Multi-Core Performance Scaling Using Graphite and SPLASH-2

Chris Baumler
CPRE 581

1. Introduction

As demand for computing power increases, chip manufacturers are increasingly turning to multi-core processing solutions. These processors contain multiple central processing units (cores) that are able to execute instructions in parallel, increasing overall program execution speed. However, the benefit of such processors depends heavily on the parallelizability of the software being executed. This project aims to explore the potential benefits to be gained from multi-core processing through the use of simulation tools and software benchmarks.

2. Objectives and Scope

The main objective of this project is to use the Graphite simulation tool to characterize the performance scaling of multi-core systems as the number of cores is increased. The characterization is limited to a range of 8 to 64 cores, and a single set of architectural parameters is used. Performance is measured using the SPLASH-2 benchmark suite.

3. Solution Approach

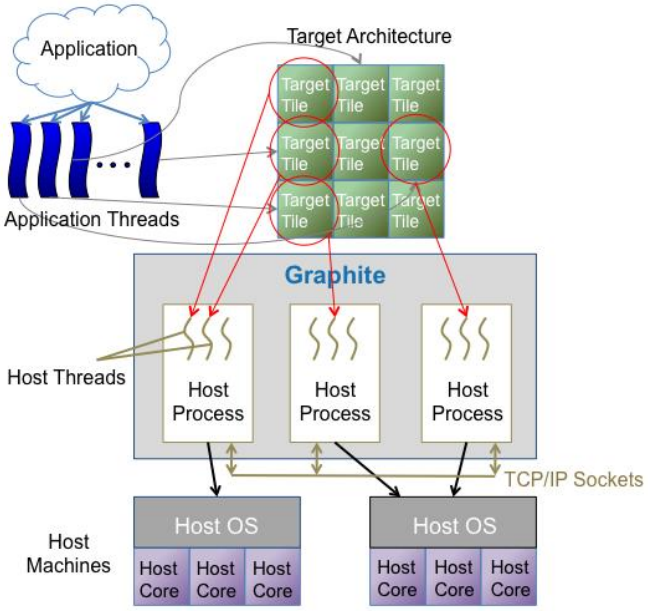
To achieve the stated objectives, the Graphite Multicore Simulator was installed on a host machine and the software programs that make up the SPLASH-2 benchmark were executed.

3.1 Graphite Multicore Simulator Overview

Graphite is an open-source tool created by the Carbon Research Group at MIT. The tool was created to allow researchers to experiment with new multi-core designs and technologies without having to focus on architecture details. This allows users to rapidly explore new ideas and to perform early software development before real hardware is available.

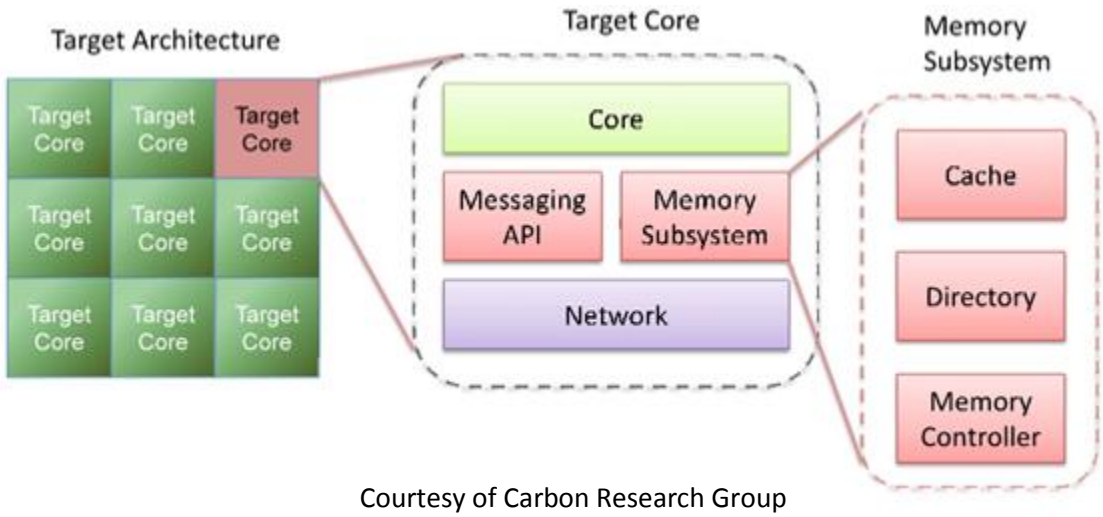
At a high level, Graphite works by creating a set of target cores and mapping application threads to those cores. The cores are then distributed among the various host processes that are available. The host processes may execute on a single multi-core host machine or may be distributed among many host machines. Graphite allows simulation of anywhere from three to several thousand cores with fast, scalable performance. Additionally, it is designed to be invisible to the applications running on it. This means that applications written using the pthreads threading library may be run “off-the-shelf.”

The following diagram, courtesy of the Carbon Research Group, demonstrates the high-level architecture of Graphite.



Graphite implements a layered communication stack to allow application threads to communicate with each other via messages. This allows for the simulation of shared memory regardless of the memory scheme of the underlying host computer(s).

The following diagram shows a more in-depth view of Graphite's architecture.



Courtesy of Carbon Research Group

Architecturally, Graphite consists of a set of models with configuration parameters. The following models currently exist:

- Network Model
- Memory Subsystem
- Core Model
- Contention Model
- Heterogeneity
- Dynamic Frequency Scaling (DFS)
- Power Model

3.2 SPLASH-2 Benchmark Suite

The SPLASH-2 benchmark suite is a collection of multithreaded applications designed to measure performance for parallel workloads. It was introduced by Stanford University in 1996 and consists of the following programs:

Kernels:

- Complex 1D FFT
- Blocked LU Decomposition
- Blocked Sparse Cholesky Factorization
- Integer Radix Sort

Applications:

- Barnes-Hut
- Adaptive Fast Multipole
- Ocean Simulation
- Hierarchical Radiosity
- Ray Tracer
- Volume Renderer
- Water Simulation with Spatial Data Structure
- Water Simulation without Spatial Data Structure

3.3 Environment and Setup

The host computer used for the simulation had the following hardware/software specifications:

- Intel Core i5 2.66 GHz CPU
- 8 GB DDR3 RAM
- Windows 7 64-bit

Since Graphite runs in a Linux environment, a virtual machine was created using VMware Player. The virtual machine was assigned 1 GB of memory, a single processor, and 20 GB of hard disk space. Ubuntu version 12.04.3 was installed in the virtual machine partition.

To support Graphite, several dependencies were installed on the host environment. First, Pin version 61206 was installed. Pin is a tool made by Intel for the instrumentation of programs. It allows another tool to dynamically insert C or C++ code into an executable. Graphite uses Pin to rewrite memory accesses when simulating a shared memory space.

In addition to Pin, a set of common libraries necessary for compiling (g++, make, libtool, etc.) and other common tasks was installed, and Boost version 1.48 was installed. Boost is another set of C++ libraries used for building projects. Appendix A includes detailed instructions for installing each component.

Once all of the dependencies were installed, the Graphite source code was downloaded, installed, and compiled. The Graphite code repository included the SPLASH-2 benchmark applications integrated into the tool.

Before executing any simulations, Graphite was configured by modifying the “carbon_sim.cfg” file. This configuration file contained all of the parameters necessary to define the multicore simulation environment. The simulation environment was configured as follows:

- Total number of cores was varied for each simulation
- Number of host processes was set to one
- Core modeling was enabled
- Shared memory simulation was enabled
- System call modeling was enabled
- Full simulation mode was enabled
- A lax barrier clock skew management scheme was selected (worst performance, best accuracy)
- Thread stack size was set to 2097152 bytes
- Each simulated core was set at 1 GHz frequency
- The following cycle costs were used for arithmetic instructions
    - add=1
    - sub=1
    - mul=3
    - div=18
    - fadd=3
    - fsub=3
    - fmul=5
    - fdiv=6
    - generic=1
    - jmp=1
- A 1-bit branch predictor was used with a 14 cycle misprediction penalty
- L1 instruction and data cache line sizes were set to 64 bytes
- L1 instruction and data cache sizes were set to 32 KB
- L1 instruction and data cache associativity was set to 4-way
- L1 instruction and data cache replacement policy was set to least-recently-used (LRU)
- L1 instruction and data cache data access time was set to 1 cycle
- L1 instruction and data cache tag access time was set to 1 cycle
- L2 cache line size was set to 64 bytes
- L2 cache size was set to 512 KB
- L2 cache associativity was set to 8-way
- L2 cache replacement policy was set to LRU
- L2 Data access time was set to 8 cycles
- L2 tag access time was set to 3 cycles

Once installation and configuration were complete, each benchmark application was executed multiple times, changing the number of simulated target cores each time. Since Graphite does not currently support assigning multiple application threads to a single target core, the number of application threads used by the benchmark application was always configured to be identical to the number of available cores.

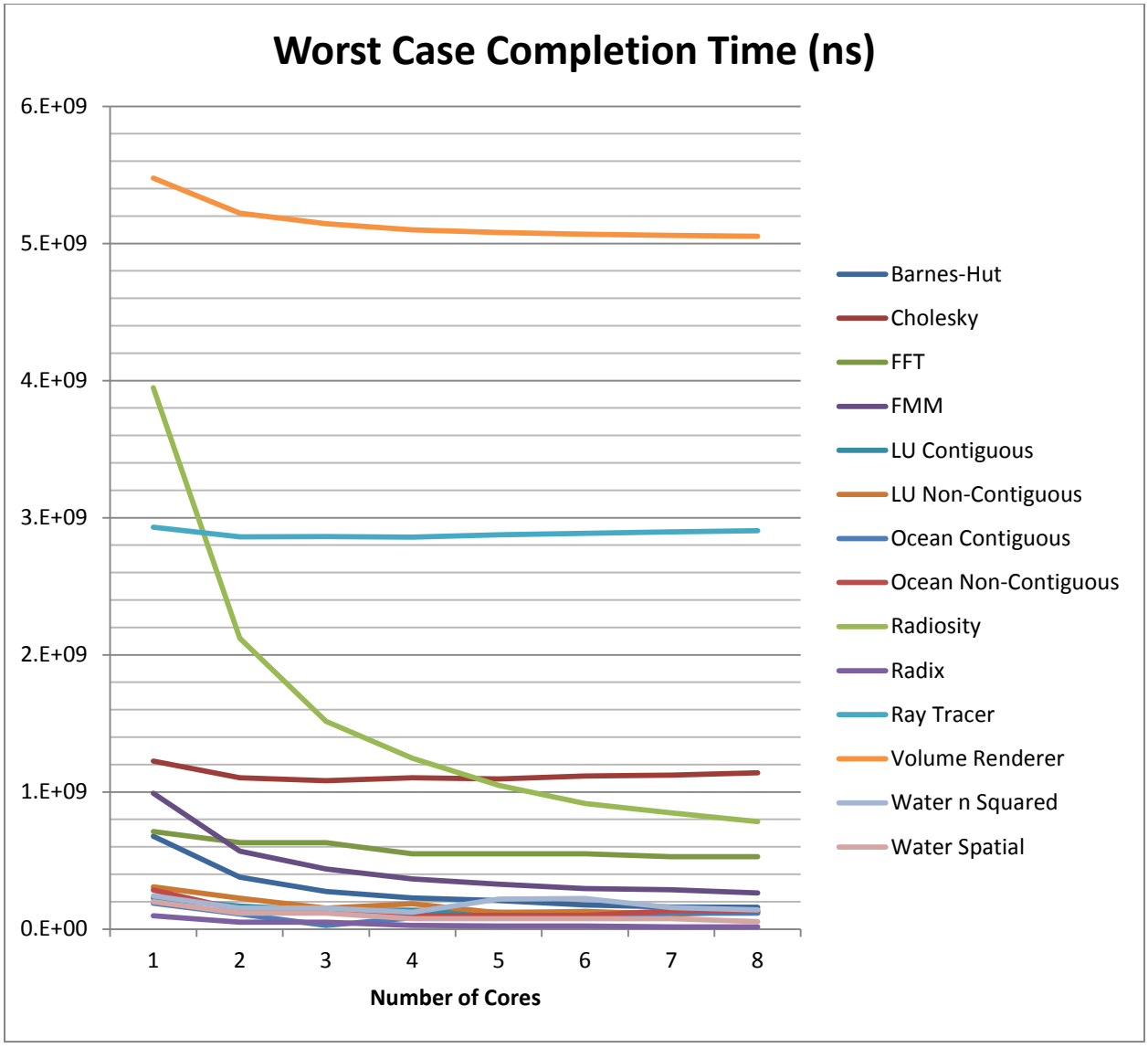
Note: To facilitate running many different test applications with different configuration settings, a shell script was created that automatically set up the configuration and executed the tests.

4. Simulation Results

Graphite tracks a vast number of performance statistics for each core which it saves in log files. The following simulation results were compiled using these statistics.

4.1 Completion Time

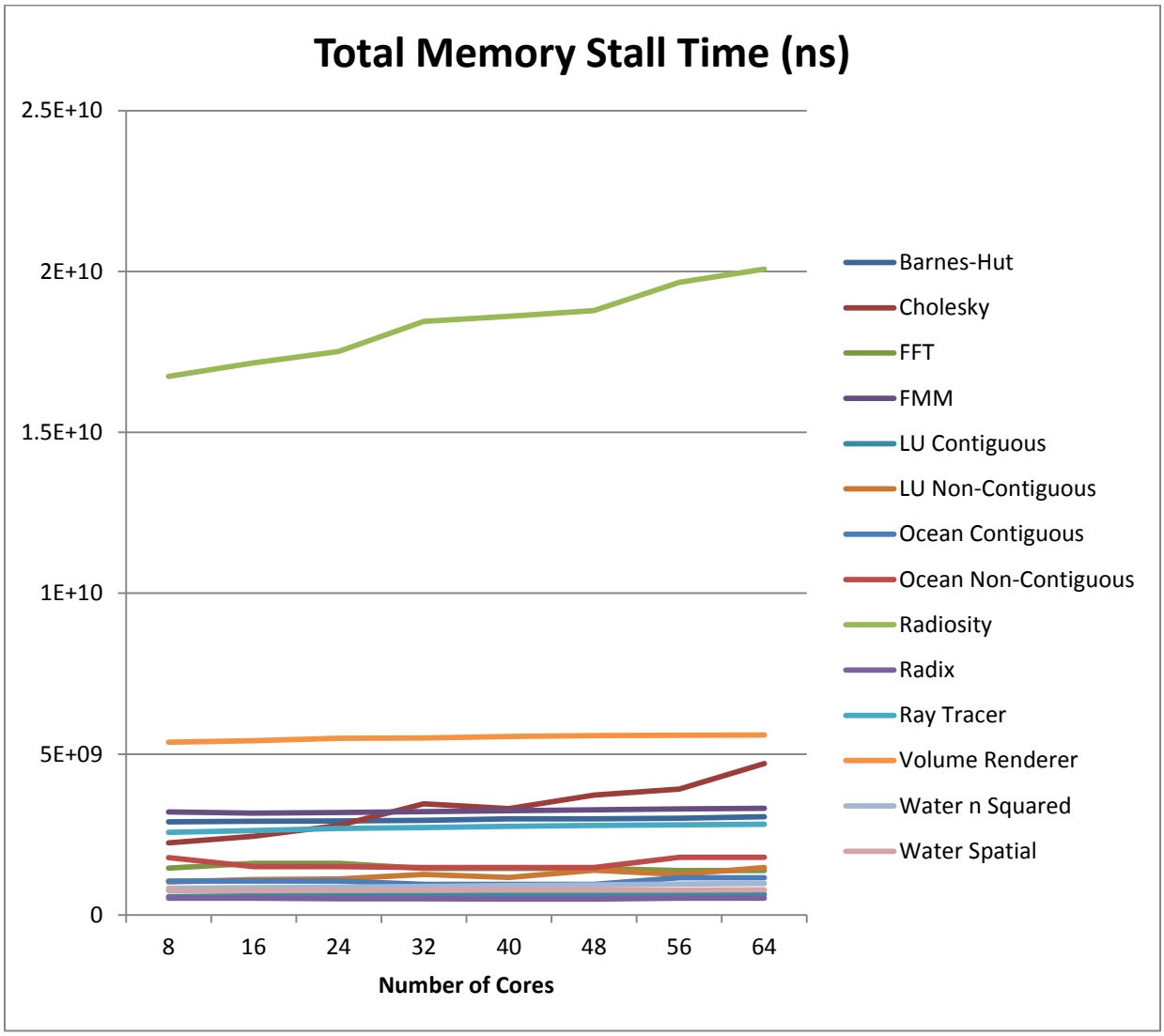
Graphite measures total completion time for the instructions running on each core. The following graph shows the worst case completion time for each application when running with different numbers of cores. Worst case completion time was determined by taking the maximum of the individual cores' completion times.



The results show a general trend of decreasing completion time. However, the slope is very pronounced in some applications (e.g. Radiosity) and not very noticeable in others. This would seem to suggest that some of the benchmark applications are more parallelizable than others.

4.2 Memory Stall Time

Graphite measures the total memory stall time for each core. The following graph was created by summing the stall times of all cores.



The results show a general trend of constant total stall time for different numbers of cores. However, there are a couple of benchmark applications (Radiosity and Cholesky) whose stall time increases when increasing the number of cores.

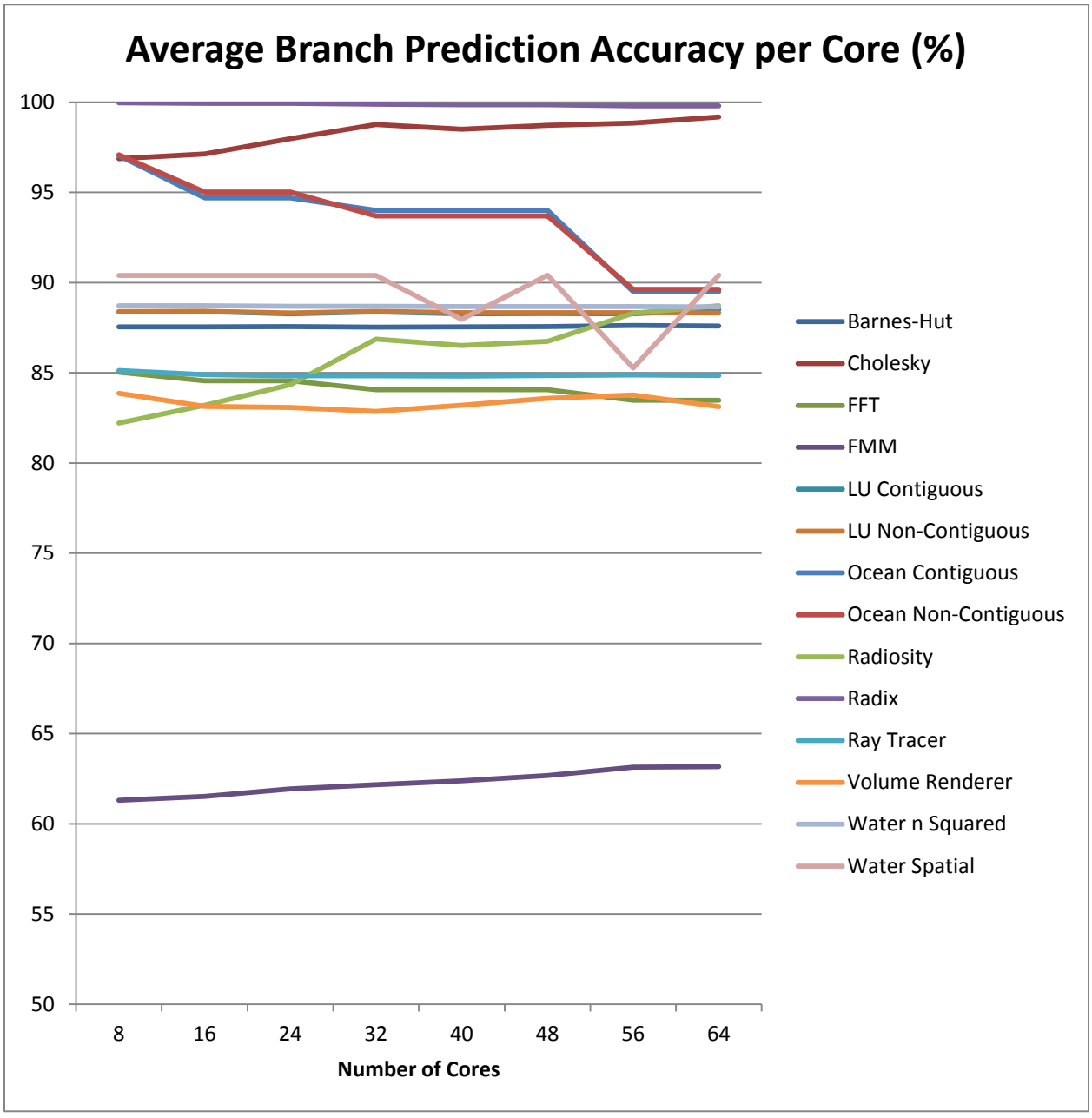
4.3 Execution Unit Stall Time

Graphite measures the total execution unit stall time for each core. The following graph was created by summing the stall times of all cores.

The results are very similar to the results for the total memory stall time. Again, there is a general trend of constant total stall time for different numbers of cores. However, there are a couple of benchmark applications (Radiosity and Cholesky) whose stall time increases when increasing the number of cores.

4.4 Branch Prediction Accuracy

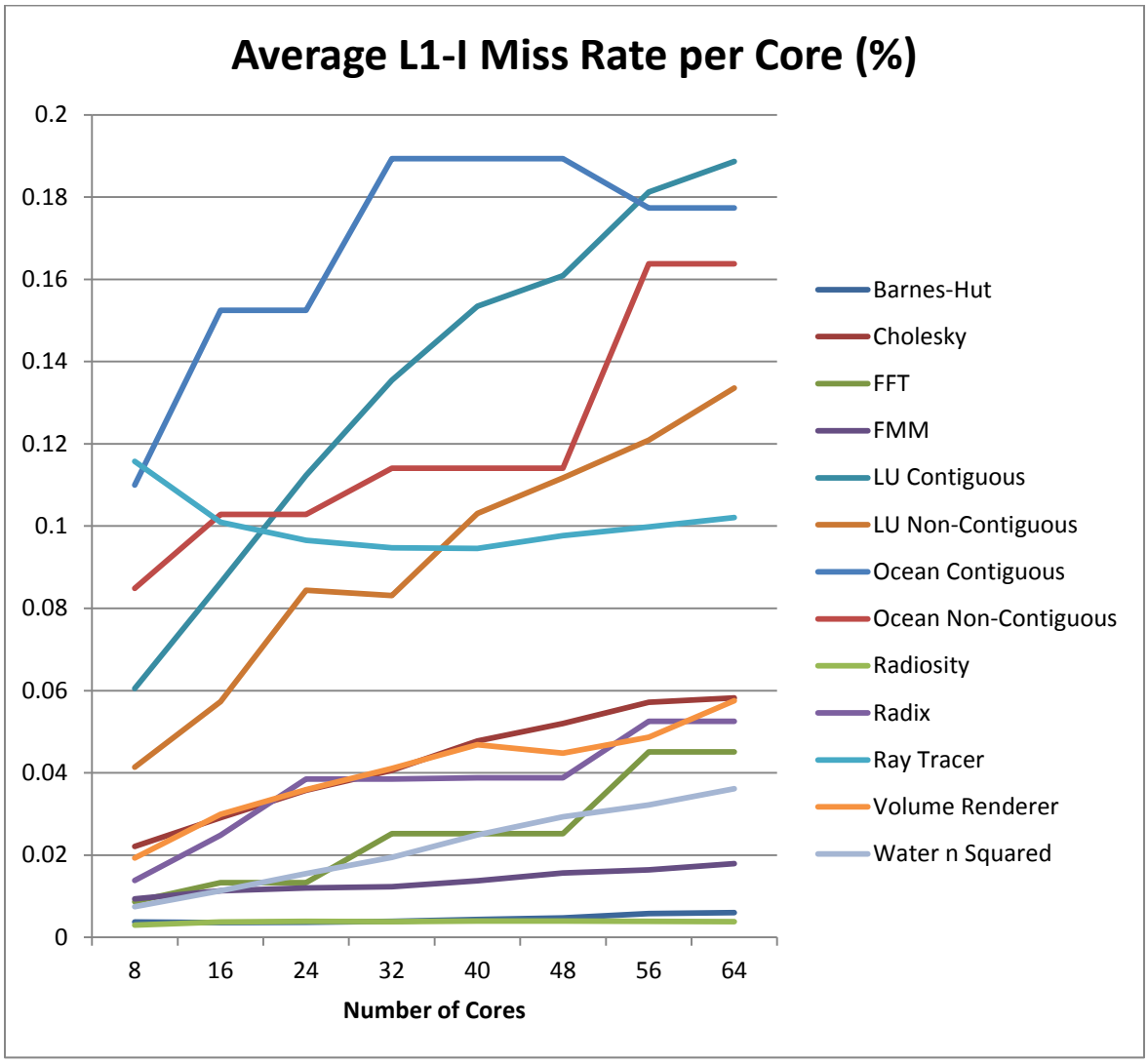
Graphite measures the number of correct and incorrect branch predictions per core. The following graph was created by dividing the number of correct predictions by the number of incorrect predictions and averaging over all of the cores.



The results are mixed. Some applications show decreasing branch prediction accuracy, while others show increasing branch prediction accuracy, and others show constant branch prediction accuracy. No general conclusion can be drawn.

4.5 L1 Instruction Cache Miss Rate

Graphite measures the L1 instruction cache miss rate (%) per core. The following graph was created by averaging the miss rates of all cores.

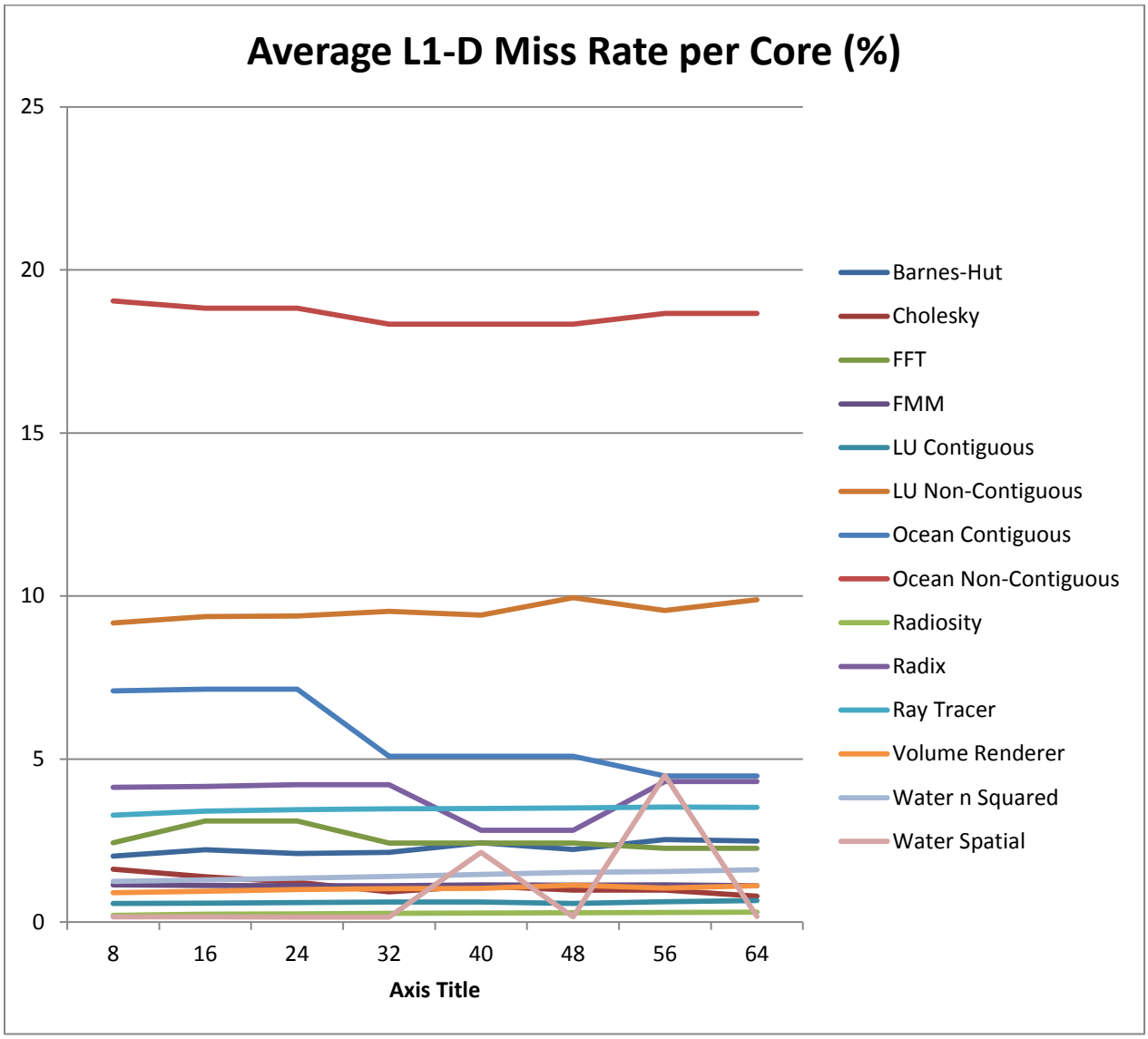


The above graph does not show the results for the Water Spatial application. It was removed, because its behavior deviated significantly from all other benchmark applications – showing steep peaks at 40 and 56 cores. Closer examination this application's data showed that the majority of the cores showed similar behavior to that of the other applications, but a few cores exhibited significantly higher miss rates. This deviation could be due to an anomaly in the test environment or a unique characteristic of that particular benchmark application.

Aside from the Water Spatial application, the general trend is a very slowly increasing miss rate with a couple of minor exceptions. The miss rate of the Barnes-Hut application appears to drop off around 56 cores, and the Ray Tracer application initially shows a decreasing miss rate before starting to increase.

4.6 L1 Data Cache Miss Rate

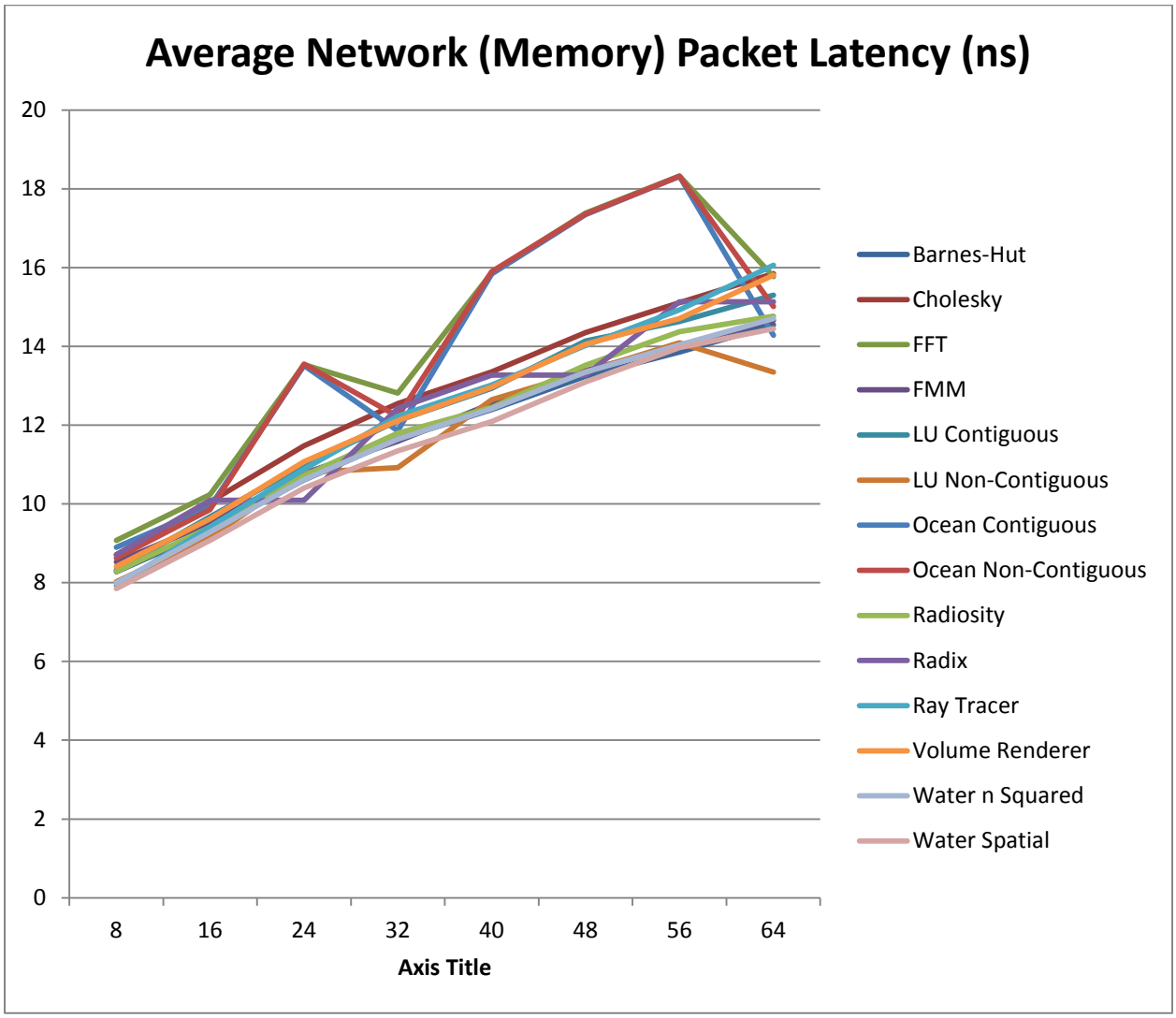
Graphite measures the L1 data cache miss rate (%) per core. The following graph was created by averaging the miss rates of all cores.



The results show that generally, the miss rate is constant. Again, there are a few exceptions. Water Spatial once again shows somewhat anomalous behavior. We also see that the data cache miss rate shows a greater amount of variability between applications than the instruction cache miss rate. Ocean Non-Contiguous shows the highest miss rate (around 19%).

4.7 Network (Memory) Packet Latency

Graphite measures various statistics related to network traffic between cores accessing the shared memory. One of these is average packet latency for each core. The following graph was created by averaging the packet latency across all cores.



The results show a general trend of increasing latency as the number of cores increases. A few applications show a deviation from this trend by sharply decreasing when going from 56 to 64 cores. The trend of generally increasing makes sense intuitively, since, as the number of cores increases, the chip area increases as well. So we would expect some cores to be located further from memory resources, increasing packet latency.

5. Conclusion

The goal of this project was to characterize the performance scaling of multi-core systems as the number of cores is increased. That goal was achieved using the Graphite simulation tool and the SPLASH-2 benchmark applications. The results showed fairly clearly that in general, execution time decreases with number of cores, while packet latency increases. These results make sense as execution is being spread among many cores working in parallel, and cores are being moved further away from memory resources. The results for various other statistics including total memory stall time, total execution unit stall time, average L1-I miss rate per core, and average L1-D miss rate per core showed little to no variation as the number of cores was increased, suggesting that these characteristics do not depend as heavily on the number of cores. The last statistic, average branch prediction accuracy per core, varied significantly, and no trend could be deduced. Overall, the project was successful in achieving its goal.

6. References

1. Carbon Research Group
http://groups.csail.mit.edu/carbon/?page_id=111

2. A View from Berkeley Wiki
http://view.eecs.berkeley.edu/wiki/SPLASH

3. Ubuntu
http://www.ubuntu.com/download/desktop

4. VMware Player
https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0%7CPLAYER-600-A%7Cproduct_downloads

5. Pin – A Binary Instrumentation Tool
http://software.intel.com/en-us/articles/pintool-downloads

Appendix A – Detailed Installation and Execution Instructions

Install Dependencies:
Download and untar Pin. Then run the following commands.

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install libtool automake autoconf autotools-dev
$ sudo apt-get install libboost1.48-dev libboost-filesystem1.48-dev libboost-system1.48-dev
```

Obtaining Graphite Source Code:

```
$ wget http://github.com/mit-carbon/Graphite/tarball/master -O graphite.tar.gz
$ tar -zxvf graphite.tar.gz
$ cd mit-carbon-Graphite-*
```

Building Graphite:
Edit “Makefile.config” by setting PIN_HOME to the directory where Pin was installed. Then run the “make” command.

Running SPLASH-2 Applications:
make barnes_bench_test
make cholesky_bench_test
make fft_bench_test
make fmm_bench_test
make lu_contiguous_bench_test
make lu_non_contiguous_bench_test
make ocean_contiguous_bench_test
make ocean_non_contiguous_bench_test
make radiosity_bench_test
make radix_bench_test
make raytrace_bench_test
make volrend_bench_test
make water-nsquared_bench_test
make water-spatial_bench_test