Understanding Performance Issues on both Single Core and Multi-core Architecture

Julian Bui
Department of Computer Science
University of Virginia
Charlottesville
VA 22903
julianbui@gmail.com

Chenguang Xu
Department of Computer Science
University of Virginia
Charlottesville
VA 22903
cx7m@virginia.edu

Sudhanva Gurumurthi
Department of Computer Science
University of Virginia
Charlottesville
VA 22903
gurumurthi@cs.virginia.edu

ABSTRACT
This research intended to find the relationships between the memory system and performance in both single core and multi-core context. For the single core part, several parameters have been considered to improve the performance. Based on these work we extend to the multi-core issues. Overall, the simulations showed results similar to configurations of many current consumer CMPs. Many results were expected, like a large L2 cache size certainly helped performance a lot and the L1 cache size, optimal system frequency was typical of many of the Intel Core 2 Duos, and that after already having 2-4 core any more cores do not help performance in FFT nearly as greatly. However, some results were more surprising like the low speedups that the n-core systems presented. Also, some of the experiments failed, like for 16-core systems and for cache coherence tests. Overall, the excess of 500 simulations recorded helped understand that in multi-core systems, communication overhead and memory latencies are a limiting factor in performance. Finding good cache configurations certainly help increase performance by taking the pressure off of the main memory and reducing communication and cache coherence latencies.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiprocessors

General Terms: Design, Performance

Keywords: multi-core, inter-processor communication, parallel program

1. INTRODUCTION
Multi-core computers, computers consisting of two or more processors working together on a single integrated circuit, have produced breakthrough performance in the beginning of the 21st century. They allow for faster execution of applications by taking advantage of parallelism, or the ability to work on multiple problems simultaneously. They have been able to perform well because they improve upon the drawbacks of the symmetric multiprocessors (SMP) machines which are multiple processor machines that shared memory among the processors.

SMPs share advantages with CMPs. SMPs and CMPs both have the ability to take advantage of parallelism. Parallelism is important because big problems can be split into smaller programs that can be executed at the same time to reduce execution time. They can make use of instruction level parallelism (ILP), which how many simultaneous instructions can be executed, and thread level parallelism (TLP), which is how many simultaneous threads can be executed, as explained in [25]. They can also take advantage of multiple processes because they treat each as a thread and split up and can assign a thread to another processor. Execution within a processor is very quick and inexpensive for time. However, with multiple processing units, the communication overhead becomes the problem. On an SMP, the main memory is out of chip. Therefore, to communicate with neighboring processors, the processors must go off chip, which has high latency. As execution speeds increase, the memory latency becomes more of the limiting factor. This is the called the memory wall.

CMPs have the same advantages of SMPs, except the main improvements are that the big memory wall problem and the power consumption problems are reduced. Because the shared memory on a CMP is on-chip, the communication latencies between processors are reduced. Cache coherence becomes less of a hindrance because the time it takes to snoop and maintain cache coherence is reduced. Also, CMPs lighten power concerns because the circuit is smaller and it requires less power to drive signals because more circuitry is on-chip and a lot of the circuitry can be shared between processors.

Since the cache and communication with caches has such an important role in the improvement from SMP to CMP, the research intended to find out what relationships exist between cache memory and performance.

2. RELATED WORKS
2.1 Related Works on Cache's Impact

Computer Organizatione'07, Month 9–12, 2007, Charlottesville, VA,

Cache memories are widely used in microprocessors to improve the system performance [1], and several works [2~8] have been done in cache fields. Cache size, cache protocols, associate numbers, etc. are all important parameters for performance, among which cache size is thought to be the most important parameter for performance. The larger the cache size is, the better the performance will be. But due to the development of technology, which makes large L1, L2 cache is available (e.g. the off-the-shelf Intel Cure Duo T7300 has 64kB L1 Cache and 4MB L2 Cache), the impact of cache size to performance is decreasing.

But energy efficiency has become more and more important in the designing of computer architecture, and perhaps become one of the most important issues in the area of computer architecture. Though large cache memories can improve the system performance dramatically, they are responsible for a large portion of the overall system's power dissipation [2]. To reduce the energy dissipation, several approaches of dynamic cache reconfiguration have been reported. Zhang et al. [3] proposed a technique called way concatenation that tunes the cache ways between one (direct-mapped), two and four. Similarly, Dropsho et al. [4] introduced a cache design, called accounting cache based on the selective ways cache [5]. The number of active ways is dynamically changed under hardware control. Powell et al. [6] applied way-prediction and selective direct-mapping to reduce the set-associative cache energy. But just as pointed out by [2], among all configurable cache parameters, cache size has the largest impact on cache performance and energy consumption. The more lager size of cache, the more energy needs to be dissipated. It is an interesting problem to find a suitable size for cache which not only minimizes the energy consumption, but also maintains a good performance.

2.2Related Works on Multi-Core CPU

The traditional way for improving CPU performance in Single Core architecture has lead dangerously close to the energy / power consumption ceiling, which stops performance progress in processor development. Multi-core or Multiprocessor System on Chips (MPSoC) is thought to be one of the most promising alternatives [8]. Processing element, communication on chips between cores, memory architecture, and hardware architecture optimization for specific application are the four main issues affecting performance and software programmability in Multi-Core system [8]. The description of multi-core technology from two of the major producers can be seen from [22~23].

Multi-core has become a hot topic research filed in recent years [9~21]. Some of the works [16, 17] focus on operation system support for multi-core architecture, or on multi-core architecture designing for easily system software programming. But most of the related works focus on high performance multi-core architecture designing. Cache coherence, memory architecture, interconnection between cores, and heterogeneity cores are some of the considered factors for improving performance. Perhaps the work in [24] is the most related to our project. The work in [24] demonstrates dynamic core assignment policies that provide significant performance gains over naive assignment, and even outperform the best static assignment. It examines policies for heterogeneous architectures both with and without multithreading cores.

3. MOTIVATION

The motivation of this project is as follows which also form the main part of this project:

1) About Cache Size —whether the larger the better for cache size? Or is there any point that the enlargement of cache size above this point makes little improvement in performance. Since as for cache itself, cache size is the most important factor for energy consumption. So to find this point is of significance. We will consider the cache size impact of L1 Cache and L2 Cache for both single core and multi-core system in this project. Also, we will take into consideration of an interesting problem that is which part of L1 Cache is more important to the performance. In some application scenarios, L1 Cache size is strictly limited, and maybe make the size for I cache and D cache different is a good choice.
2) CPU Frequency ---does higher frequency means higher performance? It supposed to be so. But we still include this part in our project, and the experiment results are somewhat curious, which show that higher frequency doesn't necessarily mean higher performance. Details and the analysis for possible reasons will be given in the later part of this report.
3) To understand the IPL (Instruction Parallelism Limit). IPL depends on the width of the processor to fetch and execute instructions, the number of memory ports, and in-order execution vs. out-of-order execution. We planed to include all of those in our project. But unfortunately, there are some unexpected difficulties (e.g. the problem for O3CPU in M5), so we only consider the impact of the width of professor in this project.
4) To under the impact of multi-core. Multi-core system can improve the performance comparing with Single core. We will figure out in this project that how much performance advantage multi-core will bring as compared to single core system. An interesting problem is to compare the performance of single core with frequency *f*, dual core with frequency *f* for each core, and single core with frequency 2*f*, and it is included in our project. In addition, the interconnection between multi-cores may change the cache size point in single core system. So the cache issues will also be considered in multi-core system.
5) The last motivation of this course project is to learn how to use M5- a powerful simulator.

4. EXPERIMENT METHODOLOGY

The research involves four major steps: 1)Setup the environment on which the code is run 2)Set up the code that the simulator will run 3)Run single-core simulations (Collect and Analyze Data) 4)Run multi-core simulations (Collect and Analyze Data).

For the first step, the group decided to run the M5 simulator, which is a "modular platform for computer system architecture research", because the simulator had a large community base and could support multi-cores [26]. Its version is M5 2.0b3 and uses gcc v4.1.1, python v2.4.4, swig v1.3.31, scons-0.97, and zlib v1.2.3. The environment was built on the Centurion Cluster at the University of Virginia which has hundreds of Dual 1.6 GHz Opteron252s with 2GB of RAM each. Also, the group had to choose a specific, commonly used benchmark. The group chose the SPLASH2 benchmark Fast Fourier Transform (FFT) because

it is widely used. This algorithm is used commonly for digital signal processing. This benchmark has been optimized and parallelized.

The second step, the code writing can be done fairly easily. Some sample CMP code was used and modified for the research's needs. The difficult part was tediously changing all the script files for every simulation – there were at least 500 simulations done in this experiment. The project made sure to model the system to common, real systems. Common latencies were and used them. Also, the instruction and data and L2 caches were 8-way associative and had 64 byte block sizes. One of the program result limitations was the topology. We could only have one L2 cache, that was shared among all processors, connected to the processors through a single bus. Also, CPUs were in-order CPUs. This does not exactly model the consumer multi-core systems but it was the only thing that M52.0b3 could model.

Thirdly, the single-core simulations were run. Six factors were changed as independent variables: L1 cache size, L2 cache size, bus width, bus/CPU frequency, varying instruction and data cache sizes, and cache coherence protocols. The testing would be done iteratively because with six independent variables, each with a wide range of values, the search space would be too large. Instead of testing the entire search space, the group first found a good L1 cache size that gave good performance in terms of execution time without being too costly in terms of size and cost of cache. Since there were no standards at the beginning, the group decided to take a wide range of independent variables and a suitable value for the L1 cache size before moving on to L2 cache size tests. Then a wide range of L2 cache sizes were tested, and a suitable value for the L2 cache size was found. Then the previous suitable L1 and L2 cache sizes were used for the next iteration of bus/CPU frequency. This type of iterative process was done until suitable values for each of the 6 independent variables were found. This data would then be collected and analyze to attempt to discover the relationships between the memory system and a uni-core system. This is done to hopefully get an idea of what range of values could be used for the multi-core system while also learning why the relationships exist and maybe try extending knowledge to the multi-core system.

The last and fourth step is the multi-core simulations. These were done to discover suitable values for the 6 independent variables discussed in the previous step, and the process was greedy and iterative just as it was in step three. Data was collected and analyzed to establish relationships again.

5. EXPERIMENT RESULTS

5.1 Single Core

5.1.1 The Impact of Cache Size

We change the size of L1 Cache, the Icache(=Dcache) size varies from 8kB to 128kB, the CPU frequency is default 1GHz, and L2 size is default 512kB. The experiment results for cache miss rate and CPU execution time can be seen from Fig.1~2. As can be seen from Fig. 1~2, 32kB is the size for Icache and Dcache that we want to find out. The enlargement of L1 Cache size above 32kB has little significance to the performance.

We also do the similar experiments to L2 Cache, by setting CPU frequency to be the default value 1GHz, and L1 Cache size to be 64kB (32kB for Icache, and Dcache respectively). The experiments results can be seen from Fig.3~4, which show that 4MB is the size for L2 Cache that we want to find out.

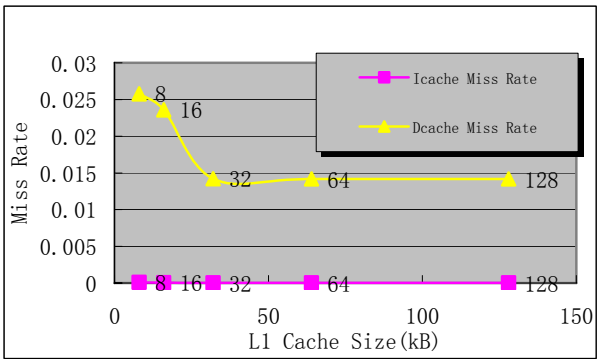

Fig.1

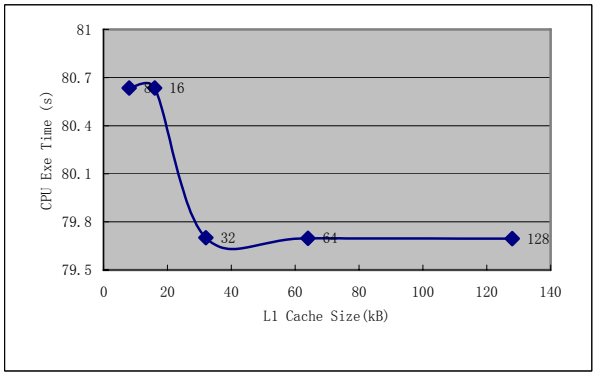

Fig. 2

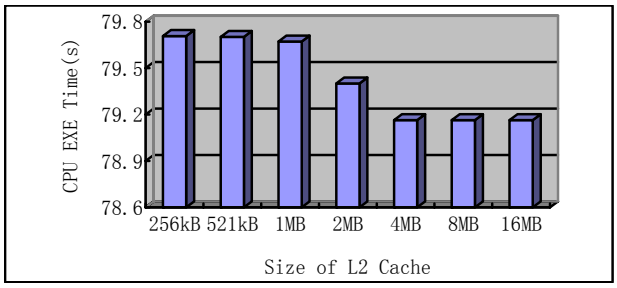

Fig. 3

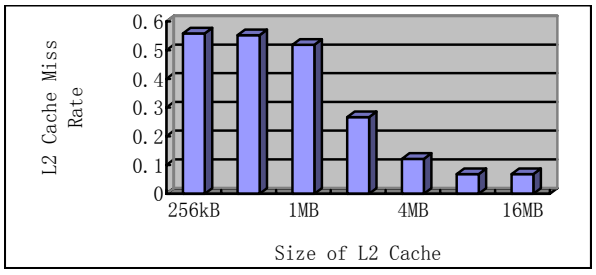

Fig.4

5.1.2 The Impact of CPU Frequency and CPU Width

In this part we change the CPU frequency from 1.0GHz to 3.6GHz, CPU width from 32 to 64, and keep other parameters constant. The experiment results can be seen in Fig.5~6. From these results we can see that:

(1)The experiment results show that the width of CPU has little impact on the performance. Or to be exact, the CPU width has little impact on the running of FFT benchmarks in M5, which is a little curious. The width of the processor to fetch and execute instructions is increased from 32 to 64, but the performance is improved so little. Anyway, it reveals the Instruction Parallelism Limit to some extent.

(2)The relationship between number of executed CPU Cycles and CPU frequency is not stable. Higher frequency may cause more CPU Cycles to execute. One of the reasons may be that although, the CPU frequency is increased, the L1 Cache and L2 Cache access delay have not been changed, so higher frequency CPU will need to wait for more cycles for that kind of access. Maybe the waited cycles are not enough for CPU to complete another job. E.g. if the CPU frequency equals 1.2GHZ, it needs to wait for about 1/4 cycles for L1 access (L1 access latency is set to be 1ns), and it is difficult to schedule 1/4 cycle for CPU to improve the performance. But if the CPU frequency is 2GHz, the CPU needs to wait for a complete cycle for L1 access, and it is easy to schedule one complete cycle to make CPU do something else within that cycle, just as the experiment results show in Fig.5. But if we take the actual CPU execution time for comparison, CPU with higher frequency will have shorter execution time as shown in Fig.6. But in later part of this project, we will see more curious results that higher CPU frequency will cause longer execution time.

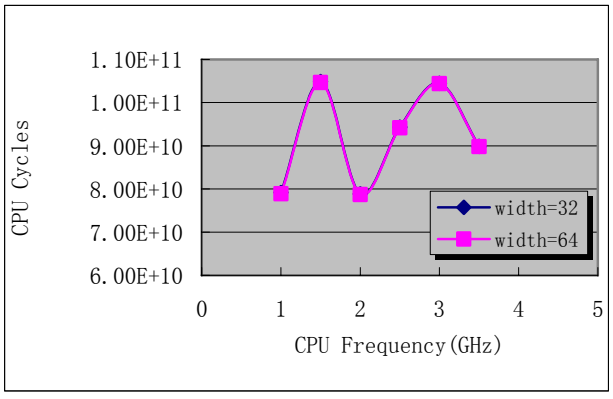

Fig. 5

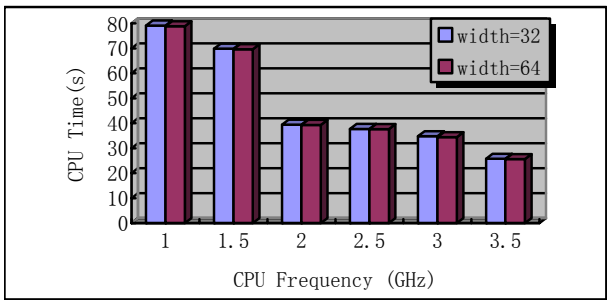

Fig. 6

5.1.3 Cache protocol

We have considered cache size, CPU width, and CPU frequency in the former part of the report. In this part, we will look into the impact of cache protocol. We only consider the L1 Cache protocol for convenient. The experiment parameters are set as follow: CPU frequency = default 1GHz, CPU width=default 32, L1Cache size=64kB, L2 Cache size=4MB. We choose two cache protocols say MESI and MSI for comparison. We change the size of benchmarks for running, and compare the I cache and D cache overall miss rate. The experiments results can be seen in Fig.7 and 8. From Fig.7 and 8, we can see that:

1) For I cache, the performance of MESI and MSI are almost the same, while for D cache, the performance of MESI is always better than MSI. In all, MESI outperforms MSI.

2) The larger of benchmark size, the low miss rate of Dcache. The explanation is that at the beginning of the execution, the cache miss rate is almost 1, and then the miss rate arrives to a stable point. So the more execution time, the low miss rate of Dcache.

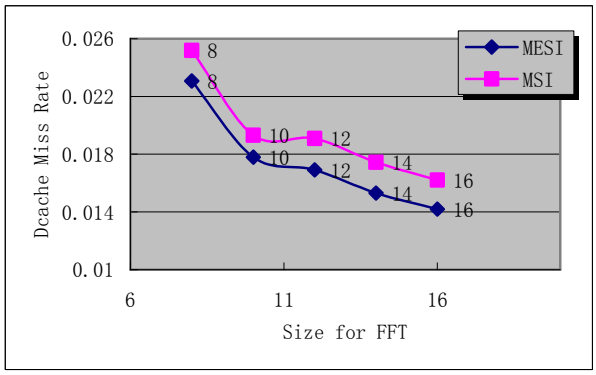

Fig. 7

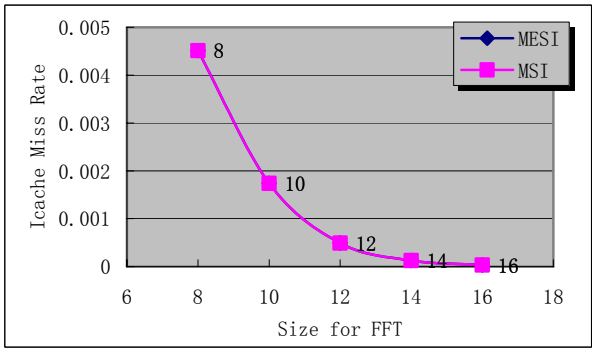

Fig. 8

5.1.4 Icache vs Dcache

In this part, we will look into which part of L1 Cache is more sensitive to performance, Icache or Dcache? For simplicity, we compare the performance of two cache size assignment scheme: (1) Isize=2Dsize, (2)Dsize=2Isize. CPU frequency and CPU width are all set to the default value, while L2 Cache size is 4MB. We change the total L1 Cache size from 6kB to 192kB. The experiment results are shown in Fig.9. As can be seen from Fig.9, Dcache is more sensitive to performance, and should be assigned

more size in some application where L1 cache size is strictly limited.

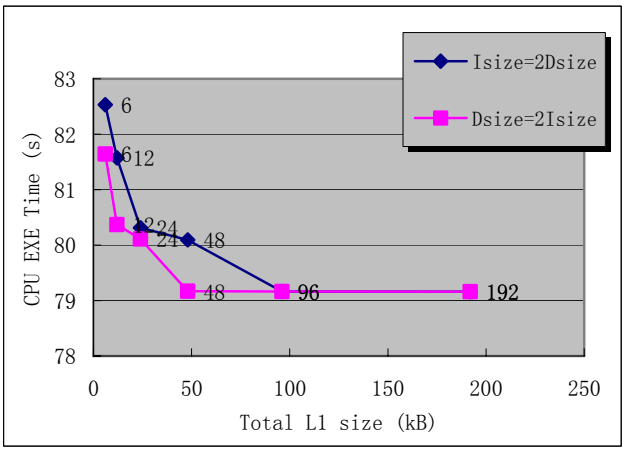

Fig. 9

5.1.5 The Performance Comparison between Single Core and Dual Core

Nowadays, dual core processor is more and more popular, and it is supposed to have better performance than single core. We want to look into how much advantage dual core can bring to the performance in this part. We will make an interesting comparison between the performance of single with frequency *f*, dual core with frequency *f* for each core, and single core with frequency 2*f*. L1 Cache size is set to be 64kB, L2 size is 4MB, CPU width is 32, we change the CPU frequency from 1 to 1.8, and the experiment results are in Fig. 10~11.

From Fig.11, we can see that as for the number of CPU Cycles executed, dual core system is of the largest number. Dual core is running at the cost of redundant computation and communication overhead. But for actual execution time, as can be seen from Fig.10, the performance of single core with frequency 2*f* is better than dual core, which is better than single core. What is strange that the actual execution time for 1.2GHz is worse than 1.0GHz (for both dual core and single core), which contradicts common sense. We have explained one of the possible reasons in the former part of this report, but it seems to be not enough.

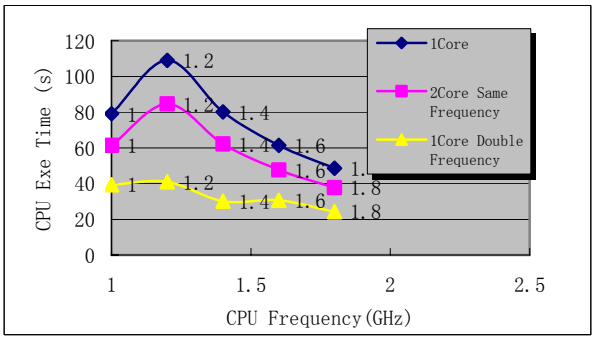

Fig. 10

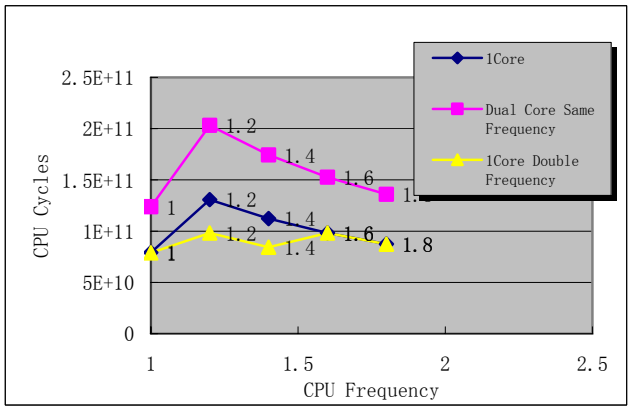

Fig. 11

5.2 Multi-core

For the multi-core simulations, the same tests were done for the single-core except for 2, 4, 8, and 16 cores. Early on in the 16-core system tests, the M5 simulator stopped producing results. It also did not produce some results on some of the cache coherence simulations. It is suspected that the M5 simulator still has bugs (it is in beta still) or maybe the benchmarks' specially created library for M5 has some bugs in it. The experiment started with 32kB L1, 4MB L2, 1.0GHz bus/CPU frequency, and a 32-bit bus. At each iteration, the independent variable in question would be changed with every independent variable that hasn't been tested kept as a default.

5.2.1 L1 Cache Size Tests

L1 cache sizes of 8, 16, 32, 64, 128, 256, and 512 kB were tested on the 2, 4, 8, and 16-core machines. Results for all machines were similar to the graph shown in Fig.12 so only the graph for 2 cores is shown. From the graph, one can see that there is a great improvement in performance when going from 16kB to 32kB. Any extra L1 cache did not give very big of an improvement.

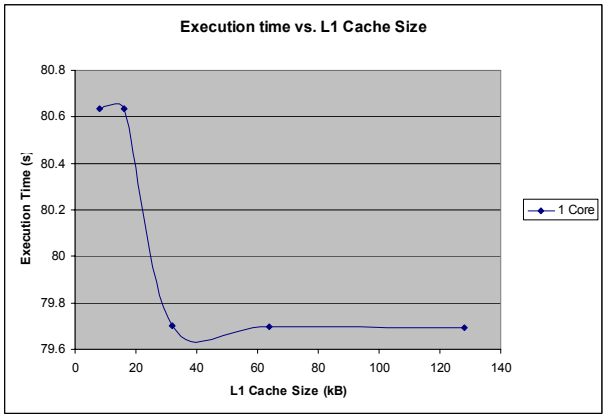

Fig. 12

5.2.2 L2 Cache Size Tests

L2 cache sizes of 256kB, 512kB, 1MB, 2MB, 4MB, 8MB, and 16MB were then tested with a machine with 32kB instruction cache sizes because 32kB was shown to give good performance in

the previous L1 iteration of the tests. Figure 13 shows the results of the L2 cache tests. Again, this graph is only for 2 cores, but the other machines produced similar results and their graphs will not be shown due to repetitiveness. The best gain comes from the 4MB L2 cache size. It brings a relatively large performance boost over the 1MB and 2MB L2 cache sizes, while increasing the cache size further than 4MB does not give such dramatic improvements. This L2 size is typical of consumer CMPs and can be explained by the fact that with more cores, a bigger L2 cache size is required to prevent having to go to slow main memory frequently.

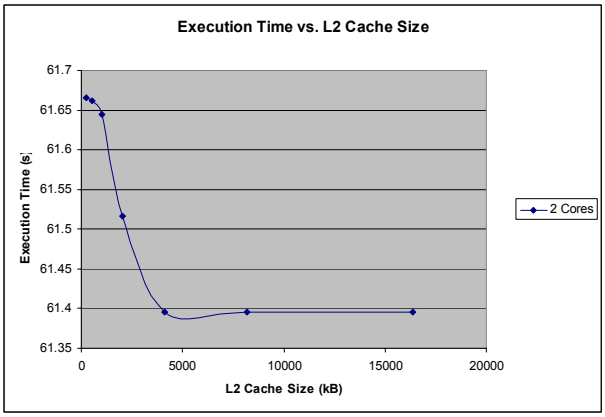

Fig. 13

5.2.3 Bus/CPU Frequency and Bus Width Tests

After the L1 and L2 cache sizes with the best gains had been determined, Bus/CPU frequencies were tested with 32kB L1 cache and 4MB L2 cache sizes chosen. For each of the 1-core, 2-core, 4-core, and 8-core machines, 1.0GHz, 1.5GHz, 2.0GHz, 2.5GHz, 3.0GHz, and 3.5GHz were tested on both 32-bit buses and 64-bit buses. This was a total of 48 (4*6*2) simulations. It was determined that the 64-bit bus width produced execution times that barely differed from the 32-bit bus. The 64-bit bus results are not shown for that reason. Figure 15 shows the relationship between execution time and system frequency. It turns out that every n-core system tested benefited most greatly when frequency was at 2.0GHz because increasing frequency after that point did not produce such dramatic results; other n-core systems besides the 2-core will not be shown. That specific frequency is common of consumer CMPs and therefore would probably not be a big power or heat problem.

5.2.4 Cache Coherency Tests

The cache coherency protocol was tested with 32kB L1, 4MB L2, 2.0GHz frequency, and a 32-bit bus. MOESI, MESI, and MSI protocols were tested. MSI keeps track of modified, shared, and invalid states. MESI keeps track of an extra exclusive state, and MOESI keeps track of an extra "occupied" state. Fig. 16 shows the results of these tests. MOESI and MESI performed exactly the same, and MSI performed .011 seconds slower. This may be explained by the theory that extra state in MOESI and MESI helps provide more efficient snooping. It is also possible that M5 is not producing valid results because it did not produce results at all for many protocol simulations. Only results from the one and two core machines were found, and four, eight, and sixteen core machines did not produce results.

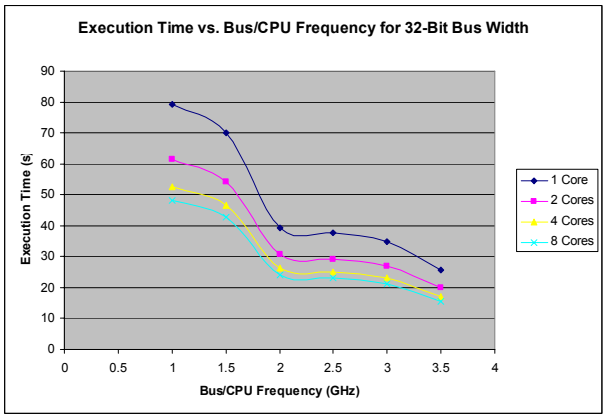

Fig. 14

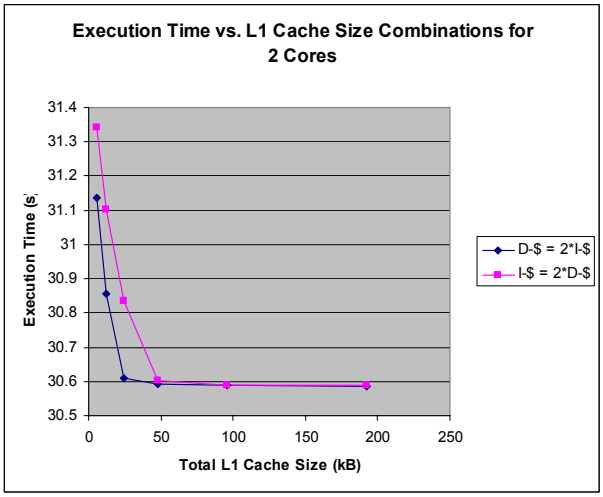

Fig. 15

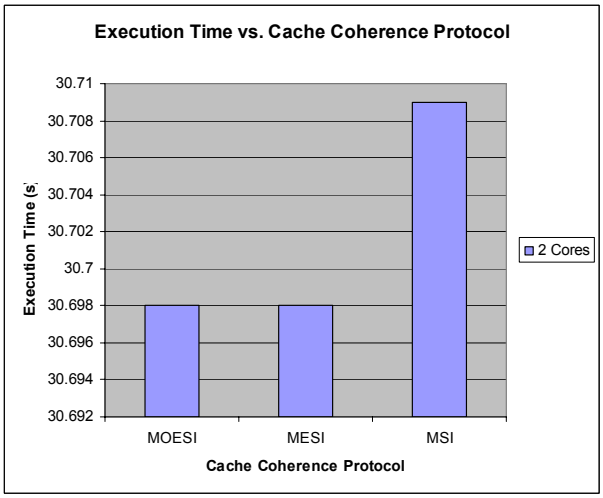

Fig. 16

5.2.5 L1 Data and Instruction Cache Size Combination Tests

Now with L1 cache size, L2 cache size, bus width, and bus frequency chosen for best gains with 32kB, 4MB, 32bits, and 2.0GHz respectively, combinations of L1 cache sizes were chosen. In other words, the project aimed to see what would happen when the instruction and data caches were not held at the same size. In Fig. 15, the blue graph represents a combination where the data cache has twice as much memory as the instruction cache. So, for example, on this line a total L1 cache size of 48kB would mean that the data cache has 32kB and the instruction cache has 16kB (because data cache = 32kB = 2*I-$ = 2*16kB). The pink line is when the instruction cache is double the size of the data cache. Similar results were found for all n-core systems, so the graph for 2-cores is shown to reduce redundancy. It was found that when the data cache was twice as large as the instruction cache the system produced faster execution times than when the instruction cache was twice as large as the data cache. This implies that either data caches are more important for system performance or maybe that the program performs many operations that requires the reuse of many previously used operands.

The number of cores for the optimized configuration was determined to be around 2-4 cores as seen in Fig. 17. Two to four cores seems to produce the best relative speedup. This can be explained by the extra overhead needed with more cores. However, this speedup does not seem as high as expected. This could be due to an imperfect simulation system or maybe the program was not run long enough to extract the full parallel aspects of the program.

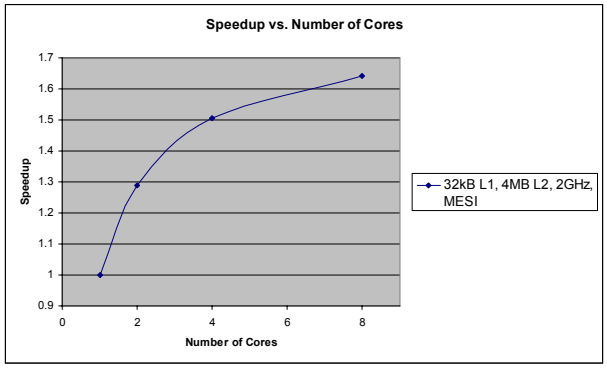

Fig. 17

6. Conclusions

This research intended to find the relationships between the memory system and performance in both single core and multi-core context. For the single core part, several parameters have been considered to improve the performance. Based on these work we extend to the multi-core issues. Overall, the simulations showed results similar to configurations of many current consumer CMPs. Many results were expected, like a large L2 cache size certainly helped performance a lot and the L1 cache size, optimal system frequency was typical of many of the Intel Core 2 Duos, and that after already having 2-4 core any more cores do not help performance in FFT nearly as greatly. However, some results were more surprising like the low speedups that the n-core systems presented. Also, some of the experiments failed, like for 16-core systems and for cache coherence tests. Overall, the excess of 500 simulations recorded helped understand that in multi-core systems, communication overhead and memory latencies are a limiting factor in performance. Finding good cache configurations certainly help increase performance by taking the pressure off of the main memory and reducing communication and cache coherence latencies.

7. ACKNOWLEDGMENTS

The authors would like to thank Meng Jiayuan for his kind help to us with M5..

8. REFERENCES

[1] [1] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishing Co. 1996.

[2] C. Zhang, F. Vahid and R. Lysecky, “A self-Tuning Cache Architecture for Embedded Systems”, in *Proc. of DATE*, 2004.

[3] C.Zhang, F. Vahid, W. Najjar, “A Highly Configurable Cache Architecture for Embedded Systems”, in *Proc. of International Symposium on Computer Architecture*, 2003.

[4] S. Dropsho et al., “Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power”, in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[5] D. H. Albonesi, “Selective cache ways: On-demand cache resource allocation”, in *Proc. of International Symposium on Micro-architecture*, 1999.

[6] M. Powell A. Agaewal, T. Vijaykumar, B. Falsafi and K. Roy, “Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct Mapping”, in *Proc. of International Symposium on Micro-architecture*, 2001.

[7] Cai, Y., Schmitz, M., Ejlali, A., Al-Hashimi, B. and Reddy, S, “Cache Size Selection for Performance, Energy and Reliability of Time-Constrained Systems,” in*11th Asia and South Pacific Design Automation Conference* (*ASP-DAC* 2006).

[8] Frank Schirrmeister, “Multi-core Processors: Fundamentals, Trends, and Challenges’, *Embedded Systems Conference* 2007 ESC351, Imperas, Inc.

[9] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny, “The power of priority: Noc based distributed cache coherency,” *NOCS*, 2007.

[10] Frederic Petrot, Alain Greiner, and Pascal Gomez, “On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures,” *Digital System Design* (*DS*D), 2006.

[11] W. J. Dally and B. Towles, “Route packets, not wires: On-chip interconnection networks,” in *DAC-38*, pages 684.689, 2001.

[12] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese.

Piranha, “A scalable architecture based on single-chip multiprocessing,” in *ISCA-27*, 2000.

[13] R. Kumar, V. Zyuban, and D. Tullsen. Exploring interconnections in multi-core architectures. Technical report, University of California, San Diego, 2005.

[14] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Single-ISA Heterogeneous Multi-core Architectures,” The Potential for Processor Power Reduction. In *MICRO-36*, Dec. 2003.

[15] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *IEEE NorChi*p Conference, Nov. 2000.

[16] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. Technical Report CSTR-2007-1605, University of Wisconsin-Madison, Aug 2007.

[17] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in future multicore systems. In Proc. Of 16th PACT (Poster), 2007.

[18] E. ´Ipek, M. Kirman, N. Kirman, and J. F. Mart´ınez, “Core fusion: accommodating software diversity in chip multiprocessors,” in *Proc.of 34th ISCA*.

[19] K. Chakraborty, P. M. Wells, and G. S. Sohi,“Computation spreading, “Employing hardware migration to specialize CMP cores on-the-fly,” in *Proc. of 12th ASPLOS*, 2006.

[20] K. Chakraborty, P. M. Wells, and G. S. Sohi, “A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs,” Technical Report CS-TR-2007-1607, University of Wisconsin-Madison, Aug 2007.

[21] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, “A novel simd architecture for the Cell heterogeneous chip-multiprocessor. In *Hot Chips* 17, 2005.

[22] Muti-Core Processors—The Next Evolution in Computing http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf

[23] Intel@ Multi-core technology, http://www.intel.com/multi-core/

[24] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance (2004),” *IEEE ISCA* '04.

[25] http://tbp.berkeley.edu/~jdonald/research/hyperthreading/mitchell_tlp.pdf

[26] http://m5.eecs.umich.edu.