A Statistical Performance Model of the Opteron Processor

Jeanine Cook
New Mexico State University
Klipsch School of Electrical and Computer Engineering
jcook@nmsu.edu

Jonathan Cook
New Mexico State University
Department of Computer Science
joncook@nmsu.edu

Waleed Alkohlani
New Mexico State University
Klipsch School of Electrical and Computer Engineering
wkohlani@nmsu.edu

ABSTRACT

Cycle-accurate simulation is the dominant methodology for processor design space analysis and performance prediction. However, with the prevalence of multi-core, multi-threaded architectures, this method has become highly impractical as the sole means for design due to its extreme slowdowns. We have developed a statistical technique for modeling multi-core processors that is based on Monte Carlo methods. Using this method, processor models of contemporary architectures can be developed and applied to performance prediction, bottleneck detection, and limited design space analysis. To date, we have accurately modeled the IBM Cell, the Intel Itanium, and the Sun Niagara 1 and Niagara 2 processors [34, 33, 10]. In this paper, we present a work in progress which is applying this methodology to an out-of-order execution processor. We present the initial single-core model and results for the AMD Barcelona (Opteron) processor.

1. INTRODUCTION

The complexity of contemporary processor architectures in combination with the large slowdowns of cycle-accurate simulators has given rise to the need for alternate performance analysis and prediction techniques that enable faster analysis and identification of performance bottlenecks. As noted recently in a panel discussion on computer architecture research directions, “Intel cycle-accurate performance simulators currently run at roughly 10 thousand instructions per second (KIPS) when simulating a unicore processor. At 10 KIPS, it will take more than nine days to simulate 1 second of the eight-way multicore processor execution. Things will only get worse as the number of simulated cores grows.” [17]. This time perturbation makes simulating a realistic workload unrealistic!

Another motivation for alternate performance analysis techniques is the lack of availability of simulators for modern multi-core architectures. A limited number of academically-available cycle-accurate simulators have various levels of multi-core simulation support, the most popular being SESC [31], m5 [11], Simics/GEMS [25], and more recently MARSSx86 [5]–the only one supporting the popular x86 architecture. Some of these have not been validated against real hardware in the multi-core mode (SESC and M5), and the others that are validated show large errors of up to 50% [5]. All cycle-accurate multi-core simulators are very slow; a few hundred thousand simulated instructions per second is the most optimistic speed with the help of native mode execution. Further, the architectures supported by many of these simulators are now obsolete.

To address these issues and to satisfy our own desire for performance models of contemporary processors, we developed a statistical modeling technique based on Monte Carlo methods that we believe can be applied to in-order and out-of-order execution, multi-core processors. We introduced this method in [34, 33]; the method was presented in detail using the Niagara 2 model as an example in [10]. These previous models were all of in-order execution processors. The model presented here of the Opteron is the first we are developing for an out-of-order execution architecture.

Monte Carlo methods are a class of algorithms that rely on repeated random sampling to compute a result. Our processor modeling technique probabilistically represents the executed instruction stream and randomly samples several histograms that are related to stall conditions. Instructions proceed to specific delay centers (e.g., execution units, cache, memory) based on probabilities. Cycles are counted for each instruction according to the stall conditions and the latency of the specific delay center that executes the instruction. The model converges and produces a result when it reaches a threshold based on the difference in CPI between consecutive intervals of a pre-defined size. All of the in-order execution models converge within seconds; the current Opteron model usually converges within seconds, but for some applications, convergence takes minutes.

New processor models using the Monte Carlo modeling technique can be developed quickly. A basic model can be created in a few months; validating and re-developing for better accuracy takes another few months. The newer models (Niagara and Opteron) are coded in C++; the older models are written in C. The Opteron model is the largest at approximately 2000 lines of code; all of the other models are implemented in hundreds of lines.

In this paper, we present the initial Monte Carlo single-core processor model that we are developing for the Opteron, Barcelona architecture. The Barcelona is a Family 10h architecture; we model the micro-architecture core of Family 10h processors. We extended the methodology to enable modeling of out-of-order execution, which represents a significant extension in comparison to prior models with respect to the micro-architecture components modeled and overall model complexity. The Opteron model is parameterized and generalized so that it is not just an attempt to model an existing processor, but can be configured to understand the performance tradeoffs when alternatives are evaluated.

Our effort is not yet complete but is producing initial results. In this paper we present initial performance prediction results. Although we have performed some initial validation and model refinement, we are still working on areas where the model is inaccurate, mostly due to iteratively understanding the details of available documentation as they pertain to actual processor behavior. The results are not final and, from experience, are expected to improve as validation is completed.

2. MONTE CARLO PROCESSOR MODELING (MCPM) TECHNIQUE

The Monte Carlo modeling methodology is a technique that statistically models processor behavior and predicts the performance of contemporary, single- and multi-core processors. Our statistical modeling technique is based on the Monte Carlo method and uses both dynamic application and processor characteristics as parameters to the model. Models are developed based on a particular existing processor and can subsequently be modified to model a different processor in the same family.

The models predict CPI (cycles per instruction) and use the equation:

$$CPI = CPI_i + CPI_s \tag{1}$$

where CPI_i is the intrinsic or ideal CPI based on the issue width, and CPI_s is the CPI due to stalls. The cause of stalls depends on the micro-architecture; data dependences, branch mis-predictions, and cache misses are stall causes common to many processors. Stalls due to a full reorder buffer, full reservation stations, and full scheduling queues are also common in out-of-order execution processors. Once stall causes are identified, we collect data on their frequency using on-chip performance counters (branch mis-predictions, cache misses, TLB misses) and binary instrumentation tools (data dependences). The cache and transition lookaside buffer (TLB) hit/miss data is used to probabilistically determine (realized as a transition probability) where in the cache hierarchy a memory reference instruction is satisfied. The branch mis-prediction rate is used to statistically determine whether a branch is predicted correctly or not.

Latency characteristics for various processor components such as caches, execution units (EUs), and branch predictors are also used in the model. These are primarily obtained from the processor hardware manual and verified using micro-benchmarks. In cases where latencies are not given in the manual or when component latencies are variable, we use targeted micro-benchmarks to determine these numbers. Targeted micro-benchmarks are short-running programs designed to extract a particular latency or to study a specific behavior or micro-architecture feature. The data dependence histograms are sampled by the model to determine the latency of subsequent dependent instructions.

Data dependence information is collected according to which type of data hazards can occur in the micro-architecture: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR). WAW and WAR hazards are typically resolved through register renaming. Therefore, in most models, only true dependences (RAW) need to be accounted for.

The data dependence information is represented as a distance in number of instructions between the operand producing and operand consuming instructions. For example, the distance between the producing and consuming instruction in the following sequence is 3.

ldub $[\%i4 + \%i2], \%i0$
nop
nop
or $\%i0, 2, \%i0$

These dependence distances are collected over the entire dynamic execution of an application and are used to produce a histogram. For the Opteron model, a histogram is generated for each executed instruction; the histogram contains a distribution of the number of dynamic instructions that separate the producing instruction (e.g.,movq %rax, -8(%rbp)) and its consumer. Figure 1 shows the MOV64-to-use his-

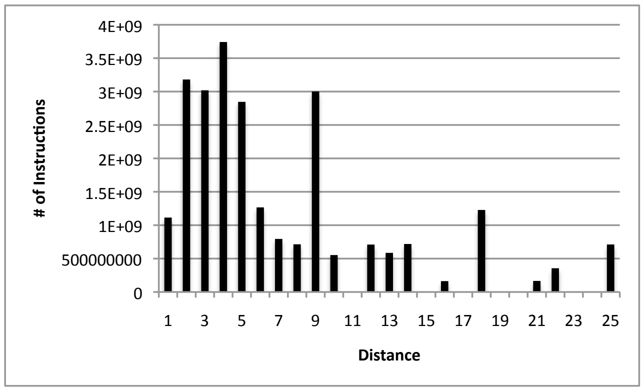

Figure 1: MOV64-to-use Histogram, *astar*

togram for the SPEC CPU2006 [4] INT benchmark, *astar* . The figure shows that the instruction dependent on a producing *mov* instruction will most likely be 4, 2, or 3 instructions, respectively, following the producing instruction.

The dynamic instruction mix and any data associated with variable latency micro-architecture components is collected using binary instrumentation. The instruction mix is used to generate an instruction stream for the model that statistically represents the instruction stream generated by the application.

Monte Carlo processor models converge to a predicted CPI based on a pre-defined threshold value. Models not only predict the CPI of an application, but they can be used for some design space analysis. A configuration file is associated with each model that defines the micro-architecture to be modeled/simulated. The latency of various components, the size of queues, and the number of execution units are just some of the characteristics that can be varied on the model by changing the configuration file.

3. THE OPTERON FAMILY 10H PROCESSOR

The Opteron family 10h processor is AMD's 64-bit architecture. There are many 10h processors with the same base core architecture, including the Barcelona, Shanghai, and Istanbul. We use a Barcelona processor (model 2352) for obtaining model data and performing validation. This is a quad-core processor, with a 2.1GHz clock. The single-core micro-architecture is shown in Figure 2.

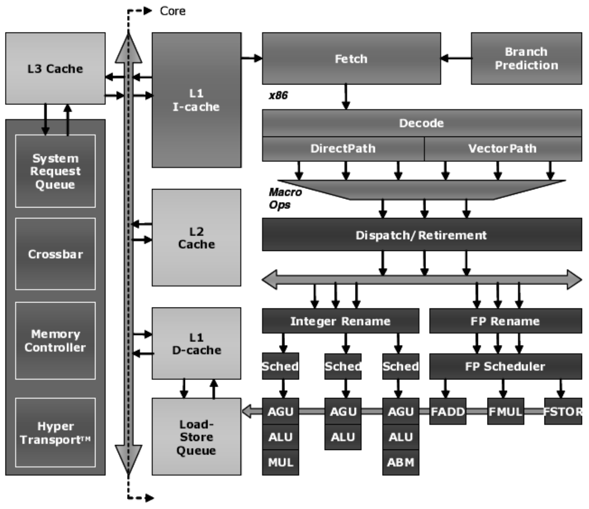

Figure 2: Opteron Core Micro-architecture (from [8]).

The AMD64 instruction set is implemented using macro-ops and micro-ops. Macro-ops (variable length) are the primary units of work and are broken down into smaller operations, called micro-ops (fixed length), that are executed in the processor's execution units. The AMD Family 10h processors are out-of-order, three-way superscalar and can fetch, decode, issue, and retire up to three AMD64 instructions (macro-ops) per cycle [8].

There are three different instruction decode categories that are associated with a specific latency. Direct Path Single decodes the less complex instructions that can be handled by the hardware as a single operation and has the smallest relative associated latency. Most basic arithmetic and logical instructions and move (*mov*) instructions are Direct Path. Direct Path Double instructions are more complex and are typically broken into two independent instructions. Vector Path decodes the most complex instructions by invoking a micro sequencer that executes a micro-code program.

The instruction control unit takes the three macro-ops that are produced during each cycle from the early decoders and places them in a centralized, fixed-issue reorder buffer (ROB). This buffer is organized into 24 lines of three macro-ops each. The reorder buffer allows the instruction control unit to track and monitor up to 72 in-flight macro-ops (integer or floating-point). The instruction control unit can then simultaneously dispatch multiple macro-ops from the reorder buffer to four schedulers, three integer, and one floating-point. These schedulers can simultaneously issue up to nine micro-ops to the three general-purpose integer execution units (ALUs), three address-generation units (AGUs), and three floating-point execution units [8].

Each integer scheduler is a reservation station that is eight entries deep, for a total queuing system of 24 integer macro-ops. The floating-point scheduler handles register renaming and has a dedicated 42-entry scheduler buffer organized as 14 lines of three macro-ops each. Each reservation station divides the macro-ops into integer/floating-point and address generation micro-ops, as required [8].

The integer execution units have a single-cycle latency and instructions can issue to these units every cycle. The floating-point execution units all have a four-stage pipeline. Many of the FP instructions are allowed to issue to an FP unit every cycle. However, there are also many FP instructions that have issue latencies varying between one issue every two cycles to as high as one instruction issue every 17 cycles [8].

The Barcelona has three levels of cache. The level one data and instruction caches are split; the L2 and L3 are unified. The cache and TLB specifics are listed in Table 1. The L2 cache has an exclusive architecture, which means it serves as a victim buffer for the L1 caches. The L3 cache is considered a non-inclusive victim cache architecture and it is optimized for multi-core AMD processors. Blocks are allocated into the L3 on L2 victim/copy-backs. Requests that hit in the L3 cache can either leave the data in the L3 cache - if it is likely the data is being accessed by multiple cores - or remove the data from the L3 cache (and place it solely in the L1 cache, creating space for other L2 victim/copy-backs), if it is likely the data is only being accessed by a single core.

Component	Opteron
L1 I-Cache	2-way assoc, 64KB,32-byte lines
L1 D-Cache	2-way assoc, 64KB, 64-byte lines, 8-way banked
L2 Cache	8-way assoc,512KB,64-byte lines
L3 Cache	32-way assoc, 2MB
L1 ITLB	fully assoc, 32 entry
L1 DTLB	fully assoc, 48 entry
L2 ITLB, DTLB	4-way assoc, 512 entry
Exe Units	3 ALUs, 3 AGUs, 3 FPUs

Table 1: Opteron Barcelona (10h) Micro-architecture Components

The AMD 10h processors implement a two-level translation lookaside buffer structure. There are two levels each of instruction and data TLBs. Specifics for these structures are also listed in Table 1.

The AMD 10h core has two load/store units. One is a pre-cache unit (LS1) that holds memory instructions that have been dispatched from the ROB that are waiting for their address to be generated in an AGU. When a memory access is dispatched to an integer queue, it is also dispatched to one of the 12 entries in the LS1. Load and store instructions will access the L1 cache from the LS1. On a hit, a load instruction will read the data. However, store instructions will not write until they retire in the ROB. If a memory access misses the cache while in the LS1, it is moved to the 32-entry post-cache (LS2) to wait while the access is satisfied. All memory accesses wait in the LS2 until retirement.

4. OPTERON SINGLE-CORE MODEL

The Opteron Monte Carlo processor model is shown in Figure 3. The model comprises a token generator, a dependency tracker, the load/store queue, the reorder buffer, FP and INT instruction queues or reservation stations, execution units, and a cache and memory model.

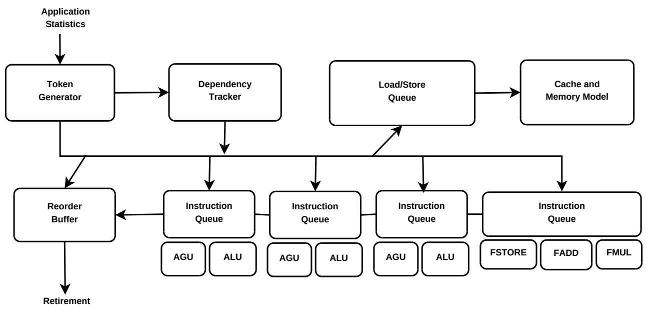

Figure 3: Opteron Single-Core Model

The *token generator* probabilistically generates instructions (i.e.tokens) based on the application instruction mix obtained through dynamic binary instrumentation. The instruction mix contains the frequency of all completed instructions and for each instruction, the probability that an operand reads (load) or writes (store) memory is also obtained. Once the token is generated and whether its operands read or write memory is known, it is assigned a unique ID representing its sequence order and a latency according to the *instruction definition table*. This table contains each instruction in the AMD64 ISA and several fields describing the instruction, the most important including the decode path, execution units used, latency, and throughput [8, 3, 1, 2]. A portion of this table is shown in Table 2. The token is matched to an entry in this table and is subsequently tagged with the proper execution unit and is assigned a latency and throughput. This information is used later when the token is dispatched to an instruction queue. Tokens are generated/fetched at a rate of three per cycle.

Insn	DecodeU	ExeU	BaseLat	MemLat	Thruput
CMPPD	single	FADD	2	2	1/1
CMPPS	single	FADD	2	2	1/1
MOVAPD	double	FSTORE	2	0	1/1
MOVAPS	double	FSTORE	2	0	1/1
DIVSD	single	FMUL	20	2	1/17
SQRTSS	single	FMUL	19	2	1/18

Table 2: Example Instruction Definition Table

Tokens are generated according to the instruction cache hit rate and the behavior of the memory hierarchy behind it. If the instruction fetch simulation indicates that fetch stalls are occurring, no tokens are generated until the cycle where they will be available.

After generation, latency and throughput look-up, the tokens proceed to the *Dependency Tracker*. The primary job of this unit is to keep a list of the producer/consumer relationships between executing tokens. When a token is generated, the Tracker examines the current dependence list to determine if there are any in-flight tokens that produce operands for the currently generated token. If the current token has any in-flight producers, those producers are connected to the current token so that it only executes after all its operands are ready. Next the Tracker randomly samples a dependence distance histogram for instruction type of the current token, which gives the distance in number of instructions of what future instruction will use this token's produced value. This new dependency is inserted into the Dependency Tracker so that the future token will be able to detect its incoming dependencies when it is generated. This dependence information is used in the *Instruction Queues* to determine the cycle in which tokens can be issued to execution units.

From the *Dependency Tracker*, tokens are dispatched in order to the reorder buffer (ROB). Three tokens per cycle can be dispatched to the ROB. Before tokens are actually dispatched, availability in the appropriate instruction queue or load/store queue is checked. If the appropriate queue for any of the three tokens is full, dispatch to the ROB will be stalled until the queue has an available entry.

Once tokens are in an instruction queue, they are issued to an execution unit. The heart of the instruction execution simulation is pulling instructions from the queues and allowing them to occupy the execution units for their required time. We do not directly simulate micro-ops, but we indirectly simulate them by allowing different instructions in the integer instruction queues be independently dispatched and occupy the ALU and AGU units.

Each instruction type has two numbers that characterize its use of execution units: its throughput and its latency. These are needed because the execution units have internal pipelines. The throughput number indicates how soon another instruction can be issued to the execution unit; the latency indicates how long it takes the instruction to finish its computation in the execution unit. These are needed for the inherent execution stage of each instruction, but the address generation stage, where the instruction uses an AGU, is always characterized by a 1/1 throughput/latency.

Many floating point instructions can utilize any of the three FP execution units, while others need to use one specific execution unit. In addition the three integer ALU's are not exactly equivalent, and certain instructions can only use one of the integer execution units, meaning it must be dispatched to that integer queue only.

Once a token is in an instruction queue, it is ineligible to proceed to execution units until its data dependencies are complete, at which point it is eligible to be dispatched to execution units. If it accesses memory it needs to use an AGU unit first to generate an address, after which it can be dispatched to its required execution unit. For tokens that are doing a memory load, the load/store queue is prevented from performing the load until the address is generated, and then the instruction cannot proceed to perform its computation until the load is complete.

Once a token has completed its use of execution units, it is marked as completed and the reorder buffer can commit the instruction. If an instruction is a branch instruction, we use the measured branch misprediction rate for the application being simulated to randomly mark it as mispredicted. When a branch completes, if it was mispredicted the reorder buffer cancels all outstanding instructions that have already been decoded and dispatched (and might be potentially executing).

5. APPLICATION CHARACTERIZATION

Since our technique is a statistical simulation of the processor, it depends on the characteristics of a particular workload type whose performance on that processor is of interest. Thus we collect and create statistical profiles of benchmark applications in order to drive the model simulations. The data for these profiles comes from two sources: binary-instrumented executions and processor performance counter data.

5.1 Binary instrumentation

We use the Pin tool [7] to instrument application binaries to collect instruction mix and dependency information. From the data collected we get a basic instruction mix distribution for the application, and also dependence distance information per instruction type, both for register usage and for memory usage. For a window of instructions our instrumentation remembers the addresses of the loads and stores that are performed, and we check any current access against that window and can thus calculate the dependence distance between memory accesses as well as register usage dependencies. This information is written out as an application profile.

5.2 Performance counter data collection

Using hardware performance counters on the same benchmark applications we also collect data regarding the architecture's behavior on that application. This is mainly memory hierarchy statistics: miss rates for all levels of cache and TLBs, and branch mispredict rates. Because we do not attempt to model some pattern of memory accesses, we simplify memory behavior into just the miss rates and the instruction dependency information.

6. EXPERIMENTAL METHODOLOGY

All experiments (model parameter collection and executing the model) are performed on a 10-node heterogeneous cluster. There are eight Opteron-based nodes on this system, each comprising two quad-core Opteron 2352 processors. The system runs the Linux operating system, 2.6.28.

We use the Opteron on-chip hardware performance counters to collect the cache, TLB, and branch predictor statistics used in the model. For this, we use the *papi* tool [6]. When collecting performance counter data, we do multiple runs and measure the variance in event counts; we take an average of these multiple runs for a particular event as the event count.

To capture dynamic application characteristics such as the instruction mix and dependence distance histograms, we use the *PIN* binary instrumentation tool [7]. When collecting dynamic application data, we run in single-user mode and do not run more than one process on a single core. We typically bind user processes to core 3 in each quad-core, since OS processes typically execute on core 0.

We report results on ten SPEC CPU2006 benchmarks [4]. We chose a subset of the SPEC CPU2006 benchmarks based on [29]. Four SPEC benchmarks are integer (INT): *perlbench*, *astar*, *libquantum*, *xalancbmk*; six are floating-point (FP): *cactusADM*, *calculix*, *lbm*, *deal*, *povray*, and *leslie3d*. For the SPEC benchmarks, we use the training input set size due to resource and time constraints.

We validate our model against the measured CPI obtained from cycle and retired instruction performance counts. We collect the data from the counters multiple times to reduce the variance; we execute the model with each application's model parameters (i.e., data gathered from performance counters and binary instrumentation tools) multiple times to ensure the variance is negligible.

7. RESULTS

The results we present here are still preliminary. The model is in the process of being further developed to increase accuracy.

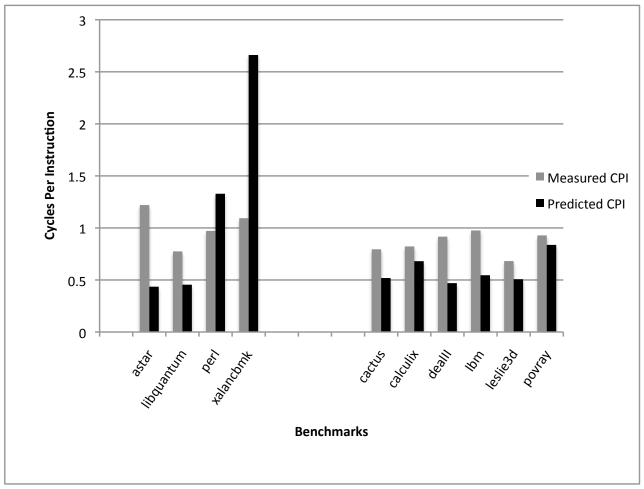

Figure 4: Measured vs Predicted CPI

Preliminary results are shown in Figure 4. As seen in the figure, the current predictions are not very accurate, ranging between 9.8 and 143% error. *Povray* has the smallest error, at 9.8%. *Xalancbmk* has the highest error. This benchmark also has a high branch mis-prediction rate and a high instruction cache miss rate relative to the other benchmarks. *Perl* also exhibits similar behavior and has a very high prediction error. This may be indicative of a problem in these components in the model.

We believe we understand some of the sources of error in our model. First, we have implemented an extremely optimistic fetch mechanism. In the Opteron processor, up to three instructions can be fetched per cycle. Our model fetches three instructions per cycle - always. From experience, we know that the maximum fetch width is not attained on every cycle. However, the performance counters on the Opteron do not provide an obvious way to measure this. We are currently studying the performance counter events to determine if we can indirectly measure or estimate the average number of instructions fetched per cycle.

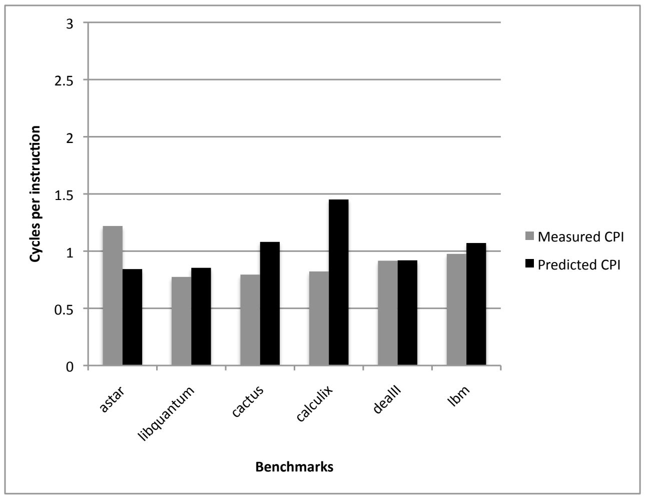

Figure 5: Measured vs Predicted CPI for Less Optimistic Fetch

To help determine if the fetch mechanism in the model could be too optimistic, we implemented a fetch that probabilistically determines the fetch width to be between zero and three instructions. Results for some benchmarks are shown in Figure 5. Although a less optimistic fetch width improved the prediction accuracy for some benchmarks, for others, the predictive accuracy degraded. We will investigate this issue further to determine whether the fetch mechanism is really modeled incorrectly.

We suspect that another source of error in the model may be the instruction latencies themselves. The latency for instructions was obtained from processor manuals [8]. We have not yet micro-benchmarked specific instructions to determine if the published latencies are correct.

8. RELATED WORK

Cycle-accurate simulators such as [38, 25, 31, 11] now provide support for CMP simulation. However, the run time for multi-core simulation increases with the number of cores. Cycle-accurate simulators exhibit other serious problems such as lack of validation and large prediction errors [35]. A number of approaches have been developed to address the large slow-down of cycle-accurate simulators. Many of them aim at increasing the simulation speed by reducing the number of simulated instructions while others, like ours, propose alternative methods to cycle-accurate simulation. These include reduced input sets [23], truncated execution [37], subsets of benchmark suites [30, 21, 28] and sampling [36, 32, 16]. Unfortunately, most of these methods compound the simulation error and they still end up using the cycle-accurate simulator [37].

Modeling is a proposed alternative approach to cycle-accurate simulation. This includes analytical models [13, 22], regression models [19, 24], and machine-learning-based models [20, 18]. Analytical models have only focused on modeling single superscalar processors. The other types of models usually require a large number of cycle-accurate simulation runs to arrive an at accurate model and usually employ execution-reducing methods such as sampling to lower the cost of running these simulations.

Statistical simulation [27, 12, 14, 15, 26] is the closest approach to our work. In this approach, the aim is to generate a small synthetic trace that is statistically similar to the full benchmark trace. The synthetic trace is then fed into a simple trace-driven simulator to generate the performance data. The first step is to compute a statistical profile of the application using cycle-accurate simulation– a modified version of *sim-outorder* [9] is used in all these experiments. This statistical profile is then used to generate the synthetic trace. This trace, in addition to synthetic instructions, contains information such as cache hit/miss probabilities and execution unit for each instruction.

The major disadvantage of these techniques is their dependence on cycle-accurate simulators to obtain the statistical profile and the difficulty in obtaining the needed data. Very fine-grained data need to be captured such as statistics for each basic block for every unique history of basic blocks, or statistics for every memory reference for every unique history of references, etc. Capturing such fine-grained data is impossible using the performance counters of modern processors, and requires the support of detailed simulators. Finally, using this method for design space exploration is difficult because a new statistical profile needs to be computed for every change in parameters related to the cache hierarchy, the branch predictor, or memory system. In contrast, input data is collected only once using our technique and since it only uses binary instrumentation, it is more readily applicable across architectures.

9. CONCLUSIONS AND FUTURE WORK

The Monte Carlo processor modeling technique has been applied successfully to accurately model in-order execution processors [34, 33, 10]. The Opteron performance model presented here represents a significant extension to the existing modeling method - this is the first out-of-order execution processor that we have modeled with this technique. We have presented initial results that show that the current model predicts performance well for only a small number of benchmarks.

There are specific components and features in the model that we will target to realize accuracy improvements. The first of these is the fetch mechanism, which we suspect does not match actual behavior in that it is too optimistic. We also suspect that some of the published instruction latencies may not be correct. From the benchmark instruction mix data that we collected during the model development process, we have identified the instructions that account for the largest percentage of the mix for all of the benchmarks. We will start by analyzing these and choosing from among these instructions to perform micro-benchmarking. Although we have micro-benchmarked the level 1 and level 2 data caches to determine access latency, we have not experimentally validated the published access latencies for the level 1 instruction cache, the level 3 cache, or memory.

10. ACKNOWLEDGMENTS

This work was supported through a PECASE award funded by Sandia National Laboratories, contract number 898047.

11. REFERENCES

[1] AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions. *http://developer.amd.com/documentation/guides.*

[2] AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions . *http://developer.amd.com/documentation/guides.*

[3] AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions. *http://developer.amd.com/documentation/guides.*

[4] CPU2006 benchmark suite. *http://www.spec.org/cpu2006/.*

[5] MARSSx86 - micro-architectural and system simulator for x86-based systems. http://www.marss86.org.

[6] Papi performance monitoring tool. *http://icl.cs.utk.edu/papi/.*

[7] The PIN tool. *http://rogue.colorado.edu/Pin/index.html.*

[8] Software Optimization Guide for AMD Family 10h Processors. *http://developer.amd.com/documentation/guides.*

[9] The SimpleScalar tool set. *http://www.simplescalar.com/.*

[10] W. Alkohlani, J. Cook, and R. Srinivasan. Extending the Monte Carlo Processor Modeling Technique: Statistical Performance Models of the Niagara 2 Processor. *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, September 2010.

[11] 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. *IEEE Micro*, 26:52–60, 2006.

[12] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. *SIGARCH Comput. Archit. News*, 32(2):350, 2004.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):1–37, 2009.

[14] D. Genbrugge and L. Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Trans. Comput.*, 57(1):41–54, 2008.

[15] D. Genbrugge and L. Eeckhout. Chip multiprocessor design space exploration through statistical simulation. *IEEE Transactions on Computers*, 58:1668–1681, 2009.

[16] G. Hamerly, E. Perelman, and B. Calder. How to use SimPoint to pick simulation points. *SIGMETRICS Perform. Eval. Rev.*, 31(4):25–30, 2004.

[17] J. C. Hoe, D. Burger, J. Emer, D. Chiou, R. Sendag, and J. Yi. The future of architectural simulation. *Micro, IEEE*, 30(3):8 –18, 2010.

[18] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. de Supinski, and M. Schulz. Efficient architectural design space exploration via predictive modeling. *TACO*, 4(4), 2008.

[19] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. pages 99 – 108, feb. 2006.

[20] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[21] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.*, 55(6):769–782, 2006.

[22] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. *SIGARCH Comput. Archit. News*, 32(2):338, 2004.

[23] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2002.

[24] B. C. Lee, J. D. Collins, H. W. 0003, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, pages 270–281, 2008.

[25] 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*, 33:92–99, November 2005.

[26] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[27] M. Oskin, F. Chong, and M. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor designs. pages 71 – 82, 2000.

[28] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:10–20, 2005.

[29] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *In ISCA Š07: Proceedings of the 34th Annual International Symposium on Computer Architecture, pages 412Ű-423, New York, NY, USA, ACM*, 2007.

[30] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 412–423, New York, NY,

USA, 2007. ACM.

[31] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[32] 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.

[33] R. Srinivasan, J. Cook, and O. Lubeck. Performance Modeling Using Monte Carlo Simulation. *IEEE Computer Architecture Letters*, 5(1):38–41, June 2006.

[34] R. Srinivasan, J. Cook, and O. Lubeck. Ultra-Fast CPU Performance Prediction: Extending the Monte Carlo Approach. *Proceedings of the IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 107–116, October 2006.

[35] V. M. Weaver and S. A. Mckee. Are Cycle Accurate Simulations a Waste of Time? *WDDD: Workshop on Duplicating, Deconstruction and Debunking*, June 2008.

[36] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. pages 84 – 95, jun. 2003.

[37] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society.

[38] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev. MPTLsim: a simulator for x86 multicore processors. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 226–231, New York, NY, USA, 2009. ACM.