

Fast and Accurate Performance Modeling of Out-of-order Issue Processors

Mikhail Smelyanskiy and Edward S. Davidson
Department of Electrical Engineering and Computer Science
Advanced Computer Architecture Laboratory
The University of Michigan
email: {msmelyan, davidson}@eecs.umich.edu

Doug Burger
Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
dburger@cs.utexas.edu

Contact: msmelyan@eecs.umich.edu

Abstract

Current techniques that estimate the performance of new processor designs are usually based on cycle-accurate simulations of benchmarks containing hundreds of millions of instructions. This simulation is very time-consuming. In addition, to determine the effect of a design change on processor performance, one has to update the simulator configuration and resimulate the entire benchmark suite.

This paper presents a first step toward a fast and accurate statistical model to estimate the performance of superscalar out-of-order processors. Program traces are analyzed and machine models are developed separately and then combined to form a Markov chain from which we compute the processor performance. Separating the program and machine models allows the benchmark to be traversed only once, rather than being repeatedly scanned for each processor model. The modeling problem is to achieve this decoupling without compromising the accuracy of the performance estimate in an otherwise idealized out-of-order machine model. The IPC estimates obtained from our initial statistical model, which incorporates dependence effects, are very close (within 7 percent) to the IPC estimates obtained from the cycle accurate trace-driven simulation of that machine model.

1. Introduction

Cycle-accurate simulation is today's prevailing technique for evaluating the performance of new processor designs. However, even SimpleScalar [1] (one of the fastest timing simulators in the public domain) takes on the order of 100,000 cycles on a host machine to simulate one cycle of the target pro-

cessor. As processor complexity grows and benchmark traces increase from hundreds of millions to hundreds of billions of instructions, cycle-accurate simulation becomes prohibitively expensive: each small change to any of the system parameters requires updating the simulator configuration and resimulating the entire benchmark suite.

This paper presents a first step toward a statistical model that can provide fast and accurate estimates of the performance of out-of-order processors. Our model builds upon work done in [5] which was applied only to a simple in-order processor. The model that we propose in this paper is the first work to-date targeted to characterizing the performance of an out-of-order machine by statistically modeling the flow of instructions through the pipeline of an out-of-order microengine. As in [5] our statistical model uses processor states similar to the states used in a traditional cycle-accurate simulator. However, instead of computing a sequence of states in time (state x at time t , state y at time $t+1$), we compute the steady-state probability of being in the each state (state x with the probability p_x , state y with probability p_y), where p_i corresponds to the fraction of run-time cycles during which the machine is in p_i .

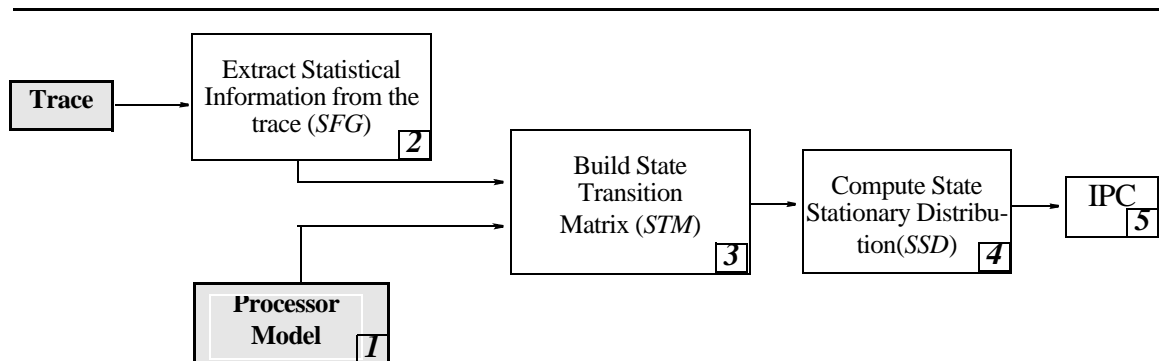


Figure 1: Flow diagram of the modeling process

Generating such a model requires five basic steps (see Figure 1).

1. Design the processor model based on the processor's microarchitecture. The level of detail is the same as that for a cycle-accurate simulator. The primary result of this step is the statistical model's state space and the rules for transitioning from one state to another.
2. This step is independent of the first and involves extracting statistical information about the sequences of instructions from the traces and incorporating it in a statistical flow graph (SFG). This is the most important step of the model, because which statistics are extracted from the trace have direct impact on the overall accuracy of the model.
3. Build a state transition matrix (STM) that captures the transitions between states statistically. Where a timing simulator computes the next state given the current state and information about subsequent instructions in the trace, we instead use the processor model and the SFG to compute the probability of transition from each state to each of its possible successor states, using the statistical information extracted from the trace. This is where the model becomes a statistics-based analytical model, rather than a simulation model.
4. Given the probability of transition from each state to each of its successor states, we can find the state stationary distribution (SSD) which corresponds to the fraction of time that the system resides in each particular state.
5. Given this distribution and observing how many instructions are committed in each state, we can compute the instructions retired per cycle (or IPC) for the particular trace.

An important property of our model is that the processor and the trace are analyzed separately, and the set of relevant characteristics are extracted from each. These characteristics are combined to yield the processor performance estimate. Separating the program and machine models allows the benchmark trace to be traversed just once; the parameters extracted from the trace can be used to study a wide variety of processor models.

The rest of this paper is organized as follows. Section 2 discusses some previously known techniques for modeling processor performance by separately characterizing the trace and the processor. Section 3 derives the general mathematical formulation of our model. Section 4 presents the microarchitecture whose performance we model. Section 5 describes our statistical approach to modeling the processor performance by describing each step in Figure 1 in detail. Section 6 presents initial experimental results. Finally, Section 7 provides some concluding remarks and discusses the directions for future research.

2. Previous Work

A number of past techniques model processor performance by separately characterizing the trace and the processor and then combining these characteristics to estimate overall performance.

Saavedra-Barrera [8] proposes a model of execution in which the frequencies of the machine instructions (obtained from the trace) are combined with their run-times on a particular machine (obtained from microbenchmarking) to predict the run-time of the trace on this machine. Although this approach requires each trace and each machine to be characterized only once, the technique does not model the high variability in the mean execution time of an instruction caused, for example, by data dependencies in the program.

A higher level approach combines the frequencies of basic blocks (obtained from the trace) with their run-times on various machines (obtained from the direct simulation) to obtain the performance estimates. Elevating the model to the block level captures the variability in the mean execution time of instructions due to dependencies within a basic block. However, it evaluates the execution time of each basic block from a cold-start state, ignoring the effects of the state of the processor that is carried over from executing previous basic blocks. *Reduced trace analysis* [9] tries to remedy the problem inherent to the simplistic technique sketched above by utilizing a “glue time” for each distinct pair of successive blocks to capture the processor state when the basic block begins executing.

Recently proposed synthetic application generation techniques [2][7] provide a means to accelerate the simulation process. A statistical profile, a set of statistical program characteristics, is extracted from a program trace. This profile is then used to generate a synthetic application which is fed into a cycle-accurate simulator that estimates the performance for an entire range of modeled microarchitectures. The main advantage of this approach is that the statistical profile (and hence the synthetic application) is much more compact than the trace, and does not grow with trace size

Emma and Davidson [3] proposed a closed form expression for the pipeline performance of a simple processor by using trace statistics to characterize the workload . The set of trace statistics is suffi-

cient for modeling a large class of in-order pipelines with data dependence and branch mis-prediction effects.

Finally, Noonburg and Shen [5] present a statistical approach to modeling superscalar in-order processor performance, which this paper extends to model an out-of-order processor. In their work each program trace is scanned once, generating a set of program parallelism parameters that model dependence distance and can be used across an entire family of machine models. The parallelism parameters are then combined with a machine model to form a Markov chain from which the average IPC is computed.

3. Statistical Model of the Processor Performance

The IPC of a processor executing a particular application is equal to the total number of committed instructions I_{total} divided by the total number of simulation cycles L as shown in (EQ 1). The I_{total} can be expressed as the sum of over c of $w(c)$, where $w(c)$ is the number of instructions committed in cycle c . Let $S = \{s_0, s_1, \dots, s_{r-1}\}$ be the state space of the processor, where r is the total number of distinct states. Each state is a vector whose elements represent relevant information about the instructions in each stage of the pipeline. Let π be a function that maps state s to the number of instructions that are committed in this state. Let n_k be the number of occurrences of state s_k during the execution of the benchmark and let $f(s_k) = \frac{n_k}{L}$, i.e. the $f(s_k)$ are frequencies that are normalized to sum to one.

$$IPC = \frac{I_{total}}{L} = \frac{\sum_{c=1}^L w(c)}{L} = \sum_{k=0}^{r-1} \pi(s_k) \times f(s_k) = \Pi \cdot \Phi \quad (\text{EQ 1})$$

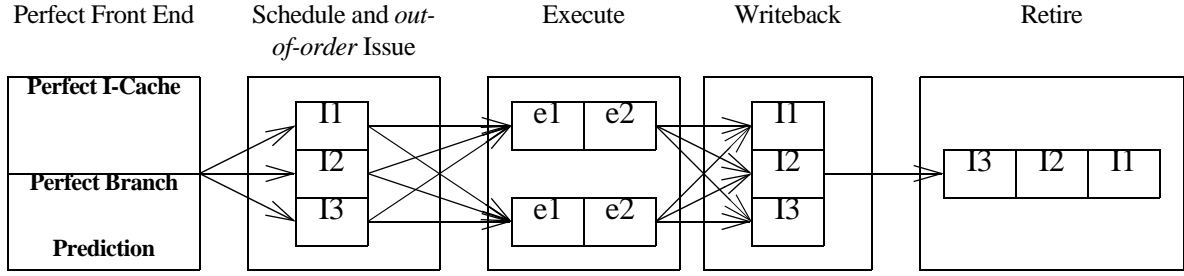


Figure 2: Microarchitecture model

Finally $\Pi = (\pi(s_0), \pi(s_1), \dots, \pi(s_{r-1}))$ is a vector that characterizes each state in terms of how many instructions are committed in that state and $\Phi = (f(s_0), f(s_1), \dots, f(s_{r-1}))$ is a vector that characterizes the dynamic distribution of the frequency of occurrence of each state.

Once we have identified all the possible states that the processor can assume for a given benchmark, we can easily determine Π . To obtain Φ is a more involved task. Each $f(s_k)$ is the “frequency” of being in state s_k measured as the fraction of time that the system spends in state s_k over the long term. Assume we can build the state transition matrix, STM , which characterizes the transition between the states, of the processor: $STM(s_i, s_j)$ is the probability of transition from state s_i to state s_j , given that we are in state s_i . Then, by Markov theory, Φ is the stationary distribution of the Markov process described by the state transition matrix, STM [4]. Assuming that this Markov process is irreducible and ergodic, its stationary distribution is obtained by solving the system of global balance equations:

$$\begin{cases} [\Phi] = [\Phi] [STM] \\ [\Phi] [e] = 1 \end{cases} \quad (\text{EQ 2})$$

Where e is a column vector of all ones.

4. Out-of-order Issue Window Microarchitecture

The out-of-order machine that we model (see Figure 2) consists of five pipeline stages: a perfect front end, schedule and issue, execution, writeback, and retire. A reorder buffer (ROB) of W entries keeps track of the state of the W instructions that are in flight within the four back end stages of this

pipeline. In this initial model we assume a perfect front end that consists of a perfect instruction cache and a perfect branch predictor. It has always fetched and buffered enough instructions in every cycle to fill up all the empty entries that occur in the ROB each cycle as retired instructions leave the ROB. The schedule and issue stage of the pipeline scans the ROB every cycle to find ready instructions (instructions with their dependencies satisfied that have not yet begun execution) and issues up to iw of them to the functional units, where iw is the maximum issue width. The execution stage of the pipeline executes the issued instructions in n functional units. We assume in this initial model generic, l -stage, and fully pipelined functional units. After an instruction is finished executing, its result is written into the instruction's ROB entry and immediately satisfies the corresponding dependencies of instructions that use this result as one of their operands. The instructions with all of their dependencies satisfied are deemed ready for issue; dependent instructions can thus begin their execution l cycles apart. The retire stage of the pipeline finds instructions in the ROB that have finished execution and retires up to rw of them from the head of the ROB (i.e. in programming order). The ROB is organized as a circular buffer so that as head entries are freed up, they become available at the tail for the front end to fill with new instructions.

5. The Statistical Performance Model

5.1. The State Space

The **state** of the machine of size W is defined as the vector $S=(s_{w-1}, s_{w-2}, \dots, s_0)$, where s_i provides state information about the instruction currently in entry i of the ROB ($i=0$ corresponds to the head and $i=W-1$ the tail of the ROB). This state information must be sufficient to allow computation of the next state given the current state plus information about instructions that enter the ROB at the end of this cycle. In this initial model, the state information, s , of an instruction is the vector (st, rl, D_{pair}) , where

- st = instruction status in the ROB which can be NONREADY, READY, EXECUTING or FINISHED
- rl = the residual latency of the instruction ($1 \leq rl \leq l$), where l is the functional unit latency
- $D_{pair} = (d0, d1)$, the dependence distance pair of the instruction ($0 \leq d0, d1 \leq W$)

The instruction's status (st) can have one of 4 values: NONREADY (if its input dependencies have not been satisfied), READY (if its input values have been produced and it can be thus issued into the functional units), EXECUTING (if it is in the execution stage of pipeline, i.e. in some functional unit), or FINISHED (if it has finished execution and its result has been written into the ROB, but it has not yet retired and still occupies an entry in the ROB).

The residual latency (rl) of the instruction is only relevant for EXECUTING instructions and is the time in machine cycles left until the completion of its execution in the functional unit. For example, $rl = 2$ means that the instruction is in the second to last stage of the functional unit pipeline.

D_{pair} is the pair of distances (in instructions) between this instruction and the two instructions that generate the inputs for this instruction. In a machine with a ROB of size W , an instruction with both dependence distances of at least W (i.e. independent of all the instructions that are in the ROB with it) is READY as soon as it enters the ROB. Thus dependence distances greater than W are simply set to W .

The size of the state space is computed by considering all possible states, i.e. all possible combinations of state vector elements. Since st can take one of 4 values, rl takes one of l values, D_{pair} takes one of $W \times W$ values, there are $4 l W^2$ possible states for one instruction. Since there are W instructions in the ROB $(4 \times l \times W \times W)^W$ is an upper-bound on the number of states. The state space size becomes problematic for large W . We are currently working on mitigating the impact of the state space size on the model, particularly as additional aspects of a microengine are incorporated into the model.

A cycle-accurate simulator uses a similar notion of state. In each cycle it computes a new state for the next cycle, given the current state and the incoming sequence of instructions. The output from the

State ID	cycle	ROB Entry I1 (tail)					ROB Entry I0 (head)					#retired
		Instr.	d0	d1	rl	st	Instr.	d0	d1	rl	st	
S1	1	i2	2	2	2	R	i1	2	2	2	R	0
S2	2	i2	2	2	1	E	i1	2	2	1	E	1
S3	3	i3	1	2	2	R	i2	2	2	0	F	1
S4	4	i4	2	2	2	R	i3	1	2	1	E	1
S5	5	i5	1	2	2	N	i4	2	2	1	E	1
S6	6	i6	2	2	2	R	i5	1	2	2	R	0
S7	7	i6	2	2	1	E	i5	1	2	1	E	1
S3	8	i7	1	2	2	R	i6	2	2	0	F	1
S8	9	i8	1	2	2	N	i7	1	2	1	E	1
S9	10	i9	1	2	2	N	i8	1	2	2	R	0
S8	11	i9	1	2	2	N	i8	1	2	1	E	1
S6	12	i10	2	2	2	R	i9	1	2	2	R	0
S7	13	i10	2	2	1	E	i9	1	2	1	E	1
S10	14	i11	-	-	-	-	i10	2	2	0	F	1

(a) Instruction trace

W=ROB size = 2
iw = issue width = 2
rw = retire width = 1
n = # of func. units = 2
l = FU latency = 2

(b) Machine Configuration

(b) Machine state at each cycle

Figure 3: Instruction trace example and machine state in each cycle as the processor executes the trace. (E - EXECUTING, R - READY, N - NONREADY, F - FINISHED)

cycle-accurate simulator is a list of states, one for each cycle. Figure 3(a) shows an example trace fragment of 10 instructions, where each instruction is represented in terms of its input operand dependence distance pair. Figure 3(c) shows the machine state in each clock cycle during the 14 cycles that it takes to execute all 10 instructions on the machine configuration specified in Figure 3(b). The state vector is (s_l, s_0) , where s_l is the state of the tail instruction *I1* and s_0 is the state of the head instruction *I0*. Each instruction state contains the instruction dependence distance pair, residual latency and status of the instruction. For example, in cycle 3, the ROB is in the state *S3*, the instruction in the tail of the ROB (*i3*) has dependence distance pair $\langle 1,2 \rangle$ and is READY to execute, and the instruction in the head of the ROB (*i2*) with dependence distance pair $\langle 2,2 \rangle$ is FINISHED and will be retired at the end of the current cycle. In the next cycle (cycle 4), *i2* has retired, *i3* has moved to the head of the ROB, and a new instruction, *i4*, with dependence distance pair $\langle 2,2 \rangle$ has entered the tail of the ROB and is ready to execute.

Figure 3 can be used to compute the retirement rate of the processor in instructions per cycle (IPC). Each state, as shown in the rightmost column, has an associated number of retired instructions. This number depends only on the current state and never exceeds rw which is 1 in this configuration. For example, in the state $S7$ (2 2 1 E 1 2 1 E), which in Figure 3(c) corresponds to cycles 7 and 13, one instruction is going to be retired at the end of the cycle. In the state $S9$ (1 2 2 N 1 2 2 R) which occurs in cycle 10, zero instructions are retired as can be deduced from the $rl=2$ status of the instruction in the head of the ROB. To get the IPC distribution vector, IPC_{distr} , the dynamic count of states corresponding to each possible retire value (0 or 1) are summed and divided by total number of cycles:

$$IPC_{distr} = \begin{bmatrix} \left(\sum_{\text{states } x \text{ in which 0 instrs retire}} 1 \right) / 14 \\ \left(\sum_{\text{states } x \text{ in which 1 instrs retire}} 1 \right) / 14 \end{bmatrix} = \begin{bmatrix} 4/14 \\ 10/14 \end{bmatrix} = \begin{bmatrix} 0.286 \\ 0.714 \end{bmatrix}$$

The average IPC is computed as follows:

$$IPC = \begin{bmatrix} 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.286 \\ 0.714 \end{bmatrix} = 0 \times 0.286 + 1 \times 0.714 = 0.714$$

Whereas the trace-driven simulator computes the next state in every cycle from the current state and the incoming sequence of instructions, our statistical model computes for each state the *probability* of transition to the each possible next state given the current state and the *probability distribution* of the incoming instructions. The state transition probabilities are used to compute the fraction of cycles spent in each state (Section 5.4.) from which IPC is computed as in (EQ 1). In the next section we show how to compute the state transition probabilities.

5.2. Finding Possible States and their Probabilities (Step 1 & 2)

When the processor is in state some S_{new} we want to obtain the probability of transition from S_{new} to each possible next state. This involves two steps. First, we set the simulator to state S_{new} and run it for

exactly one clock cycle to compute as much of the next state as possible without going to the trace.

During this one cycle simulation of the state the following actions will take place:

- Up to iw READY instructions will be issued into available functional units to begin execution
- The residual latency of each EXECUTING instruction will be decremented.
- The instructions in the ROB that are FINISHED will be committed up to the retire width rw . FINISHED instructions satisfy the corresponding dependency of instructions that use its result as of one of their operands. If both of an instruction's operands are satisfied, this instruction is marked as READY.

Thus the first step enables us to compute some portion of the next state. This portion will be shared by all next states. The rest of this section describes the second step of the procedure.

If we commit k instructions in this cycle, we know that the ROB will shift by k instructions to free up k available slots for the new instructions to come in from the trace. Some statistics from the trace are needed to determine which instructions may occupy the empty spots in the ROB and with what probabilities. The operations of this model differ from trace simulation primarily by filling empty ROB positions statistically, rather than sequentially from the trace at the cost of often retraversing the same state sequence.

Let the dependence distance pairs of the instructions currently in the ROB be $D1_{pair}^{w-1}, D1_{pair}^{w-2}, \dots, D1_{pair}^0$, as defined above. When k instructions are committed in the current cycle, we use the following trace statistic as the probability that the next k instructions that will enter the ROB will be $D2_{pair}^{k-1}, \dots, D2_{pair}^0$:

$$P(D2_{pair}^{k-1}, \dots, D2_{pair}^0 | D1_{pair}^{w-1}, \dots, D1_{pair}^0) = \frac{\text{number of occurrences of } D2_{pair}^{k-1}, \dots, D2_{pair}^0, D1_{pair}^{w-1}, \dots, D1_{pair}^0}{\text{number of occurrences of } D1_{pair}^{w-1}, \dots, D1_{pair}^0} \quad (\text{EQ 3})$$

The left side of this formula is the conditional probability that next k instructions will have dependence distance pairs of $D2_{pair}^{k-1}, \dots, D2_{pair}^0$ given that the previous W instructions have dependence distance pairs of $D1_{pair}^{w-1}, \dots, D1_{pair}^0$. The right side estimates this probability using counts of the occurrences of

the relevant strings in the trace. These conditional probabilities are used to model how empty ROB slots are filled. In the next subsection we show how to efficiently obtain these conditional probabilities from the trace.

5.2.1. Statistical Flow Graph (SFG)

We analyze each trace only once to compute the state transition probabilities. The result of this analysis is the statistical flow graph (SFG), named after a similar graph described in [6]. There are, however, important differences between these two graphs. Our SFG is specifically tailored to capture sequences of dependence distances from the trace for our out-of-order instruction flow model, whereas the SFG in [6] captures the information needed for the in-order processor model. We first describe how we build the SFG, and then discuss how it can be used to compute the state transition probabilities.

The statistical flow graph consists of rw (retire width) sub-graphs (G_1, \dots, G_{rw}) where each sub-graph corresponds to a particular value of k from 1 to rw . All rw subgraphs are built simultaneously during the single pass through the trace. Here we only show how to build one such subgraph G_k which is used when k instructions retire.

To build G_k , we iterate over the instructions in the trace. Each instruction is represented by a dependence distance pair, as above. When each instruction is read it is pushed into a FIFO of depth $W+rw$. Then the state, S , represented by the first W instructions in the FIFO is looked up in a hash table (and added if not found) and its count is incremented. Then the state, represented by the next W instructions that appear in the FIFO after the first k instructions is looked up in a table of successors of S and its count is incremented for use as a transition edge weight in G_k . After the entire trace is traversed, the states in the hash table are the states of G_k , and arc is added for each successor of S and labeled with a conditional probability equal to the transition count of that successor of S divided by the count of state S itself. which exactly corresponds to the conditional probability computed in (EQ 3). Note that the

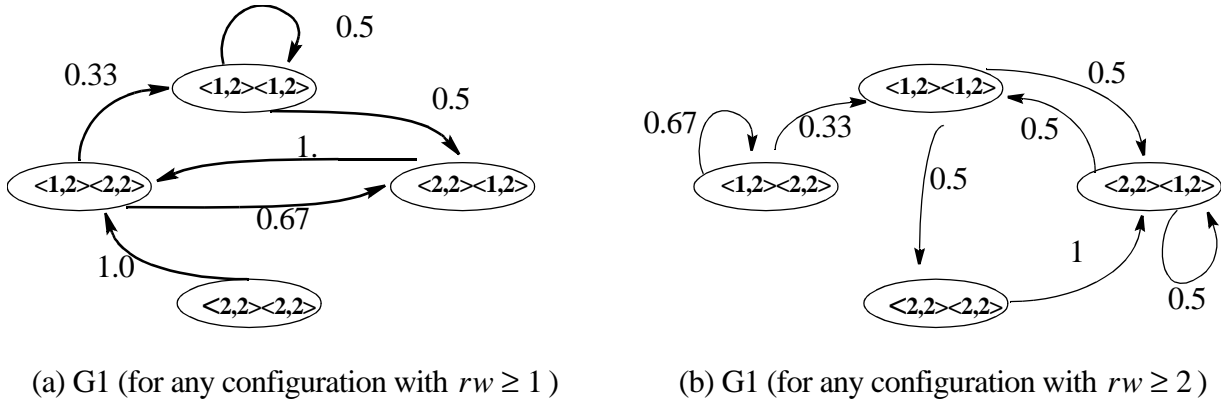


Figure 4 A Statistical Flow Graph

states of all rw subgraphs and their counts are exactly the same, but the transitions probabilities are different.

Figure 4 shows the statistical flow graph that corresponds to the trace sequence from Figure 3(a). Although we show two subgraphs $G1$ and $G2$ that are used, respectively, when 1 or 2 instructions retire, only $G1$ is used in obtaining the state transitions of our example since its configuration has $rw=1$.

$G1$ and $G2$ provide the next-state transition probabilities for the trace fragment in Figure 3(a). For example, given the sequence of instructions $\langle 2,2 \rangle \langle 1,2 \rangle$ in the trace, $G1$ indicates that the next sequence that follows it (overlapping by one instruction $\langle 2,2 \rangle$) is $\langle 1,2 \rangle \langle 2,2 \rangle$ with probability 1.0. In the trace fragment we see that in fact whenever $\langle 1,2 \rangle$ is followed by $\langle 2,2 \rangle$, the next instructions is $\langle 1,2 \rangle$ always. However, given the sequence of instructions $\langle 2,2 \rangle \langle 1,2 \rangle$ in $G2$, there are two sequences ($\langle 2,2 \rangle \langle 1,2 \rangle$ and $\langle 1,2 \rangle \langle 1,2 \rangle$) of two instructions that displace the sequence $\langle 2,2 \rangle \langle 1,2 \rangle$ with equal probability. Similarly, in the trace fragment $\langle 1,2 \rangle$ is followed by $\langle 2,2 \rangle$ twice and one time the next instructions are $\langle 1,2 \rangle$ then $\langle 2,2 \rangle$, whereas the other time they are $\langle 1,2 \rangle$ then $\langle 1,2 \rangle$.

5.3. State transition matrix computation (Step 3)

We use information from both SFG and the processor microarchitecture to compute the machine state transition matrix, STM . STM is of dimension is $n \times n$, where n is total number of machine states,

	State	1	2	3	4	5	6	7
1	1 2 2 N 1 2 2 R	0	1	0	0	0	0	0
2	1 2 2 N 1 2 1 E	0.5	0	0.5	0	0	0	0
3	2 2 2 R 1 2 2 R	0	0	0	1	0	0	0
4	2 2 1 E 1 2 1 E	0	0	0	0	1	0	0
5	1 2 2 R 2 2 0 F	0	0.333	0	0	0	0.667	0
6	2 2 2 R 1 2 1 E	0	0	0	0	0	0	1
7	1 2 2 N 2 2 1 E	0.333	0	0.667	0	0	0	0

Figure 5: Complete state transition matrix

Table 1: The stationary distribution for the trace in Figure 3

state	probability	#retired
1 2 2 N 1 2 2 R	0.1249	0
1 2 2 N 1 2 1 E	0.1785	1
2 2 2 R 1 2 2 R	0.1607	0
2 2 1 E 1 2 1 E	0.1607	1
1 2 2 R 2 2 0 F	0.1607	1
2 2 2 R 1 2 1 E	0.1072	1
1 2 2 N 2 2 1 E	0.1072	1

and element (i,j) of *STM* is the probability that the next state is j , given that the current state is i . *STM* is used to compute the state stationary distribution (SSD).

The algorithm to construct the state transition matrix (STM) chooses a new state and generates all the successors of this new state until no more new states are generated.

Figure 5 shows the complete state transition matrix constructed for the trace fragment in Figure 3(a). The actual state vector for each state ID is at the left of the matrix. The transition matrix is constructed by considering all possible transitions from each state. For example, consider the processor state 1 2 2 N 1 2 1 E in cycle 9 (Figure 3 (c)). The potential next states are computed as follows:

- The instruction $\langle 1,2 \rangle$ at the head of the ROB retires and leaves the ROB
- The following instruction $\langle 1,2 \rangle$ moves into the head of the ROB. Its status becomes READY (R) and its residual latency (rl) remains 2.

At this point, we have computed the deterministic portion of the next state vector. To fill the empty slot in the ROB, we perform a look-up in the statistical flow graph (SFG) . Looking at the G_j in Figure

4(a), it can be seen that there are two possible successors of the $\langle 1,2 \rangle \langle 1,2 \rangle$ pair: $\langle 1,2 \rangle \langle 1,2 \rangle$ and $\langle 2,2 \rangle \langle 1,2 \rangle$, each with the probability 0.5. A new instruction always enters with $rl=2$. Since the new ROB head instruction has $rl=2$, a new instruction with a 1 in its distance pair will have $st=N$ and otherwise will have $st=R$. There are thus two possible next states:

- 1 2 2 N 1 2 2 R with probability of 0.5, and
- 2 2 2 R 1 2 2 R with probability of 0.5.

Note, that the next states and their probabilities were computed using only the current state vector, knowledge of the machine model, and information from the SFG. Note that the 4 SFG states in this example expand to 7 machine states.

5.4. Computing the State Stationary Probability Distribution and IPC (Step 4 & 5)

After the state transition matrix has been constructed, the next step is to find the State Stationary Probability Distribution (SSD). This distribution is obtained by solving the system of equations (EQ 2) to obtain the state stationary distribution vector Φ . Given the state stationary distribution vector and number of instructions retired in each state, we can obtain the IPC distribution vector in which i th element corresponds to the fraction of all cycles in which i instructions are committed. From IPC distribution we can obtain the average IPC.

Table 1 shows the Stationary State distribution computed from the *STM* of Figure 5. To get the IPC distribution vector IPC_{dist} , the probability of the states corresponding to each possible retire value (0 or 1) are summed and divided by total number of cycles.

$$IPC_{distr} = \begin{bmatrix} \sum_{\text{states } s \text{ in which 0 instrs retired}} p_s \\ \sum_{\text{states } s \text{ in which 1 instrs retired}} p_s \end{bmatrix} = \begin{bmatrix} 0.1249 + 0.1607 \\ 0.1785 + 0.1607 + 0.1607 + 0.1072 + 0.1072 \end{bmatrix} = \begin{bmatrix} 0.286 \\ 0.714 \end{bmatrix}$$

The average IPC is computed as follows:

$$IPC_{avg} = [0 \ 1] \begin{bmatrix} 0.286 \\ 0.714 \end{bmatrix} = 0 \cdot 0.286 + 1 \cdot 0.714 = 0.714$$

We can see that average IPC matches the average IPC obtained from the cycle accurate simulation (Section 5.1.). The next section describes the results of applying this model to the CPU2000 benchmark suite.

6. Experimental Results

Figure 6 shows some initial performance results which compare the timing simulator with our Markov model for seven CPU2000 benchmarks: compress, go, li, m88ksim, jpeg, applu and hydro. Each trace consists of 200,000 instructions. There is one table each of four different processor configurations.

The second and third column show the average IPC results for direct trace-driven simulation and for the Markov model respectively. The fourth column shows the error which is the percent difference between two IPC values. The fifth column shows total number of states produced by the Markov model. Finally, the sixth column shows the average number of transitions per state, which indicates the sparsity of the state transition matrix.

The important point to note here is that we construct the set of statistical flow sub-graphs in a single pass through the trace and can use them for a vast range of machine configurations. The maximum rw of interest tells us how many subgraphs to construct.

We can see that the Markov model comes very close to the results obtained from the cycle accurate simulator. The error generally ranges from 0 to 2% with one case at 7%. We can also see from the figure that the state transition matrix is very sparse as the number of non-zero entries ranges only between one to three times the total number of states.

	Simulator	Markov Model	error %	#States	Avg. Transition Per State
compress	0.837	0.846	1	8	2
go	0.804	0.793	1.3	12	1.8
li	0.879	0.873	0.6	8	2
m8ksim	0.882	0.876	0.6	8	2
ijpeg	0.855	0.833	2	8	2
applu	0.885	0.878	0.7	14	1.9
hydro	0.866	0.855	1	14	1.9

(a) Processor Model: $W = 2, rw = 2, iw = 2, n = 2, l = 2$

	Simulator	Markov Model	error %	#States	Avg. Transition Per State
compress	0.938	0.937	0.1	2371	2.2
go	0.777	0.777	0	1711	2.1
li	0.897	0.902	0.5	2546	2.2
m8ksim	0.856	0.860	0.4	3247	2.4
ijpeg	0.836	0.846	1	2179	2.1
applu	0.891	0.895	0.4	3086	2.4
hydro	0.870	0.862	1	4032	2.7

(b) Processor Model: $W = 4, rw = 2, iw = 2, n = 4, l = 3$

	Simulator	Markov Model	error %	#States	Avg. Transition Per State
compress	3.738	3.746	0.2	1336	1.3
go	3.393	3.341	1.5	1304	1.4
li	3.395	3.327	2	1908	1.3
m8ksim	3.147	3.143	0.1	2984	1.3
ijpeg	3.300	3.320	0.6	959	1.2
applu	3.257	3.270	0.3	2736	1.4
hydro	3.346	3.352	0.2	5476	1.4

(c) Processor Model: $W = 8, rw = 4, iw = 2, n = 8, l = 1$

	Simulator	Markov Model	error %	#States	Avg. Transition Per State
compress	4.976	4.984	0.1	1014	1.1
go	4.460	4.549	2	1110	1.4
li	4.292	4.233	1.3	1521	1.3
m8ksim	3.841	3.557	7	2196	1.2
ijpeg	3.801	3.797	0.1	486	1.1
applu	4.042	3.923	3	1581	1.1
hydro	4.468	4.459	0.2	3416	1.2

(d) Processor Model: $W = 12, rw = 6, iw = 12, n = 12, l = 1$

Figure 6: Comparison between direct simulation and the Markov model using seven CPU2000 benchmarks and 4 processor configurations.

The Markov model for the ROB of size 4 (Figure 6(b)) has a larger number of states in 6 of the 7 benchmarks than the Markov models for the size 8 and size 12 ROBs (Figure 6(c,d)). This is due to the higher latency of its functional units ($l=3$ vs. $l=1$). The higher the latency, the more time instructions spend in the ROB waiting for their dependencies to be satisfied, which produces a larger number of states.

Figure 7 shows the timing comparison (in seconds) between the cycle-accurate simulator and the Markov model for a variety of processor configurations running 200,000 instructions of the *compress* benchmark. In both Figure 7(a) and Figure 7(b) the ROB size varies from 2 to 8. All of the ROB configurations in Figure 7(a) have a retire width of 1, whereas the retire width is 2 in Figure 7(b).

The first column shows instruction window size. Second column shows simulator run time for this processor configuration. The third column shows the time it takes to build the statistical flow graph. The fourth and fifth sub-columns show the time it takes to construct the state transition matrix (STM) and find the stationary distribution of states. The time it takes to find IPC from the state probability distribution is negligibly small.

Traversing the trace and building the statistical flow sub-graphs (G_1, G_2) takes significantly less time than the trace-driven simulation which also processes the trace one instruction at a time.

For the retire width of one (Figure 7(a)) the Markov model always runs faster than simulator. As window size increases, so does the state size, which causes the Markov model to take more time to run. As retire width is increased to 2 (Figure 7(b)), the simulation time is less than with a retire width of one and it also grows more slowly. The state space in this case is larger than for the retirement width of 1. This causes the Markov model to become slower than the simulator for larger ROB sizes.

Issue Window Model	Simulation	Markov Model			
		SFG	Build STM	Steady State	#States
W=2	24	1	1	0	6
W=3	49	1	1	0	52
W=4	58	2	1	0	326
W=5	68	2	4	4	870
W=6	73	2	10	15	1454
W=7	80	2	14	27	1844
W=8	84	2	18	35	2122

(a) issue window model with varying window size ($iw=2, rw=1, n=2, l=1$)

Issue Window Model	Simulation	Markov Model			
		SFG	Build STM	Steady State	#States
W=2	16	1	1	0	4
W=3	25	2	1	0	72
W=4	28	3	1	2	440
W=5	34	3	5	8	1035
W=6	37	3	10	17	1503
W=7	38	3	16	27	1871
W=8	40	4	20	41	2135

(b) issue window model with varying window size ($iw=2, rw=2, n=2, l=1$)

Figure 7: Run time comparison of the direct simulator the Markov model for window size varying from 2 to 8. 200,000 instructions of the compress benchmark are used. In (a) retire width 1 is used. In (b) retire width of 2 is used. Other parameters remain the same. All times are in seconds.

Thus, one important fact that cannot be overlooked is the large number of states generated for ROB's model of larger size. For the Markov model to be usable for the larger ROB sizes, the state space must be kept small. Our current research is focused on reducing the number of states.

7. Conclusion and Future Directions

In this work we demonstrated how a Markov model can be used to model the performance of an out-of-order superscalar processor. We perform a one-time scan of the trace to generate the set of characteristics that allow us to quickly obtain performance estimates for a variety of machine configurations. We combine this characterization with the machine model to form Markov state transition matrix. Once the relevant characteristics are extracted from the trace, building the state transition matrix does not depend on the trace length, but only on the complexity of the machine model. The state

transition matrix is used to obtain the stationary state distribution from which we obtain the performance estimate.

We found that the experimental results of our initial models are very close to the results obtained from the trace-driven cycle-accurate simulation.

This work represents initial step in the attempt to model the performance of realistic out-of-order superscalar processor systems. We have seen that the state space scales poorly even for moderate ROB sizes. One way to reduce the state space requirements is to treat each new instruction entering the ROB as independently distributed, regardless of the current state of the instructions in the ROB. This is part of our future work. We expect that this model will scale well to longer traces, but we need to verify this. We also plan to extend this model to more complex processors with variable functional unit latencies, realistic instruction and data caches and imperfect branch prediction.

8. References.

- [1] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, 25(3):13-25, 1997.
- [2] Lieven Eeckhout, Koen De Bosschere and Henk Neefs. Performance Analysis Through Synthetic Trace Generation. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*. Apr., 2000, pp. 1-6.
- [3] Philip G. Emma and Edward S. Davidson. Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance. In *IEEE Trans. Computers*, No. 7, pp. 859-875, July 1987.
- [4] Roy Goodman. Introduction to Stochastic Models, the Benjamin/Cummings Publishing Company, Inc., 1988.
- [5] Derek B. Noonburg, John Paul Shen. A Framework for Statistical Modeling of Superscalar Processor Performance. In *Proceeding of HPCA-3*, 1997.
- [6] Derek B. Noonburg, A Framework for Statistical Modeling of Superscalar Processor Performance. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997.
- [7] Mark Oskin, Frederic T. Chong and Matthew Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 71-82, June, 2000.
- [8] Rafael H. Saavedra-Barrera, Alan Jay Smith, and Eugene Miya. Machine characterization Based on an Abstract High-Level Language Machine. In *IEEE Transactions on Computers*, Vol. 38, 12, December 1989.
- [9] John-David Wellman. Ph.D. Thesis. University of Michigan, Ann Arbor, 1996.