Accelerating Multi-core Simulators

Aparna Mandke Keshavan Varadarajan Amrutur Bhardwaj
Y. N. Srikant

IISc-CSA-TR-2009-10
http://archive.csa.iisc.ernet.in/TR/2009/10/

Computer Science and Automation
Indian Institute of Science, India

October 2009

Accelerating Multi-core Simulators

Aparna Mandke Keshavan Varadarajan
Amrutur Bharadwaj Y. N. Srikant

Abstract

Simulation is an important means of evaluating new microarchitectures. With the invention of multi-core (CMP) platforms, simulators are becoming larger and more complex. However, with the availability of CMPs (i.e. host/hardware processor) with larger caches and higher operating frequency, the wall clock time required for simulating an application has become comparatively shorter. Reducing this simulation time further is a great challenge, especially in the case of multi-threaded workload due to indeterminacy introduced due to simultaneously executing various threads. In this paper, we propose a technique for speeding multi-core simulation. The model of the processor core and cache are replaced with functional models, to achieve speedup. A pre-constructed timed Petri net helps to estimate the execution time of the processor and the memory access latencies are estimated using hit/miss information obtained from the functional model of the cache. This model can be used to predict performance of data parallel applications or multiprogramming environment on CMP platform with various cache hierarchies and shared bus interconnect. The error in estimation of the execution time of an application is within 6%. The speedup achieved ranges between an average of 2x-4x over the cycle accurate simulator.

1 Introduction

Microarchitects develop and use simulators to evaluate their ideas. Developing a simulator requires careful trade-off between speed and accuracy. Making these simulators faster has always been a topic of interest among researchers. Lot of research has been done to make simulation of single processor platform faster. Performance prediction of an application can be obtained quickly by creating analytical or statistical models of the processor

[11], [20]. However, these models are not flexible. These models fall into different category than simulation based models.

Simulators can be classified into two basic types, namely: Compiled Simulators and Instruction Set Interpreter based simulators (ISS).

1.1 Simulation Speedup in Compiled Simulators

Miles et al. [4] proposed static compiled simulator, where application binary is translated into C code that is then compiled with the simulator to achieve speedup over ISS. Lei Gao et al. [9] used this technique to get faster simulator for the multiprocessor platform. They use hybrid simulation technique in which they simulate target independent part of the application code on the native simulation host and target dependent part of the code on ISS. They have done this for Multiprocessor System-On-Chip (MPSoC) environment with up-to 8 DSPs. Their accuracy is within 3% and average speedup achieved is between 3x-5x over the detailed simulator. The same technique improves single processor simulation speed by 10-50x. This demonstrates that improving speedup and at the same time retaining good accuracy is not an easy task in case of a multiprocessor. The presence of synchronization variables in multi-threaded applications makes it more complex to strike a balance between speedup and accuracy.

1.2 Simulation Speedup in ISS Simulators

Traditional simulators like SESC [18], Opal [14] etc. fall under the category of ISS simulators. Sherwood et al. [19] achieve simulation speedup for uniprocessors by fast-forwarding simulation based on the phase behaviour of an application. These simulators (SimPoint) simulate a single interval from each phase and fast-forward the rest of the intervals in the same phase. Applying the same technique is not very straight forward for multi-threaded applications. In [17], Erez et al. have explored phases in parallel programs for executions with 2 and 4 threads. They have implemented SimPoint technique for parallel programs. Their results show that the clock cycles per instruction (CPI) error is 15% or less, with an average error of 3% for executions with 2 threads and lesser error for 4 threads execution. Accuracy and speedup gains are not known when the same technique is applied to execution with 8 or more number of threads.

Small CMPs can be analyzed using cycle-accurate simulators like SESC[18]. However, the running time of such simulator becomes large, for bigger CMP designs. Hence an alternative speedier approach is required for design space exploration. Further, with the proliferation of multi-cores, multi-threaded

applications are becoming the norm. In this paper, we propose a technique to make simulations faster through the use of functional components i.e. independent of time. The execution time is estimated, unlike ISS simulators where this time is computed through the execution of instructions in a step-by-step manner. Some research has already been done which allows user to explore memory hierarchy for multi-cores. The authors in [20, 7] have proposed analytical models of overall system for shared memory multiprocessors/CMPs. However, one analytical model cannot cover all the diverse options for the memory subsystem. A lot more can yet be done to achieve higher speedup and accuracy.

We estimate the execution time required for the multi-threaded application on varying the number of threads. The results are presented for different applications from the SPLASH suite[21] for different number of threads (1 to 32 in power of 2). As per our knowledge, ours is the first paper which attempts to estimate the performance up-to 32 threads of multi-threaded application. Rest of the papers, either deal with multi-programming applications[3] or have dealt with up-to 4[1] or 8 threads[9].

We have made the following assumptions in this work.

- Core has in-order execution.
- Each thread executes on a separate core.

Mobile PCs, mobile Internet appliances and other embedded platforms have low power requirements. These platforms and newer larger CMPs with many cores prefer processors with in-order execution, e.g., Intel Atom, VLIW processors like Trimedia TM32. In [8], Erik Tol et al. map H.264 decoding on four VLIW multiprocessors with shared memory architecture. Hence our first assumption does not limit the usability of our framework. Our second assumption is in-line with other recent work done on CMP simulation[16],[1].

The rest of the paper is organized as follows. Section 2 describes the abstract implementation of the core. Section 3 describes implementation of our framework. The experimental setup and results are described in section 4. The error analysis is presented in section 5. Related work and the direction of our future work are discussed in sections 6 and 7 respectively.

2 Abstract Implementation of the Multi-Core Platform

To estimate the time required for the complete execution of an application, we need to find the time spent by an instruction in every pipeline stage

and also in the memory subsystem if it is load or store. In case of parallel programs, we have to estimate stalls due to synchronization variables as well. To find out the time spent in pipeline stages, we model the program executed by the thread on a core as a *Timed Petri-net*. Following subsection explains this in greater details.

2.1 Abstract model of a core

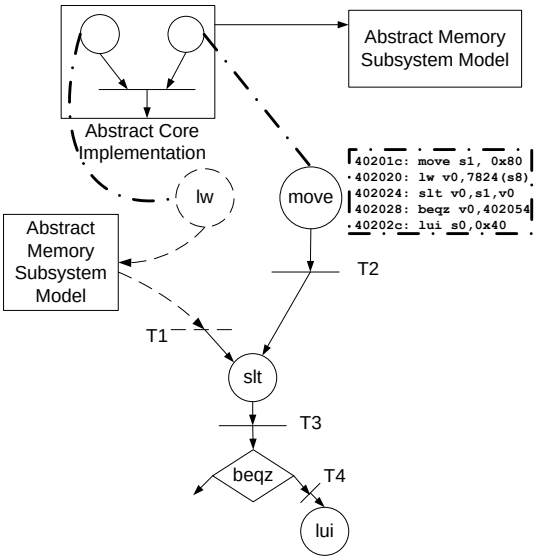

Figure 1: **Figure shows the abstract implementation of the core and the memory subsystem. Core executes *Timed Petri net* model of the program representation.**

A program can be viewed as a collection of instructions executed within a core [*internal events(IE)*] and references done to the memory subsystem, which are visible out of the core[*external references(ER)*]. These *ERs* are either data region accesses due to load/store instructions or instruction fetches done by the core. Hence a core can be visualized as an entity which emits out *ERs* to the memory subsystem at specific time instants. This can be captured appropriately in *Timed (place/transition) Petri Net Model* [15]. *Timed Petri nets* assign *firing times* to the transitions or places of a net. Tokens are removed from the input places at the beginning of the *firing period*, and they are deposited to the output places at the end of this period. We model a program execution as a *timed Petri net* and assign the *firing time* to the transition. Figure 1 shows a piece of assembly code. The load instruction is an example of *ER* which is fed to the memory subsystem. *Firing time* of the transition T_1 will be decided by the memory subsystem. It depends upon the state of the cache and resource contention of the shared interconnect. It is

shown with *dashed lines* in the figure whereas, other *places* and *transitions* in the *timed Petri net*, dependent on the core and program dependencies, are shown with the solid lines. *Firing time* and *Firing period* of these transitions are decided by the nature of the core and program dependencies. In our implementation, the core calculates the time at which instruction enters execution stage and completes execution. It *neither performs actual execution of the instruction nor it models the complex pipeline stages or various queues or registers*. It just takes their effect into an account while calculating the time when an instruction starts and completes execution. To get the above *IE* and *ER* events, we use compiled instruction interpreter as the front-end and pass these events for timing estimation to our abstract core model. Each abstract core executes *timed Petri net* model on these events.

2.2 Data Dependency Analysis

Instructions can be marked executed only after its source instruction completes execution. As an example, consider an *IE-ER* trace in Figure 2. Type (IE/ER) of each node will be decided by the instruction type. (Here nodes for instruction fetches are not shown separately.) Instructions in the same row are issued in the same cycle by the core if there is no data dependency (Assuming the core has an issue-width of 4). Data dependency between instructions I_{11} and I_{21} is denoted by a directed edge. The instructions I_{11}, I_{12}, I_{13} and I_{14} go into the execution stage simultaneously at t_1. However, the instruction I_{21} goes into execution stage only after I_{11} completes execution. Since we consider in-order execution, I_{22} has to wait along with I_{21}. Both these instructions go into execution stage simultaneously after I_{11} completes execution at t_2. This delay is caused due to instruction dependency. If I_{11} is a load instruction and if it is a cache hit then I_{21} waits for an additional time of cache hit latency. On the other hand, if I_{11} incurs a cache miss, then I_{21} experiences a longer delay in scheduling. Thus, we need to distinguish between such instruction dependency and cache related dependency.

This data dependency information (edges in *IE-ER trace*) can be determined statically using assembly code and reaching definitions [2]. Each assembly instruction can be considered as a program statement and a register as an input/output variable. With the reaching definition algorithm, we can determine all instructions defining a source register in an instruction. However, this yields an approximate and conservative information. A register may be set by some instruction in some other procedure. Doing such inter-procedural analysis[2] is very complex and the call graph of functions

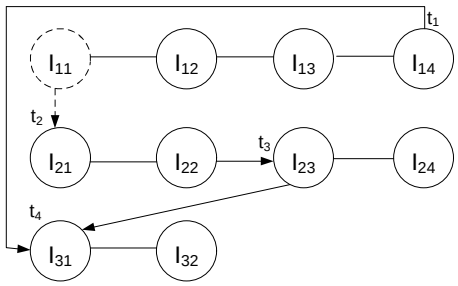

Figure 2: ***IE-ER* Graph. Dependency between different nodes are marked with directed edges. I_{21} can go into execution stage after I_{11} completes execution.**

cannot be constructed accurately except at run-time. We overcome these problems by doing dependency analysis at run-time using simulation. We collect dependency information during the simulation with ideal cache and interconnect configurations. This gives us data dependencies between instructions (dependencies between *IEs* or *IE-ER* pair). However, since we set cache latencies to zero in the *ideal memory trace*, all dependencies on load instructions (between *ER-ER* pair or between *ER-IE* pair) may not be captured. To capture these dependencies, we set L1 data cache hit and miss latencies to some large value (approximately equal to the latency of memory) and then perform a one-time simulation of an application with two threads. Dependency information obtained during this simulation can be used for an execution with any number of threads and with any memory configuration. Paths traversed during the execution with two threads include all the paths traversed by all the threads if the same application-input pair is executed with a different number of threads. This dependency is more accurate for a given input of the application than statically calculated information.

Thus to summarize,

- The data dependency on the load instructions is computed only once on a cycle-accurate simulator by executing the application with two threads, for a given input of the program and with very large hit/miss delay values of L1 data cache and once with ideal cache and interconnect configuration. It is reused for all the other executions with different number of threads.

- Front-end instruction emulator gives *IE*/*ER* events and abstract core model calculates timing information for these events.

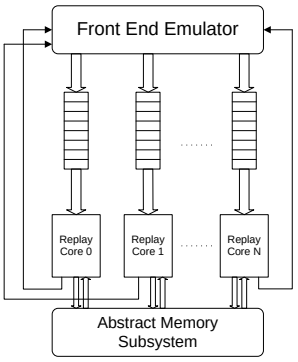

Figure 3: **Front-end emulator executes application binary and generates traces for respective replay cores. Replay cores issue instruction and data references to the abstract cache simulator.**

3 Framework Implementation

The block diagram of our framework is shown in Figure 3. Following section describes each component in details.

3.1 Instruction Emulator

Instruction emulator is the front end of our framework. It executes compiled instructions on the native host and generates traces for these instructions. This trace is passed through the buffer to our abstract core model. Core estimates time when all these instructions will go into the execution stage. It then releases this buffer back to the emulator. This method slows down the front-end and we do not have to make provision for large buffers between front-end and the replay core as done in[16]. The number of such buffers maintained is equal to the sum of maximum number of instruction bundles issued for fetch and the pipeline length of the abstracted core. The number of instructions contained in every buffer are either equal to the fetch width of the processor or lesser if branch predictor predicts branch mispredictions in the current fetch bundle.

In case of branch misprediction, front-end emulator generates separate event and the replay core stops releasing buffers back to the front-end emulator till the marked branch instruction completes execution.

Functional emulator ensures synchronization variable semantics in the application is honoured. Front-end spins to acquire the lock, till lock is available and all these instruction traces generated during spinning are passed to the replay core for timing simulation. The difference between the number of instructions executed by the reference simulator and by our framework is

less than 3% for applications using synchronization variables.

3.2 Replay Core

Replay Core replays its own *IE-ER* trace. It feeds instruction or data addresses to the cache model at the estimated time. It uses dependency annotations in the trace to calculate the time at which each instruction goes into the execution stage and completes its execution. Each data reference in the trace is either a data read or data write. Every instruction belongs to a class depending on its execution stage latency. Table 1 gives different instruction classes and latency of each class. In SESC simulator, each instruction waits for nine clock cycles which includes the delay required for instruction decoding and register renaming. We add this delay to every instruction after it is fetched.

Table 1: **Execution Unit Latency. These are used to calculate *firing period* for *IE* transitions.**

Class Type	Latency
Integer operations	8
Branch	8
Float addition,subtraction	8
Float multiplication	9
Float division	17

These latencies are specific to the SESC simulator, and will vary depending on the processor being used. In addition to the above latencies, instruction does not go into the execution stage, if its data dependencies are not satisfied.

3.3 Abstract Model of Memory Subsystem

We have implemented fast cache model which takes memory references and returns hit/miss information. It also returns latency incurred for each memory access. We have been able to avoid traditional cycle-accurate implementation of the memory subsystem due to following reasons:

- Cache configuration on the multi-core platform is not "strictly" inclusive. Hence cache evictions do not take place in upper levels of cache (e.g. L1 cache) due to conflict misses in lower levels of the shared cache. Cache line will get evicted in L1 cache only in case of a coherence miss.
- Most of the accesses are hits in L1 cache. Hence timing information returned by our abstract cache model is accurate enough to estimate the total time taken by the application to complete its execution.

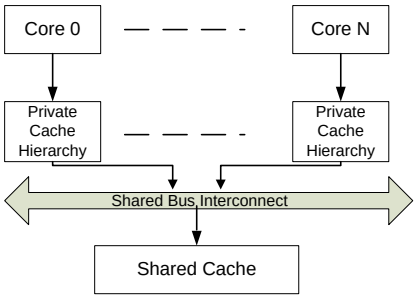

Figure 4: **Current framework simulates caches with different sizes and levels of hierarchies. Our framework considers shared bus interconnect.**

Our implementation can be considered as extension of Dinero[6] type of trace-driven cache model. Dinero gives cache hit/miss information for uniprocessor environment. However we have attempted to estimate the same for multiple threads and multi-core environment. In addition to that, our model also returns latency associated with each memory access.

We do not model miss status handling registers(MSHR)[12]. We also do not model cache coherency protocol, since most of the data parallel applications divide data equally among the threads and these threads work on their portion of the data. Synchronization variables are the only frequently accessed shared variables among these threads. The number of accesses to these variables is negligible compared to the total memory references. This enables us to avoid cycle-accurate simulation of cache coherence messages.

Our framework can simulate caches with different hierarchies and sizes as shown in the Figure 4. We have assumed shared bus interconnect in this work. For other types of interconnects, separate abstract model should be created to estimate the contention and delay incurred at different links. Our framework can estimate performance of data parallel applications which divide data among threads. This includes applications which distribute loop iterations among different threads. It can be used to predict the performance of multiple programs as well.

4 Experimental Setup and Results

We compare our results against the results obtained on the SESC cycle-accurate simulator. Table 2 describes applications used for simulation. MPGEnc and MPGDec are from Alpbench[13] and do not employ any synchronization constructs. The remaining applications are from Splash-2 [21] benchmark suite and use synchronization constructs. FFT and RADIX are cpu bound applications. Only 25% of the instructions in FFT and Radix

are memory access operations, whereas in the case of LU close to 50% of the instructions are load/store instructions. Ocean uses 5 mutexes and 13 synchronization barriers. It makes approximately 52,000 synchronization calls with 32 threads. For all these applications, we skip initialization and then simulate them to completion. Table 3 gives the processor specification. Our

Table 2: **Application Specification**

Application	Description
LU	Continuous 256x256 Matrix, B=16
LU	Non-Continuous 256x256 Matrix, B=16
MPGEnc	128x128, 3 Frames
MPGDec	128x128, 3 Frames
FFT	64K Complex Data points
OCEAN	Continuous 256x256 grid
RADIX	256K keys

Table 3: **Processor Specification**

Processor	
Execution	In-order
Frequency	3GHz
Branch Penalty	8 cycles
Branch Unit	Alpha style Hybrid
Fetch/Issue/retire	4/4/5
Issue Queue Size	8
INT registers	96
FP registers	80
ROB Size	176
LdSt/Int/FP units	2/2/3

framework has been evaluated for four different cache configurations as shown in Table 4. L2_1048 is the base configuration and rest of the configurations show variations applied to this base configuration. We have performed our experiments on a PC with 2GHz dual core Intel Xeon processor with 6MB cache. We have taken readings for issue-width of 4 and varied the number of threads from 1 to 32 in power of 2.

4.1 Evaluation of Simulation Speedup

The results of our experiments are presented in Figures 6–11. The results show the percentage error recorded with respect to the reference simulator. The speedup achieved for the various configurations is also presented. LU (Continuous and Noncontinuous) show higher average speedup within the range of 3x-4x. The speedup achieved can be accounted to the memory bound nature of these applications. Also these applications have a longer simulation time compared to the other applications. The average speedup for the other applications is in the range of 2x-3x. Clearly, our approach gives

Table 4: **Cache/Interconnect Test Configurations**

BASE Configuration - L2_1048

IL1	32KB,2-way,2 ports
Hit/Miss lat	1/1
ITLB entries	64
DL1	32KB, 4-way,2 ports,WB
Hit/miss lat	2/2
DTLB entries	64
Coherence protocol	MESI protocol
Unified, shared L2	1024KB 8-way, WB, 2ports
Hit/Miss latency	9/11
Interconnect	shared bus interconnect
L1L2DBus	1 port, 1 cy. port occupancy
Memory Hit/Miss latency	469/469

L3

DL1	Private, 32KB, 4-way,2 ports,WT
Hit/miss lat	2/2
L2	Private,512KB,8way WB, 2ports
Hit/Miss latency	7/9
Coherence protocol	MESI protocol
L3	shared,1024KB,8way WB, 2ports
Hit/Miss latency	9/11

L2_2048

Unified, shared L2	2MB,8way
Hit/Miss latency	10/12

L1

DL1	64KB, 8way
Hit/miss lat	2/2

better speedup for long running applications. This speedup is dependent on memory configuration used. We see lesser speedup for the configuration with shared L3 cache. This is because, due to larger private caches, most of the cache accesses are cache hits which reduces simulation time in the reference simulator as well.

4.2 Evaluation of Execution Time of Application

Error in estimation of execution time of applications like MPGEnc, MPGDec and LU is independent of the number of threads. In case of Radix, this error increases on increasing the number of threads. The execution time of Radix decreases nearly linearly with an increase in the number of threads. The execution time drops nearly 25x when the number of threads is increased from 1 to 32. The percentage error that is seen with single threaded execution is amplified with increase in the number of threads due to decreased execution time of the application with 32 threads. Despite of this, the percentage error is below 5%.

In case of FFT, a drop in percentage error is seen with an increase in

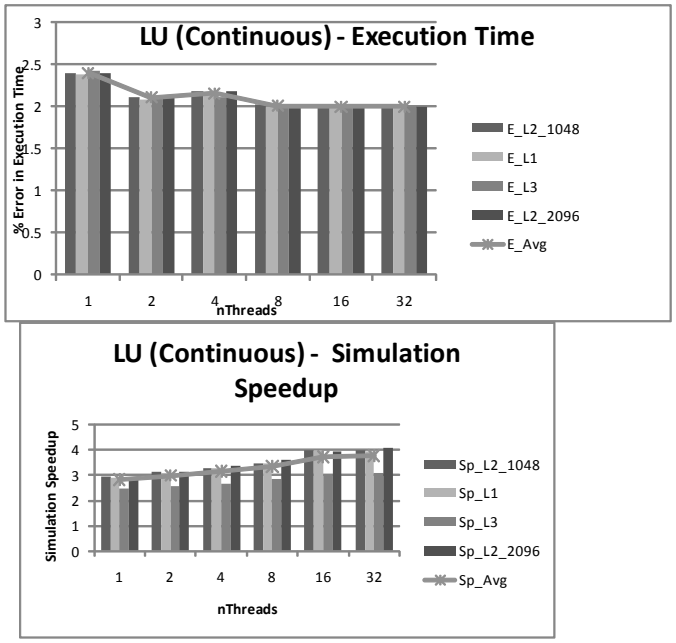

Figure 5: **First graph shows % error in estimation of Execution Time of LU (Continuous). Next graph shows simulation speedup obtained over the reference simulator by our framework.**

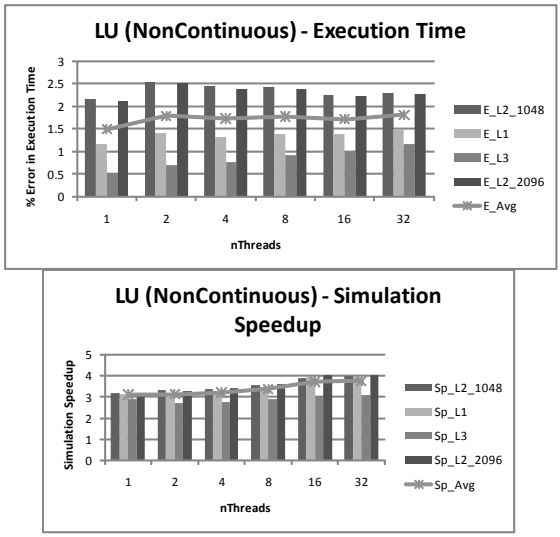

Figure 6: **First graph shows % error in estimation of Execution Time of LU (NonContinuous). Next graph shows simulation speedup obtained over the reference simulator by our framework.**

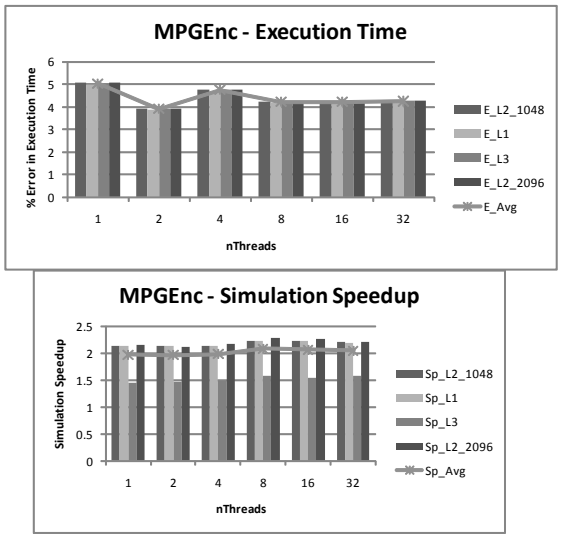

Figure 7: **First graph shows % error in estimation of Execution Time of MPGEnc. Next graph shows simulation speedup obtained over the reference simulator by our framework.**

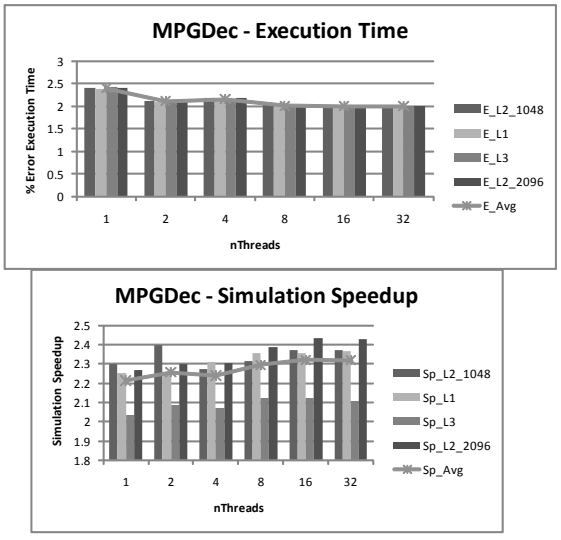

Figure 8: **First graph shows % error in estimation of Execution Time of MPGDec. Next graph shows simulation speedup obtained over the reference simulator by our framework.**

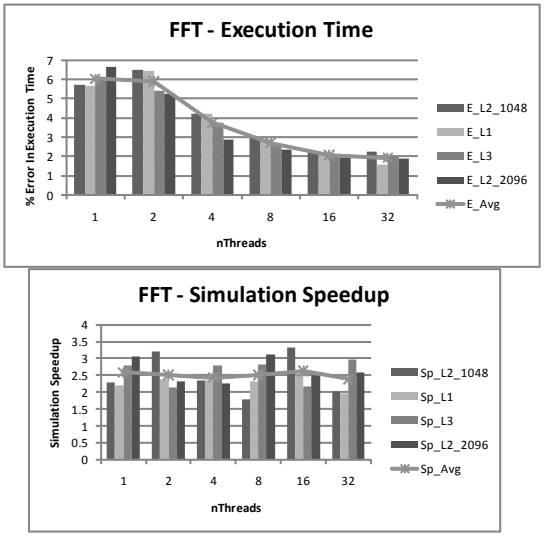

Figure 9: **First graph shows % error in estimation of Execution Time of FFT. Next graph shows simulation speedup obtained over the reference simulator by our framework.**

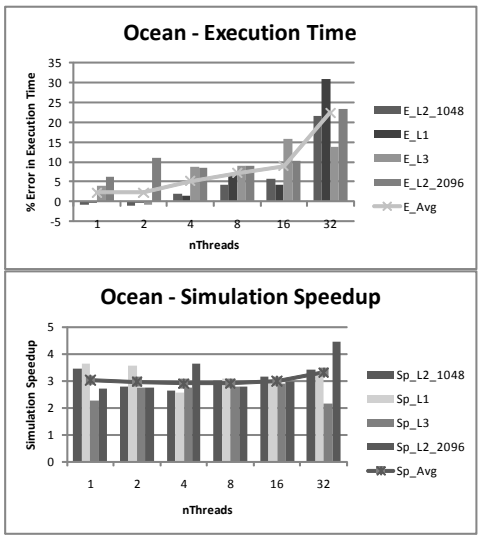

Figure 10: **First graph shows % error in estimation of Execution Time of Ocean. Next graph shows simulation speedup obtained over the reference simulator by our framework.**

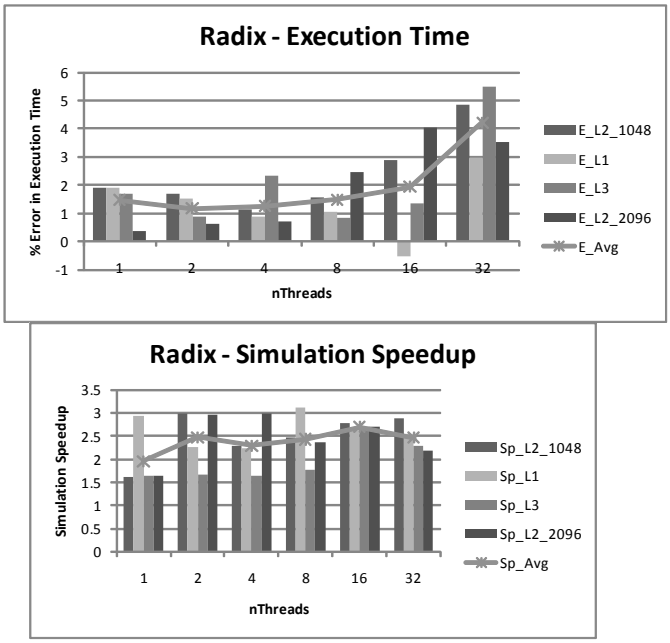

Figure 11: **First graph shows % error in estimation of Execution Time of Radix. Next graph shows simulation speedup obtained over the reference simulator by our framework.**

the number of threads, which seems quite counter intuitive. For a single threaded execution, error in estimation of execution time is an average of 6% and goes down to 2% for 32 number of threads. This is because, FFT with a single threaded execution has 2.74% of data miss rate. So for every data miss, it accesses shared interconnect. Contentions at the shared interconnect are not modelled in our framework. We assume contention to be negligible compared to cache latencies. The number of shared interconnect accesses remain the same, even on increasing the number of threads but this delay gets overlapped with other memory accesses. So we ignore contention and do not add it to the shared cache latency.

Above mentioned factor accounts for the large percentage in error seen in Ocean. Error in estimation of execution time is very large in case of Ocean which is an average of 22% for 32 threads. Ocean makes approximately 52000 concurrent synchronization variable calls which triggers lot of cache coherence messages. This causes contentions at the shared interconnect. When readings are taken for the reference simulator with an ideal interconnect, this error drops to an average of less than 4% for 32 threads.

5 Error Analysis of the Framework

Simulation of an multi-threaded application is inherently non-deterministic on varying memory subsystem configuration. We have attempted to predict the execution time of such non-deterministic applications. In our framework, error can be introduced due to various reasons such as,

- Abstraction of the core
- Abstraction of the memory subsystem

5.1 Discussion

The time at which an instruction enters the execution stage depends upon contention generated at various core components like unavailability of ROBs, instruction queue buffers etc. By not modelling core components like load/store queue sizes, the number of ports of load/store queue, size of ROBs, size of the instruction queue etc, introduces error in calculating the time at which an instruction goes into the execution stage and time at which *external references* are made to the memory subsystem. Our reference simulator does not execute instructions from mis-speculated path. It only adds misprediction penalty to the cycle count.

Another source of inaccuracy in our framework is due to abstraction of the memory subsystem. We have not implemented MSHR in our cache model but reference simulator has MSHR at all cache levels. All references waiting in MSHR in the reference simulator will add this waiting period in addition to hit latency. But in our model it returns only hit latency. This introduces error in the estimation of execution time. We do not model cache coherency. Therefore estimation of miss latency will not be very accurate. But overall error is still within 6% of an average.

6 Related Work

In [20], Daniel Sorin et al. have carried out analytical modeling for shared memory multiprocessors. This model supports homogeneous applications, i.e. application memory requests are equally distributed across the relevant memory modules. However, if the cache size is changed, the simulator needs to be re-run to estimate some of the application parameters used in the analytical model. In [10][7], Engin et al. have used artificial neural networks to develop the analytical model for exploring CMP design space.

HySim[9] alternates between native execution of the application and detailed simulation on ISS to achieve speedup over the detailed simulation. In [17], Erez et al. detect phases in parallel application on the shared memory architecture and use it to guide simulation at those points. In [3], authors use co-phase matrix to find individual program phases and guide simultaneous multi-threading simulation in case of multi-program environment. In [16], Matteo et al. has built framework to simulate splash-2 programs on 1024 in-order cores CMP platform. Their focus of study is on the characterization of splash-2 programs so they do not model delays caused by the memory subsystem and use ideal memory and single cycle processor functional unit latencies in their experiments.

In[5], Penry et al. show automated parallelization of CMP simulators and achieve speedup of 7.6x for a 16-processor CMP model on conventional 4-processor shared memory multiprocessor.

7 Our Contributions and Future Work

- We have implemented abstract models of the core and memory subsystem. Our experiments show that for most of the data parallel applications user need not capture the exact time at which memory reference arrives at the shared caches since most of the data accesses are first level cache hits. Approximate estimation of cache hit or miss and hence latency of the memory accesses can be used to estimate the execution time of the application.

- We differentiate instructions executed by the core as *external references* and *internal events*. Petri-net modelling of a thread execution allows us to avoid modelling of pipeline stages, load-store queues and ROBs. Our framework achieves average percentage error below 6% and an average simulation speedup ranging from 2x-4x.

- We find out data dependencies between the instructions at runtime. So we can estimate more accurate program dependencies than usual static time analysis methods[2].

In the future, we plan to explore various cache configurations simultaneously in a single simulation. We also need to improve the accuracy of the interconnect model to account for contentions. Our experiments show that even on abstracting both core and memory model, we could achieve maximum speedup of up to 5x over the cycle-accurate simulator. Greater speedup can be achieved by making simulator multi-threaded. We plan to explore this possibility in future.

References

[1] R. S. C. Aamer Jaleel. Cmpsim: A pin-based on-the-fly multi-core cache simulator. *Workshop on Modeling, Benchmarking and Simulation*, 2008.

[2] R. S. Alfred Aho and J. Ullman. Compilers: Principles, techniques and tools.

[3] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 45–56, Washington, DC, USA, 2004. IEEE Computer Society.

[4] M. Chrystopher, A. Stanley, and F. jim. Compiled instruction set simulation. *Software, Practice and Experience*, 21(8), 1999.

[5] P. David, F. Daniel, H. David, W. Ryan, S. Graham, A. David, and C. Dan. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. *High-Performance Computing Architecture*, 2006.

[6] J. Edler and M. Hill. Dinero trace-driven uniprocessor cache simulator.

[7] S. A. M. Engin Ïpek, Bronis R. An approach to performance prediction for parallel applications. *International Euro-Par Conference*, 2005.

[8] R. G. Erik B. van der Tol, Egbert Jaspers. Mapping of h.264 decoding on a multiprocessor architecture.

[9] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM.

[10] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGOPS Oper. Syst. Rev.*, 40(5):195–206, 2006.

[11] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. *SIGARCH Comput. Archit. News*, 32(2):338, 2004.

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[13] M.-L. Li, R. Sasanka, S. A.-K. Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *IEEE International Symposium on Workload Characterization*, 2005.

[14] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[15] P. Marwedel. Embedded system design. Springer International Edition.

[16] M. Monchiero, Ahn, J. Ho, Falconi, Ayose, Ortega, Daniel, Faraboschi, and Paolo. How to simulate 1000 cores. *dasCMP Workshop*, 2008.

[17] E. Perelman, M. Polito, J. yves Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, 2006.

[18] J. Renau, B. Fraguela, J. Tuck, W. Lui, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Struss, and P. Montesinos. Simulator for cmp architecture.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[20] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ilp processors. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 380–391, Washington, DC, USA, 1998. IEEE Computer Society.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.