A Framework for Statistical Modeling of Superscalar Processor Performance

Derek B. Noonburg John Paul Shen

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{derekn,shen}@ece.cmu.edu

Abstract

This paper presents a statistical approach to modeling superscalar processor performance. Standard trace-driven techniques are very accurate, but require extremely long simulation times, especially as traces reach lengths in the billions of instructions. A framework for statistical models is described which facilitates fast, accurate performance evaluation. A machine model is built up from components: buffers, pipelines, etc. Each program trace is scanned once, generating a set of program parallelism parameters which can be used across an entire family of machine models. The machine model and program parallelism parameters are combined to form a Markov chain. The Markov chain is partitioned in order to reduce the size of the state space, and the resulting linked models are solved using an iterative technique. The use of this framework is demonstrated with two simple processor microarchitectures. The IPC estimates are very close to the IPCs generated by trace-driven simulation of the same microarchitectures. Resource utilization and other performance data can also be obtained from the statistical model.

1 Introduction

This paper presents a statistical approach to modeling superscalar processor performance. Current performance evaluation techniques generally involve some sort of trace-driven simulation. Using a very detailed model of the processor microarchitecture, these techniques can produce accurate performance figures [BHLS96]. However, these results come at the cost of very long simulation times. Current trace-driven or timing simulators have an overhead of two to four orders of magnitude, i.e., it takes 100–10,000 cycles on a host machine to simulate a single target processor cycle (trace generation takes only 5–100 of those cycles) [BHLS96, CK94]. This is an especially important consideration now that program traces can run into billions of instructions. The statistical approach presented here achieves a significant decrease in execution time while maintaining good accuracy.

Our statistical model uses processor states similar to those used in a trace-driven simulator. However, instead of computing a time-based list of states ("state x in cycle t, state y in cycle $t+1$, ..."), we compute a probability for each state ("state x with probability p_x, state y with probability p_y, ..."). The probability of being in a particular state is equivalent to the fraction of cycles in which the processor is in that state.

Generating such a model involves two basic steps. The first is designing the processor model. This model is based on the processor's microarchitecture, and is at the same level of detail used in a timing simulator. In fact, designing this model is similar to designing a traditional trace-driven timing simulator. The primary result of this step is the statistical model's state space. The second step involves the transitions between states. Where a timing simulator computes the next state from the current state plus information about subsequent instructions in the trace, we instead compute probabilities of transitions from each state to every possible successor state, using information extracted from the trace. This is where the statistical component of the model comes in.

It is important to observe that the processor and program are analyzed separately. This is an extension of the concept of machine and program parallelism [Jou89] with performance being the resultant interaction between the two. After a program trace is analyzed, the parameters which are extracted from the trace can be used to estimate the performance of that program on different processors. The trace need be analyzed only once in order to model performance on a family of fairly similar processors. This analysis can even be done on the fly, while generating the trace, to avoid having to store large trace files.

The model — a state space plus transition probabilities — forms a Markov chain. This Markov chain is partitioned into a set of smaller models in order to get state spaces of reasonable size. Each partition is made up of one or more processor components: an issue buffer, an execution pipeline, etc. This allows a processor model to be built up from interchangeable components, similar to the way a SPICE circuit model is built up from circuit elements.

The work presented here is only a first step. It is an effort to explore the feasibility of an analytical approach to superscalar performance estimation. This work should be considered a *framework* for statistical models. Several simple examples are presented here, and more complex models will be possible using similar techniques.

Section 2 describes some previous work which led up to this paper. Section 3 presents the framework used to model processor microarchitectures. Section 4 introduces the statistical techniques used. Sections 5–7 present three models which illustrate the modeling framework. Finally, Section 8 summarizes the results and suggests potential future directions.

2 Previous Work

Jouppi describes a simple theoretical performance model which uses the concepts of machine parallelism and benchmark (i.e., program) parallelism [Jou89]. Machine parallelism is defined as the product of the average degree of superpipelining and the degree of parallel issue. This is effectively the maximum number of instructions "in flight" (in the execution stages) in the processor. Program parallelism is defined as the average speedup when the program is executed on an infinitely parallel superscalar processor, compared to execution on a single-issue processor.

If the machine parallelism is low relative to program parallelism, overall performance is limited by the machine parallelism. If, on the other hand, the machine parallelism is high, the performance is limited by the program parallelism. Most performance analysis, however, is concerned with cases where the program and machine parallelisms are comparable. In these cases, the performance is dependent on complex interactions between the program and machine.

Dubey, Adams, and Flynn propose an analytical performance model [DAF94]. They analyze a program trace in order to extract two parameters:

$$
\begin{aligned}
p_\delta &= \text{Prob}\,[\text{instructions } i \text{ and } i+\delta \text{ are} \\
&\qquad \text{mutually independent}] \\
p_\omega &= \text{Prob}\,[\text{two instructions scheduled in the same} \\
&\qquad \text{cycle are separated by } \omega \text{ branches}]
\end{aligned}
$$

Like Jouppi, we view processor performance as a result of the interaction of machine parallelism and program parallelism. Trace-drive simulation makes use of a machine model, but does not explicitly model the program and the interaction between program and machine parallelism. Our approach makes use of both machine and program models.

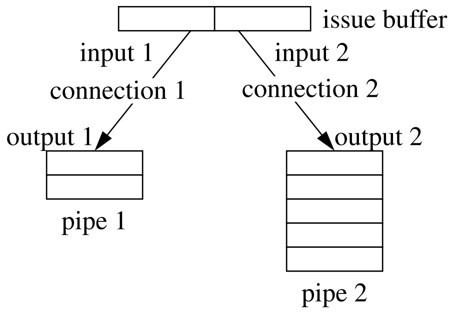

Figure 1: Three components — an issue buffer and two pipes — and two connections.

We extract a set of data from the trace, which is similar in idea to the p_δ and p_ω parameters mentioned above, but more detailed. The interaction between program and machine is then explicitly modeled as a Markov chain.

3 Model Framework

This section presents the general framework used for modeling a processor. The model structure takes the form of a block diagram of the processor. The blocks are called **components**. Components come in various types: issue buffers, pipelines, and so on. The arcs between components are called **connections**. The component at the tail of a connection is called its **input**. The component at the head is the connection's **output**. From the point of view of a component, the flow(s) coming in are called **input flows** and the flow(s) leaving are called **output flows**. Instructions flow from component to component along the connections. See Figure 1.

The instruction flow between two components in a given cycle (i.e., the number of instructions which flow from the input to the output of a connection) is determined by three things. Clearly, only instructions which are currently ready to leave the input component can flow. The number of available instructions in a particular cycle is called the **push** into that connection. Second, some of those instructions may be blocked by the output component. This may be caused by dependences between instructions (control or data dependences) or by structural hazards (e.g., a stalled pipeline). The number of instructions which can be accepted by the output component in a particular cycle is called the **pull** out of that connection. It is possible for pull to be greater than push, e.g., if the input component is empty (push = zero) and the output component is free of dependences (pull > zero). Finally, the flow is limited by the bandwidth of the connection, e.g., it may be possible for no more than one instruction to flow, regardless of how many are ready (push) and how many are independent (pull). The flow across a connection in a given cycle is equal to the minimum of the push, pull, and bandwidth.

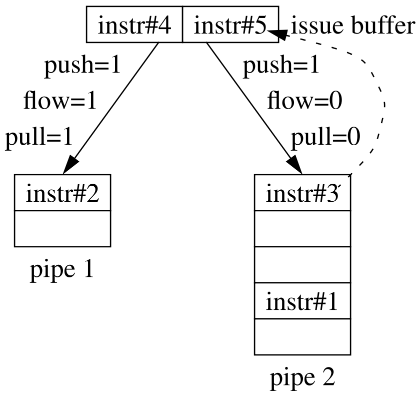

Figure 2: The same components shown in Figure 1, with five instructions. The dotted arc indicates that instruction #5 is data dependent on #3. The push into each pipe is 1, since there is one instruction of each type in the issue buffer. The pull into pipe 1 is 1, since there is no data dependence, and therefore the flow is also 1. The pull into pipe 2 is 0 because of the data dependence, and therefore the flow is 0. (Both connection bandwidths are 1.)

See Figure 2.

This definition of flow is equivalent to what is used in simulators. Flow is split into push and pull here so that the contributions from the input and output components of a connection can be independently computed. This becomes important for partitioned models (see Section 7).

The following sections describe several specific components. However, the framework itself is general, and can be extended by adding new component types. The ultimate goal is to provide a wide variety of components which can be "wired" together to generate the machine model, something like circuit elements in the SPICE model of a circuit.

4 Statistical Modeling

4.1 States and probabilities

The structure of the processor microarchitecture (components and connections) defines the model's state space. A **state** represents the current state of the processor. It is a vector, consisting of information about the instructions in each component in the machine (the "in flight" instructions).

This information must be sufficient to allow computation of the next state given the current state plus information about subsequent instructions. Thus the state must include instruction type and dependence information for each instruction currently in flight. As used here, the instruction type is an identifier for the pipeline which will execute the instruction, and also marks the instruction as a branch or non-branch. The dependence information indicates the preceding instructions on which the instruction is data-dependent.

A trace-driven simulator uses a similar notion of state. In each cycle, it uses the current state to compute a new state for the next cycle. At the lowest level, the output of a trace-driven simulator is a list of states, one for each cycle. This output is, of course, not used directly. Certain interesting values — IPC, resource usage, etc. — are extracted from it as it is generated.

The statistical model presented here uses states somewhat differently. Instead of a list of states, the low-level output is the probability of the processor being in each state in any particular cycle, i.e., the fraction of cycles spent in each state. For example, a trace-driven simulator which had only three states might produce the output 0,1,2,2,2,1,0,2,2,0. The statistical model would instead produce a probability distribution: [0.3 0.2 0.5]. This indicates that 30% of cycles were spent in state 0, 20% in state 1, and 50% in state 2. In general, the same performance figures which are extracted from a simulation run can also be extracted from this state distribution.

The most important thing to note about this difference — a time-series of states vs. state probabilities — is that the simulation output size and run-time are proportional to the number of cycles simulated, while the statistical model output size and run-time are proportional to the number of states. This is where the statistical model gains its execution time advantage. While the statistical model still depends on an analysis of each trace, this need only be done once per program and is significantly faster than simulation.

4.2 Markov chains

In order to compute the state probability distribution, the statistical model requires information from both the processor microarchitecture and from the program trace. As described above, the microarchitecture determines the state space. The program, on the other hand, determines the probabilities of transitions between states.

The state distribution is computed by forming a Markov chain, using the processor's state space. In order to form this Markov chain, we need the probabilities of transitions between every pair of states. These transition probabilities form an $n \times n$ matrix $\mathbf{P}$, where n is the number of states, and P_{xy} is the probability of a transition from state x to state y, i.e., the probability of going to state y in cycle $t+1$, given that the processor is in state x in cycle t. The state distribution that we want is then the stationary distribution of the Markov chain, which can be computed from the transition matrix using standard techniques [Res92].

A trace-driven simulator can afford to be generous with its state space — it only has to store the state vector for the current cycle. If there are n states, this amounts to $\log_2 n$ bits. To build the Markov chain transition matrix, however, requires n^2 probabilities. (The $n \times n$ matrix is actually very sparse, and the number of non-zero elements is significantly less than n^2, however it can still be quite large.) This means that the state space must be designed very carefully. It must be large enough to be useful, i.e., to produce interesting and meaningful performance information, but small enough to be amenable to solution using Markov chain techniques.

5 One-Pipe Processor

The next three sections present simple machine models to demonstrate the modeling technique described above. All of the models use the DEC Alpha instruction set and use traces generated with DEC's Atom tool [SE94]. Trace-driven simulators for the same machine models were developed using MW, a second-generation tool based on VMW (Visualization-based Microarchitecture Workbench) [DSP93], in order to provide a basis for comparing the performance estimates.

5.1 The one-pipe model

The first processor is a simple, single-pipeline, scalar processor (see Figure 3). It consists of three components: a fetch buffer, an issue buffer, and the execution pipeline. The fetch and issue buffers each hold one instruction. The fetch buffer comes in two versions: one which does perfect branch prediction (no stalls), and one which does no branch prediction (fetching stalls after every branch, until the branch is issued). The issue buffer has very little effect in this model — it models a decode stage, which will never cause a stall. It does however introduce a one-cycle bubble after every branch, in connection with the no-prediction fetch buffer. The pipeline executes all instructions with a latency of two cycles (writeback is assumed not to generate stalls, and is thus ignored here). There is a forwarding path from the last pipe stage to the first, so an instruction dependent on the immediately previous instruction stalls in the issue buffer for only one cycle.

Instructions have two important characteristics with respect to this processor. First, an instruction can be either a branch or a non-branch. (There are no other instruction types since there is only one pipe.) Second, each instruction has a **dependent instruction distance**, s, which is defined as the distance, measured in dynamic instructions, to the source of its most recent data dependence. If the instruction has no data dependences, or if the distance to the nearest source is greater than some predefined maximum, the dependence distance is considered to be infinite.

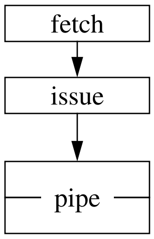

Figure 3: A one-pipe processor microarchitecture.

For example, consider this excerpt from a trace:

```
     ldl   0(a2)   --> a3
     blbc  a3, A             ; s = 0
A:   addl  a0, a3  --> a0    ; s = 1
     br    B                 ; s = ∞
B:   ldq   8(a2)   --> a2
```

The `blbc` instruction is dependent on its immediate predecessor (`ldl`), so its dependence distance is 0. The first `addl` is also dependent on the `ldl`, which is its second predecessor, so the dependence distance is 1. The `br` is not dependent on any other instructions, so its dependence distance is infinity.

5.2 A monolithic state space

This processor is simple enough that we can use a single state vector to represent the state of the entire processor. This is a **monolithic** model, in contrast to the **partitioned** model described in Section 7.

The state vector consists of the following elements:

- x^F = # instructions in the fetch buffer ($0 \leq x^F \leq 1$)
- x^I = # instructions in the issue buffer ($0 \leq x^I \leq 1$)
- x_i^P = # instr's in pipe stage i ($0 \leq x_i^P \leq 1$ for $0 \leq i \leq 1$)
- y_i = the types of the next two instructions: 0 = non-branch, 1 = branch ($y_i \in \{0, 1\}$ for $0 \leq i \leq 1$)
- s_i = the dependent instruction distances of the next two instructions ($0 \leq s_i \leq s_{\max}$ for $0 \leq i \leq 1$)

The type and distance values are for the next two instructions to be issued. If there are instructions in both the fetch and issue buffers (i.e., if there is no branch bubble), then the instruction in issue is type y_0 and distance s_0 and the instruction in fetch is y_1/s_1. If there is an instruction in fetch but none in issue (i.e., an unpredicted branch just issued), the instruction in fetch is y_0/s_0, and the next instruction (which has yet to be fetched) is y_1/s_1.

Type information must be kept for at least the next two instructions, since we need to know about any branches currently in the fetch and issue buffers. Keeping type information for more than two instructions would improve (slightly) the model's accuracy. (See the discussion on computing transition probabilities below.)

It is not necessary to maintain type/distance information for instructions in the pipe, since this information does not influence the machine behavior. An instruction issues when it is ready (independent), and then moves through the pipe and leaves the machine without stalling.

A dependence distance can be between 0 and $s_{\max}-1$, representing an actual distance, or it can be $s_{\max}$, representing a distance greater than $s_{\max}-1$ or no dependence at all. A distance of zero means that the instruction is dependent on the immediately preceding instruction. In this processor, an instruction with a dependence distance of at least one, i.e., independent of the preceding instruction, can always issue immediately. Thus we could use $s_{\max}=1$. However, the s values are also used for instruction sequencing (see below), which is made more accurate by using a higher value of $s_{\max}$ ($s_{\max}=5$ is chosen arbitrarily to match the three-pipe models presented later).

There are $2\times 2\times 2^2\times 2^2\times 6^2=2304$ possible states. This is easily small enough to be able to use a single monolithic Markov model.

5.3 Example

Consider the loop shown in Figure 4. This loop scans a linked list, extracting an integer from each node and adding it to one variable (`a0`) if even, or a different variable (`a1`) if odd. The linked list is constructed so that the trace alternates between these two paths.

A section of the trace is shown in Figure 5, along with the instruction type and dependent instruction distance for each dynamic instruction. For this example, it is assumed that the processor executes iterations of this loop and nothing else.

Figure 6 shows the processor state at each cycle. The upper half of the figure is a standard cycle-by-cycle pipeline diagram, showing how the instructions flow through the processor. (The instructions are numbered as in Figures 5 and 4.) The lower half shows the corresponding state vector for each cycle.

In cycles 2, 7, 11, and 18, the processor is in the same state: $x^F=0$, $x^I=1$, $x^P=01$, $y=[10]$, $s=[01]$. In cycle 2, the next two instructions to be issued are numbers 1 and 4. Instruction 1 is a branch ($y_0=1$) and instruction 4 is not ($y_1=0$). Instruction 1 is dependent on instruction 0, the immediately previous instruction, so $s_0=0$. Instruction 4's most recent dependence is also instruction 0, which is the second previous instruction (in trace order), so $s_1=1$.

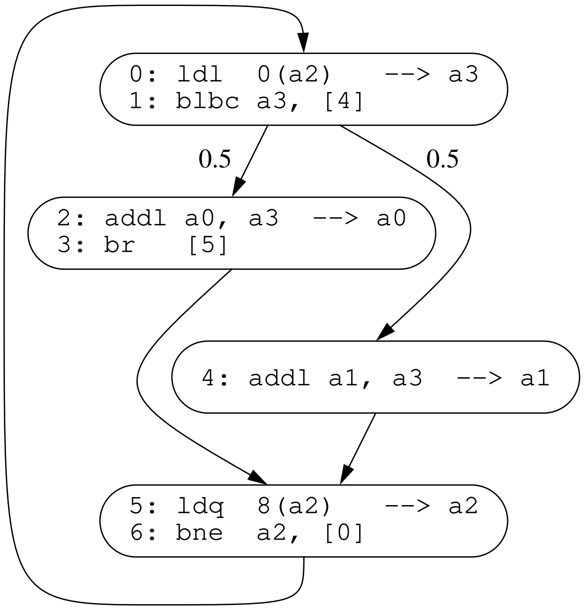

Figure 4: Assembly code for a simple loop. Arc weights indicate the taken / not-taken probabilities for the conditional branch.

instr. number	0	1	4	5	6	0	1	2	3	5	6	...
type (y)	0	1	0	0	1	0	1	0	1	0	1	...
dep. instr. dist. (s)	1	0	1	4	0	1	0	1	5	5	0	...

Figure 5: A section of the trace for the code shown in Figure 4. The instructions are listed in the order they appear in the trace, along with their type and dependent instruction distance. Note that $s_{\max}=5$, so an s value of 5 indicates either a dependence distance of at least 5 or no dependence at all.

When the processor is in this state:

- the push out of the pipe is 1, since there is an instruction in the last stage;
- the flow out of the pipe is 1, since the push is 1 and the pipe never stalls (i.e., the pull out is always 1);
- the pull into the pipe is also 1 since there is nothing for the next instruction to be dependent on (if there had been an instruction in the first pipe stage, the pull would be 0, since $s_0=0$ means the next instruction would be dependent on this one);
- the push out of the issue buffer into the pipe is 1, since there is an instruction in the issue buffer;
- the flow from issue to the pipe is 1, since both push and pull are 1;

	cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
instructions in processor	fetch	1			4	5	6			0	1			2	3		5	6			0
	issue	0	1	1		4	5	6	6		0	1	1		2	3		5	6	6	
	pipe 0		0		1		4	5		6		0		1		2	3		5		6
	pipe 1	6		0		1		4	5		6		0		1		2	3		5	
Markov chain state	x^F	1	0	0	1	1	1	0	0	1	1	0	0	1	1	0	1	1	0	0	1
	x^I	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	0
	x^P	01	10	01	10	01	10	11	01	10	01	10	01	10	01	10	11	01	10	01	10
	y_0,y_1	0,1	1,0	1,0	0,0	0,0	0,1	1,0	1,0	0,1	0,1	1,0	1,0	0,1	0,1	1,0	0,1	0,1	1,0	1,0	0,1
	s_0,s_1	1,0	0,1	0,1	1,4	1,4	4,0	0,1	0,1	1,0	1,0	0,1	0,1	1,5	1,5	5,5	5,0	5,0	0,1	0,1	1,0

Figure 6: A cycle-by-cycle diagram of the one-pipe processor executing the code from Figure 4. The upper half shows the instructions (labeled by instruction number) in each buffer and pipe stage. The lower half shows the corresponding Markov chain state vector.

- the pull into the issue buffer is 1, since it will be empty after the flow into the pipe;
- the push out of the fetch buffer is 0, since it is empty;
- the flow from fetch to issue is 0, since the push is 0;
- the pull into the fetch buffer is 1, since there will be no branch remaining in fetch or issue;
- the flow into the issue buffer is 1, since the pull is 1 (and the push is always 1).

Given this information, we can compute most of the next state vector: $x^{F'} = 1$, $x^{I'} = 0$, and $x^{P'} = 10$. We also know that the y and s values "shift over" by one instruction, i.e., $y_0' = y_1 = 0$ and $s_0' = s_1 = 1$, since one instruction was issued. However, some statistics from the trace are needed to determine y_1' and s_1'.

The information extracted from the trace for this model is:

$$\text{parSeq}\left[y_0, s_0, y_1, s_1, y_1', s_1'\right] = \text{Prob}\left[\text{next instr is type } y_1', \text{ distance } s_1' \mid \text{previous instr's are } y_0/s_0 \text{ and } y_1/s_1\right]$$

In this case, we need parSeq$[1,0,0,1,y_1',s_1']$ for each possible y_1', s_1' pair. Looking at the trace (Figure 5), there are four places where this y_0, s_0, y_1, s_1 sequence occurs:

1. instructions 1 and 4, followed by instruction 5, in which case $y_1' = 0$ and $s_1' = 4$;
2. instructions 6 and 0 (first occurrence), followed by instruction 1, in which case $y_1' = 1$ and $s_1' = 0$;
3. instructions 1 and 2, followed by instruction 3, in which case $y_1' = 1$ and $s_1' = 5$;
4. instructions 6 and 0 (second occurrence, remembering that the trace "wraps around", repeating the pattern shown in Figure 5), followed by instruction 1, in which case $y_1' = 1$ and $s_1' = 0$.

From this, we see that:

- $y_1' = 0, s_1' = 4$ with probability 0.25;
- $y_1' = 1, s_1' = 0$ with probability 0.5;
- $y_1' = 1, s_1' = 5$ with probability 0.25.

Going back to the state described above — $x^F = 0$, $x^I = 1, x^P = 01, y = [10], s = [01]$ — there are three possible transitions:

1. $x^{F'} = 1$, $x^{I'} = 0$, and $x^{P'} = 10$, $y' = [00]$, $s = [14]$ with probability 0.25;
2. $x^{F'} = 1$, $x^{I'} = 0$, and $x^{P'} = 10$, $y' = [01]$, $s = [10]$ with probability 0.5;

3. $x^{F'} = 1, x^{I'} = 0$, and $x^{P'} = 10, y' = [01], s = [15]$ with probability 0.25.

These probabilities match the four successors of this state in Figure 6: cycles 3, 8, 12, and 19.

The transition probabilities for every other state are computed similarly, using the complete parSeq table which is extracted from the trace. With the resulting transition matrix, we can compute the state probabilities. In this simple example, the computed probabilities exactly match the states shown in Figure 6.

We now have the probability of the processor being in each state, as well as the transition probabilities between every pair of states. A state transition implies a specific number of instructions flowing from each component to the next. Given this information, we can compute the flow probability distribution for each connection. For example, the probability that one instruction flows from fetch to issue is:

$$\sum_x \left(\text{Prob}\,[\text{state } x] \cdot \sum_y \text{Prob}\,[x \to y \text{ transition}] \right)$$

where the outer sum is over all states x, and the inner sum is over all transitions $x \to y$ which cause a flow of one instruction from fetch to issue. The IPC is then just the weighted average of the flow probability distribution. (This weighted average will be the same at the fetch-issue and issue-pipe connections.)

5.4 Results

Table 1 shows the performance estimates generated by this model, compared to the simulated performance, for a few small benchmarks. The first benchmark (linked list) is the linked list traversal described above. The second (floating point) is another simple loop, which traverses two vectors, doing a floating point multiplication and addition in each iteration. The third (compress) is from the SPECint92 suite. The fourth (Livermore loops) is a standard floating point benchmark. Compress and Livermore loops are run with small data sets (around a half million cycles).

For this processor microarchitecture, the statistical model produces results within 1% of the simulator. Both the statistical model and simulator produce more detailed information in addition to the IPC value, e.g., resource usage. For example, Table 2 shows the fraction of cycles in which each component contains 0, 1, or 2 instructions, as well as the average number of instructions in the component, for the compress benchmark on the no-branch-prediction processor. The data generated by the statistical model is again very close to the simulated data.

	perfect br. pred.		no branch pred.	
benchmark	model	sim.	model	sim.
linked list	0.7363	0.7362	0.5542	0.5542
floating point	0.9514	0.9513	0.9162	0.9162
compress	0.7784	0.7871	0.7205	0.7291
Livermore lps	0.8875	0.8875	0.8371	0.8372

Table 1: Modeled vs. simulated IPC for the single-pipe processor model.

	fetch		issue	
	model	sim.	model	sim.
0 instructions	0.1040	0.1052	0.0743	0.0737
1 instruction	0.8960	0.8948	0.9257	0.9263
avg. # instr's	0.8960	0.8948	0.9257	0.9263

	pipe	
	model	sim.
0 instructions	0.0000	0.0000
1 instruction	0.5590	0.5418
2 instructions	0.4410	0.4582
avg. # instr's	1.5589	1.4582

Table 2: Modeled vs. simulated component occupancies for the compress benchmark, on the single-pipe processor model. The numbers are the fraction of cycles in which there are a particular number of instructions in the specified component.

6 The Monolithic Three-Pipe Model

This section presents a more complex microarchitecture and introduces some new concepts which are necessary to model it.

This processor has three pipelines: the first executes integer instructions with a latency of one cycle, the second executes floating point instructions with a latency of five cycles, and the third executes memory instructions with a latency of two cycles. All branches are executed by the integer pipe. The fetch and issue buffers are similar to their counterparts in the one-pipe model, but they can each hold two instructions. As before, there are two zversions of the fetch buffer: one with perfect branch prediction and one with no branch prediction (branches are resolved on issue and cause a one-cycle bubble). Up to two instructions can be issued per cycle, and all instructions are issued in order. See Figure 7.

There are four instruction types: integer, floating point, memory, and branch. Each instruction has three dependent instruction distances, one for each pipe. Consider this small

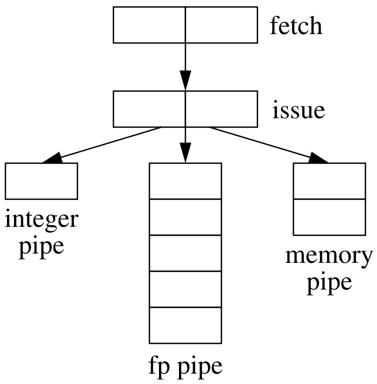

Figure 7: A three-pipe processor microarchitecture.

piece of an assembly code trace:

```
0:  addt f1, f2 --> f3
1:  ldt 0(i1) --> f4
2:  addt f6, f7 --> f8
3:  addt f3, f4 --> f5 ;s = [s_max 1 0]
```

The `addt`s are floating point instructions, and the `ldt` is a memory instruction. The dependent instruction distances are shown for the last `addt` (instruction #3). It is not dependent on any integer instruction, so the integer distance is $s_{\max}$. It is dependent on the second previous floating point instruction (instruction #0, which writes to register `f3`), so the floating point distance is 1. Finally, it is dependent on the previous memory instruction (#1), so the memory distance is 0.

The state vector for this processor model is an extension of the one-pipe processor's state:

- x^F = # instructions in the fetch buffer ($0 \le x^F \le 2$)
- x^I = # instructions in the issue buffer ($0 \le x^I \le 2$)
- x_i^{int} = # instructions in integer pipe stage i ($0 \le x_i^{\text{int}} \le 1$ for $i = 0$)
- x_i^{fp} = # instructions in floating point pipe stage i ($0 \le x_i^{\text{int}} \le 1$ for $0 \le i \le 4$)
- x_i^{mem} = # instructions in memory pipe stage i ($0 \le x_i^{\text{int}} \le 1$ for $0 \le i \le 1$)
- y_i = the type of the i^{th} next instruction: 0 = integer, 1 = floating point, 2 = memory, 3 = branch ($y_i \in \{0,\ldots,3\}$ for $0 \le i \le 3$)
- s_{ij} = the dependent instruction distance of the i^{th} next instruction ($0 \le s_{ij} \le s_{\max}$ for $0 \le i \le 3, 0 \le j \le 2$)

The instruction types range from 0 to 3 (three pipes plus branches). There are three distances for each instruction, i.e., the i^{th} next instruction is dependent on the s_{ij}^{th} previous instruction in pipe j.

Since there can be up to four instructions in fetch and issue, type information must be kept for at least four instructions, in order to correctly deal with branches.

The floating point pipe is deepest and thus determines the value of $s_{\max}$. (A single $s_{\max}$ value is used for simplicity, but we could actually use a different one for each pipe.) Consider the second instruction in the issue buffer. If the instruction ahead of it in issue is a floating point instruction, and the floating point pipe is full, then a dependence on any of the five previous floating point instructions will cause a stall (one in issue plus four in the pipe; not counting the last one in the pipe because its result can be forwarded). So we need s values from 0 to 4, plus one more to indicate a longer or no dependence. This implies $s_{\max} = 5$.

This state vector definition results in a state space size of

$$3 \times 3 \times 2^1 \times 2^5 \times 2^2 \times 4^4 \times 6^{4\times 3} = 1.28 \times 10^{15}$$

This is far too large to solve successfully.

One optimization is to remove the s values from the state vector. For each state, the probability of each possible s value can be computed using information extracted from the trace (the conditional probabilities of s, depending on the current y sequence). To do this accurately requires adding the types of the four most recently issued instructions to the state vector:

- y_i^{prev} = the types of the two previously issued instructions ($y_i^{\text{prev}} \in \{0,\ldots,3\}$ for $0 \le i \le 3$)

This results in a state space with:

$$3 \times 3 \times 2^1 \times 2^5 \times 2^2 \times 4^4 \times 4^4 = 1.51 \times 10^8 \text{ states}$$

which is still too large.

The next section shows how the Markov chain can be partitioned into smaller, more manageable pieces.

7 The Partitioned Three-Pipe Model

7.1 Partitioning the state space

The processor model used in this section is the same three-pipe model used previously (see Figure 7). The difference is in the design of the state space. Instead of modeling the processor with one large Markov chain, we partition it, modeling each partition with its own Markov chain.

As with the one-pipe model, control dependences are modeled by the pull into the fetch buffer. This pull depends on instructions in both the fetch and issue buffer. If there is a branch (assuming the no-prediction model) in either buffer, the pull is zero, i.e., fetching is stalled until the branch is issued. Because of this tight coupling between the two buffers, they are lumped together in one Markov chain.

The pipes are relatively independent of the issue buffer and of each other. Each pipe is represented by a separate Markov chain.

This partitioning leads to a fetch/issue state vector:

- x^F: # instructions in fetch
- x^I: # instructions in issue
- y_i: types of the next four instructions to be issued
- y_i^{prev}: types of the four previously issued instructions

and three pipe state vectors of the form:

- x_i^P: instructions in pipe

The state vector elements are the same ones used in the monolithic model.

7.2 Push and pull probabilities

There are now four Markov chains, which form a set of simultaneous equations. For the pipe-i (0=integer, 1=floating point, or 2=memory) model, the unknown is the push out of the issue buffer:

$$p_i^{\text{issue}} = \text{the number of instructions ready to flow from issue to pipe-}i$$

For the fetch/issue model, the unknowns are the pulls into the pipes:

$$q_i^{\text{pipe}-j} = \text{the number of pipe-}i\text{ instructions which are independent of all instructions in pipe-}j$$

There is a pull *by* each pipe *into* each pipe. This is because an instruction can, in general, be dependent on an instruction in any pipe. For an instruction to flow from issue into pipe-i, it must be independent of the instructions in all of the pipes ($q_i^{\text{pipe}-0} = q_i^{\text{pipe}-1} = q_i^{\text{pipe}-2} = 1$).

The push value depends on the current state of the fetch/issue model. The pull values depend on the current states of the pipe models. Since the fetch/issue model does not know the current state of the pipe models, and vice versa, we instead use push and pull probability distributions:

$$\begin{aligned}
\tilde{p}_{ik}^{\text{issue}} &= \text{Prob}\,[\text{there are } k \text{ instr's ready to flow from issue to pipe-}i] \\
\tilde{q}_{ik}^{\text{pipe}-j} &= \text{Prob}\,[\text{there are } k \text{ pipe-}i \text{ instr's independent of all instructions in pipe-}j]
\end{aligned}$$

Given the pull distributions, the fetch/issue Markov chain can be solved for its state distribution. From this, the push distribution can be directly computed. Similarly, given the push distributions from the fetch/issue buffer, the pipe Markov chains can be solved, producing the pull distributions. An iterative relaxation technique can be used to generate a simultaneous solution for all four Markov chains.

The above technique relies on an implicit assumption. The fetch/issue and pipe states must be statistically independent. More specifically, the push probabilities (from issue) must be independent of the pipe states, and the pull probabilities (from the pipes) must be independent of the fetch/issue state. This turns out to be an inaccurate assumption. For example, consider the following sequence of instructions:

```
0:  addt f1, f2 --> f3
1:  addt f4, f5 --> f6
2:  stt f3
3:  stt f6
4:  subt f7, f8 --> f9
```

In this example, the compiler has separated the floating point instructions from the stores which depend on them. The first add-store pair (#0 and #2) are separated by another floating point add. The second pair (#1 and #3) are separated by a store. This means that the first store is dependent on the second previous floating point instruction, so the dependence distance is 1, while the second store is dependent on the first previous floating point instruction, so its dependence distance is 0. Considering both memory instructions, the probability of dependence distance 0 is 0.5 and the probability of dependence distance 1 is 0.5. However, these distances are correlated with the current state of the issue buffer: when there are two stores in issue, the dependence distance will be 1, and when there are one store and one floating point subtract, the dependence distance will be 0. Thus the pull by the floating point pipe is dependent on the issue state, which violates the independence assumption.

7.3 Correlated Markov chains

The solution to this problem is to drop the assumption that the fetch/issue and pipe models are entirely independent. The model must allow for some correlation between them. To do this, the pipe state vectors are augmented:

- x_i^P (same as before)
- y_i: types of next two instructions to be issued
- y_i^{prev}: types of the four previously issued instructions

instr. number	0	1	4	5	6	0	1	2	3	5	6	...
type (y)	2	3	0	2	3	2	3	0	3	2	3	...
dep. instr. dist's (s^{int})	5	5	5	5	5	5	5	5	5	5	5	...
(s^{fp})	5	5	5	5	5	5	5	5	5	5	5	...
(s^{mem})	0	0	0	1	0	0	0	0	5	1	0	...

Figure 8: A section of the trace for the code shown in Figure 4, with types and dependence distances for the three-pipe model.

The y and y^{prev} values in the pipe states correlate with the other pipes and with the first two y values and the four y^{prev} values in the fetch/issue state. That is, if the fetch/issue Markov chain is in a particular state, each pipe must be in a state with matching y values, and vice versa. The push and pull probabilities can then be made conditional on $y_{0\ldots1}$ and $y^{\text{prev}}_{0\ldots3}$ to allow for the correlation. (The pipe states contain only the next two instruction types, as opposed to four like the fetch/issue state. This is done to reduce the sizes of the pipe model state spaces.)

The resulting state space sizes are:

- fetch/issue: $3 \times 3 \times 4^4 \times 4^4 = 589,824$ states
- integer pipe: $2^1 \times 4^2 \times 4^4 = 8,192$ states
- floating point pipe: $2^5 \times 4^2 \times 4^4 = 131,072$ states
- memory pipe: $2^2 \times 4^2 \times 4^4 = 16,384$ states

7.4 Example

This example uses the same program as is used with the single-pipe example (Figure 4). The trace is the same, but there are now four possible instruction types and three different dependent instruction distances for each instruction (see Figure 8).

Figure 9 shows the processor states at each cycle. Remember that there are now four separate states: one for fetch/issue and one for each of the three pipes. (The floating point pipe is omitted from the figure.)

In cycles 2 and 3, the fetch and issue buffers are in the state: $x^F = 0$, $x^I = 1$, $y^{\text{prev}} = [3232]$, $y = [3023]$. In both of these cycles, the four previously issued instructions are 3, 5, 6, and 0 (types 3, 2, 3, and 2), and the next four instructions to be issued are 1, 4, 5, and 6 (types 3, 0, 2, and 3).

Since the instruction in issue is an integer instruction (all branches are executed by the integer pipe), we need the pull *by* each pipe *into* the integer pipe. In this example, the integer pull is always 1 (since the integer dependence distances are all 5) and the floating point pull is always 1 (since there are no floating point instructions). The pull by the memory pipe can be either 0 or 1, with the following probabilities:

$$\begin{aligned}\text{Prob}[\text{pull by mem into int pipe} = 0 \mid & \\ y^{\text{prev}} = [3232], y = [30]] &= 0.5 \\ \text{Prob}[\text{pull by mem into int pipe} = 1 \mid \ldots] &= 0.5\end{aligned}$$

These probabilities are generated at the same time the memory pipe transition is computed (see below).

The instruction sequencing information extracted from the trace for this model is:

$$\begin{aligned}\text{instrSeq}[y_{0\ldots6}] = \text{Prob}[&\text{next instr is type } y_6 \mid \\ &\text{previous instr's are types } y_{0\ldots5}]\end{aligned}$$

In the case when the pull is 1, we need $\text{instrSeq}[3,2,3,0,2,3,y'_3]$ for all possible values of y'_3. An examination of the trace (Figure 8) shows that $y'_3 = 2$ with probability 1.

Given these probabilities, there are two possible transitions:

1. if pull = 0: $x^{F'} = 0, x^{I'} = 1, y^{\text{prev}'} = [3232], y' = [3023]$ (no change in state), with probability 0.5 (this corresponds to the cycle 2 $\rightarrow$ cycle 3 transition)

2. if pull = 1: $x^{F'} = 2, x^{I'} = 0, y^{\text{prev}'} = [2323], y' = [0232]$ (the branch has left issue and two new instructions have been fetched), with probability 0.5 (this corresponds to the cycle 3 $\rightarrow$ cycle 4 transition)

While building the fetch/issue transition matrix, we also have to compute the conditional push out of issue (which will be used to compute the transition probabilities for the pipe models). In this particular state, we have:

$$\begin{aligned}\text{Prob}[\text{push into int pipe} = 0 \mid & \\ y^{\text{prev}} = [3232], y = [30]] &= 0 \\ \text{Prob}[\text{push into int pipe} = 1 \mid \ldots] &= 1 \\ \text{Prob}[\text{push into fp pipe} = 0 \mid \ldots] &= 1 \\ \text{Prob}[\text{push into fp pipe} = 1 \mid \ldots] &= 0 \\ \text{Prob}[\text{push into mem pipe} = 0 \mid \ldots] &= 1 \\ \text{Prob}[\text{push into mem pipe} = 1 \mid \ldots] &= 0\end{aligned}$$

(Normally, we would have to consider all states with these y^{prev} and y values; in this example, it just happens that this is the only state with these values.)

In cycle 2, the memory pipe is in the state: $x^P = 10$, $y^{\text{prev}} = [3232]$, $y = [30]$. First, we compute the pull by the memory pipe into each pipe. For this purpose, we extract dependence distance probability information from the trace:

	cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
instr's in proc.	fetch	01				45	6			01				23			56			
	issue		01	1	1		45	6	6		01	1	1		23	3		56	6	6
	int pipe	6	-	-	-	1	-	4	-	6	-	-	-	1	-	2	3	-	-	-
	mem pipe	--	--	0-	-0	--	--	5-	-5	--	--	0-	-0	--	--	--	--	--	5-	-5
fetch/ issue state	x^F	2	0	0	0	2	1	0	0	2	0	0	0	2	0	0	2	0	0	0
	x^I	0	2	1	1	0	2	1	1	0	2	1	1	0	2	1	0	2	1	1
pipe states	int x^{P0}	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0
	mem x^{P2}	00	00	10	01	00	00	10	01	00	00	10	01	00	00	00	00	00	10	01
shared state	y^{prev}	0323	0323	3232	3232	2323	2323	2302	2302	3023	3023	0232	0232	2323	2323	3230	2303	2303	3032	3032
	y	2302	2302	3023	3023	0232	0232	3230	3230	2303	2303	3032	3032	0323	0323	3232	2323	2323	3230	3230

Figure 9: A cycle-by-cycle diagram of the three-pipe processor executing the code from Figure 4. The upper half shows the instructions (labeled by instruction number) in each buffer and pipe stage. The lower half shows the corresponding Markov chain state vectors for the partitioned model. The y^{prev} and y values are part of all four state vectors. The x^{P1} (floating point pipe state) values are omitted because the trace contains no floating point instructions.

$$\begin{aligned}
&\text{distProb}[y^{\text{prev}}_{0\ldots3}, y_{0\ldots1}, s_{0\ldots1}] = \\
&\quad \text{Prob}\,[\text{next two instr's have distances } s_0, s_1, s_2 \mid \\
&\qquad \text{previous instr's are types } y^{\text{prev}}_{0\ldots3} \text{ and} \\
&\qquad \text{next instr's are types } y_{0\ldots1}]
\end{aligned}$$

(There is a separate distProb table for each pipe i, with distances for dependences on instructions in pipe i.)

The next instructions to be issued are a branch ($y_0 = 3$) and an integer instruction ($y_1 = 0$). They have a memory dependent instruction distances of 0 and 0 with probability distProb$[3232, 30, 00] = 1$. Since there is an instruction in the first stage of the memory pipe, this branch is dependent and cannot issue. Thus the pull by the memory pipe into the integer pipe is 0 (with probability 1). There is another state (cycle 3) with the same y^{prev} and y values. In this state, the pull is 1 with probability 1. It turns out that both states have equal probabilities in the state distribution, so:

$$\begin{aligned}
\text{Prob}\,[\text{pull by mem pipe into int pipe} = 0 \mid & \\
y^{\text{prev}} = [3232], y = [30]] &= 0.5 \\
\text{Prob}\,[\text{pull by mem pipe into int pipe} = 1 \mid \ldots] &= 0.5
\end{aligned}$$

The pull by the memory pipe into the floating point and memory pipes turns out to be 1 with probability 1. These are the pull probabilities used by the fetch/issue model above.

The push into the memory pipe, given $y^{\text{prev}} = [3232]$ and $y = [30]$, as computed by the fetch/issue model (see above), is 0 with probability 1. There will therefore be zero flow into the pipe. Since the pipe state includes y^{prev} and y, we also need to look at the flow into the other pipes. The push into the integer pipe is 1, with probability 1. However, the pull into the integer pipe by the memory pipe is 0. The push into the floating point pipe is 0, so no instructions will flow into any of the three pipes. From this, we can compute the one possible state transition:

- $x^P = 01$, $y^{\text{prev}} = [3232]$, $y = [30]$, with probability 1

The full transition matrix for each partition is built and solved for the state distribution. The conditional push and pull probabilities are also computed. This process is iterated until it settles on a simultaneous solution.

7.5 Results

Table 3 shows the performance estimates generated by this model, compared to simulated performance. The benchmarks are the same as used for Table 1.

The modeled IPCs for the first three benchmarks are within 2% of the simulated value. The Livermore loops results, however, are off by about 10%. We have identified the problem and are looking for ways to further improve the accuracy without significantly increasing the size of the state spaces.

As an example of more detailed performance information, the component utilization numbers for the compress benchmark running on the no-branch-prediction processor are given in Table 4. Even more detailed information —

benchmark	perfect br. pred.		no branch pred.	
	model	sim.	model	sim.
linked list	1.0060	1.0039	0.5867	0.5859
floating point	1.0564	1.0147	0.9781	0.9426
compress	1.1656	1.1427	1.0061	0.9884
Livermore lps	1.0629	0.9692	0.9776	0.8970

Table 3: Modeled vs. simulated IPC for the three-pipe processor model.

	fetch		issue	
	model	sim.	model	sim.
0 instructions	0.1879	0.2000	0.1038	0.1019
1 instruction	0.1164	0.1176	0.0887	0.0870
2 instructions	0.6957	0.6824	0.8076	0.8111
avg. # instr's	1.5078	1.4823	1.7039	1.7091

	int pipe		mem pipe	
	mod.	sim.	mod.	sim.
0 instructions	0.4190	0.4292	0.3983	0.3964
1 instruction	0.5810	0.5708	0.3532	0.3720
2 instructions			0.2484	0.2316
avg. # instr's	0.5810	0.5708	0.8500	0.8352

Table 4: Modeled vs. simulated component occupancies for the compress benchmark, on the three-pipe processor model.

e.g., the number of integer instructions in the fetch buffer — can be computed from the model.

8 Conclusions

The work presented here shows that statistical models for superscalar processor performance are potentially very useful. Based on a one-time-only analysis of each benchmark trace, performance estimates can quickly be generated for an entire family of processors. This facilitates exploring the processor design space far more quickly than is possible with trace-driven simulation.

The three-pipe model described in the previous section takes only a few minutes to run. Even more importantly, this run-time is independent of the length of the trace. Once the trace analyzer is built into the benchmark instrumentation code (using Atom), trace analysis will take only slightly longer than it currently takes just to generate the trace.

This work is only a first step. There are two concerns in extending the model to more complex processors. The first is the size of the state space. We believe that the state space can be kept reasonable by partitioning the Markov chain. This should be especially effective for processors which consist of several relatively decoupled units. The second concern is the accuracy of the model. This paper shows that very accurate results can be achieved for simple processors. We plan to experiment with more complex processors to determine whether our framework scales up well.

Acknowledgments

This research has been supported by NSF Grant CCR9423272.

References

[BHLS96] Bryan Black, Andrew S. Huang, Mikko H. Lipasti, and John Paul Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proc. ICCD*, 1996. (to appear Oct., 1996).

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[DAF94] Pradeep K. Dubey, George B. Adams, III, and Michael J. Flynn. Instruction Window Size Trade-Offs and Characterization of Program Parallelism. *IEEE Transactions on Computers*, 43(4):431–442, April 1994.

[DSP93] Trung A. Diep, John P. Shen, and Mike Phillip. EXPLORER: A Retargetable and Visualization-Based Trace-Driven Simulator for Superscalar Processors. In *Proc. MICRO-26*, pages 225–235, 1993.

[Jou89] Norman P. Jouppi. The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[Res92] Sidney I. Resnick. *Adventures in Stochastic Processes*. Birkhäuser, 1992.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proc. PLDI*, pages 196–205, 1994.