

Is Queuing Model Good for Processor Fast Benchmarking?

Afzal Hossain

Nanova Corporation, Los Altos, California, U.S.A.

E-mail: afzal@nanova.com

ABSTRACT

Processor architecture simulations take days during design when silicon not available. Long simulation time is impractical, yet cost of early design mistake is high. Analytical models execute fast and recently introduced method named Fast Benchmarking produce performance results in milliseconds. With accurate analytical model, results produced by Fast Benchmarking are also accurate. For example, instruction fetch results differ by $\pm 7\%$ on average with simulations. This paper examines queuing model of a processor suitable for Fast Benchmarking. Compared to artificial neural network or simulation, the queuing model gives better insight about a micro-architecture design.

KEY WORDS: Processor, Architecture, Benchmarking, Modeling, and Simulation.

1. INTRODUCTION

Processor designers are often challenged with a maze of entangled and competing design goals. Expensive mistakes are made during the early stage of architecture definition and those mistakes are difficult to fix later. Simulation remains a primary tool to cope with the challenge. Benchmarking is a primary tool for studying processor performance. For example, the SPEC CPU2006 benchmarks stress processor, memory, and compiler performances. Timing-accurate architecture simulations can take even years. Hundreds of simulations are often needed to produce meaningful data, and analysis of massive amount of simulation data is a daunting task. Developing insight about a design to guide hundreds of simulations is even more challenging. The problem, frustrations, and the state of the art of architecture simulation can be found concisely in a special issue of IEEE Micro [1-5]. **Case study** [6]: Jane, a CPU architect, wants to know what size memory is ideal for a unified Trace Cache in a 4-thread SMT processor. She is using the CPU2000 benchmarks with 26 programs. Her research design requires 4 independent benchmarks as threads in SMT simulations. For the given workload, each benchmark has roughly 308 Billion dynamic instructions on average. Jane is using modified SimPoint with MRRL, with simulation time 2.25 hours per thread. Her exhibits require a graph showing ten data points establishing the vantage memory size. Jane wants to consider all possible combinations of threads and take averages. In addition, from past experience she knows, roughly 50% of all simulations may be wasted if she does nothing to prevent it, so she plans for prospecting. She wants to estimate simulation time. The number of ways 4 threads can be selected from 26 programs, ${}^{26}C_4 = 14,950$. For each combination she needs 10 simulations for 10 data points. Therefore, she needs 149,500 simulations without wasted runs. She has found a way to reduce wasted runs by selecting 4 arbitrary

threads for prospecting the vantage memory size. The prospecting involves 15 exploratory simulations. SMT simulations run slower compared to one thread. Each SMT simulation takes 4×2.25 hours or 9 hours. Therefore, the prospecting takes 15×9 or 135 hours of simulation. After prospecting she needs $149,500 \times 9$ hours for full simulation, that is about 153.6 years, which is impractical. So, she decides to simulate only 8 benchmarks, four each from FP and INT. This generates 8C_4 , or 70 simulations and requires 70×9 hours or 26.25 days on a single machine or 2.6 days on 10 distributed machines. Although sampling simulators have known accuracy issues and the benchmarks are partial, Jane, without other choices, accepts the compromise. The case shows that modest research goals can unearth large number of simulations and compromises. ||

1.1 Queuing Models

A queuing system consists of one or more servers providing services to arriving customers. When servers are busy, the customers wait in queues. Fig. 1 shows a single queue three-server system in abstract representation. Traditionally, queuing systems are characterized by three components: arrival process, service mechanism, and the queue discipline. Arrivals can happen from one or more customer sources. Customer population may be limited or infinite. The number of servers and the service time distribution of each server can describe the service mechanism. The queue discipline describes the rules by which servers select the next customer to serve. $[A/B/c];[d/e/f]$ is a popular notation for a queuing system, where A is the probability distribution of the inter-arrival time, B is the probability of the service time, c is the number of servers, d is the size of the queue, e is the size of the customer population, and f is the queue discipline. For example, a single server queue with exponential arrival and service times, unlimited queue size, unlimited customer population and FIFO rule can be denoted by $[M/M/1];[\infty/\infty/\text{FIFO}]$ or $M/M/1$ for short. $M/M/1$ queues are easy to analyze using Markovian birth-death process. Other queues are harder for manual analysis, but computer simulations can be used [7].

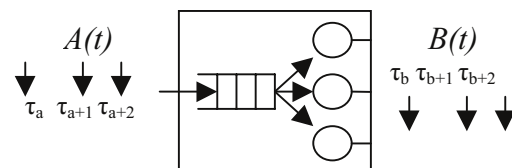


Figure 1: A three server queuing system.

A processor micro architecture can be modeled using queues and servers: where key hardware resources are represented as servers, buffers as queues, and the requests for resources as customers. A Functional Unit in the execution core of a processor is an example of a server. An Instruction Queue is an

example of a *queue*. An instruction is an example of a *customer*. There are significant challenges in the application of queuing theory for modeling micro architectures. Many queues do not follow particular queuing discipline. For example, instructions can be issued to the Functional Units out-of-order. Instructions are not fully independent from one another, so their arrivals are dependent on one another. Also, the service time of servers are not randomly distributed for all servers. For example, different functional unit may take different number of cycles to execute. The deviations from randomness impose difficulties in the application of queuing theory for modeling processor behaviors. Despite challenges, is queuing model a good design tool? Can it serve as a micro-architecture model in Fast Benchmarking [6]? This paper has a modest goal of introducing a new modeling technique.

1.2 Architecture Simulators and Related Works

The merit of queuing model is relative compared to the merits of other modeling choices. SimpleScalar is a popular architecture simulator: although the simulators are compiled themselves, instructions of the simulated architecture are interpreted. As a result, SimpleScalar is slow [8]. In contrast, hybrid compiled simulators can improve interpretive simulation performance by applying compiled simulation [9]. M5 provides a capability to simulate multiple systems in a network, but accuracy is costly in terms of modeling time and model execution time [3]. To avoid long run time, there are efforts to avoid full benchmark simulations by taking samples of executions. SimFlex, and SimPoint are two good examples [2,4]. To reduce randomness in SimPoint, machine learning can be used to pick representative samples [4]. Configurations of SimPoint can use the memory reference reuse latency proposed in [10]. On another front, *Virtualization* can be used to emulate new architectures. However, the tool is yet to be explored for application in architecture design [11].

1.3 Fast Benchmarking And Related Works

Despite recent advances in simulation, the need for accuracy, persistently long runtime, and the need for running hundreds of simulations remain mostly unaddressed. How to use massive amount of simulation data to make sense for design decisions remains another big challenge. An analytical approach has advantages in this area [12-14]. A recently introduced methodology, termed *Fast Benchmarking (FB)*, based on analytical models, can produce benchmark performance of a processor in milli-seconds without running extensive simulations [6]. Key benefits of Fast Benchmarking are the dramatic reduction in number of simulations, reduced runtime, and greater insight about a design. With accurate models, performance data produced by Fast Benchmarking is also accurate. Hence the FB

methodology is a compelling alternative to simulation only methodologies. Fast Benchmarking uses hierarchical models of a processor. It is easier to understand analytical modeling at lower component levels, where models are essentially algebraic equations of performance parameters. The bottom-up model development process can be summarized as following: start with a micro-architecture sub-system at lower level. A cache memory or a router in an on-chip network is a good place to start. Identify a list of parameters that describe the selected component physically and its performance. Next, come up with the performance metrics for the subsystem, but, use the measurements that are useful for describing benchmark performance. Next, write algebraic equations for each of the measurements.

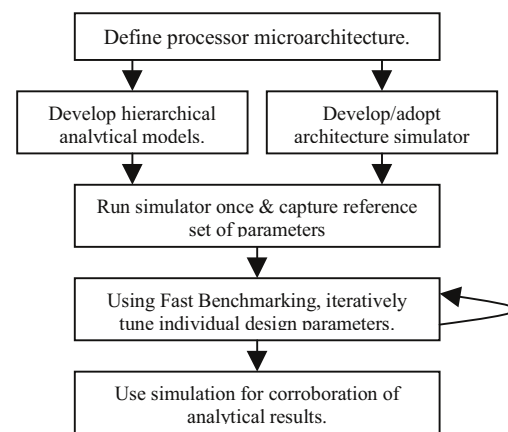


Figure 2: Fast Benchmarking process flow.

Once smaller subsystems are modeled this way, modeling moves to larger components where the smaller subsystem is a building block. Measurements from the lower-level blocks become input parameters at the higher levels. The *reference set of parameters* is a key concept in Fast Benchmarking. Through architecture simulation, preferable only once, a set of values of the model parameters are collected and referred as reference parameters. Then, an iterative process, while keeping all or most other parameters constant, tunes a particular parameter. Once an optimal set of design parameters is identified, using minimal architecture simulation is used to corroborate the performance predicted by FB. The process dramatically reduces the number of required simulations. The Fast Benchmarking process is shown in Fig. 2. There are many ways to model micro-architectures for FB [12-15]. Models based on artificial neural networks are presented in [14,15]. Rest of the paper is organized as following - Section 2 presents the modeled micro architecture. Section 3 presents the queuing model. Section 4 presents experimental setup. Section 5 concludes the paper.

2. REFERENCE MICROARCHITECTURE

We use a single-core SMT processor for

demonstration, which is shown in Fig. 3 [12]. The processor uses separate Instruction and Data Caches at Level 1. It has a unified cache for both instructions and data at Level 2. Notice that a Trace Cache augments the L1 Instruction Cache for better instruction fetch performance. The processor supports a fixed p number of threads in hardware. We have made following assumptions:

- Each thread has logically dedicated Trace Predictor, Trace Buffer, and Fill Unit.
- Each thread has dedicated read ports at TC and IC.
- Width of Trace Cache lines, Instruction Cache lines, and read ports are equal in number of instructions.
- Each thread has logically dedicated program counter and branch predictor.
- All threads share the Trace Cache memory, the Instruction Cache memory and the L2 Cache.
- All threads share the instruction issue logic and the functional units.

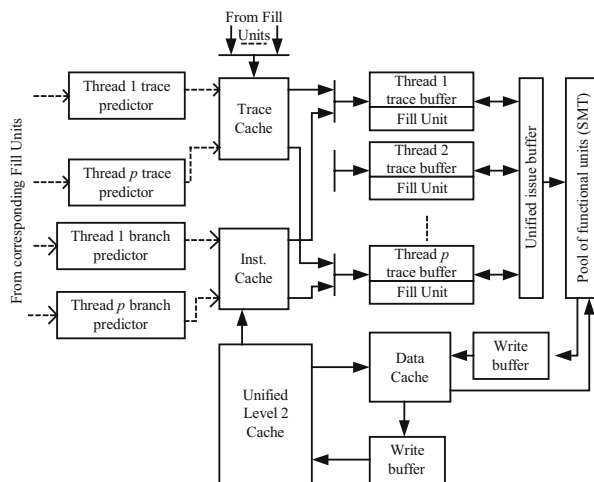


Figure 3: An SMT processor with Trace Cache

If there is only one read port for all threads, the fetch rate of the all threads combined is limited to the fetch rate from one thread alone. This is on the average seven instructions per cycle [12]. Therefore, there is a need for more than one read port for higher performance. The cost of supporting a thread in hardware in an SMT processor is high [16]. Increasing the thread to an arbitrary number does not improve performance due to contention at shared resources. The maximum number of supported threads in an SMT machine, which leads to optimal performance, is four to five [12]. Therefore, it is reasonable for each thread to have a dedicated read port. Finally, when our objective is to find the performance limit of the micro architecture, having dedicated port per thread is appropriate. Similar reasoning is used for dedicated branch predictors. The issue buffer is unified for all threads. This will bring all ready instructions in one queue so that both the instruction level parallelism of one thread and the process level parallelism among multiple threads contribute to higher utilization of the functional units. We have only one write port for the Trace Cache. Writing at the Trace Cache takes place after

fetching and completing a trace by the Fill Unit. The writes in a Trace Cache are much less frequent than the reads [12]. Therefore, for a limited numbers of supported threads, only one write port for the Trace Cache may be enough. The Trace Predictors, the Fill Unit, and the reading of the Trace Cache for all hardware threads are concurrently active.

3. THE QUEUING MODEL

We now present a Queuing Model for predicting performance of the reference micro-architecture of Fig. 3. The objective is to calculate the overall instruction execution throughput and to understand performance bottlenecks so that necessary action can be taken for improvements. The micro-architecture model is a high level concept and many of the input parameters are themselves functions of lower level component models. For example, the author presents models of Trace Cache miss rate and instruction fetch performance in [17,18]. Trace and Instruction Cache fetch rates, TCFR and ICFR respectively, as defined in [17] are used as input parameters in the micro-architecture queuing model. On one hand those are good examples of lower level models, on the other hand, parameter of those models are used in the current queuing model. Although dependant, detailed knowledge of these models is not required to understand the current paper. Input parameters used in the micro-architecture model are shown below:

p - Number of SMT threads in memory; p_{is} - Probability that an instruction is a memory access instruction; p_w - Probability that a memory access instruction is a write instruction; p_{dirty} - Probability that a replaced line in Data Cache is dirty; n_{st} - average number of memory writes that cause a store buffer flush to the Data Cache; n_{fu} - Number of functional units; y - Ratio of cache lines width to bus width; N - Number of instructions present in the microarchitecture in steady state execution; $ndcport$ - Number of front-end access ports of the Data Cache; $nl2port$ - Number of front-end access ports of the L2 Cache; h_{DC} - Data Cache hit rate (Average number of success per access); mr_{ic1} - Instruction cache misses per instruction.

Following intermediate terms are also used:

IPC - Overall instruction execution throughput of the microarchitecture, all threads combined; $PIPC$ - Instruction execution throughput of a thread; WBR - Data Cache requests due to processor write buffer flushes; RR - The rate of execution of instructions that are not blocked for cache accesses; $L2R$ - The rate at which instructions are blocked for the L2 Cache access; DCR - The rate at which instructions are blocked for the Data Cache access; $DCMR$ - The rate of unsuccessful access attempts at the Data Cache due to miss; $DCOTR$ - Open network component of the Data Cache traffic. (The open network traffic is introduced later); $L2ICR$ - L2 Cache requests due to misses at the IC; $L2DCR$ - Open network traffic due to write back of modified cache lines from Data Cache; $L2BR$ - L2 Cache open network traffic due to narrow bus width; $L2OTR$ - Open network part of L2 Cache traffic.

We make the following assumptions in the analysis:

- The ratio y of width of the cache lines to the width of the bus, which carries data between two caches, is the same for all cases. An access to Data Cache or L2 Cache line results in y number of transfers on the bus.
- Processor has write buffers. Write buffer is written to Data Cache only after certain number of writes.
- Write-back and write allocate on write miss policy is used for the Data Cache.

- d) Any functional unit can executing any instruction.
- e) The service time of a server has general probability distribution. The distribution coefficient of the service time of a server can be obtained by applying Khintchine-Pollaczek equations, where the mean and variance of the service time can be measured from simulations or using analytical models [7]. We also assume general distributions for the arrival rates of customers. FIFO discipline is used in all queues. We assume that the total number of instructions in the system is fixed to N , where N is large.

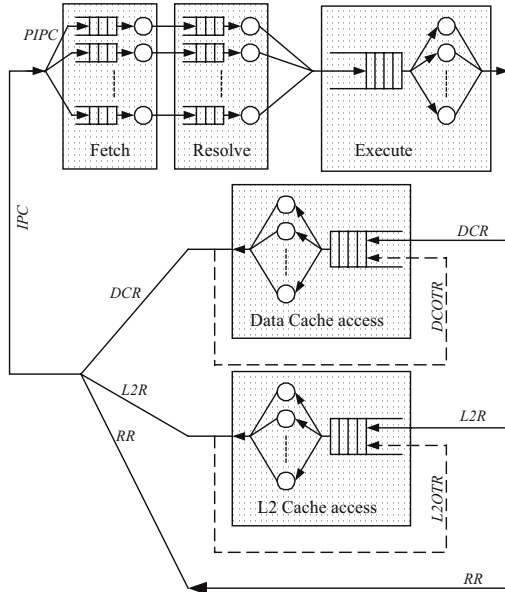


Figure 4. Micro-architecture Queuing Model

A mixed network queue model of the micro architecture is presented in Fig. 4. The closed part of the network contains a fixed number of instructions. This is the number of instructions present in the micro architecture in various buffers and functional units. In steady state, this is assumed to be a constant number N . The closed part of the network is the primary focus of our analysis. The average flow rate in the closed part of the network represents the processor's overall instruction execution rate. There are p number of single server queues in the Fetch section of the model. Each queue in the Fetch section incorporates the effect of the Trace Cache and the associated hardware dedicated to instruction fetching of a thread. It also incorporates the effect of the Instruction Cache access and cache miss penalties on the fetch rate. These queues also incorporate the penalties paid for branch miss predictions for each thread. For the Trace Cache microarchitectures, the service rate of these queues are related to the instruction fetch rate of the microarchitecture. For the microarchitectures where a Trace Cache is missing, the instruction Cache fetch rate, ICFR is related to the service rate of these servers. There are p number of queues in the Resolve section of the model, one for each thread. This is where instruction of each thread waits for available ILP of the thread, mainly due to dependencies on other instructions of the same thread. The Execute section of the model

has a single queue and multiple servers. When an instruction of a thread is clear of all dependencies it enters the *Execute* queue. There are n_{fu} number of servers each representing a functional unit. Some part of the closed network instruction traffic must go through the Data Cache. These are the instructions, which are blocked for Data Cache access. There is also an open network component of the Data Cache traffic. Occasional write buffer flushes, voided data cache accesses due to misses are the main source of the open network component of the traffic. The multiple servers at the Data Cache represent multiple read ports available to the processor. L2 Cache in the model is similar to the Data Cache model. Only instructions, which are blocked for L2 Cache access, go through closed network component of the L2 Cache traffic. Open network part of the traffic is caused by Data Cache write backs of dirty lines, Instruction Cache misses, and multiple reads or writes belonging to a single access. An access to L2 Cache causes y number of transactions on the bus. The open network traffics are shown with dotted lines in Fig. 4. No open network traffic enters the closed loop traffic of the network. The net impact of the open network traffic is the overall slowing down of the closed network traffic. Notice that both the open network and the closed network traffic are dependent on overall instruction execution rate, and are not independent of each other. The closed network traffic is fed back to the Fetch section of the queue model. This is a modeling technique, it represents that the buffers ahead of the Fetch engine must be free to fetch additional instructions in the system. In other words, the fetch rate need not exceed the overall execution rate. The instructions, which do not depend on the Data Cache access or the Level 2 Cache access are fed back to the Fetch section through a direct path. Table 1 presents the number of servers for each queue, mean arrival rates and mean service times of the servers.

Table 1. Queue parameters

Queue name	# servers, c	Arrival Rate	Service Time
Fetch	1	$PIPC$	$1/TCFR$
Resolve	1	$PIPC$	$1/Avg. ILP$
Execute	n_{fu}	IPC	Operation latency
Data Cache	$ndcport$	$DCOTR+DCR$	DC read hit latency
L2 Cache	$nl2port$	$L2OTR+L2R$	L2 read hit latency

Additional queue parameters used in the queue analysis algorithm are shown below:

WFetch [N]- Waiting time at Fetch queue with N instructions in the closed network; WResolve [N]- Waiting time at Resolve queue; WEX [N]- Waiting time at Execute queue; WDC [N]- Waiting time at Data Cache queue; WL2 [N]- Waiting time at L2 Cache; X[N]- Instruction throughput at the closed part of network.

$X[N]$ = IPC; ESFQ- Expected service time at Fetch queue. $ESFQ = 1/TCFR$, or $1/ICFR$; ESRQ- Expected service time at Resolve queue; ESEQ- Expected service time at Execute queue; ESDC- Expected service time at Data Cache queue; ESL2- Expected service time at L2 Cache queue; LFetch [N]- Number of instructions in the Fetch queue; LResolve [N]- Number of inst. In the Resolve queue; LEX [N]- Number of instructions in the Execute queue; LDC [N]- Number of instructions in the DC queue; LL2 [N]- Number of instructions in the L2 queue; LODC [N]- Number requests through open network that is in the Data Cache queue; LOL2 [N]- Number requests through open network that is in L2 Cache queue; UtilFetch- Utilization of Fetch servers; UtilResolve- Utilization of the Resolve servers. Indicates if a thread's ILP limits execution throughput; UtilEX- Utilization of a functional unit; UtilDC- Utilization of a Data Cache port; UtilL2- Utilization of a L2 Cache port.

3.1 Traffic Analysis

This section presents the analysis of traffic through various parts of the queuing network. The equations presented here allow us to see the rates at which instructions are going through different paths of the network. These rates are used in the next section in an algorithm for analysis of the queuing network. In the closed part of the network, traffic can be thought of as instructions passing through various parts of the network. In the open part of the network, traffic needs further clarification. As an example, only part of L2 Cache access traffic is created by blocked instructions. Other part of L2 traffic is created by back-end activities, such as write-backs of dirty lines from Data Cache. These requests do not directly represent an instruction. However they require service of the L2 cache just like the blocked instructions. Therefore, for L2 traffic analysis, we will treat these service requests from apparent dissimilar sources equally. A similar situation can be seen in the Data Cache traffic also. Notice that, these traffics are represented by the dotted lines in Fig. 4. This assumption does not affect our overall objectives, which are, determining network utilization, analyzing performance bottlenecks, and calculating overall instruction execution throughput. PIPC is the overall execution rate of a single thread, whereas IPC is the instruction execution throughput of the entire microarchitecture. IPC is used as an input parameter in the equations presented in this section. An algorithm for calculating IPC is presented in the next section. If we assume that all threads have equal priority, then an instruction has equal probability of coming from any thread. Then, the traffic through hardware for one thread, PIPC, is

$$PIPC = IPC / p \quad (1)$$

The closed loop component of traffic through Data Cache, which is also the rate at which instructions are blocked for Data Cache accesses, DCR, is

$$DCR = IPC * p_{is} * h_{DC} * (1 - p_w) \quad (2)$$

Closed loop part of the L2 Cache traffic, L2R, is

$$L2R = IPC * p_{is} * (1 - h_{DC}) \quad (3)$$

Instructions, which are not blocked for cache

accesses are fed back through a direct path between the Execute section and the Fetch section of the model. These are mainly the register only instructions or instructions which worked with internal data forwarding or buffers. The traffic through this path, RR, is

$$RR = IPC * (1 - p_{is} + p_{is} * h_{DC} * p_w) \quad (4)$$

Equations (2), (3), and (4) need to add up to IPC.

$$IPC = DCR + L2R + RR \quad (5)$$

We now derive expressions for the open network traffic at the Data Cache. There are two components of such traffic: delayed writes to the Data Cache due to write buffer flushes, and access to data cache that results in a miss. Traffic due to write buffer flush, WBR, is

$$WBR = \frac{IPC * p_{is} * p_w * y}{n_{st}} \quad (6)$$

Traffic due to accesses that result in miss, DCMR, is

$$DCMR = IPC * p_{is} * (1 - h_{DC}) \quad (7)$$

Therefore, the open network component of the Data Cache traffic, DCOTR, is

$$\begin{aligned} DCOTR &= WBR + DCMR \\ &= \frac{IPC * p_{is} * p_w * y}{n_{st}} + IPC * p_{is} * (1 - h_{DC}) \end{aligned} \quad (8)$$

The open network traffic of the Level 2 Cache have three components: write back of the replaced dirty lines from the Data Cache, traffic due to Instruction Cache misses, traffic due to narrow buses. This happens when y is greater than one. Open network traffic due to instruction cache misses, L2ICR, is

$$L2ICR = IPC * mrICi * y \quad (9)$$

Open network traffic resulted from write back of modified Data Cache lines, L2DCR, is

$$L2DCR = IPC * p_{is} * (1 - h_{DC}) * p_{dirty} * y \quad (10)$$

Open network traffic resulted from narrower bus width, L2BR, is

$$L2BR = IPC * p_{is} * (1 - h_{DC}) * (y - 1) \quad (11)$$

Therefore, the open network traffic of the Level 2 Cache, L2OTR, is

$$L2OTR = L2ICR + L2DCR + L2BR \quad (12)$$

Notice that all traffic components are function of IPC. When IPC is zero, all traffics reduce to zero.

3.2 Network Analysis

We now present an algorithm for the analysis of the queuing network. The algorithm includes calculation of instruction execution throughput and calculation of utilization of different microarchitecture elements. We are interested in the final iteration of the loop when $I=N$, $X[N] = IPC$. The Mean Value Analysis based algorithm appears below:

```

LFetch(0) = 0; LResolve(0)=0; LEX(0)=0; LDC(0)=0; LL2(0)=0; LODC(0) = 0; LOL2(0)=0;
FOR I = 1 TO N
  WFetch[I] = ESFQ*(1+LFetch(I-1))

```

$$\begin{aligned}
W_{Resolve}[I] &= ESRQ*(1+L_{Resolve}(I-1)) \\
W_{EX}[I] &= (ESEQ/nfu) * (1.0 + L_{EX}[I-1]); \\
W_{DC}[I] &= (ESDC/ndcport)*(1.0+L_{DC}[I-1] + LO_{DC}[I-1]) \\
W_{L2}[I] &= (ESL2/nl2port) * (1.0 + L_{L2}[I-1] + LO_{L2}[I-1]) \\
X[I] &= 1 / (W_{Fetch}[I] + W_{Resolve}[I] + W_{EX}[I] + \\
&\quad ((p_{ls} * h_{DC} * (1.0 - p_w)) * W_{DC}[I]) + \\
&\quad ((p_{ls} * (1.0 - h_{DC})) * W_{L2}[I])) \\
L_{Fetch}[I] &= X[I] * (1/p) * W_{Fetch}[I] \\
L_{Resolve}[I] &= X[I] * (1/p) * W_{Resolve}[I] \\
L_{EX}[I] &= X[I] * W_{EX}[I] \\
L_{DC}[I] &= X[I] * (p_{ls} * h_{DC} * (1.0 - p_w)) * W_{DC}[I] \\
LO_{DC}[I] &= X[I] * mdcq * W_{DC}[I] \\
L_{L2}[I] &= X[I] * (p_{ls} * (1.0 - h_{DC})) * W_{L2}[I] \\
LO_{L2}[I] &= X[I] * ml2q * W_{L2}[I]
\end{aligned}$$

END FOR

Above, $mdcq$ is the open network traffic multiplier for the Data Cache and $ml2q$ is the open network traffic multiplier for the Level 2 Cache:

$$mdcq = \frac{p_{ls} * p_w * y}{n_{st}} + p_{ls} * (1 - h_{DC}) \quad (13)$$

$$ml2q = p_{ls} * (1 - h_{DC}) * (y - 1) + p_{ls} * (1 - h_{DC}) * y * p_{dirty} + mrIC_i * y \quad (14)$$

Equations 15 through 19 show the utilization of different servers. These equations can be used to locate network performance bottlenecks.

$$util_{Fetch} = (IPC * ESFQ) / p \quad (15)$$

$$util_{Resolve} = (IPC * ESRQ) / p \quad (16)$$

$$util_{EX} = (IPC * ESEQ) / nfu \quad (17)$$

$$util_{DC} = ((DCR + IPC * mdcq) * ESDC) / ndcport \quad (18)$$

$$util_{L2} = ((L2R + IPC * ml2q) * ESL2) / nl2port \quad (19)$$

4. EXPERIMENTAL SETUP AND RESULTS

Our experiments involved four types of architectures: Superscalar, Superscalar with Trace Cache, SMT, and SMT with Trace Cache. We have used the First Benchmarking process as shown earlier in the Fig. 2. We have used the Simplescalar *sim-outorder* simulator and an SMT derivative POSM simulator and added Trace Cache [16]. We modeled 16-wide processors. A bimodal branch predictor used a BTB with 2-bit counters. BTB had 2048 entries. The default line width was 16 instructions. The 4-way set-associative IC had 256 sets; each line was 64B long. The IC was capable of supplying one line per cycle. Data Cache had identical parameters and was independent. The 4-way set-associative unified L-2 Cache had 1024 sets, with a line size of 64B. The 2-way set associative TC had 512 sets. Instruction throughput of *gcc*, *go*, and *perl* are shown in Fig. 5.

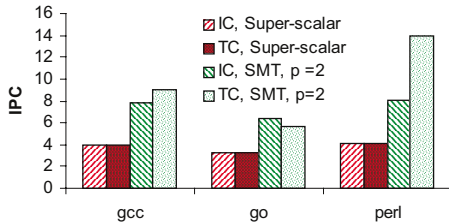


Figure 5: IPC of gcc, go, and perl

5. CONCLUSIONS

The newly introduced *Fast Benchmarking*

methodology can produce processor performance in milliseconds, when others can take hours. This paper presents a micro-architecture modeling technique, using queues, for applications in Fast Benchmarking. The modeling of probability distributions of underlying random variables can be improved in many areas: currently, the resource requests are modeled as customers, where a request can influence others as the instructions are dependent on each other. Service rates of servers often depend on the requests. As an example, different thread can have different instruction level parallelism. And the dynamic instruction level parallelism of a thread can change during execution. A FIFO dispatch discipline does not represent an out-of-order machine well. Despite limitations, we think queuing models are good for processor performance analysis, especially for Fast Benchmarking. Artificial neural network based models can augment the queuing models. Other more accurate micro-architecture models should also be explored for Fast Benchmarking.

6. ACKNOWLEDGEMENTS

The author thanks James Burns of Intel Corporation and Nasima Parveen of Nanova for their supports.

REFERENCES

- [1]. T Sherwood, and J J. Yi, "Guest Editors' Introduction: Com Arch. Simul. and Model.", *IEEE MICRO*, Jul-Aug 2006.
- [2]. Wenisch, et. al., "SimFlex: Statistical Sampling of Computer Sys Simulation", *MICRO*, July-Aug 2006.
- [3]. Binkert, et. al., "The M5 Simulator: Modeling Networked Sys", *IEEE Micro*, Jul-Aug 2006.
- [4]. Biesbrouck, et. al., "Efficient Sampling Startup for SimPoint", *IEEE MICRO*, July-August 2006.
- [5]. Standard Performance Evaluation Corp: spec.org
- [6]. Hossain, Pease, and Burns, "Fast Benchmarking", *accepted ACTA J of Modeling and Simulation*, 2012.
- [7]. Gross, Donald; Carl M. Harris (1998). *Fundamentals of Queueing Theory*. Wiley. ISBN 047132812X.
- [8]. D. Burger, T. Austin, "The SimpleScalar Tool Set", U. Wisconsin-Madison Comp Sci Tech Report #1342, June 1997.
- [9]. Reshadi, et. al., "Hybrid-Compiled Simulation", *Tran on Embedded Computing Sys*, 8(3), 2009.
- [10]. Haskins, Skadron, "Accelerated Warmup for Sampled Microarchitecture Simulation", *Tran on Architecture and Code Optimization*, V2, N1, March 2005.
- [11]. J. Smith, R. Nair, "The Architecture of Virtual Machines", *IEEE Computer*, pp32-38, May 2005.
- [12]. A Hossain, "Trace Cache in Simultaneous Multi-threading", Ph.D. diss, Syracuse University, 2002
- [13]. Derek B. Noonburg, and John P. Shen, "Theoretical Modeling of Superscalar Processor Performance", *Proc. 27th Int. Symp. on Microarchitecture*, Dec 1994.
- [14]. A. Beg, "Incorporating Program Characteristics into a Processor Model", *World Congress on Engineering and Computer Science*, San Francisco, 2008.
- [15]. Dubach, et. al, "An Emp. Architecture-Centric Approach to Microarch Dgn", *IEEE Tr. Computers*, pp1445, Oct 2011.
- [16]. J. S. Burns, "Parallel On-Chip Simultaneous Multi-threading", Ph.D. Dis, U Southern Calif, May 2000.
- [17]. Hossain, et. al, "Trace Cache Perf. Parameters", *IEEE Int Conf on Computer Design: VLSI in Computers and Processors*, Freiburg, Germany, Sept, 2002.
- [18]. Hossain, et. al., "Trace Cache Miss Rate", *ACTA J. of Modeling & Simulation*, Vol 27, No. 3, 2007.