CHARACTERIZING SHARED MEMORY MULTIPROCESSOR BENCHMARKS FOR FUTURE CHIP MULTIPROCESSOR ARCHITECTURES USING INSTRUCTION FLOW ANALYSIS

An Honors Fellows Thesis

by

PHILIP JAMES JAGIELSKI

Submitted to the Honors Programs Office
Texas A&M University
in partial fulfillment of the requirements for the designation as

HONORS UNDERGRADUATE RESEARCH FELLOW

April 2011

Major: Computer Engineering

CHARACTERIZING SHARED MEMORY MULTIPROCESSOR BENCHMARKS FOR FUTURE CHIP MULTIPROCESSOR ARCHITECTURES USING INSTRUCTION FLOW ANALYSIS

An Honors Fellows Thesis

by

PHILIP JAMES JAGIELSKI

Submitted to the Honors Programs Office
Texas A&M University
in partial fulfillment of the requirements for the designation as

HONORS UNDERGRADUATE RESEARCH FELLOW

Approved by:

Research Advisor: Paul V. Gratz
Director for Honors and Undergraduate Research: Sumana Datta

April 2011

Major: Computer Engineering

ABSTRACT

Characterizing Shared Memory Multiprocessor Benchmarks for Future Chip Multiprocessor Architectures Using Instruction Flow Analysis. (April 2011)

Philip James Jagielski
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Paul V. Gratz
Department of Electrical and Computer Engineering

For forty years, transistor counts on integrated circuits have doubled roughly every two years, enabling computer architects to double the clock speed of processors. Recently, heat dissipation and power consumption trends have forced chip designers to add larger caches and more cores per chip, instead of increasing clock speed with the extra transistors. This has provided challenges for programmers who wish to continue increasing application performance as though the speed of a uniprocessor had continued doubling. In this characteristic study, we examine the effect of the operating system on a set of parallel benchmarks run on a simulated many-core processor. Past research has shown that the performance of the OS code has a large impact on application performance; however, most studies ignore the OS and focus on the application code. This work will characterize performance bottlenecks and show possible areas that could be improved. We found that resource contention in the kernel was limiting the efficiency of the benchmarks.

DEDICATION

To Baby Zane

ACKNOWLEDGMENTS

First and foremost, I want to thank my advisor, Dr. Gratz, for his patience and insight. He has given me the most non-renewable resource there is, his time. I am forever grateful to him for giving me the opportunity to learn so much about computer architecture, and research in general.

I would also like to thank Ehsan Fatehi for introducing me to M5. His experience getting PARSEC to cross compile to the Alpha architecture was invaluable during my similar struggle with SPLASH-2. Anusha Shankar also gave me advice on figuring out how to correllate M5 data with Linux kernel code.

I am also grateful to my brother, who helped me normalize the data for some of the plots so he could learn Python.

Finally, I would like to thank my wife, Jennifer, for understanding when we could not spend much time together during our last weeks together before our son was born.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION, BACKGROUND AND RELATED WORK

This chapter first examines the historical context of microprocessor research and improvements that led to the current state of the art. It goes on to analyze both software and hardware innovations that have allowed the rapid increase in microprocessor performance over the past half century.

Background

Almost forty five years ago, Gordon Moore postulated that transistor densities in integrated circuits would double every two years [1]. Since then, Moore's Law has held remarkably true.

For forty years, doubling the transistor count of a processor enabled computer engineers to roughly double it's clock speed. However, in the past five years, heat dissipation and power consumption have caused the clock frequency trend to level off. Moore's Law continues, but increasing transistor numbers now translate into increasing the number of cores per chip as well as the size of the on-chip caches.

Unfortunately, adding cores and cache memory does not linearly scale computing power, even in a perfectly parallelizable problem domain. Accesses to the same memory addresses from different processes must be coordinated so they don't conflict. Not coordinating memory accesses would result in inconsistent data and nondeterministic behavior.

The journal model is *IEEE Computer Architecture Letters.*

The problem of efficiently adding cores that can use a shared memory space is a current research problem. It is being attacked from many different angles, both on the hardware and software side. Most current approaches to improving performance ignore the performance of the operating system on the system as a whole. In this thesis I investigate the effect of the operating system on a suite of benchmarks and present methods to increase the efficiency of multithreaded programs. Figure 1 shows the primary motivation for this research: the poor scaling of the SPLASH-2 FFT benchmark as the number of cores doubles from 1 to 32.

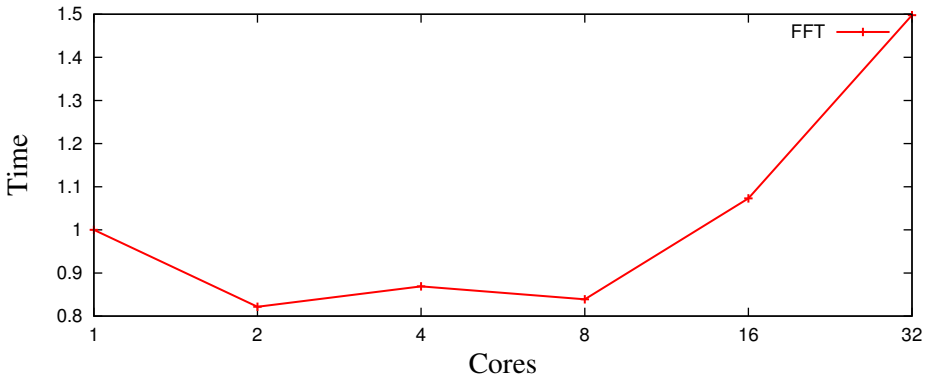


Fig. 1.: FFT execution time

Literature survey

The array of processor technologies available in the early years was much more heterogeneous than today's market. This led to an equally varying array of processor architectures. Today's market is dominated by two philosophies, x86 and ARM. But in the 1960's different target markets led to different architectures. Lines of mainframes were considered wildly successful if they sold a hundred units, so there were so many different companies that no unified standard existed. For example, today you can count on either 32-bit or 64-bit technology, but in the sixties 48-bit and 60-bit architectures also existed [2]. Different

process technologies also allowed different properties in the architectures that used them. The CRAY-1 used different gate level technology (ECL) and as a result it's clock speed was not exceeded by standard CMOS circuits for two decades [3]. Today the feature size of transistors can be measured in the number of atoms, and processors contain over a billion transistors on a single chip. These facts, plus the economics of industry consolidation, mean that only a few architectures survive today. A desktop computer is likely to run Intel's x86 architecture, and a mobile phone or tablet is likely to use the ARM architecture. However, advances in the field can be generalized to apply to these competing lines of processors.

Hardware improvements

Smaller transistor size translates into a higher clock speed because the transistors can switch on and off faster. This makes the processor linearly faster because it can do more work in the same time. However, that is the domain of physicists and materials engineers, and thus beyond the scope of this thesis. Given a fixed process technology to work with, computer architects employ two techniques to increase the throughput of the processor. The first is harvesting parallelism from a sequential instruction stream, or instruction level parallelism (ILP). The second harvests parallelism from the applications running on the CPU by adding more cores to the CPU and allowing it to process multiple instruction streams simultaneously.

Instruction level parallelism

Besides hiding latency to cache, RAM, and disk, the pipeline depth of a processor affects how much parallelism can be extracted from the instructions. Multiple issue processors

examine the pipeline beyond the next element, and can issue multiple instructions to be executed on the same cycle. This requires multiple copies of internal components such as the ALU. There are two ways to implement multiple issue:

- Static Multiple Issue - the compiler packages instructions in such a way that the processor knows which instructions it can safely execute at the same time.
- Dynamic Multiple Issue - the processor, called a superscalar, decides which instructions can be issued for each clock cycle.

Speculation can increase the parallelism of either approach. Speculation guesses about properties of instructions or branches and issues instructions based on those guesses. If the guess turns out to be wrong it must roll back the issued instruction as if it had never executed [4]. Therefore, too much speculation can actually hinder performance.

Dataflow computing is an extreme form of ILP where the entire program is represented as a dependency graph of instructions instead of a sequential, ordered stream. This is very different from a standard sequential list of instructions, and programs written this way require a dataflow architecture to run such as the MIT Tagged-Token Dataflow Architecture. The MIT researchers designed a language, Id, that was built from the ground up to use a dataflow representation of code. Their compiler compiles Id into a tagged-token graph, where each token represents the dependencies of a particular instruction [5].

Although dataflow computing is a fruitful research topic, it has not appeared in mainstream processor architectures. One reason is that debugging is harder on a dataflow architecture

because there are no ordering constraints. Also, serial programming languages and instruction set architectures are thoroughly embedded in modern industry [6]. Despite these setbacks, dataflow research serves as an inspiration for superscalar processors today.

Multithreaded hardware

The two broad categories of parallel machines that exist today are symmetric multiprocessing (SMP), and message passing machines. SMP computers have multiple processors that access a single shared memory space. Message passing involves clusters of machines that have their own memory systems, but communicate to share data.

The fundamental difficulty in parallel computing is keeping a coherent and safe version of data at all times. Safe parallelism in SMP systems is achieved in software by using atomic instructions to lock or unlock resources or sections of code. Message passing machines divide the memory into private spaces for every CPU so that concurrent data modification can not happen.

Message Passing systems consist of nodes running independent programs or operating systems. Since each program is on a uniprocessor system, no cache coherence must be performed, but data can only be passed between programs via explicit send() and receive() calls [7]. Two problems prevented message passing machines from proliferating. First, the overhead of sending and receiving messages is usually large ($>$1ms when the technology was developed). Also, the message passing programming model severely limited what types of applications could run on the machine [3].

However, computing clusters similar to message passing machines thrive in the modern computing landscape, just in a different form. Large Internet companies, cloud computing

platforms, and many other types of websites use clusters of computers to render web pages and serve database results. Arguably, every computer connected to the Internet could be considered a large message passing machine.

In SMP systems, each CPU has a local cache of the data in RAM, and a modification of the data in single cache must be communicated to all caches in the system to prevent another cache from serving stale data. To keep the memory system safe from concurrent accesses by different processors, a cache coherence protocol must be implemented in hardware. Early cache coherence protocols were based on snooping bus traffic to determine when modifications happened [8]. Snooping does not scale well because it depends on connecting every cache to every other cache in the memory system (an $O(n)$ increase in cache connections).

The Stanford Dash Multiprocessor [9] was the first attempt to build a scalable shared memory machine out of commodity processors. Their research showed it was possible to achieve near linear performance up to 64 processors while using cheap individual processors and existing programming languages. The authors implemented a distributed directory based cache coherence scheme. In this scheme special nodes called directories hold the cache state information so that each cache line request does not need to broadcast to all 64 caches.

As more processors or nodes are added to a parallel system, the memory latencies rise to handle synchronization [10]. Hardware multithreading allows software threads to share the functional units of a processor core when a thread blocks for a memory access. Fine-grained multithreading changes threads every instruction if necessary, while coarse-grained multithreading switches only when a thread performs a blocking operation. The first allows maximum instruction throughput, and the latter is optimized for the time to execute a single

thread [3].

As all applications have sections that must be computed serially, single-thread performance can outweigh increased parallelism [11]. Also, switching threads rapidly can actually impede performance by reducing memory temporality. If the different threads use data from disparate addresses, the cache churn will increase the memory latencies that multithreading tries to avoid. Multithreading is compatible with multiple issue processors, and the two concepts are often combined.

SMP systems have historically been the domain of scientific and industrial computing, but they have steadily taken over the consumer space over the last five years. This has happened because, as is explained in the introduction, all the transistors that Moore's law provides can not be used to linearly scale a single processor's performance.

Software improvements

One way to improve the speed of software is to improve the quality of the algorithm. However, this is beyond the scope of this thesis. Instead I focus on the interaction between the algorithm and the operating system. Computer architects often use the average instructions that a processor can execute per cycle to calculate its efficiency. Most research papers discount the effects of the operating system when performing experiments. Architectural decisions play a large role in the efficiency of an OS, for example, system calls and interrupts can interfere with useful data in the cache because they jump to memory locations not used frequently. This causes system code to have 50-85% higher cycles per instruction (CPI) than user level code [12].

OS parameters such as page mapping can also affect the cache miss rates and TLB usage,

which directly affects the speed of the system. Lo et al. found that improving the operating system mapping policy could reduce the cache miss rates for a symmetric multithreaded superscalar processor to the same level as a superscalar uniprocessor [13]. Virtual-to-physical page mapping and the per-process offset affected how the critical set fit into the cache and how badly interference affected the cache.

Since SMT processors share scarce resources among threads, the operating system can choose to run threads that share the available CPU resources efficiently as opposed to threads that compete inefficiently for the same hardware resources. Thread sensitive scheduling can potentially boost system performance by 15% [14].

CHAPTER II

METHODOLOGY

Cycle level simulation of an entire processor architecture is the primary method to determine the efficacy of modern processor designs. One can simulate how a program runs on similar processor architectures in a design space to determine the optimum configuration of processor elements and/or cache levels [15]. In this study we used the open source M5 Simulator [16] to gather instruction stream from executing a subset of the SPLASH-2 benchmarks. SPLASH-2 is a suite of twelve parallel benchmarks specifically designed for the academic study of multiprocessor architectures [17].

The M5 simulator

M5 is a discrete-event, general-purpose architectural simulator released under an open source license. Although it is highly extensible, in this thesis we used the basic CPU and memory models built into M5. They key reasons why M5 was chosen are its free license and its ability to run in full system mode. Full system mode boots a simulated Linux kernel, instead of just emulating syscalls. This allows us to include the effect of the operating system in our performance analysis.

We used revision `7026` of M5 from the mercurial repository. Few deviations were made from M5's default full system settings. Since we were investigated future systems that have not been made yet, no effort was made to use novel architecture techniques; the `AtomicSimpleCPU` model is realistic enough to capture general performance trends. The other system options were based on an average computer, with a 2GHz CPU frequency, two levels of cache, and M5's default idealized snoop cache coherence algorithm.

The SPLASH-2 benchmarks

The SPLASH-2 benchmark suite was released in 1992 and patched in 1995 as a tool to study shared address-space multiprocessor systems. It consists of 4 kernels, scientific and engineering parallel computations, and 8 applications, complete programs more realistic than the kernels. Because SPLASH-2 uses homogeneous parallelization, we can use simpler timing models when simulating its execution [18]. The SPLASH-2 benchmarks focus more on pure computation than processing large amounts of data like more modern benchmark suites such as PARSEC [19].

We examine three of the kernels in this thesis, FFT, Radix, and LU. The FFT benchmark performs a complex 1-D FFT transform. It uses the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication [20]. Communication happens three times during matrix transpose steps, yielding low overall throughput requirements, but a very bursty communication profile.

The Radix benchmark performs an iterative integer radix sort. Each iteration, every processor calculates a local histogram with its assigned keys, then communicates to obtain a global histogram each iteration. It then uses the global histogram to permute its keys; this step requires all-to-all communication [21].

The LU benchmark factors a dense matrix into the product of an upper and lower triangular matrix. The matrix is divided into blocks, which are assigned to processors using 2-D scatter decomposition to reduce communication. Each block is allocated locally on the processor that owns it.

How to compile SPLASH-2 for full system simulation in M5

The first step in investigating the performance of the benchmarks is getting the programs to compile. Since the SPLASH-2 Suite was released in 1995 it does not compile in modern compilers in its released form. We used the Modified SPLASH-2 patches [22] to modernize the code and hand modified the code in several benchmarks to get the suite to compile.

Another constraint is the Instruction Set Architectures (ISAs) that M5 supports. To get the most realistic data from the simulator possible it needs to be run in full system mode where the simulator actually boots a copy of Linux and runs the piece of code being investigated inside Linux. Currently the Alpha ISA is the only ISA supported by M5's full system mode, so one must cross-compile the benchmarks to run on M5.

To build SPLASH-2 for M5, first download M5 and it's necessary files and build it. Most of the default full system files don't work well with SPLASH-2; download the Parsec in M5 [23] files to replace them. Note that `tsb_osfpal` from the obsolete full system files is required to run more than 8 processors.

```
hg clone http://repo.m5sim.org/m5-stable
scons build/ALPHA_FS/m5.opt
wget http://www.m5sim.org/dist/current/m5_system_2.0b3.tar.bz2
wget http://www.cs.utexas.edu/~parsec_m5/vmlinux_2.6.27-gcc_4.3.4
wget http://www.cs.utexas.edu/~parsec_m5/linux-parsec-2-1-m5.img.bz2
wget http://www.m5sim.org/dist/current/tsb_osfpal
```

Keep the directory structure of the system files from m5sim.org, but move the parsec disk image in to the `disks` directory and modify `./configs/common/Benchmarks.py` line `53` to return `env.get('LINUX_IMAGE', disk('linux-parsec-2-1-m5.img'))` instead of `linux-latest.img`. Do the same to line 67 of `./configs/common/FSConfig.py`. Move

vmlinux_2.6.27-gcc_4.3.4 into the binaries directory, but change its name to just vmlinux.

To allow support for more than four processors move tsb_osfpal into the binaries directory and modify ./configs/common/FSConfig.py by changing the line self.pal = binary('ts_osfpal') from ts_osfpal to tsb_osfpal.

The console file is the only original file that you will use from the m5 system files provided by m5sim.org. It may be necessary to chmod +x the other files in binaries. To tell M5 where the disk images are, export M5_PATH as an environment variable or edit ./configs/common/SysPaths.py line 53. Run the regression tests to ensure that all variables are correctly set.

SPLASH-2 is no longer maintained, so one must download the source code from an internet archival resource. Note that when the file is untarred it may say unexpected end of file, but the files are still usable. Because of their age, the SPLASH-2 benchmarks will not compile with modern compilers. The Modified SPLASH-2 patches [22] attempt to modernize the code.

```
wget http://web.archive.org/web/20080528165352/http://www-flash.stanfo
rd.edu/apps/SPLASH/splash2.tar.gz
tar -xzvf splash2.tar.gz
cd splash2
wget http://www.capsl.udel.edu/splash/splash2-modified.patch.gz
gzip -d splash2-modified.patch.gz
patch -p1 < splash2-modified.patch
```

In codes/Makefile.export change the BASEDIR, and on line 9 change the macros to c.m4.null.POSIX to support parallelism. The following commands will compile the FFT benchmark and test it.

```
cd kernels/fft
make
./FFT -t
```

To cross-compile the benchmarks to run in M5 first download the Alpha cross-compiler.

```
wget www.m5sim.org/dist/current/alphaev67-unknown-linux-gnu.tar.bz2
tar -xjvf alphaev67-unknown-linux-gnu.tar.bz2
```

Next, change `Makefile.export` to use the crosscompiler by modifying `CC`, `CFLAGS`, and `LDFLAGS`. Also force `make` to compile statically since the disk image does not have the correct library versions for dynamic linking.

```
CC := /path/to/alphaev67-unknown-linux-gnu/bin/alphaev67-unknown-linux-
gnu-gcc
CFLAGS := $(CFLAGS) -I/path/to/alphaev67-unknown-linux-gnu/alphaev67-un
known-linux-gnu/sys-root/usr/include
LDFLAGS := $(LDFLAGS) -L/path/to/alphaev67-unknown-linux-gnu/alphaev67-
unknown-linux-gnu/lib/
```

After compiling FFT for the Alpha ISA, the executable can be loaded onto the disk image and run inside M5. To copy the executable onto the disk image run the following:

```
mount -o loop,offset=32256 linux-parsec-2-1-m5.img /mnt/m5_disk
mkdir -p /mnt/m5_disk/benchmarks/
cp FFT /mnt/m5_disk/benchmarks/
umount /mnt/m5_disk/
```

Add the fft benchmark to `configs/common/Benchmarks.py` by adding the line `'fft': [ SysConfig('fft.rcS', '512MB')],` to the `Benchmarks` data structure. Then make the rcS script which will run the benchmark inside M5 in `./configs/boot` with this text:

```
#!/bin/sh
cd benchmarks
echo "Running FFT now..."
./FFT -t -p1
#Gracefully exit M5
/sbin/m5 exit
```

All the SPLASH-2 kernels compile without modifications. To compile the applications refer to the following.

- **raytrace** - runs without modifications, but the program to verify output does not work.
- **ocean** - runs without modification and the output matches correct.out within rounding tolerances.
- **radiosity** - runs without modification, but there are significant differences between correct.out and M5's output.
- **water-*** - runs without modification, but no known correct output is given to verify the results.
- **fmm** - Rename the `_Complex` type and use `__DBL_DIG__` and `__DBL_MAX__` to compile. The output does not match correct.out.
- **barnes** - runs without modification, but no known correct output is given to verify the results.
- **raytrace** - Use `libtiff` from volrend to compile `rltotiff`. Comment out the `BYTESWAP` macro in `tiff_rgba_io.c`.
- **volrend** - This application depends on libtiff, which needs several modifications to compile. First remove the `-ansi` flag in the `Makefile`. In `tiffcompat.h` comment out lines `173-187` and leave the `#include <malloc.h>` statement. To build the library for alpha, add `include ../../../Makefile.config`. However, the makefile actually builds a C program and runs it to generate some source code for libtiff. To

ensure that this program is compiled for the local machine and not Alpha, change the `${CC}` under the target `g3state.h` to `gcc` or another default compiler.

Data collection from M5

To obtain the dynamic instruction stream we modified the `traceInst` function in `src/exe/exetrace.cc` to output data about every instruction as it executes. The function printed the disassembled instruction and its operands, the program counter (PC), and the CPU the instruction executed on. Only instructions during the execution of the benchmark under test were printed; booting the Linux kernel was ignored. This was accomplished by running M5 twice. The first time a checkpoint was taken right before the benchmark was executed, and the second time the `--trace-flags=ExecNoTicks` and `--checkpoint-restore=1` flags were passed to M5 to print the instruction stream only while the benchmark was running.

All the plots presented in this document except the timing plot showw in the figure on page 18 were parsed from the instruction streams explained above. The data for the total execution time plot was taken directly from the statistics output by M5 after each simulation.

We used the address of each instruction to classify it into either PAL code, kernel code, the SPLASH-2 benchmark, or the launcher process that started the benchmark. The PAL code occupies addresses `0x4000` to `0x7000`. The benchmark instruction's PC starts at `0x2000000`, and the BASH launcher process starts at `0x200000000`. The kernel code occupies the high end of the address range, anything above `0xFFFF00000000000`. Similarly, counting the number of memory barriers, as shown in the figure on page 23, was done by a simple grep command searching for `mb` or `wmb` instructions.

Counting the number and types of different locks was done by matching the dynamic in-

struction stream with specific lock implementations found in the Linux kernel for the Alpha architecture. Locks in Alpha are implemented by load-locked/store-conditional pairs [24]. Load-locked loads the current value of a location in memory and sets a special flag. The next store-conditional on that address stores a new value only if no changes have been stored since the load-locked. Atomic operations such as locking and unlocking can be implemented using these instructions. For example, a raw_spin_lock that succeeds executes the following instructions in order: `ldl_l bne lda stl_c`. We implemented a finite-state machine in Python that kept track of the instruction stream for each CPU and incremented appropriate counters for every `ldl_l/stl_c` instruction pairing it found.

CHAPTER III

RESULTS AND ANALYSIS

In this chapter we present a detailed explanation of the data parsed from M5. In addition to plots, tables with the instruction counts are presented in Appendix A so the reader can precisely view the results.

Ideal multithreaded application performance

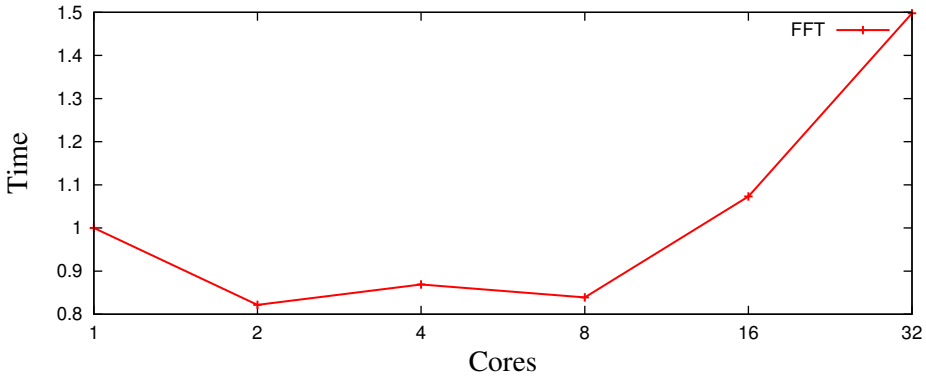
Amdahl's law is used to predict the maximum speedup of a program when only part of the program is sped up. In parallel computing, Amdahl's law is applied to define how much a program can be improved by adding more cores. For example, if a program consists of 50% sequential code, and 50% parallelizable code, the maximum speedup is 2x because the sequential 50% will always need to execute. Amdahl's law is often stated as:

$$\text{speedup} = \frac{1}{(1-P) + \frac{P}{N}}$$

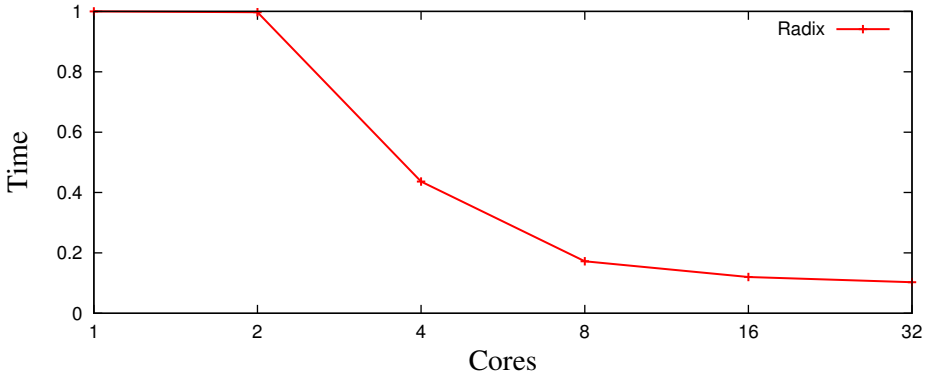
where P is the proportion of the program that can be made parallel, and N is the number of processors. Applications are said to be "embarrassingly parallel" if P is near 1 because it is trivial to parallelize the entire problem. The SPLASH-2 benchmarks are not embarrassingly parallel because the programs all have small sequential bottlenecks where communication is required. Amdahl's law neglects more complex factors such as memory bottlenecks, but it indicates that adding more cores should always offer some benefit, even if it is minor. If adding cores does not benefit, there is a problem that needs to be investigated. Figure 1 on page 2 of the introduction illustrates the problem we saw in the FFT benchmark.

SPLASH-2 performance

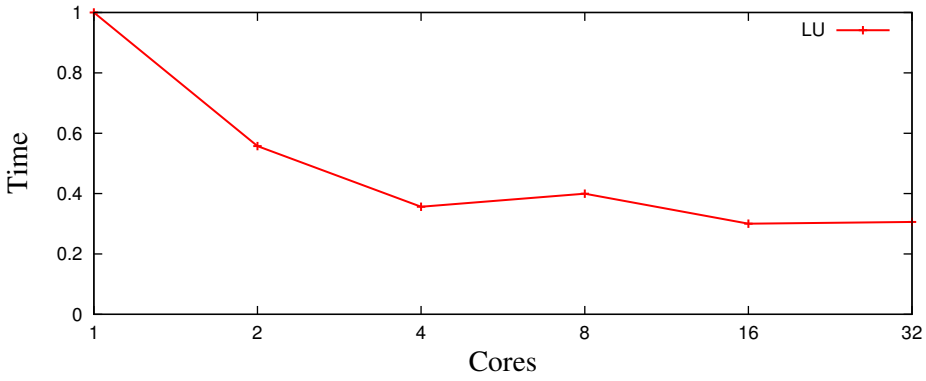
Figure 2 shows the execution time of three benchmarks, normalized against 1 core.



(a) FFT Execution Time



(b) Radix Execution Time



(c) LU Execution Time

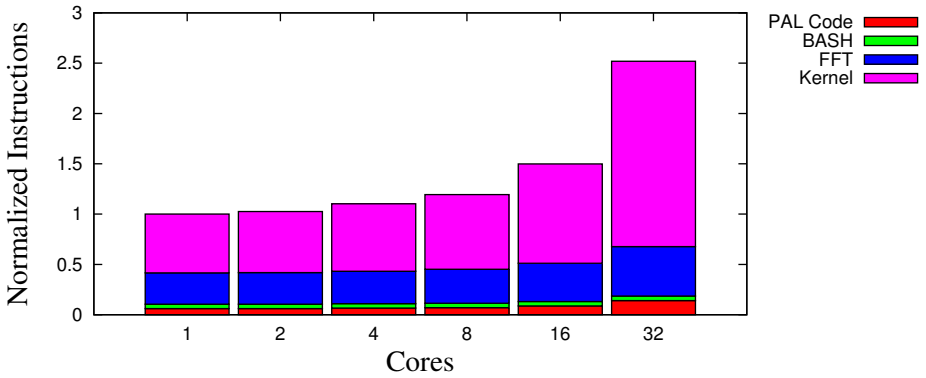
Fig. 2.: Normalized execution times of three SPLASH-2 benchmarks

FFT shows the worst performance since it actually takes more time to run the program with 16 and 32 cores than with a single core. Radix scales the best of the three benchmarks, disregarding the extremely modest gain from single to dual core configuration. At this time, the author does not know what causes such an inconsequential performance gain. The Radix execution time almost halves every time the number of cores are doubled, but after 8 cores the decrease is less than half.

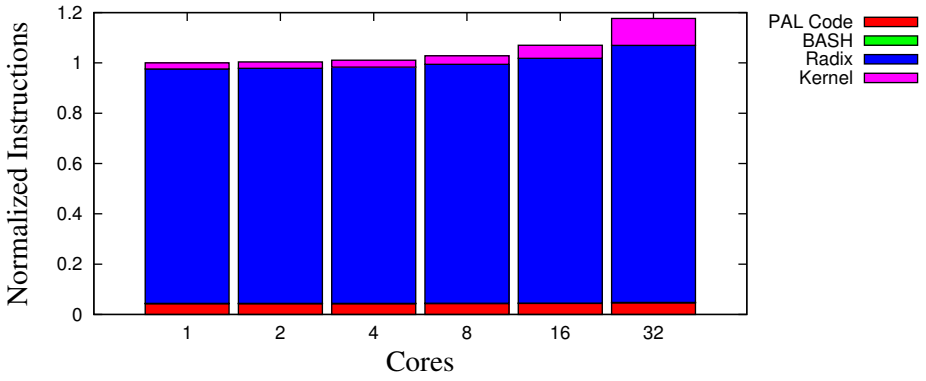
We see a strong decrease in execution time for the first 2 core doublings, but the performance flatlines above 4 cores. The execution time actually increases slightly at 8 cores and at 32. This indicates that the LU algorithm is sensitive to the architectural parameters of the system, and even possibly unstable.

The performance of all three benchmarks could be improved. In an ideal system the execution time of a program running on 32 cores would be 0.03125 times the time it takes the same program to run on a single processor. Radix comes closest to this goal with the 32 core program taking only 10% of the time of the single core program.

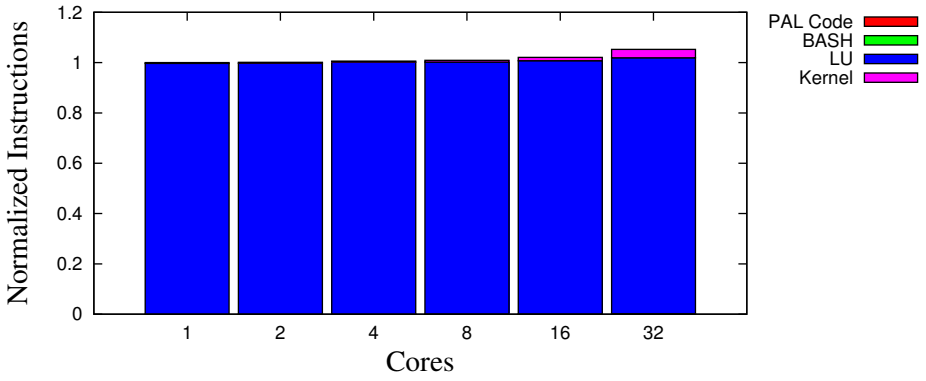
To study why the performance degrades so badly we counted the number of instructions and plotted their increase in Figure 3.



(a) FFT Instruction Growth



(b) Radix instruction growth



(c) LU instruction growth

Fig. 3.: Normalized instruction count, grouped by address, of three SPLASH-2 benchmarks

These plots show the number of instructions executed, divided in categories. We found four different types of instructions: PAL code, BASH, the benchmark itself, and kernel code. These are essentially 4 different processes that interact with each other. PAL code, or Privileged Architecture Library code, provides a hardware abstraction layer to the kernel. It provides cache management, interrupt and exception handling, as well as other firmware related tasks [25].

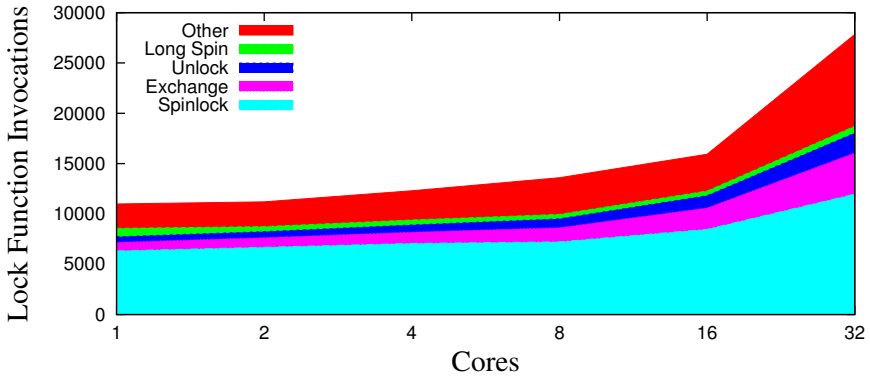
The BASH code is the launcher script that spawns the benchmark process. The benchmark code, labelled in the figures with the name of the benchmark, is the multithreaded SPLASH-2 benchmark being tested. Finally, the kernel code is the Alpha version of Linux kernel 2.6.

Ideally, the total number of instructions should have stayed roughly the same because the program is solving the exact same problem. If the number of instructions increases, it means the processor is doing unnecessary work that is not directly contributing to the solution. By comparing the growth rates of the categories of instructions in Figure 3 we can determine which process is the bottleneck.

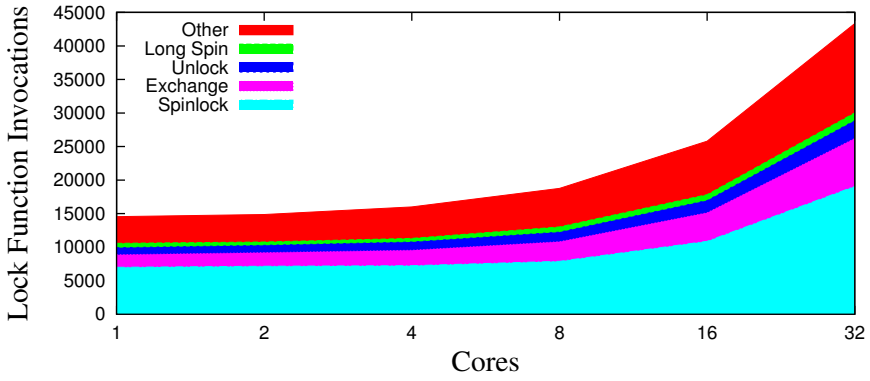
For FFT, the kernel code is clearly the largest increase. For Radix and LU the exponential increase is still there, it is just amortized better by the large instruction count of the actual benchmark. For all three benchmarks the application code increases slightly, while the kernel code increases exponentially as the number of cores doubles from 1 to 32. This means the program can be sped up if the processor spent less time executing instructions for the kernel. We hypothesized that contention for locks inside the kernel was driving the increase. To verify the hypothesis we analyzed the dynamic instruction stream to look for locks.

Analyzing the dynamic instruction stream

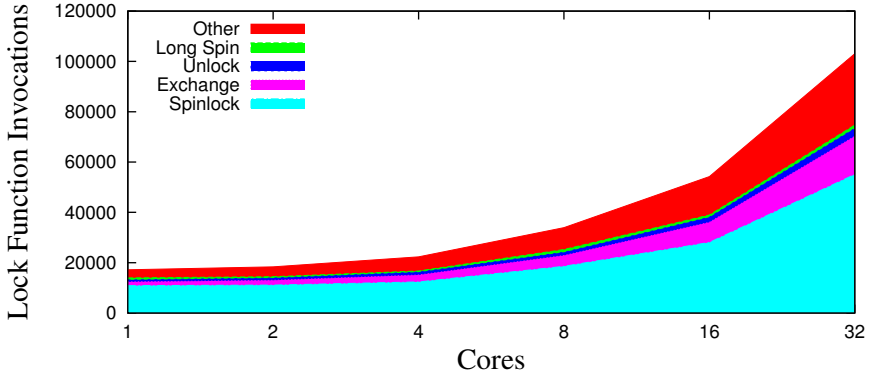
Figure 4 shows the different types of locks as the number of cores increases. This only measures locking functions inside the kernel that use the `ldl_l` and `stl_c` instructions.



(a) FFT lock growth



(b) Radix lock growth



(c) LU lock growth

Fig. 4.: Growth of different locking functions

Spinlocks dominate the types of locks we detected. The exchange category consists of functions that atomically swap a variable with a new value. The unlock function unlocks a lock that is not implemented using spinning. The long spin category is comprised of spin locks that spun for more than one loop. Functions that had only a few hundred invocations and unidentified assembly streams are assigned the other category.

One must note that Figure 4 is a measure only of load-lock/store-conditional pairs, not all synchronization functionality within the kernel. Locks are generally locked using those two instructions, and then freed using a memory barrier. For example, the `raw_spin_unlock` function is implemented in two instructions with the code `mb(); lock=0;` in the kernel. Figure 5 shows the increase in memory barrier instructions. We counted both regular barriers (`mb`) and write memory barriers (`wmb`) for this plot.

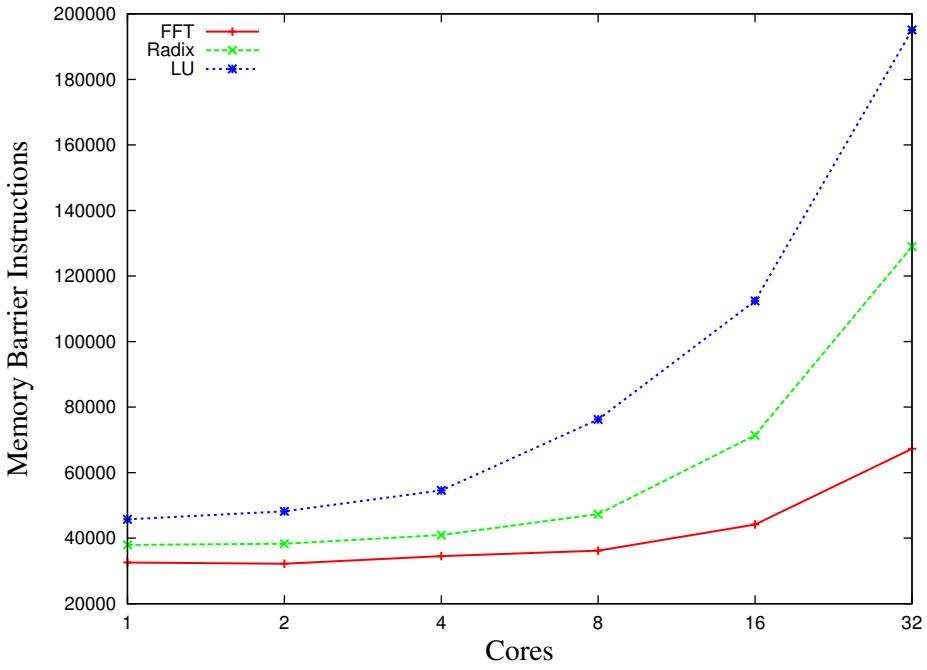


Fig. 5.: Memory barriers executed

Looking at Figures 4 and 5 one can see that the number of locks stays fairly flat until 4 or 8 cores, at which point it begins a near exponentially increasing pattern. This correlates with the good performance in Figure 2 up to the same point, confirming our hypothesis that resource contention in the kernel is reducing the performance gains of the applications when we add more cores.

It is also important to note that the ”Spinlock” category only represents spinlocks that succeeded on the first try. Locks that had many repeated instructions between the load-lock instruction and the store-conditional instruction are put into the ”Long Spin” category. Figure 6 shows total number of cycles these spinlocks had to spin before they were available, and Figure 7 shows the average number of cycles spent in each spinlock that looped more than once.

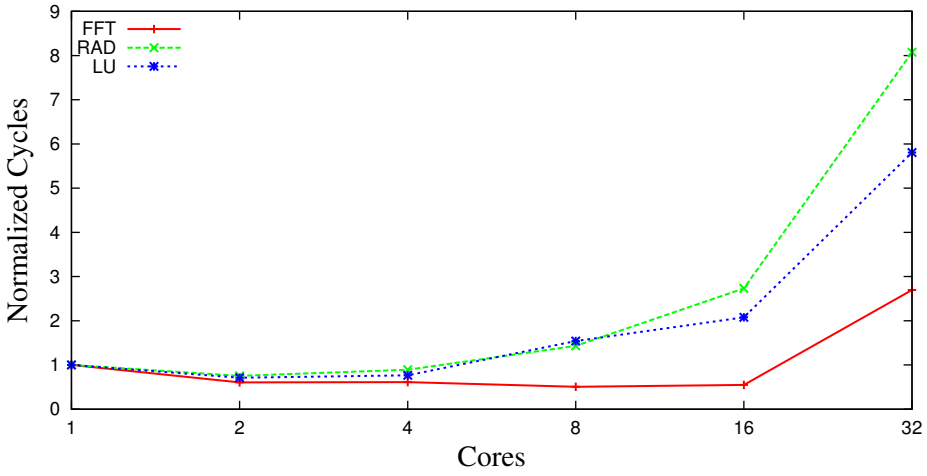


Fig. 6.: Normalized total cycles spent in spinlocks

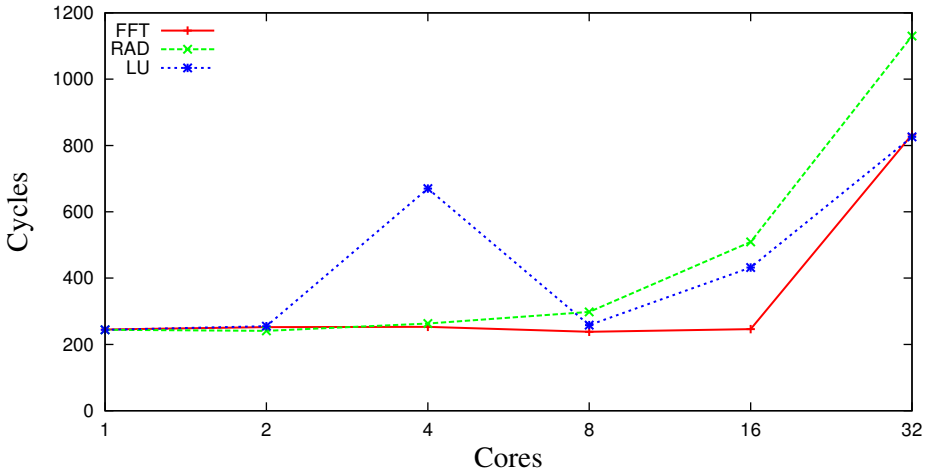


Fig. 7.: Average cycles spent per spinlock

The number of long spinlocks and the number of cycles spent in spinlocks both increase drastically as the number of cores reaches 32. This indicates that there is heavy contention for resources in the kernel.

Note the anomalous increase in cycles for the LU benchmark on 4 cores in Figure 7. This corresponds to the slight rising of execution time in Figure 2c, showing that more time spent in locking constructs corresponds to a higher execution time.

From Figures 6 and 7 it may seem that radix has the worst performance of the three benchmarks because it spends the longest time looping in spinlocks. However, if you consider the total number of locks executed, the Radix benchmark has less than LU and FFT per total instructions executed.

From these results, it is clear that locking and synchronization mechanisms in the kernel are hindering how well these benchmarks scale. To increase the performance of these

benchmarks with many core processors we must reduce the synchronization overhead in the kernel.

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

In this thesis we examined the reason why the FFT benchmark scales poorly in the M5 Simulator. By examining the dynamic instruction stream we found that the instruction count of the kernel code was increasing exponentially as the number of cores increased. Further analysis showed that both the number of locks and the average number of cycles spent inside a spinlock increased rapidly after the number of cores exceeded eight.

Conclusions

In conclusion, load balancing is a big problem facing modern shared-memory, chip multiprocessor designers. Resource contention causes too many lock and unlock calls, which slow down the code by preventing the actual application code from running. The first step to increasing the performance of these benchmarks is to utilize each core to its fullest.

Potential changes that could benefit include changing the granularity at which the code communicates. This could reduce the synchronization overhead because the code could run longer in between communicating. Since critical structures are a source of locks during communication, this would lead to less locks overall.

Another possible remedy is to use a different memory consistency model. Alpha has the weakest memory consistency model, so memory barriers are commonly used to ensure valid results. Using a stronger model like x86's model could lower the synchronization penalty, with the potential downside that the processor would have to work harder to enforce the memory model.

Future work

More work is necessary to understand exactly what causes the number of locks to explode as the core count increases. To understand where the locks are coming from, we could take into account the address of each instruction and compare it to disassembled binaries of the Linux kernel and the SPLASH-2 application. This would allow us to analyze the sources of the most expensive locks.

Higher fidelity simulations would also benefit this topic because they were yield more realistic results. Using the most realistic CPU model in M5, as well as a realistic cache coherence scheme for many cores, would give better results. This could lead to experiments tweaking architectural parameters to reduce the operating system overhead.

REFERENCES

[1] Gordon E. Moore, "Cramming more components onto integrated circuits," *in IEEE Solid-State Circuits Newsletter*, 1965.

[2] James E. Thornton, "Parallel operation in the control data 6600," in *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, 1965.

[3] Mark Donald Hill, Norman Paul Jouppi, and Gurindar Sohi, *Readings in Computer Architecture*, Gulf Professional Publishing, 2000.

[4] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/software Interface*, Morgan Kaufmann, 2008.

[5] K. Arvind and Rishiyur S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *in IEEE Trans. Comput.*, 1990.

[6] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data flow machines and languages," *in Computer*, 1982.

[7] Charles L. Seitz, "The cosmic cube," *in Communications of the ACM*, 1985.

[8] Mark S. Papamarcos and Janak H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *in SIGARCH Comput. Archit. News*, 1984.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford Dash multiprocessor," *in Computer*, 1992.

[10] K. Arvind and Robert A. Iannucci, "Two fundamental issues in multiprocessing," in *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, 1988.

[11] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "April: A processor architecture for multiprocessing," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1991.

[12] Philip M. Wells and Gurindar S. Sohi, "Serializing instructions in system-intensive workloads: Amdahls Law strikes again," *in 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[13] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh, "An analysis of database workload performance on simultaneous multithreaded processors," *in ACM SIGARCH Computer Architecture News*, 1998.

[14] Sujay Parekh, Susan Eggers, and Henry Levy, "Thread-sensitive scheduling for SMT processors," `http://www.cs.washington.edu/research/smt/`.

[15] Erez Perelman, Greg Hamerly, and Brad Calder, "Picking statistically valid and early simulation points," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[16] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt, "The M5 simulator: Modeling networked systems," *in Micro, IEEE*, 2006.

[17] Steven Cameron Woo, Moriyoshi Ohara, and Evan Torriet, "The SPLASH-2 programs: characterization and methodological considerations," *in Communication*, 1995.

[18] Christian Bienia, Sanjeev Kumar, and Kai Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Proceedings of the 2008 International Symposium on Workload Characterization*, 2008.

[19] Christian Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[20] David H. Bailey, "FFTs in external or hierarchical memory," *in Journal of Supercomputing*, 1990.

[21] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha, "A comparison of sorting algorithms for the connection machine cm-2," in *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[22] Ioannis E. Venetis, "The modified SPLASH-2 home page," `http://www.capsl.udel.edu/splash/`, 12 November 2010.

[23] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler, "Running PARSEC 2.1 on M5," Tech. Rep., The University of Texas at Austin, Department of Computer Science, 2009.

[24] Richard L. Sites, "Alpha axp architecture," *in Communication*, 1993.

[25] Hewlett-Packard, "HP OpenVMS systems documentation - FAQ," `http://h71000.www7.hp.com/faq/vmsfaq_021.html`.

APPENDIX

TABLES OF DATA

Table I.: Instruction growth of the FFT benchmark

Cores	PAL Code	BASH	FFT	Kernel
1	244454	178726	1250556	2351064
2	244065	177235	1263858	2441599
4	265146	178564	1294561	2696372
8	281163	178564	1361343	2984038
16	350852	178991	1529432	3971110
32	566413	178991	1978882	7411790

Table II.: Instruction growth of the Radix benchmark

Cores	PAL Code	BASH	Radix	Kernel
1	7197206	179564	157079855	4036759
2	7214189	178073	157453158	4305760
4	7237617	179402	158287076	4628311
8	7289889	179402	160103272	5640328
16	7435617	179815	163977971	8683744
32	7842547	179815	172181621	18015925

Table III.: Instruction growth of the LU benchmark

Cores	PAL Code	BASH	LU	Kernel
1	538093	179572	1457205541	4357391
2	628014	177998	1458173393	4724298
4	782206	179327	1463962946	5848504
8	1204206	179327	1463962946	9986505
16	1740988	178570	1471687943	19268020
32	3137760	178570	1487130971	48808270

Table IV.: Memory barrier instructions executed

Cores	FFT	Radix	LU
1	32559	37929	45772
2	32205	38311	48195
4	34518	40937	54570
8	36199	47327	76226
16	44152	71378	112395
32	67282	128941	195076

Table V.: Lock types for the FFT benchmark

Cores	Spinlock	Exchange	Unlock	Long Spin	Other	Cycles on Long
1	6320	848	557	837	2418	205106
2	6675	954	607	492	2470	124449
4	7066	1111	722	493	2903	125271
8	7219	1404	887	437	3651	104148
16	8472	2119	1227	457	3651	112566
32	11962	4083	1996	664	9116	552920

Table VI.: Lock types for the Radix benchmark

Cores	Spinlock	Exchange	Unlock	Long Spin	Other	Cycles on Long
1	6991	1866	1067	658	3963	160589
2	7146	2029	1102	504	4095	121216
4	7227	2310	1230	546	4682	143368
8	7898	2922	1430	774	5744	230288
16	10881	4242	1834	863	7985	438838
32	19071	7129	2663	1148	13318	1296710

Table VII.: Lock types for the LU benchmark

Cores	Spinlock	Exchange	Unlock	Long Spin	Other	Cycles on Long
1	10888	1455	806	769	3261	187380
2	11090	1894	901	522	3950	133241
4	12359	2756	1093	514	5608	143368
8	18525	4320	1291	1119	8668	288939
16	28028	8015	1995	901	15259	389398
32	55053	15152	3148	1316	28385	1087661

CONTACT INFORMATION

Name:	Philip James Jagielski
Professional Address:	c/o Michael Jagielski 2017 Irongate Dr. Arlington TX, 76012
Email Address:	philipjagielski@gmail.com
Education:	B.S., Computer Engineering, Texas A&M University, May 2011 Undergraduate Honors Fellow