a significant amount of time allocating and deallocating memory saw consistent improvement. This increased performance was obtained simply by linking in the threaded dynamic memory library and did not require any recompiling of application source code. The library performed the best under newer processor architectures than have more effective multithreading hardware support. The results of the research show that the increased performance occurs regardless of the compiler used. Testing has also shown that the library in general performs better in Solaris than in Linux.

By shifting the bulk of the processing associated with dynamic memory management to a separate thread, the threaded library allows the dynamic memory system to take advantage of other cores in the processor that may have otherwise gone unused. Traditionally dynamic memory libraries have had to deal with the trade off between the complexity of the algorithm and the speed at which it operates. Multithreading the dynamic memory system opens up more processing power without negative performance effects on the main application thread. This extra processing capability could allow more complex but efficient algorithms to be used to control and distribute dynamic memory. This would not only provide faster allocation and deallocation functions, but could also further enhance program speed by reducing the number of cache misses by using smarter block allocation. Multithreading dynamic memory provides one way for applications (even single threaded ones) to take advantage of the increased parallelism provided by modern desktop processors. This same approach could be used in the future to thread other computation-intensive dynamic library functions.

As the shift to many-core processing approaches, computer programmers and architects must shift their perspective on multithreaded computing. There are limits to the amount of parallelism that can be programmed into applications and new approaches are needed to fully take advantage of massively parallel architectures. Extra processing cores open up additional computation power that allows optimization methods such as argument prediction to become realistic possibilities. Recent advances in multithreaded hardware support can be seen through the tests with Intel's new Core i7 processor architecture. Although recent improvements can be seen, more work needs to be done in order for fine-grained threading algorithms to be effective. Although the threaded library reduces the number of instructions executed in the dynamic memory functions by around 50%, full speedup is not seen due to bottlenecks in the hardware (atomic operation latency and cache coherency) and in software (operating system scheduling). It is crucial that future processors continue to make headway in improving locking and cross-core communication in order to allow parallel applications to fully take advantage of the benefits that many-core computing provides.

Appendix A

Profiling Results

A.1 SPEC Profiling Results

Percent of total computation captured by callgrind profiling					
Benchmark	**malloc**	**realloc**	**calloc**	**free**	**total**
bwaves	0	0	0	0	0
gamess	0	0	0	0	0
milc	0.01	0	0	0	0.01
zeusmp	n/a	n/a	n/a	n/a	n/a
gromacs	0	0	0	0	0
cactusADM	0	0	0	0	0
leslie3d	0	0	0	0	0
namd	0	0	0	0	0
dealII	n/a	n/a	n/a	n/a	n/a
soplex	0.01	0	0	0	0.01
povray	0.03	0	0	0.02	0.05
calculix	0.02	0	0	0.01	0.03
GemsFDTD	0.01	0	0	0	0.01
tonto	7.52	0	0	5.96	13.48
lbm	0	0	0	0	0
wrf	0.17	0	0	0.09	0.26
sphinx3	0	0	0.23	0.06	0.29
perlbench	0.96	0.16	0	0.57	1.69
bzip2	0	0	0	0	0
gcc	0.19	0	9.68	0.28	10.15
mcf	0	0	0	0	0
gobmk	0.01	0	0	0	0.01
hmmer	0.02	0.02	0.01	0.02	0.07
sjeng	0	0	0	0	0
libquantum	0	0	0	0	0
h264ref	0	0	0	0	0
omnetpp	5.86	0	0	4.24	10.10
astar	0.21	0	0	0.12	0.33
xlancbmk	2.26	0	0	1.24	3.5

$^{n/a}$Data for these programs were not gathered due to their extremely long callgrind runtimes (still running after 7 days)

Table A.1: SPEC CPU2006: Percent of computation in `malloc`/`free` functions

Number of function calls by callgrind profiling					
Benchmark	**malloc**	**realloc**	**calloc**	**free**	**total**
bwaves	1255	0	14	1255	2524
gamess	154892	0	14	154892	309798
milc	25	8	6513	6464	13010
zeusmp	n/a	n/a	n/a	n/a	n/a
gromacs	2063	73	21663	24988	48787
cactusADM	130477	1024	223	126039	257763
leslie3d	308712	0	14	308691	617417
namd	1324	0	14	1322	2660
dealII	n/a	n/a	n/a	n/a	n/a
soplex	231684	69499	1	231684	532868
povray	2451493	46806	14	2415353	4913666
calculix	2339144	124	14	2339532	4678814
GemsFDTD	1101083	0	14	1101083	2202180
tonto	1522510972	0	14	1522510956	3045021942
lbm	25	0	10	4	39
wrf	4975	0	14	4653	9642
sphinx3	387190	0	13837512	14023887	28248589
perlbench	6409331	434213	0	5152610	11996154
bzip2	28	0	8	24	60
gcc	2786034	5697	944655	3710526	7446912
mcf	25	0	10	5	40
gobmk	219868	9659	10	219857	449394
hmmer	982086	368631	122564	1105494	2578775
sjeng	20	21	44	1	86
libquantum	25	21	10	42	98
h264ref	2115	0	102726	105294	210135
omnetpp	267064929	0	14	266998683	534063626
astar	3683334	0	14	3683334	7366682
xlancbmk	135183711	0	14	135183718	270367443

$^{n/a}$Data for these programs were not gathered due to their extremely long callgrind runtimes (still running after 7 days)

Table A.2: SPEC CPU2006: Number of calls to `malloc`/`free` functions

Cache behavior w/ System library malloc/free				
Benchmark	**instr. reads**	**data reads**	**data writes**	**L1 misses**
tonto	2.90456E+12	1.19999E+12	4.36845E+11	34196723504
Perlbench	4.10037E+11	1.4004E+11	89264105432	3135406330
gcc	1.83145E+11	47892244056	52583251009	3481106921
omnetpp	7.47492E+11	2.5168E+11	1.64777E+11	12479977159
astar	8.23048E+11	3.92198E+11	1.29797E+11	11133088784
xlancbmk	1.18955E+12	4.087E+11	1.65069E+11	16867583625
Benchmark	**L2 misses**	**L3 misses**	**total cycles**	**CPI**
tonto	16635135900	926451704	1.05056E+13	3.62
perlbench	1546861571	41683188	1.42256E+12	3.47
gcc	1559180940	362745042	7.00363E+11	3.82
omnetpp	4252388122	3975200916	2.90698E+12	3.89
astar	5558265914	16556956	3.24552E+12	3.94
xlancbmk	8062219260	743145106	4.01294E+12	3.37

Table A.3: SPEC CPU2006: Cache effects w/ System `malloc`/`free` library

Cache behavior w/ LFB malloc/free				
Benchmark	**instr. reads**	**data reads**	**data writes**	**L1 misses**
tonto	2.66818E+12	1.1366E+12	3.83722E+11	33724322120
perlbench	3.85193E+11	1.33619E+11	82771151699	2950155645
gcc	1.82563E+11	47722105880	52458818564	3470181773
omnetpp	7.00072E+11	2.37593E+11	1.53149E+11	12053399531
astar	8.22744E+11	3.9211E+11	1.29729E+11	11190829139
xlancbmk	1.160E+12	3.99297E+11	1.58126E+11	15685476832
Benchmark	**L2 misses**	**L3 misses**	**total cycles**	**CPI**
tonto	16233238298	1257845525	9.79893E+12	3.67
perlbench	1453636767	42882111	1.34054E+12	3.48
gcc	1553997929	362185916	6.98258E+11	3.82
omnetpp	3959248784	4134901963	2.74869E+12	3.93
astar	5588850075	13128990	3.24622E+12	3.96
xlancbmk	7.289E+09	1107572947	3.89333E+12	3.36

Table A.4: SPEC CPU2006: Cache effects w/ LFB `malloc`/`free` library

Intel i7 cache latencies	
L1 access	4 cycles
L2 access	10 cycles
L3 access	40 cycles
mem access	100 cycles

Table A.5: Numbers used for Intel i7 cycle estimation calculation

Notes: L2 misses were estimated using the equation L2 miss = (L1 miss - L3 miss)/2
L3 misses were halved in calculating total cycles to account for hardware prefetching
total cycles = instr. reads + 4*data reads + 4*data writes + 10*L1 miss + 40*L2 miss + 100*L3 miss*0.5

A.2 ns2 Profiling Results

Percent of total computation captured by callgrind profiling					
ns2 benchmark	**malloc**	**realloc**	**calloc**	**free**	**total**
speedtest1	1.68	0.01	0	1.7	3.39
speedtest2	1.79	0.03	0	1.83	3.65
speedtest3	1.43	0	0	1.52	2.95
speedtest6	1.14	0.19	0	0.92	2.25
speedtest7	0.21	0.01	0	0.18	0.4
speedtest8	0.02	0	0	0.02	0.04
speedtest9	0.02	0	0	0.02	0.04
speedtest10	0.21	0.03	0	0.16	0.4
speedtest11	0.04	0	0	0.03	0.07
speedtest12	0.02	0	0	0.02	0.04

Table A.6: ns2: Percent of computation in `malloc`/`free` functions

Percent of total computation captured by callgrind profiling			
ns2 benchmark	**Packet::alloc**	**Packet::free**	**total**
speedtest1	2.42	2.21	4.63
speedtest2	2.38	2.18	4.56
speedtest3	2.45	1.52	3.97
speedtest6	7.59	7.28	14.87
speedtest7	8.76	8.02	16.78
speedtest8	8.80	8.07	16.87
speedtest9	8.87	8.14	17.01
speedtest10	8.71	7.99	16.70
speedtest11	8.85	8.11	16.96
speedtest12	8.86	8.13	16.99

Table A.7: ns2: Percent of total computation in internal memory functions

Appendix B

Runtime Performance Results

B.1 Linux Benchmark Results

Linux gcc scores		
Benchmark	**System**	**LFB**
bwaves	14.9	14.2
gamess	14.9	14.9
milc	16.4	16.7
zeusmp	12.3	12.3
gromacs	6.6	6.6
cactusADM	8.7	8.2
leslie3d	9.1	8.7
namd	12.1	12.1
dealII	18.9	18.2
soplex	22.2	21.9
povray	15.3	15.3
calculix	4.5	4.5
GemsFDTD	10.5	10.0
tonto	11.1	11.5
lbm	33.8	32.0
wrf	n/a	n/a
sphinx3	27.1	27.0
perlbench	19.5	19.8
bzip2	12.4	12.4
gcc	20.2	20.6
mcf	29.2	29.3
gobmk	17.2	17.2
hmmer	8.2	8.2
sjeng	17.1	17.1
libquantum	28.8	28.8
h264ref	21.5	21.5
omnetpp	16.6	16.9
astar	11.8	12.0
xlancbmk	21.8	23.9

[n/a]Data for these programs were not gathered due build or runtime failures.

Table B.1: Performance results running SPEC CPU2006 benchmarks compiled w/ Linux gcc compiler on Intel i7

Linux llvm scores		
Benchmark	**System**	**LFB**
bwaves	14.8	14.2
gamess	14.9	14.8
milc	15.0	15.2
zeusmp	12.3	12.3
gromacs	6.5	6.5
cactusADM	8.6	8.0
leslie3d	9.1	8.7
namd	11.1	11.1
dealII	18.9	18.1
soplex	22.1	21.7
povray	n/a	n/a
calculix	4.3	4.3
GemsFDTD	10.5	9.9
tonto	11.2	11.6
lbm	31.6	31.2
wrf	n/a	n/a
sphinx3	19.7	19.8
perlbench	16.3	16.6
bzip2	11.5	11.5
gcc	18.0	18.3
mcf	26.9	27.0
gobmk	15.0	15.0
hmmer	8.5	8.5
sjeng	14.7	14.7
libquantum	22.1	22.1
h264ref	21.9	21.8
omnetpp	15.7	16.0
astar	11.1	11.3
xlancbmk	17.0	18.6

[n/a]Data for these programs were not gathered due build or runtime failures.

Table B.2: Performance results running SPEC CPU2006 benchmarks compiled w/ Linux llvm compiler on Intel i7

B.2 Solaris Benchmark Results

Solaris Sun Studio scores		
Benchmark	**System**	**LFB**
bwaves	28.7	28.7
gamess	14.5	14.5
milc	18.0	17.8
zeusmp	18.3	18.3
gromacs	11.1	11.1
cactusADM	14.7	14.2
leslie3d	20.6	18.7
namd	13.3	13.3
dealII	21.7	20.8
soplex	24.1	23.4
povray	16.1	16.2
calculix	14.0	13.3
GemsFDTD	20.0	19.1
tonto	17.2	19.0
lbm	38.0	34.5
wrf	16.7	16.4
sphinx3	25.8	25.9
perlbench	19.0	20.3
bzip2	13.5	13.5
gcc	18.2	18.2
mcf	29.4	29.3
gobmk	16.8	16.8
hmmer	16.3	16.3
sjeng	17.7	17.7
libquantum	38.9	38.6
h264ref	26.3	26.2
omnetpp	14.9	16.6
astar	12.2	12.6
xlancbmk	21.5	21.6

Table B.3: Performance results running SPEC CPU2006 benchmarks compiled w/ Solaris Sun Studio compiler on Intel i7

Solaris gcc scores		
Benchmark	**System**	**LFB**
bwaves	15.9	11.5
gamess	n/a	n/a
milc	16.6	16.5
zeusmp	n/a	n/a
gromacs	6.7	6.7
cactusADM	n/a	n/a
leslie3d	n/a	n/a
namd	12.1	n/a
dealII	n/a	n/a
soplex	22.2	21.7
povray	n/a	n/a
calculix	n/a	n/a
GemsFDTD	n/a	n/a
tonto	9.3	10.4
lbm	33.0	32.7
wrf	n/a	n/a
sphinx3	27.1	27.0
perlbench	n/a	n/a
bzip2	12.3	12.2
gcc	19.1	19.1
mcf	28.4	28.4
gobmk	17.1	17.1
hmmer	8.2	8.1
sjeng	17.3	17.3
libquantum	n/a	n/a
h264ref	22.1	22.1
omnetpp	15.3	16.9
astar	11.4	11.8
xlancbmk	19.7	20.6

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.4: Performance results running SPEC CPU2006 benchmarks compiled w/ Solaris gcc compiler on Intel i7

B.3 Hardware Platform Results

Core 2 Duo scores		
Benchmark	**system**	**LFB**
bwaves	13.1	8.6
gamess	10.8	10.8
milc	9.5	8.4
zeusmp	9.6	9.6
gromacs	6.4	6.5
cactusADM	5.8	5.0
leslie3d	5.8	4.2
namd	9.9	10.0
dealII	16.4	11.6
soplex	13.3	12.0
povray	11.5	12.2
calculix	4.2	3.5
GemsFDTD	7.0	5.0
tonto	7.8	6.1
lbm	10.6	8.2
wrf	n/a	n/a
sphinx3	16.5	16.1
perlbench	14.9	14.5
bzip2	10.4	10.4
gcc	13.6	13.1
mcf	16.5	16.5
gobmk	13.0	13.0
hmmer	7.3	7.3
sjeng	12.7	12.7
libquantum	14.2	14.2
h264ref	17.3	16.1
omnetpp	10.1	9.5
astar	8.5	8.7
xlancbmk	13.1	14.2

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.5: Performance results running SPEC CPU2006 benchmarks compiled w/ Linux gcc compiler on Intel Core 2 Duo

AMD64x2 benchmark scores		
Benchmark	**system**	**LFB**
bwaves	5.1	6.9
gamess	11.4	11.4
milc	8.4	8.2
zeusmp	3.5	3.5
gromacs	5.6	5.6
cactusADM	6.0	5.0
leslie3d	5.8	4.7
namd	9.3	9.3
dealII	11.9	9.6
soplex	7.5	6.7
povray	14.5	14.3
calculix	3.7	3.4
GemsFDTD	3.0	2.8
tonto	6.7	6.2
lbm	9.0	7.0
wrf	n/a	n/a
sphinx3	11.2	10.8
perlbench	12.1	12.0
bzip2	6.4	6.6
gcc	5.6	5.5
mcf	7.4	7.8
gobmk	12.9	12.8
hmmer	7.2	7.2
sjeng	10.6	10.6
libquantum	10.1	10.4
h264ref	11.7	11.4
omnetpp	6.2	5.6
astar	5.8	5.9
xlancbmk	9.1	8.7

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.6: Performance results running SPEC CPU2006 benchmarks compiled w/ Linux gcc compiler on AMD64 X2

B.4 Speedup Comparisons

Comparing SPEC CPU2006 speedups by architecture			
Benchmark	**Core 2 Duo**	**AMD64x2**	**Core i7**
bwaves	0.65	1.36	0.95
gamess	1.00	1.00	1.00
milc	0.89	0.98	1.02
zeusmp	1.00	1.01	1.00
gromacs	1.00	1.00	1.00
cactusADM	0.86	0.84	0.95
leslie3d	0.72	0.80	0.95
namd	1.00	1.00	1.00
dealII	0.71	0.81	0.96
soplex	0.90	0.89	0.99
povray	1.06	0.99	1.00
calculix	0.84	0.92	1.00
GemsFDTD	0.72	0.92	0.95
tonto	0.78	0.92	1.04
lbm	0.77	0.78	0.95
wrf	n/a	n/a	n/a
sphinx3	0.98	0.96	1.00
perlbench	0.97	0.99	1.02
bzip2	1.00	1.04	1.00
gcc	0.96	0.99	1.02
mcf	1.00	1.05	1.00
gobmk	1.00	0.99	1.00
hmmer	1.00	1.00	1.00
sjeng	1.00	1.00	1.00
libquantum	1.00	1.03	1.00
h264ref	0.93	0.97	1.00
omnetpp	0.94	0.90	1.02
astar	1.03	1.01	1.02
xlancbmk	1.08	0.95	1.10

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.7: SPEC benchmark score speedups with the LFB library on different architectures

Comparing SPEC CPU2006 speedups by compiler		
Benchmark	**gcc**	**llvm**
bwaves	0.95	0.96
gamess	1.00	0.99
milc	1.02	1.01
zeusmp	1.00	1.00
gromacs	1.00	1.00
cactusADM	0.95	0.93
leslie3d	0.95	0.95
namd	1.00	1.00
dealII	0.96	0.96
soplex	0.99	0.98
povray	1.00	n/a
calculix	1.00	0.99
GemsFDTD	0.95	0.95
tonto	1.04	1.04
lbm	0.95	0.99
wrf	n/a	n/a
sphinx3	1.00	1.01
perlbench	1.02	1.02
bzip2	1.00	1.00
gcc	1.02	1.02
mcf	1.00	1.00
gobmk	1.00	1.00
hmmer	1.00	1.00
sjeng	1.00	1.00
libquantum	1.00	1.00
h264ref	1.00	1.00
omnetpp	1.02	1.02
astar	1.02	1.02
xlancbmk	1.10	1.09

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.8: SPEC benchmark score speedups with the LFB library using different compilers

Comparing SPEC CPU2006 speedups by O/S			
Benchmark	**linux-gcc**	**Solaris-ss**	**Solaris-gcc**
bwaves	0.95	1.00	0.72
gamess	1.00	1.00	n/a
milc	1.02	0.99	0.99
zeusmp	1.00	1.00	n/a
gromacs	1.00	1.00	1.00
cactusADM	0.95	0.97	n/a
leslie3d	0.95	0.91	n/a
namd	1.00	1.00	n/a
dealII	0.96	0.96	n/a
soplex	0.99	0.97	0.98
povray	1.00	1.01	n/a
calculix	1.00	0.95	n/a
GemsFDTD	0.95	0.96	n/a
tonto	1.04	1.10	1.11
lbm	0.95	0.91	0.99
wrf	n/a	0.98	n/a
sphinx3	1.00	1.00	1.00
perlbench	1.02	1.07	n/a
bzip2	1.00	1.00	0.99
gcc	1.02	1.00	1.00
mcf	1.00	1.00	1.00
gobmk	1.00	1.00	1.00
hmmer	1.00	1.00	1.00
sjeng	1.00	1.00	1.00
libquantum	1.00	0.99	n/a
h264ref	1.00	1.00	1.00
omnetpp	1.02	1.11	1.10
astar	1.02	1.03	1.04
xlancbmk	1.10	1.00	1.05

[n/a]Data for these programs were not gathered due build or runtime failures.

Table B.9: SPEC benchmark score speedups with the LFB library on different operating systems

B.5 Solaris Allocators

Solaris allocation library benchmarks			
Benchmark	**dlmalloc**	**system**	**LFB**
bwaves	28.2	28.70	28.6
gamess	14.50	14.50	14.50
milc	15.10	18.00	17.80
zeusmp	18.30	18.30	18.30
gromacs	10.90	11.10	11.10
cactusADM	14.80	14.70	14.20
leslie3d	17.70	20.60	18.70
namd	13.30	13.30	13.30
dealII	21.60	21.70	20.80
soplex	23.10	24.10	23.40
povray	16.20	16.10	15.90
calculix	13.30	14.00	13.30
GemsFDTD	16.20	20.00	19.10
tonto	15.90	17.20	19.00
lbm	26.80	38.00	24.80
wrf	14.40	16.70	16.40
sphinx3	25.70	25.80	25.90
perlbench	16.10	19.00	20.30
bzip2	13.50	13.50	13.50
gcc	17.40	18.20	18.20
mcf	29.30	29.40	29.30
gobmk	16.80	16.80	16.80
hmmer	16.30	16.30	16.30
sjeng	17.70	17.70	17.70
libquantum	38.90	38.90	38.60
h264ref	26.30	26.30	26.20
omnetpp	16.30	14.90	16.60
astar	12.40	12.20	12.60
xlancbmk	21.10	21.50	21.60

Table B.10: SPEC benchmark scores with various memory allocation libraries on Solaris

B.6 Memory Usage

Library percent memory usage			
Benchmark	**system**	**LFB**	**difference**
bwaves	28.13	29.40	0.05
gamess	0.10	0.10	0.00
milc	20.77	23.00	0.11
zeusmp	16.50	16.50	0.00
gromacs	0.40	0.50	0.25
cactusADM	20.50	20.50	0.00
leslie3d	4.00	4.00	0.00
namd	1.50	n/a	n/a
dealII	3.43	9.57	1.79
soplex	2.80	7.20	1.57
povray	0.10	0.20	1.00
calculix	1.43	10.27	6.16
GemsFDTD	27.20	27.30	0.00
tonto	0.30	0.97	2.22
lbm	13.40	13.50	0.01
wrf	n/a	n/a	n/a
sphinx3	1.10	1.30	0.18
perlbench	4.67	9.27	0.99
bzip2	19.53	19.70	0.01
gcc	2.17	7.03	2.25
mcf	27.60	27.60	0.00
gobmk	0.60	0.60	0.00
hmmer	0.73	2.83	2.86
sjeng	5.70	5.70	0.00
libquantum	2.10	3.30	0.57
h264ref	0.77	0.73	-0.04
omnetpp	3.23	4.73	0.46
astar	5.67	8.93	0.58
xlancbmk	6.67	9.67	0.45
Average:	7.90	9.79	1.90

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.11: SPEC benchmark memory usage with various memory allocation libraries

Notes: Memory measurements are percent of total memory (3GB) measured with ps command
Measurements were taken at 60s, 180s, and 300s into each benchmark and averaged

	system library memory usage				**LFB library memory usage**			
Benchmark	**meas. 1**	**meas. 2**	**meas. 3**	**Average**	**meas. 1**	**meas. 2**	**meas. 3**	**Average**
bwaves	27	28.7	28.7	28.13	29.4	29.4	29.4	29.40
gamess	0.1	0.1	0.1	0.10	0.1	0.1	0.1	0.10
milc	20.8	21.4	20.1	20.77	21.8	24.2	23	23.00
zeusmp	16.5	16.5	16.5	16.50	16.5	16.5	16.5	16.50
gromacs	0.4	0.4	0.4	0.40	0.5	0.5	0.5	0.50
cactusADM	20.5	20.5	20.5	20.50	20.5	20.5	20.5	20.50
leslie3d	4	4	4	4.00	4	4	4	4.00
namd	1.5	1.5	1.5	1.50	n/a	n/a	n/a	n/a
dealII	3.2	0.3	6.8	3.43	3	9.5	16.2	9.57
soplex	2.8	2.8	2.8	2.80	7	7.3	7.3	7.20
povray	0.1	0.1	0.1	0.10	0.2	0.2	0.2	0.20
calculix	2.5	0.9	0.9	1.43	7.5	10.8	12.5	10.27
GemsFDTD	27.2	27.2	27.2	27.20	27.3	27.3	27.3	27.30
tonto	0.1	0.2	0.6	0.30	0.2	1.3	1.4	0.97
lbm	13.4	13.4	13.4	13.40	13.5	13.5	13.5	13.50
wrf	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
sphinx3	1.1	1.1	1.1	1.10	1.3	1.3	1.3	1.30
perlbench	3.9	4.7	5.4	4.67	6.1	9.3	12.4	9.27
bzip2	27.7	27.8	3.1	19.53	27.8	28.1	3.2	19.70
gcc	1.1	3.8	1.6	2.17	1.5	5.7	13.9	7.03
mcf	27.6	27.6	27.6	27.60	27.6	27.6	27.6	27.60
gobmk	0.6	0.6	0.6	0.60	0.6	0.6	0.6	0.60
hmmer	0.7	0.7	0.8	0.73	1.8	2.8	3.9	2.83
sjeng	5.7	5.7	5.7	5.70	5.7	5.7	5.7	5.70
libquantum	2.1	2.1	2.1	2.10	3.3	3.3	3.3	3.30
h264ref	0.8	0.4	1.1	0.77	1	0.5	0.7	0.73
omnetpp	3.2	3.2	3.3	3.23	4.7	4.7	4.8	4.73
astar	9.2	1.6	6.2	5.67	9.3	8.1	9.4	8.93
xlancbmk	3.5	6.4	10.1	6.67	5.1	9.3	14.6	9.67

$^{n/a}$Data for these programs were not gathered due build or runtime failures.

Table B.12: SPEC benchmark memory usage with the system and LFB libraries.

Notes: Memory measurements are percent of total memory (3GB) measured with ps command
Measurements were taken at 60s, 180s, and 300s into each benchmark and averaged

Appendix C

Atomic Operation Latencies

Core i7	
code measured	**latency (s)**
Base Cost of for loop	2.65068E-09
Cost of (counter=counter + 1) - baseCost	3.50369E-10
Cost of __sync_fetch_and_add(&counter,1) - baseCost	6.72792E-09

Table C.1: Increment code latencies for a Core i7 processor system

AMD Phenom II X4	
code measured	**latency (s)**
Base Cost of for loop	3.49930E-09
Cost of (counter=counter + 1) - baseCost	1.70315E-10
Cost of __sync_fetch_and_add(&counter,1) - baseCost	4.41543E-09

Table C.2: Increment code latencies for a AMD Phenom II X4 processor system

Quad-core Xeon	
code measured	**latency (s)**
Base Cost of for loop	3.03104E-09
Cost of (counter=counter + 1) - baseCost	4.16176E-10
Cost of __sync_fetch_and_add(&counter,1) - baseCost	1.64959E-08

Table C.3: Increment code latencies for a quad-core Xeon processor system

Appendix D

Sin/Cos Argument Prediction

Prediction algorithm notes: The algorithm that has been used to predict the future data values is based on context. This predictor learns from the previous values that follow certain context which is a finite sequence of values and predicts the future value when the same context repeats. The paper by Yiannakis Sazeides *et al* states that the context predictor has an accuracy from 50%-90% [45]. This context prediction algorithm was implemented with slight modification. This context prediction algorithm considers a matrix to store different contexts as the learning process progresses. Based on the previous values a particular context is chosen and if the context contains a value after the previous value a prediction is made. However, if it does not contain any future value after the previous value a prediction based on the maximum frequency of occurrences in the particular context is made. When two or more values have the same maximum frequency in the same context then we go for a random selection among them. If a prediction previously made is wrong then the predictor corrects the existing contexts or creates new contexts so that it becomes more and more accurate as the predictions progress. This algorithm when implemented in Matlab has given correct predictions from about 40%-50% of the times for the considered data values. This prediction process was also repeated using other algorithms based on clustering the data values and using decision trees. These have also given approximately the same result in predicting correct values for the considered data.

Sin/Cos argument prediction		
Function	**PID**	**% Correct**
tan	4972	99.90
tan	5041	99.90
tan	5043	99.90
tan	5109	99.90
tan	5226	99.90
tan	5249	99.90
tan	16885	99.90
sin	5226	36.23
sin	5026	38.90
sin	5109	40.21
sin	5041	43.10
sin	5249	45.21
sin	16885	46.38
sin	5043	46.71
sin	4972	47.20
sin	5163	99.90
sin	5165	99.90
sin	5180	99.90
sin	5210	99.90

Continued on next page

Sin/Cos argument prediction

Function	**PID**	**% Correct**
sin	5215	99.90
sin	5218	99.90
cos	5043	40.21
cos	16885	41.01
cos	5226	42.56
cos	5041	45.40
cos	5249	46.23
cos	4972	47.23
cos	5026	50.10
cos	5109	50.23
cos	5165	96.78
cos	5180	99.10
cos	5210	99.30
cos	5215	99.38
cos	5163	99.90
cos	5218	99.90
all	16885	34.38
all	5041	40.30
all	5026	41.23
all	4972	42.11

Continued on next page

Sin/Cos argument prediction		
Function	**PID**	**% Correct**
all	5249	42.11
all	5043	43.87
all	5226	46.32
all	5109	54.32
all	5165	95.70
all	5180	99.30
all	5215	99.40
all	5210	99.50
all	5163	99.90
all	5218	99.90

Table D.1: Argument prediction correctness for sin/cos functions

Appendix E

LFB Library Code

```
// Lock-free bins (LFB) threaded malloc library
//
// Edward Herrmann
// University of Cincinnati
// May 2010

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <errno.h>
#include <signal.h>
#include <string.h>

#define IS64BIT 0
#define MULTI_THREADED 1   //uses extra locks around malloc and free calls if multithreaded
#define SOLARIS 0   //use different sleep function if using solaris
#define MEMALIGN_USED 0   //enable if memalign functions are used
#define MAX_BIN_SIZE 10000
#define MIN_BIN_SIZE_SHIFT (2+IS64BIT)  //size shifted by this many bits to determine smallest
    bin size
#define NUM_BINS 32  //change to 64 if in 64 bit mode
#define MALLOCS_NEEDED_FOR_REFRESH 1000
#define FREES_NEEDED_FOR_REFRESH 100
#define BIN_SIZE_TYPE short int
#define OFFSET_TYPE short int
#define BIN_SIZE_BYTES (sizeof(block)-sizeof(block*)) //bytes reserved for storing bin size
#define MIN_BLOCK_SIZE sizeof(block*) //bytes reserved for storing bin size

#define findbinsize(x) ( sizeof(int)*8-__builtin_clz((unsigned)((((x)-1)>>(MIN_BIN_SIZE_SHIFT-1))
    |1))-1 )


// ---- function declarations ----
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void *mem_manager_thread(void *ptr);
void create_mem_manager_thread();
void fork_cleanup();
void ex_program(int sig);
void immediate_malloc();

#ifdef MEMALIGN_USED
void *valloc(size_t size);
void *memalign(size_t biundary, size_t size);
int posix_memalign(void **memptr, size_t alignment, size_t size);
#endif

//function pointers
void *(*mallocp)(size_t size) = NULL;
void (*freep)(void*) = NULL;

// ---- structures ----
typedef struct block{
    BIN_SIZE_TYPE size;
    OFFSET_TYPE offset;
    struct block *next;
} block;

typedef struct bininfo{
    int blocksinuse;
    int lastrefill;  //refreshes since bin last refilled
    int lastused;    //refreshes since bin last used
    int binsize;
    int blocksfree;
    int locked;
    block *firstblock;
    block *lastblock;
} bininfo;

// ---- global variables ----

//locks and inter-thread message variables
volatile int mallocrefreshneeded = 0;
volatile int freerefreshneeded = 0;
volatile int lastmallocrefresh = 0;
volatile int lastfreerefresh = 0;
volatile int emptybin = -1;
volatile int initializing = 0;
volatile int freelist_initialized = 0;

volatile void** freelist_head = NULL;
volatile void** freelist_tail = NULL;
volatile bininfo bins[NUM_BINS];

pthread_mutex_t initlock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t datalock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t requestready = PTHREAD_COND_INITIALIZER;

pthread_t managerthread;
int mallocmisses = 0;
int freemisses = 0;
int largemallocs = 0;
volatile int threadid = 0;


void *calloc(size_t nelem, size_t elsize){
    void *returnval;

    returnval = malloc(nelem*elsize);
    if (returnval != NULL){
        memset(returnval, 0, (size_t)(nelem*elsize));
    }
    return returnval;
}

void *realloc(void *ptr, size_t size){
    void *returnval;

    if (size == 0) free(ptr);
    if (ptr == NULL) return malloc(size);
    if (findbinsize(size) <= ((block*)((void*)ptr-BIN_SIZE_BYTES))->size) return ptr;
    else {
        returnval = malloc(size);
        if (returnval != NULL){
            size = 1<<(((block*)((void*)ptr-BIN_SIZE_BYTES))->size + MIN_BIN_SIZE_SHIFT);
            memcpy(returnval, ptr, size);
        }
        return returnval;
    }
}

#ifdef MEMALIGN_USED
void *valloc(size_t size)
{
    return memalign(getpagesize(),size);
}

void *memalign(size_t boundary, size_t size)
{
    void * returnaddr;
    int curbinindex;
    int blockoffset = 0;
    size += boundary;
    returnaddr = malloc(size);
    curbinindex = ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->size;
    while (((int)returnaddr % boundary) != 0){
        returnaddr++;
        blockoffset++;
    }
    ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->size = (short)curbinindex;
    ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->offset = blockoffset;
    return returnaddr;
}

int posix_memalign(void **memptr, size_t alignment, size_t size)
{
    void * returnaddr;
    int curbinindex;
    int blockoffset = 0;
    size += alignment;
    returnaddr = malloc(size);
    curbinindex = ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->size;
    while (((int)returnaddr % alignment) != 0){
        returnaddr++;
        blockoffset++;
    }
    ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->size = curbinindex;
    ((block*)((void*)returnaddr-BIN_SIZE_BYTES))->offset = blockoffset;
    *memptr = returnaddr;
    return 0;
}
#endif

void *malloc(size_t size)
{
    void *memLocation = NULL;
    int binindex;
    int blockindex;
    //create memman thread if not done so already
    if (!threadid)
    {
        if (initializing) {
            if (mallocp == NULL){
                return NULL;
            }
            if (size < MIN_BLOCK_SIZE)
                size = MIN_BLOCK_SIZE;
            memLocation = mallocp((size)+BIN_SIZE_BYTES);
            if (memLocation != NULL){
                ((block*)memLocation)->size = findbinsize(size);
                ((block*)memLocation)->offset = 0;
                memLocation+=BIN_SIZE_BYTES;
            }
            return memLocation;
        } else {
            create_mem_manager_thread();
        }
    }

    //find proper bin index
    binindex = findbinsize(size);
    __builtin_prefetch (bins[binindex].firstblock, 1, 2);

    //check if bin is empty
    if (bins[binindex].blocksfree < 1){

        if (size <= 0) return NULL;

        //increment bin size
        if (bins[binindex].binsize != MAX_BIN_SIZE){
            __sync_fetch_and_add(&bins[binindex].binsize,1);
        }

        //signal bin refresh
        mallocrefreshneeded = 1;
        if (emptybin == -1) emptybin = binindex;
        lastmallocrefresh = 0;
        pthread_cond_signal(&requestready);

    //check if periodic refill required
    } else if (++lastmallocrefresh > MALLOCS_NEEDED_FOR_REFRESH){
        lastmallocrefresh = 0;
        mallocrefreshneeded = 1;
        pthread_cond_signal(&requestready);
    }

    do
    {
        //check if free block exists
        if (bins[binindex].blocksfree > 0){

             //atomically decrement free index
            blockindex = __sync_fetch_and_sub(&(bins[binindex].blocksfree),1);

            //check if valid block obtained
            if (blockindex < 1){

                //just missed the last block
                //undo decrement
                __sync_fetch_and_add(&(bins[binindex].blocksfree),1);

                //signal refresh
                mallocrefreshneeded = 1;
                if (emptybin == -1) emptybin = binindex;
                pthread_cond_signal(&requestready);
```

```
#ifdef SOLARIS
                sched_yield();
#else
                pthread_yield();
#endif
            } else {

                //grab first block from bin
                do {
                    memLocation = (void*)bins[binindex].firstblock;
                } while (__sync_bool_compare_and_swap(&(bins[binindex].firstblock),memLocation,((
                    block*)memLocation)->next) == 0);
                __builtin_prefetch (((block*)memLocation)->next, 0, 2);

                //return new block
                return &(((block*)memLocation)->next);
            }
        } else {
            //bin empty

            //signal refresh
            if (!mallocrefreshneeded) mallocrefreshneeded = 1;
            if (emptybin == -1) emptybin = binindex;
            pthread_cond_signal(&requestready);
#ifdef SOLARIS
            sched_yield();
#else
            pthread_yield();
#endif
        }
    } while (1);
}

void free(void *ptr)
{
#if MULTI_THREADED
    void **oldaddr;
#endif
    int binindex;

    //add to freelist
    if (ptr == NULL) return;

#if MEMALIGN_USED
    //remove offset if block was aligned
    if (((block*)((void*)ptr-BIN_SIZE_BYTES))->offset){
            ptr -= ((block*)((void*)ptr-BIN_SIZE_BYTES))->offset;
    }
#endif

    //find binindex
    binindex = ((block*)((void*)ptr-BIN_SIZE_BYTES))->size;

    //if bin not full, add block back into bin
    if (bins[binindex].blocksfree < bins[binindex].binsize){

            do {
                *(void**)(ptr) = (void*)bins[binindex].firstblock;
            } while (__sync_bool_compare_and_swap(&(bins[binindex].firstblock),*(void**)(ptr),(
                void*)ptr-BIN_SIZE_BYTES) == 0);
            //if we are adding the first block, set the last block pointer
            if (*(void**)(ptr) == NULL) bins[binindex].lastblock = (block*)((void*)ptr-
                BIN_SIZE_BYTES);

            //increment number of free blocks
            __sync_fetch_and_add(&(bins[binindex].blocksfree),1);
    } else {
        //bin full, add block to free list

        //set address value to NULL
        *((void**)ptr) = NULL;

#if MULTI_THREADED
        //update tail pointer
        oldaddr = (void**)__sync_lock_test_and_set(&freelist_tail,ptr);

        //add new block into free list
        if (oldaddr != NULL)
            *oldaddr = ptr;

        if (!freelist_initialized){
            __sync_bool_compare_and_swap(&freelist_head,NULL,ptr);
            freelist_initialized = 1;
        }

#else
        //add new block into free list
        if (freelist_tail != NULL)
            *freelist_tail = ptr;

        freelist_tail = ptr;
        if (!freelist_initialized){
            freelist_head = ptr;
            freelist_initialized = 1;
        }
#endif

        //check if periodic free list processing required
        if (++lastfreerefresh > MALLOCS_NEEDED_FOR_REFRESH){
            mallocrefreshneeded = 1;
            freerefreshneeded = 1;
            lastfreerefresh = 0;
            pthread_cond_signal(&requestready);
        }
    }
}

void *mem_manager_thread(void *ptr){
        void **cur_head;
        void **prev_head;
        int emptyfraction = 0;
        int binindex = 0;
        int refillindex = 0;
        int refillsneeded = 0;
        int blocksadded = 0;
        void* freeaddress;
        block *curblock = NULL;
        block *prevblock = NULL;
        block *firstblock = NULL;
        block *lastblock = NULL;
        (void) signal(SIGINT, ex_program);

        pthread_mutex_lock(&datalock);
        threadid = 1; // signal initialization ready

        while(1){
            //wait for new request
            if ((!mallocrefreshneeded) && (!freerefreshneeded)){
                pthread_cond_wait(&requestready, &datalock);
            }

            //if bin miss, create new block immediately
            if (emptybin != -1) immediate_malloc(emptybin);
            if (mallocrefreshneeded || (!freerefreshneeded)){
                //refill bins
                if (mallocrefreshneeded) mallocrefreshneeded = 0;

                //refill nearly empty bins first
                for (emptyfraction = 8; emptyfraction > 1; emptyfraction/=2){
                        for (binindex = 0; binindex < NUM_BINS; binindex++){
                            if (bins[binindex].blocksfree < ((bins[binindex].binsize+1)/
                                emptyfraction)){
                                //bin needs to be refilled

                                refillsneeded = ((bins[binindex].binsize+1)/2) - bins[binindex].
                                    blocksfree;

                                if (refillsneeded > 0){
                                        prevblock = NULL;
                                        firstblock = NULL;
                                        blocksadded = 0;
                                        cur_head = (void*)freelist_head;
                                        prev_head = NULL;

                                        //use empty blocks from freelist if available
                                        while ((void*)cur_head != (void*)freelist_tail){
                                            if (emptybin != -1) immediate_malloc(emptybin);
                                            if ((freelist_initialized)&&(cur_head != NULL)){
                                                if ((void*)(*cur_head) != NULL){
                                                        curblock = (block*)((void*)cur_head -
                                                            BIN_SIZE_BYTES);
                                                        if (curblock->size == binindex){
                                                                if (prevblock != NULL){
                                                                        prevblock->next =
                                                                            curblock;
                                                                } else {
                                                                        firstblock = curblock;
                                                                }
                                                                prevblock = curblock;
                                                                blocksadded++;
                                                                refillsneeded--;
                                                                if (prev_head == NULL){
                                                                        freelist_head = (void*)(*
                                                                            cur_head);
                                                                } else {
                                                                        *prev_head = (void*)(*
                                                                            cur_head);
                                                                }
                                                                if (refillsneeded == 0) break;
                                                        }
                                                        else prev_head = cur_head;
                                                        cur_head = ((void**)(*cur_head));
                                                } else break;
                                            } else break;
                                        }

                                        // malloc any remaining blocks needed
                                        for (refillindex = 0; refillindex < refillsneeded;
                                            refillindex++){
                                                if (emptybin != -1) immediate_malloc(emptybin);
                                            curblock = ((block*)(mallocp((1<<(binindex+
                                                MIN_BIN_SIZE_SHIFT))+BIN_SIZE_BYTES)));
                                            if (curblock != NULL){
                                                curblock->size = binindex;
                                                curblock->offset = 0;
                                                if (prevblock != NULL){
                                                        prevblock->next = curblock;
                                                } else {
                                                        firstblock = curblock;
                                                }
                                                prevblock = curblock;
                                                blocksadded++;
                                            }
                                        }
                                        if (prevblock != NULL){
                                          lastblock = prevblock;
                                          lastblock->next = NULL;
                                        }

                                        //place the linked list of new blocks into the bin
                                        if (firstblock != NULL){
                                                //reserve last block during pointer redirection
                                                if (__sync_fetch_and_sub(&(bins[binindex].
                                                    blocksfree),1) < 1){
                                                    //no free blocks left
                                                    //add new blocks to front of list
                                                    do {
                                                        lastblock->next = bins[binindex].
                                                            firstblock;
                                                    } while (__sync_bool_compare_and_swap(&(bins[
                                                        binindex].firstblock),lastblock->next,
                                                        firstblock) == 0);
                                                    if (lastblock->next == NULL) bins[binindex].
                                                        lastblock = lastblock;
                                                } else {
                                                    //last block reserved, add new blocks to the
                                                        end of the list
                                                    bins[binindex].lastblock->next = firstblock;
                                                    bins[binindex].lastblock = lastblock;
                                                }

                                                //unreserve last block and all new blocks
                                                __sync_fetch_and_add(&(bins[binindex].blocksfree)
                                                    ,blocksadded+1);
                                        }
                                }
                            }
                        }
                }
            }
            if (freerefreshneeded || (!mallocrefreshneeded)){
                //process free queue

                while ((void*)freelist_head != (void*)freelist_tail){

                    if (emptybin != -1) immediate_malloc(emptybin);
                    if ((mallocrefreshneeded)&&(!freerefreshneeded)){
                        //stop processing free queue on bin miss
                        break;
                    }

                    if (freelist_initialized){
                        if ((void*)(*freelist_head) != NULL){
                            //remove from free queue
                            freeaddress = ((void*)freelist_head - BIN_SIZE_BYTES);
                            freelist_head = (void*)(*freelist_head);
                            //free the block
                            freep(freeaddress);
                        }
                    }
                }

                //free queue processing complete
                if ((void*)freelist_head == (void*)freelist_tail) lastfreerefresh = 0;
                if (freerefreshneeded) freerefreshneeded = 0;
            }
```

```
        } //end while

        return NULL;
}

void create_mem_manager_thread(){
    int binindex;
    char *error;

    //mutex to ensure two threads aren't started
    pthread_mutex_lock(&initlock);

    if (threadid != getpid()){
        threadid = 0;
        initializing = 1;

        //get address of system malloc function
        if (mallocp == NULL){
        error = dlerror();
            mallocp = dlsym(RTLD_NEXT, "malloc");
            if ((error = dlerror()) != NULL){
                fputs(error, stderr);
                exit(1);
            }
        }

        // get address of system free function
        if (freep == NULL){
            freep = dlsym(RTLD_NEXT, "free");
            if ((error = dlerror()) != NULL){
                fputs(error, stderr);
                exit(1);
            }
        }

        //initialize variables
        pthread_mutex_init(&datalock, NULL);
        for (binindex=0; binindex<NUM_BINS; binindex++){
            bins[binindex].binsize = 0;
            bins[binindex].blocksinuse = 0;
            bins[binindex].blocksfree = 0;
            bins[binindex].lastrefill = 0;
            bins[binindex].lastused = 0;
            bins[binindex].locked = 0;
            bins[binindex].firstblock = NULL;
            bins[binindex].lastblock = NULL;
        }
        freelist_head = NULL;
        freelist_tail = NULL;
        freelist_initialized = 0;

        //create thread
        pthread_create(&managerthread, NULL, mem_manager_thread, (void*)NULL);
        pthread_detach(managerthread);

        //schedule new manager thread creation on fork
        pthread_atfork(NULL,NULL,fork_cleanup);

        while (threadid == 0){}  //wait for manager thread to get initialized

        //set manager thread id
        threadid = getpid();
        initializing = 0;
    }

    //release initialization lock
    pthread_mutex_unlock(&initlock);
```

```
}

void fork_cleanup(){
    //on fork, create new manager thread
    create_mem_manager_thread();
}

//create one block for a specific bin
void immediate_malloc(){
    block *newentry;

    //emptybin is bin index
    if (emptybin != -1){
        if (bins[emptybin].firstblock == NULL){

                //allocate new block
                newentry = (block*)(mallocp((1<<(emptybin+MIN_BIN_SIZE_SHIFT))+BIN_SIZE_BYTES));

                if (newentry != NULL){
                    newentry->size = emptybin;
                    newentry->offset = 0;

                    //add to front of bin
                    do {
                            newentry->next = bins[emptybin].firstblock;
                    } while (__sync_bool_compare_and_swap(&(bins[emptybin].firstblock),newentry->
                        next,newentry) == 0);

                    //if we are adding the first block, set the last block pointer
                    if (newentry->next == NULL) bins[emptybin].lastblock = newentry;

                    //increment free block count
                    __sync_fetch_and_add(&(bins[emptybin].blocksfree),1);

                    //reset empty bin index
                    emptybin = -1;
                }
        }
    }
}

//debug messages on ctrl+c
void ex_program(int sig) {
 int i;
 printf("pid: %d\n",getpid());
 if (pthread_mutex_trylock(&initlock) == EBUSY){
        printf("init lock busy\n");
 }
 for (i=0; i<NUM_BINS; i++){
        printf("bin %d: size %d free %d\n",i,bins[i].binsize,bins[i].blocksfree);
 }
 printf("\nlarge mallocs: %d\n",largemallocs);
 printf("malloc misses: %d free misses %d\n",mallocmisses, freemisses);
 printf("Caught signal: %d ... !!\n", sig);
 (void) signal(SIGINT, SIG_DFL);
}
```

Appendix F

Atomic Operation Latency Benchmark Code

```
// atomic latency benchmark
// Ryan Miller
// University of Cincinnati
// April 2010

#include<iostream>
#include<sys/time.h>
#include<cmath>
using namespace::std;

int counter=0;
const unsigned int MAX_COUNT=100000000;
const unsigned int NUM_TRIALS=5;

double difference[NUM_TRIALS];
struct timeval start_tv; // has tv_sec and tv_usec elements (from sys/time.h)
struct timezone start_tz; // needed for gettimeofday() call
struct timeval stop_tv;
struct timezone stop_tz;

inline double timeDifference(timeval& start, timeval& stop){
  double d1, d2;
  d1 = (double)start.tv_sec + 1e-6*((double)start.tv_usec);
  d2 = (double)stop.tv_sec + 1e-6*((double)stop.tv_usec);
  // return result in seconds
  return (d2-d1);
}

//Functions to be tested
inline void *BaseCost() {  }
inline void *PlusEquals() { counter += 1; }
inline void *PlusOne() { counter = counter + 1; }
inline void *IncrementOne() { counter = counter + 1; }
inline void *AtomicFetchAdd1() { __sync_fetch_and_add(&counter, 1); }
inline void *AtomicAddFetch1() { __sync_add_and_fetch(&counter, 1); }
inline void *AtomicSubFetch1() { __sync_sub_and_fetch(&counter, 1); }
inline void *AtomicCAS() { __sync_bool_compare_and_swap(&counter,counter,counter+1);}
```

```
//Base Test Function
double TestFunction(void *function() , int c)
{
  for (unsigned int trial=0; trial<NUM_TRIALS; trial++)
  {
    counter = c;
    gettimeofday(&start_tv, &start_tz);
    for (unsigned int i=0; i<MAX_COUNT; i++)
    {
      //Test function
      function();
    }
    gettimeofday(&stop_tv, &stop_tz);
    difference[trial] = timeDifference(start_tv,stop_tv);
  }
  //Statistics Calculations
  double totalTime=0;
  for (unsigned int trial=0; trial<NUM_TRIALS; trial++)
  {
    totalTime += difference[trial];
  }
  double mean=totalTime/NUM_TRIALS/MAX_COUNT;
}


int main()
{
  double baseCost = TestFunction(BaseCost, 0);
  cout << "Base Cost of for loop\t\t\t\t\t";
  cout << baseCost << " s" << endl;

  double plus1Time = (TestFunction(PlusOne, 0)-baseCost);
  cout << "Cost of (counter=counter + 1) - baseCost \t\t";
  cout << plus1Time << " s" << endl;

  double atomicTime = (TestFunction(AtomicFetchAdd1, 0)-baseCost);
  cout << "Cost of __sync_fetch_and_add(&counter,1) - baseCost\t";
  cout << atomicTime << " s" << endl;

  cout << "Cost: " << atomicTime / plus1Time << endl;

  return 0;
}
```

Bibliography

[1] R. Kalla, “Power7: Ibm’s next generation power microprocessor,” in *Hot Chips 21*, Aug. 2009.

[2] Intel Press Release, Intel Corporation, “Futuristic intel chip could reshape how computers are built, consumers interact with their pcs and personal devices,” tech. rep., Intel Press Release, Intel Corporation, Dec. 2009.

[3] Intel Press Release, Intel Corporation, “Intel research advanced ’era of tera’,” tech. rep., Intel Press Release, Intel Corporation, Feb. 2007.

[4] P. Douglas Eadline, “Hpc madness: March is more cores month,” *Linux Magazine*, Mar. 2010.

[5] Standard Performance Evaluation Corporation, “Spec cpu 2006.” http://www.spec.org/cpu2006/, Aug. 2008 (last updated).

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth ed., 2007.

[7] D. Haggander and L. Lundberg, “Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor,” *Parallel Processing, International Conference on*, vol. 0, 1998.

[8] B. Calder, D. Grunwald, and B. Zorn, “Quantifying behavioral differences between c and c++ programs,” *Journal of Programming Languages*, 1995.

[9] D. Dice and A. Garthwaite, “Mostly lock-free malloc,” in *Proceedings of the 3rd International Symposium on Memory Management*, 2002.

[10] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *Proceedings of the 5th international symposium on Memory Management* (E. Petrank and J. E. B. Moss, eds.), ACM, 2006.

[11] D. R. Butenhof, *Programming with POSIX Threads*. Professional Computing Series, Addison-Wesley, 1997.

[12] M. M. Michael, "Scalable lock-free dynamic memory allocation," *Conference on Programming Language Design and Implementation*, vol. 39, no. 6, 2004.

[13] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg, "Mcrt-malloc: a scalable transactional memory allocator," in *Proceedings of the 5th International Symposium on Memory Management* (E. Petrank and J. E. B. Moss, eds.), ACM, 2006.

[14] Doug Lee, "A memory allocator." http://g.oswego.edu/dl/html/malloc.html, Apr. 2000.

[15] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[16] S. Kahan and P. Konecny, "Mama!: a memory allocator for multithreaded architectures," *PPOPP*, 2006.

[17] J. Evans, "A scalable concurrent malloc(3) implementation for freebsd." http://people.freebsd.org/ jasone/jemalloc/bsdcan2006/jemalloc.pdf, 2006.

[18] S. Ghemawat and P. Menage, "Tcmalloc : Thread-caching malloc." http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2010.

[19] J.-M. Gurney, "mmalloc." http://resnet.uoregon.edu/ gurney_j/jmpc/mmalloc.html, 2002.

[20] S. Microsystems, "mtmalloc: Mt hot memory allocator," 2005.

[21] R. Benson, "Identifying memory management bugs within applications using the libumem library," *Sun Developer Network*, June 2003.

[22] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin, "Mmt: Exploiting fine-grained parallelism in dynamic memory management," in *IEEE International Parallel & Distributed Processing Symposium*, Apr. 2010.

[23] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin, "Memory management thread for heap intensive sequential applications," in *MEDEA Workshop PACT*, 2009.

[24] H. Baker and C. Hewitt, "The incremental garbage collection of processes," *Proceedings of the Symposium on Artifical Intelligence and Programming Languages, SIGPLAN Notices*, vol. 12, pp. 55–59, Aug. 1977.

[25] Valgrind Developers, "Valgrind." http://www.valgrind.org/, Jan. 2010.

[26] Oprofile Developers, "Oprofile." http://oprofile.sourceforge.net/, Jan. 2010.

[27] M. Behm, "Using valgrind to detect and prevent application memory problems," *Redhat Magazine*, vol. 15, 2006.

[28] Standard Performance Evaluation Corporation, "465.tonto spec cpu2006 benchmark description." http://www.spec.org/auto/cpu2006/Docs/465.tonto.html, Mar. 2006 (last updated).

[29] University of Western Australia Theoretical Chemistry, "tonto." http://www.theochem.uwa.edu.au/tonto/, Dec. 2005 (last updated).

[30] Standard Performance Evaluation Corporation, "471.omnetpp spec cpu2006 benchmark description." http://www.spec.org/auto/cpu2006/Docs/471.omnetpp.html, June 2006 (last updated).

[31] OMNeT++ Community, "Omnet++." http://www.omnetpp.org/, Apr. 2010 (last updated).

[32] Standard Performance Evaluation Corporation, "403.gcc spec cpu2006 benchmark description." http://www.spec.org/cpu2006/Docs/403.gcc.html, June 2006 (last updated).

[33] GCC team, "Gcc, the gnu compiler collection." http://gcc.gnu.org/, Apr. 2010 (last updated).

[34] E. D. Berger, K. S. McKinley, and B. G. Zorn, "Reconsidering custom memory allocation," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2002.

[35] mubench project, "mubench." http://mubench.sourceforge.net/, Apr. 2010.

[36] Intel Corporation, "Intel 64 and ia-32 architectures software developers manual - volume 3a: System programming guide, part 1," tech. rep., Intel Corporation, Mar. 2009.

[37] D. Kanter, "Inside nehalem: Intel's future processor and system," *Real World Technologies*, Apr. 2008.

[38] Savant Developers, "Savant." http://www.cliftonlabs.com/vhdl/savant.html, Jan. 2010.

[39] J. Davis, "Sun intros eight-core processor," *Electronic News*, Nov. 2005.

[40] Sun Microsystems, "Opensparc t1." http://www.opensparc.net/opensparc-t1/index.html, Jan. 2010.

[41] Xilinx, "Xilinx university program." http://www.xilinx.com/univ/, Jan. 2010.

[42] M. Bruning, "Comparison of solaris os and linux for application developers," tech. rep., Sun Developer Network, Sun Corporation, June 2006.

[43] J. Attardi and N. Nadgir, "A comparison of memory allocators in multiprocessors," tech. rep., Sun Developer Network, Sun Corporation, June 2003.

[44] 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*, Sept. 1995.

[45] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.