CARNEGIE MELLON UNIVERSITY

A FRAMEWORK FOR STATISTICAL MODELING OF SUPERSCALAR PROCESSOR PERFORMANCE

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

by

Derek B. Noonburg

Pittsburgh, Pennsylvania
September 1997

Abstract

This dissertation presents a statistical approach to modeling superscalar processor performance. Instead of directly modeling an execution trace, as with standard simulation-based performance models, a statistical model works with the probabilities of instruction types, instruction sequences, and processor states.

The program trace and machine are analyzed separately, and the performance is computed from these two inputs. The statistical flow graph is introduced as a compact representation for program traces. The characterization of a specific processor and the statistical flow graph for a specific benchmark are combined to form a Markov chain. In order to reduce the state space size, this Markov chain is partitioned into several smaller sub-models.

Simulation-based techniques require extremely long run times, especially as traces reach lengths in the billions of instructions. The statistical approach presented here dramatically reduces the time required to explore a microarchitectural design space. Separating the program and machine models allows the time-consuming part of the modeling process, which takes time proportional to the trace length, to be done only once per benchmark. The statistical model and program trace representation are extended to include various microarchitectural features, including branch prediction, I-caches, D-caches, and value prediction. Enough information is extracted from a single trace analysis to allow modeling of that benchmark on a large family of processors.

The results show that the statistical model produces IPC estimates very close — within a few percent — to the IPCs measured by a cycle-accurate simulator. An explanation for the modeling error is presented, and a technique is demonstrated by which statistical model run time can be traded for improved accuracy.

Acknowledgments

There are many people who deserve my thanks for their help and support throughout my tenure as a graduate student. First, I want to thank my advisor, John Paul Shen. John started me on the topic of statistical modeling by suggesting the idea of a physics of computer architecture, and has encouraged me through the project, despite many wrong paths and dead ends. In addition, I want to thank the rest of my thesis committee — Randy Bryant, Hyong Kim, and Daniel Siewiorek from CMU and Keith Diefendorff from Apple Computer — who provided many useful suggestions. I would like to thank all of my officemates and other students in our research group for their friendship and for conversations about pretty much everything: Shirish Sathaye, Chriss Stevens, Bryce Cogswell, Trung Diep, Chris Newburn, Andrew Huang, Alex Dean, and Bryan Black. Finally, I want to thank my family and friends for their support through this whole thing.

Contents

1 Introduction **1**

1.1 Performance Modeling 1

1.2 Statistical Modeling 3

1.2.1 The Modeling Process 6

1.2.2 Advantages and Disadvantages 8

1.2.3 Potential Uses 8

1.3 Previous Work 9

1.3.1 The Piecewise Linear Model 9

1.3.2 Smoothability 11

1.3.3 Instruction Independence Probabilities 13

1.3.4 Memory Stack Distance 13

2 Parallelism Distributions **15**

2.1 Definition 15

2.2 Transforms 17

2.3 Why Distributions? 18

2.4 Matrix Multiplication Model 20

2.4.1 Specifics 24

2.4.2 Benchmarks 30

2.4.3 Experimental Results 30

3 Using Markov Models 33

3.1 Markov Models 33

3.2 Instruction Characteristics 36

3.3 Processor Models 39

3.4 A One-Pipeline Processor 41

3.4.1 The Model 41

3.4.2 Transition Matrix Computation 44

3.4.3 Experimental Results 55

3.5 A Three-Pipeline Processor 57

4 Partitioned Markov Models 61

4.1 Partitioning 61

4.2 A Three-Pipeline Processor 62

4.2.1 The Model 62

4.2.2 Transition Matrix Computation 68

4.2.3 Experimental Results 77

4.3 Aliasing 80

5 The Statistical Flow Graph 83

5.1 The Statistical Flow Graph 83

5.2 The Three-Pipeline Processor 87

5.3 Aliasing 91

5.4 Experimental Results 93

5.5 Adding Instruction Addresses 97

6 Modeling Branch Prediction **101**

6.1 Branch Predictors . . . 101

6.2 Statistical Modeling . . . 102

6.3 Experimental Results . . . 104

7 Modeling Caches **107**

7.1 Instruction Cache . . . 107

7.2 Data Cache . . . 109

7.3 Trace Analysis . . . 109

7.4 Experimental Results . . . 111

8 Modeling Value Prediction **115**

8.1 Value Prediction . . . 115

8.2 Statistical Modeling . . . 116

8.3 Experimental Results . . . 117

9 Conclusions **119**

9.1 Contributions . . . 119

9.2 Summary of Results . . . 121

9.3 Limitations . . . 122

9.4 Future Work . . . 124

9.4.1 Increasing Accuracy. . . . 124

9.4.2 Increasing Processor Complexity. . . . 125

9.4.3 Other Uses . . . 126

A Markov Model Mathematics **129**

A.1 Markov Chains . . . 129

A.2 Partitioned Markov Chains . . . 131

B Implementation **133**

B.1 Overview 133

B.2 Fetch Buffer 134

B.3 Issue Buffer 139

B.4 Pipeline 143

List of Figures

1.1 Simulator and statistical model output . . . 4
1.2 Trace-driven simulator framework . . . 5
1.3 Statistical model framework . . . 6
1.4 Jouppi's piecewise linear model . . . 10
1.5 Theobald et al.'s smoothability function . . . 12

2.1 Time sequence compared to probability distribution . . . 19
2.2 Matrix multiplication model . . . 21
2.3 Parallelism distribution function . . . 24
2.4 Matrix multiplication model results . . . 31

3.1 Microarchitecture components and connections. . . . 40
3.2 The one-pipe processor. . . . 42
3.3 Control flow graph for a simple loop. . . . 45
3.4 Trace for the simple loop. . . . 46
3.5 State transition in one-pipe processor. . . . 50
3.6 One-pipe transition matrix for the simple loop. . . . 53
3.7 Markov model results for one-pipe processor with perfect branch prediction 55
3.8 Markov model results for one-pipe processor with no branch prediction . . 56
3.9 A three-pipe processor. . . . 57

4.1 Trace for the simple loop. 71

4.2 Partitioned Markov model results, perfect branch prediction 78

4.3 Partitioned Markov model results, no branch prediction 79

5.1 The statistical flow graph. 84

5.2 Instruction sequences in the aliasing example. 91

5.3 Instruction sequence model results, perfect branch prediction 93

5.4 Instruction sequence model results, no branch prediction 94

5.5 Model and simulation run times 95

5.6 Model run times versus number of sequences 96

5.7 Model results as floating point pipeline depth is decreased 97

5.8 Parallelism distribution results 98

5.9 Error rate for instruction sequences augmented with address bits 99

6.1 Branch prediction model results 105

7.1 Cache model results 112

7.2 Cache model results with different miss latencies 113

8.1 Value prediction model results 118

A.1 Solving a system of partitioned Markov chains. 132

List of Tables

2.1 The SPEC95 benchmarks 31

3.1 One-pipe instruction sequencing probabilities for the simple loop. 46

3.2 One-pipe model states for the simple loop. 49

3.3 One-pipe stationary distribution for the simple loop. 53

4.1 Three-pipe model states for the simple loop. 70

4.2 States and transitions of the three-pipe model. 76

4.3 Instruction distance (instrDist) table for aliasing example 81

7.1 Example states for fetch buffer with I-cache model 108

9.1 Statistical model characteristics. 122

Chapter 1

Introduction

This dissertation presents the framework for a new type of superscalar processor performance model. These models use statistical techniques in place of the usual simulation-based techniques. Instead of directly modeling sequential execution, a statistical model works with the probabilities of instruction types, instruction groups, and processor states. Instructions are no longer considered individually. Instead, they are characterized by their essential properties so that an instruction trace can be replaced by a statistical analysis of instruction properties. Statistical modeling enables much faster design space exploration. Whereas a simulation-based model takes time proportional to the number of instructions executed, a statistical model takes time related to the complexity of the machine model, independent of the number of instructions in the trace.

1.1 Performance Modeling

Currently, the most widely used and well understood technique for modeling microprocessor performance is timing simulation. This involves constructing a software model of the processor microarchitecture that simulates the behavior of each individual instruction as it

flows through the machine. These models can produce very accurate performance figures [BHLS96, BS97].

The basic simulation technique is **trace-driven simulation** [DSP93, DS95]. A trace is generated for each benchmark program, either by running an instrumented binary [BKW90, SCH+91, SE94] or by hardware monitoring [CE85, GAR+93]. Traces must include all of the executed instructions and instruction addresses, as well as any other necessary data such as memory reference addresses. Because current benchmarks execute billions of instructions, complete instruction traces are extremely large. Once the traces have been collected and stored, the trace-driven simulator is run with each benchmark trace, in turn, as input. If the microarchitecture design is changed, the simulator must be updated, and all of the simulations rerun.

One technique for reducing the size of traces is **trace sampling** [LPI88, Lau94, FP94, DN95, CHM96]. Various methods can be used to select a series of small sections from a complete trace. For example, a trace sample might select 100 sections, each consisting of 1000 consecutive instructions, from a trace containing 100 million instructions. These sections are concatenated and simulated as a single trace. Trace sampling is very effective at reducing trace size and simulation time, but the nature of the induced error is not well understood.

Execution-driven simulation avoids traces altogether [SF89, SF91]. The timing simulator is teamed with an instruction-level functional simulator. The functional simulator runs the benchmark, and effectively supplies an instruction trace to the timing simulator. The timing simulator accepts instructions on the fly, directly from the functional simulator, thus avoiding the storage of any sort of trace. If the functional simulator is fast enough (relative to the timing simulator), execution-driven simulation can be nearly as fast as trace-driven simulation. One disadvantage here is that functional simulators are very complex, since they must simulate the complete system environment.

A hybrid hardware and execution-driven approach is described by Rose and Flanagan [RF96]. Here, a hardware monitor is used to record events on the processor's external bus. This information is used as input to a functional simulator. The bus trace contains all information read into the processor, including memory fetches into the on-chip caches. This allows a very accurate simulation which can include the effects of interrupts, DMA, and other “random” events.

There are two major drawbacks to simulation. First, trace-driven simulators require storage of complete program traces. Benchmark suites such as SPEC95 [SPE95] require simulation of hundreds of billions of instructions. Storing a complete set of instruction traces for SPEC95 requires an enormous amount of disk space. This problem, however, can be solved by using one of the techniques described above.

The second problem is more troublesome. Timing simulators typically take 1,000–100,000 cycles on a host machine to simulate one target processor cycle [BHLS96, CK94]. As an example, simulating 100 billion instructions on a 200 MHz host with a simulator that takes 10,000 host cycles per simulated instruction would take over 50 CPU days. Trace sampling is a possible solution to this problem, but it comes at the cost of accuracy. This dissertation presents another alternative.

1.2 Statistical Modeling

A microprocessor performance model typically has some notion of **processor state**. For example, a trace-driven timing simulator keeps track of all of the instructions currently in flight, as well as its current position within the trace. It might keep a list of the instructions in various buffers and pipeline stages, as well as the current contents of the caches and branch prediction tables.

At the lowest level, the output of a timing simulator is simply a time sequence of states.

cycle	state
0	0
1	1
2	2
3	2
4	2
5	1
6	0
7	2
8	2
9	0

simulator

state	probability
0	30%
1	20%
2	50%

statistical model

Figure 1.1: The time sequence output of a simulator compared to the state probability distribution output of a statistical model.

This sequence is processed to generate information useful to the architect: IPC, buffer usage, etc.

A statistical model also utilizes the concept of processor state. Although it is somewhat smaller than a simulator's state, the statistical model state still contains enough information to determine the behavior of the instructions in flight. The fundamental difference is that, where a simulator generates a time sequence of states, a statistical model produces a state probability distribution. That is, the output of the statistical model is a list of probabilities, one for each state. The probability associated with a particular state is the probability that the processor is in that state in any given cycle, or equivalently, the fraction of cycles in which the processor is in that state.

Figure 1.1 compares the simulation approach to the statistical approach. This very small example is for a program that runs for ten cycles on a processor that has only three possible states. The simulator produces a list of ten states, one for each simulated cycle. The statistical model produces a list of three probabilities, one for each state. Both tables

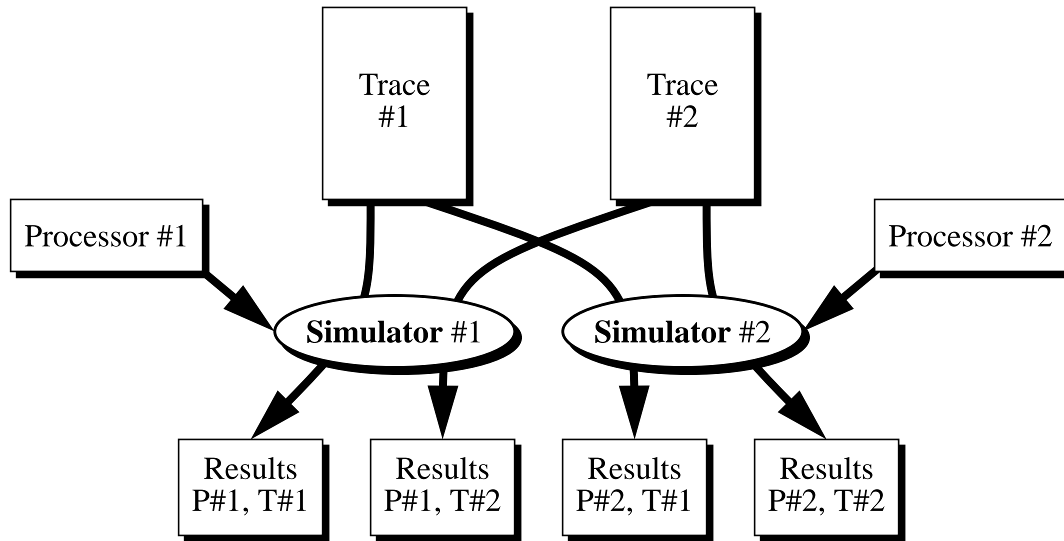


Figure 1.2: Framework for a trace-driven simulator. The simulator has to process each trace repeatedly for every processor.

describe the same data. Three out of ten of the simulated cycles are spent in state 0, hence state 0 has a probability of 30%, and similarly for states 1 and 2.

The most important consequence of this difference — a time series of states vs. state probabilities — is that the simulation run time is proportional to the number of cycles simulated while the statistical model computation time is a function of the number of states. Nearly all of the useful information which can be extracted from the series of states can also be extracted from the state probabilities. Time sequences and probability distributions are discussed in more detail in Chapter 2.

This dissertation presents a framework for statistical performance models. It also presents several processor models built on this framework.

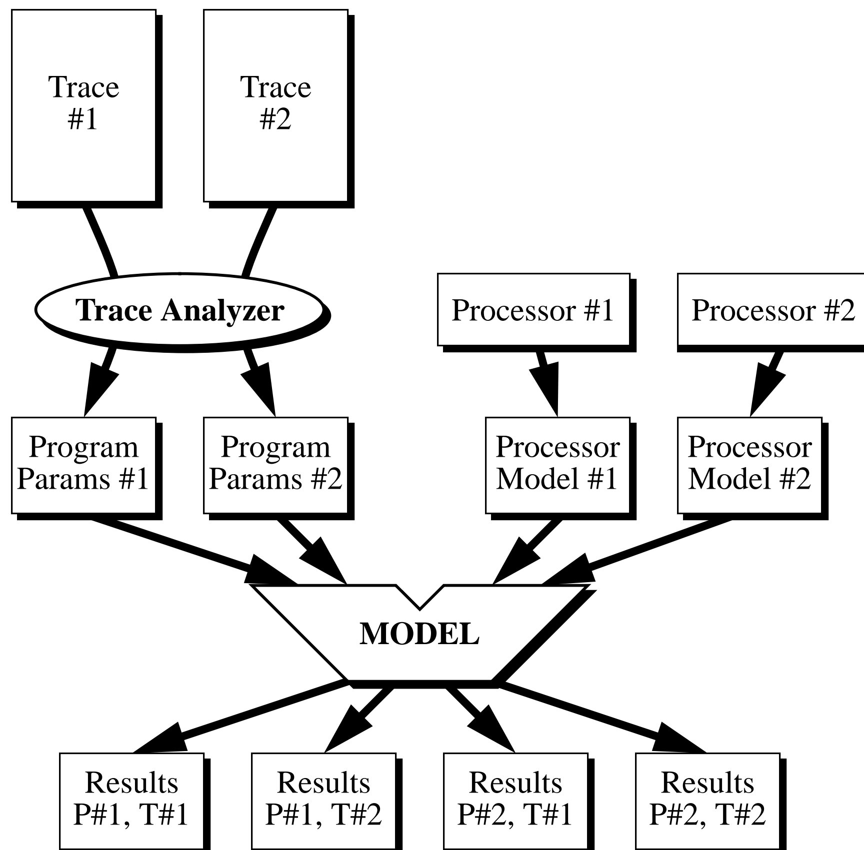


Figure 1.3: Framework for a statistical model. The statistical model processes each trace only once.

1.2.1 The Modeling Process

Exploring a processor design space using a simulator involves two steps (see Figure 1.2). First a simulator is constructed for each processor to be examined. In general, there will be several variants of the design. The simulator may accept parameters (cache sizes, pipeline depths, etc.) describing the specific microarchitecture, or there may be separate simulators for each processor variant. Next, a trace is generated for each benchmark program. Finally, to generate a complete set of performance information, each of the traces is run through each simulator.

The analogous framework for the statistical model is shown in Figure 1.3. As with the simulator, a model is built for each processor. As before, this model may be parameterized for different microarchitectures, or there may be several separate models. Constructing these processor models is very similar to constructing timing simulators. The important difference from the simulation framework is that the statistical model uses two separate steps in place of the simulation step. First, the trace analyzer processes each trace once to generate a set of program parameters. Then, to model a specific benchmark running on a specific processor, the corresponding program parameters and processor model are combined by the statistical modeling step. The effect is that each trace is analyzed only once, not once for every processor as with the simulator. Simulating m benchmarks on n processors requires $m \times n$ separate simulations. Statistical modeling, on the other hand, requires only m trace analyses. It still needs $m \times n$ model computations, but these are relatively fast, compared to simulation or trace analysis. The output of the model is the state probability distribution described above, which can be further analyzed to compute the issue rate and other useful performance information.

Superscalar processor performance can be broken into three components: instruction-level parallelism (ILP), clock speed, and instruction count [HP90]. This work concentrates on the ILP component of performance. Instruction-level parallelism is the interaction of program parallelism and machine parallelism [Jou89] (see Section 1.3.1). The statistical model presented here treats the division between program and machine parallelism explicitly. The program parameters in Figure 1.3 are a representation of program parallelism, and the processor models embody the machine parallelism. The statistical model computes the interaction between the two.

1.2.2 Advantages and Disadvantages

Statistical modeling has a key advantage over timing simulation: each program trace is processed only once. This can be done by storing the trace and running a post-processing tool on it (like trace-driven simulation), or by running an on-the-fly trace analyzer (similar to execution-driven simulation). Trace analysis takes time proportional to the number of instructions executed by the benchmark, just like simulation, but it is done only once, instead of once per processor configuration.

Since the trace is processed once and then discarded, the trace storage problem associated with trace-driven simulation is avoided. Since running the model takes time related to the complexity of the model and independent of the trace length, the execution time problem of simulation is avoided.

Due to its nature, statistical modeling will be slightly less accurate than full timing simulation. However, the results described in later chapters show that it can be quite accurate, and certainly accurate enough for many uses.

1.2.3 Potential Uses

Statistical modeling allows a processor design space to be explored much more quickly than timing simulation. Many different microarchitecture features and parameters can be tried out very quickly. This narrows the design space to a small region which can be further evaluated by a cycle-accurate timing simulator.

There are many other potential uses which are not considered further in this dissertation. By directly examining the state space of a statistical model, it may be possible to gain useful insight into microarchitecture-level factors which affect performance.

Statistical modeling may also be of value to compiler designers and software developers. The program parameters extracted by the trace analyzer could provide a method for

comparing or classifying different programs, as well as comparing different versions of a single program, e.g., with and without a certain optimization.

1.3 Previous Work

Numerous researchers have examined the limits of instruction-level parallelism available in benchmark code, using various ideal machine models [Wal91, AS92, LW92]. There have been relatively few papers that attempt to analytically model the instruction-level parallelism that can be achieved by realistic processors. This section briefly summarizes several ideas, taken from four of these papers, that influenced the work in this dissertation.

1.3.1 The Piecewise Linear Model

Jouppi describes a theoretical performance model that 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, i.e., the maximum number of instructions which can be in execution stages in any one cycle. **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.

The performance (ILP) vs. machine parallelism curve is divided into two linear regions (see Figure 1.4). In the first region, the machine parallelism (MP) is less than the program parallelism (PP), and so the performance is limited by machine parallelism. In this region, performance improves with increased machine parallelism. In the second region, the machine parallelism is greater than the program parallelism, and the performance is limited by the program parallelism. In this region, performance does not change as the machine parallelism is increased further.

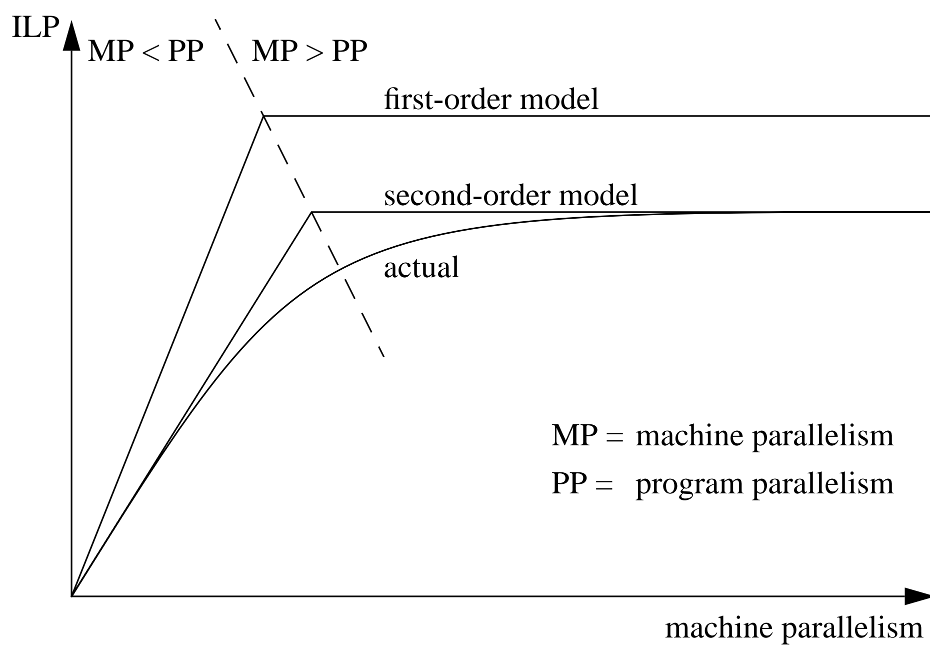


Figure 1.4: Jouppi's piecewise linear superscalar performance model. The upper curve is the first-order model; the middle curve is the second-order model. The lowest curve is the actual performance.

Jouppi describes several factors which account for the differences between the modeled and actual behaviors. These include variations in instruction latency, variations in per-cycle instruction-level parallelism, and variations in parallelism by instruction class. His suggested second-order model takes these nonuniformities into account to explain the lowering of the two performance asymptotes, i.e., the tilting of the machine parallelism-limited asymptote and the dropping of the program parallelism-limited asymptote (see Figure 1.4).

The area in which program and machine parallelism are roughly equal — where the asymptotes meet — is extremely interesting because it contains the optimal operating point. Jouppi's model concentrates on asymptotic behavior: where machine parallelism is significantly less than or greater than program parallelism. It does not provide an explanation for the rounding of the knee in the performance curve in this transition area.

Jouppi's idea of treating program parallelism and machine parallelism as two separate elements of a processor model is fundamental to the work presented in this dissertation.

1.3.2 Smoothability

Theobald et al. introduce the concept of **smoothability of program parallelism** to explain this rounded knee in the transition region [TGH92]. Let P be the average program parallelism of a benchmark. If the program parallelism is perfectly smooth, i.e., evenly distributed over time, then a machine parallelism of exactly P is sufficient to achieve maximum performance. In general, program parallelism is not perfectly smooth over time: more than P instructions are available in some cycles, less than P in others. Theobald et al. define smoothability as the ratio of the performance with a machine parallelism of $\lceil P \rceil$ to the performance with infinite machine parallelism. A perfectly smoothable program would thus have a smoothability of one.

Graphically, smoothability is then a measure of the distance between the corner of

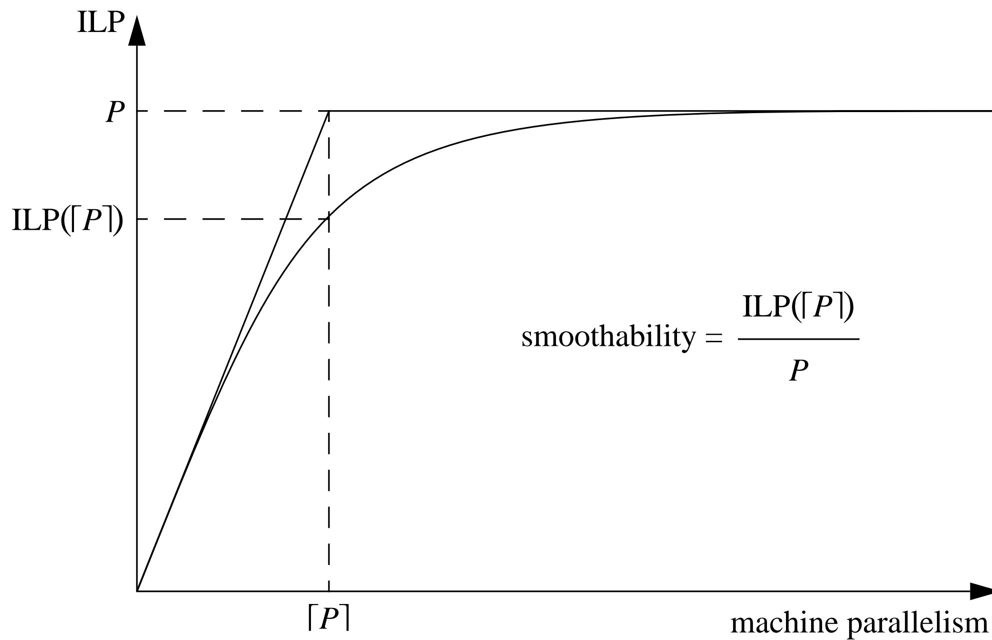


Figure 1.5: Theobald et al.'s smoothability function.

Jouppi's second-order piecewise linear model and the knee of the actual performance curve (see Figure 1.5). This distance reflects the inefficiency in the interaction of program parallelism and machine parallelism due to dynamic mismatching of their nonuniformities.

Because the parallelism present in most real programs is not perfectly smooth, single average values are not the best way to characterize it. This motivated the use of parallelism distributions (see Chapter 2).

1.3.3 Instruction Independence Probabilities

Dubey, Adams, and Flynn propose an analytical performance model [DAF94]. They characterize the parallelism present in a program with two parameters:

$$p_\delta = \text{Prob}\left[\text{instructions } i \text{ and } i+\delta \text{ are mutually independent}\right]$$

$$p_\omega = \text{Prob}\left[\text{two instructions scheduled in the same cycle are separated by } \omega \text{ branches}\right]$$

These parameters are extracted from program traces. Note that the values are averaged across an entire trace. The performance model then uses these parameters to estimate the ILP, i.e., the average number of instructions issued per cycle.

The important point here is that these two parameters characterize a trace, independent of the processor. The idea of characterizing data dependences with the p_δ parameter is extended to the dependent instruction distance concept (see Section 3.2).

1.3.4 Memory Stack Distance

Jacob et al. present a model for memory hierarchy performance [JCSM96]. This model uses the concept of **stack distance**. The ith reference (load or store instruction) to memory address A is denoted A_i. Each word of memory is denoted by a different letter. Each reference, A_i, has a stack distance:

$$\text{stack distance of } A_i = \text{the number of distinct memory blocks referenced between } A_{i-1} \text{ and } A_i$$

For example, consider a series of data memory references: $A_0\ B_0\ C_0\ B_1\ B_2\ A_1$. There is one block ($C$) referenced between B_0 and B_1, so the stack distance of B_1 is 1. B_2 immediately

follows B_1, so the stack distance of B_2 is 0. The stack distance of A_1 is 2 because blocks B and C are referenced between A_0 and A_1. Only the number of distinct blocks referenced between A_0 and A_1 is important; the number of times these other blocks are referenced does not affect the stack distance. For a fully associative cache with LRU replacement, the stack distance of A_i is the largest cache size for which reference A_i would be a miss. The cache model presented in Chapter 7 uses stack distance data collected from traces.

Chapter 2

Parallelism Distributions

Instruction-level parallelism is a measure of instruction throughput, i.e., the average rate at which instructions are processed by the machine. The common convention of measuring the instructions per cycle (IPC) at the issue stage is used. In this dissertation, instruction throughput is modeled, but instead of a single average value, a probability distribution is used. This chapter explains how distributions are used and why they are better than averages. In addition, a simple model which makes use of probability distributions to model performance is presented.

2.1 Definition

Computer architects are often interested in the parallelism achieved by a processor implementation on a particular benchmark. In general, they measure (or model) the number of instructions issued each cycle and compute an average IPC value over the execution of the entire program. In this work, the **parallelism distribution** is used as a measure of parallelism. A parallelism distribution is a vector (denoted by boldface symbols). The ith

element of a distribution is the probability that i instructions are issued in any given cycle:

$$\begin{aligned}
\mathbf{IPC} &= \begin{bmatrix} \mathbf{IPC}_0 & \mathbf{IPC}_1 & \dots & \mathbf{IPC}_n \end{bmatrix} \\
\mathbf{IPC}_i &= \text{Prob}\left[i \text{ instructions are issued}\right] \\
&= \text{the fraction of cycles in which } i \text{ instructions are issued}
\end{aligned}$$

The ith element, $\mathbf{IPC}_i$ can be defined as either a probability or a fraction of cycles. The two definitions are equivalent. The probability definition is useful when constructing a statistical model. The time definition has a more intuitive feel when examining performance modeling results.

A probability distribution must sum to one, i.e.,

$$\sum_{i=0}^{n} \mathbf{IPC}_i = 1$$

This just says that every cycle must be accounted for by exactly one element of the distribution.

The concept of a probability distribution can be applied elsewhere as well. For example, instead of measuring the average number of instructions in a buffer, one can look at the occupancy distribution:

$$\begin{aligned}
\mathbf{b} &= \begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \dots & \mathbf{b}_n \end{bmatrix} \\
\mathbf{b}_i &= \text{Prob}\left[\text{the buffer is holding } i \text{ instructions}\right] \\
&= \text{the fraction of cycles in which the buffer holds } i \text{ instructions}
\end{aligned}$$

The value $\mathbf{b}_i$ is the probability that there are i instructions in the buffer in any given cycle, or equivalently, the fraction of cycles in which there are i instructions in the buffer.

The common element in these two cases is a time sequence of values. In the case of issue rate, we have the number of instructions issued in cycle 0, cycle 1, cycle 2, etc. Similarly, for the buffer, we have the number of instructions in the buffer in each cycle. A probability distribution can be used in any situation in which there is a sequence of discrete values over time.

2.2 Transforms

three different forms for information such as issue rate or buffer occupancy have been described:

- a sequence of values over time (usually one value per cycle)
- a probability distribution
- an average value

Each of these forms can easily be converted to the next. Consider a time sequence consisting of m values:

$$x_0,\ x_1,\ \ldots,\ x_{m-1}$$

where x_t is the tth value, e.g., the number of instructions issued in cycle t. It is assumed that the values are bounded: $\forall t,\ 0 \le x_t \le n$. This sequence can be converted to a probability distribution:

$$\mathbf{d}_i = \frac{1}{m} \cdot (\text{the number of cycles for which } x_t = i) \quad 0 \le i \le n$$

This distribution clearly sums to one.

A distribution can be converted to a single average value:

$$d_{\text{avg}} = \sum_{i=0}^{n} i \cdot \mathbf{d}_i$$

This is the same distribution as would have been obtained by simply averaging the time sequence:

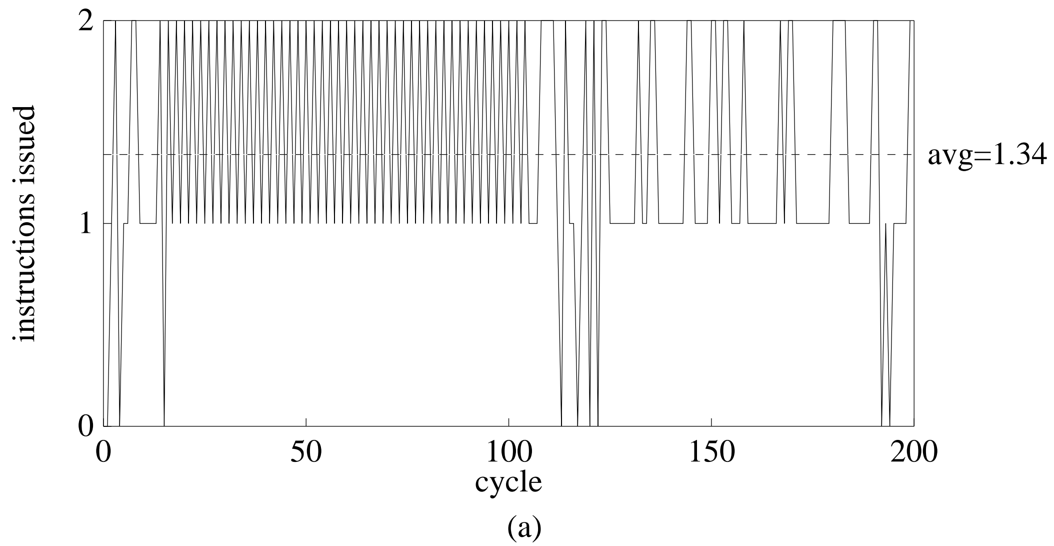
$$d_{\text{avg}} = \frac{1}{m} \cdot \sum_{t=0}^{m-1} x_t$$

As an example, consider a sequence of 200 values (the issue rate of a 2-issue processor over the first 200 cycles of the SPECint95 compress benchmark). Figure 2.1(a) shows the time sequence: the number of instructions (0, 1, or 2) issued each cycle. Figure 2.1(b) shows the corresponding probability distribution. The average value is shown as a horizontal dashed line in (a) and a vertical dashed line in (b). The important thing to note here is the quantity of information involved in each form. The time sequence has 200 points; the distribution has three values; and the average is a single value.

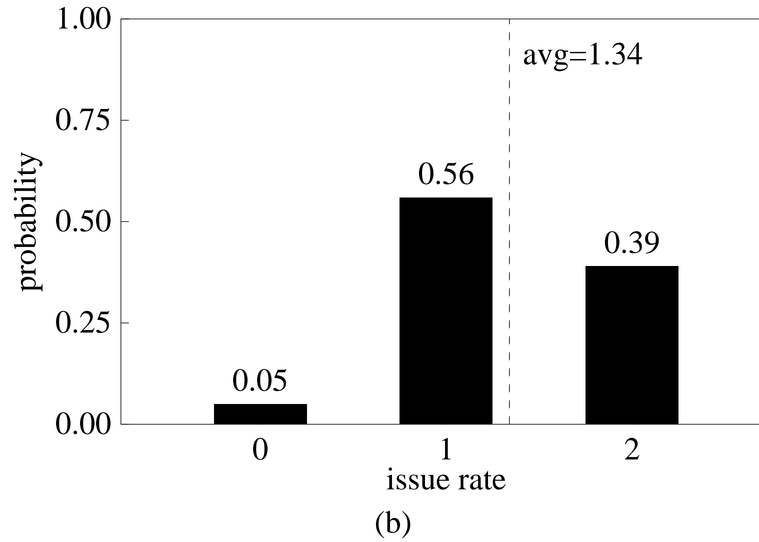
2.3 Why Distributions?

A distribution and an average are both "condensed" versions, or summaries, of a time sequence. Distributions are more useful than averages because they supply more information.

In the case of issue rate, the distribution **IPC** provides an idea of the smoothability of the parallelism (see Section 1.3.2). If the parallelism is smooth, i.e., if the processor usually issues close to IPC_{avg} instructions, the **IPC** distribution will have a peak at the average and much lower probabilities for the other issue rates. If, on the other hand, the parallelism is not smooth, this will be indicated by an **IPC** distribution that is more spread out, with relatively high values away from IPC_{avg}.



(a)



(b)

Figure 2.1: Issue rate for 200 cycles shown as (a) a time sequence, and (b) a probability distribution. The average value is marked by a dashed line in both graphs.

Buffer occupancy distributions can be used in a similar way. For example, the fraction of time the buffer is empty is just $\mathbf{b}_0$. The fraction of time it is completely full is $\mathbf{b}_n$. The fraction of time it is "almost full" — has zero or one empty slot — is $\mathbf{b}_{n-1} + \mathbf{b}_n$, and so on.

2.4 Matrix Multiplication Model

This section presents a simple statistical performance model which uses parallelism distributions and builds on the idea of treating instruction-level parallelism as the interaction of program parallelism and machine parallelism. The concept of parallelism distribution functions is introduced. These take the form of matrices and the modeling process involves multiplying them, hence the name "matrix multiplication model".

This model serves as an example of the use of parallelism distributions. It also demonstrates how program parallelism and machine parallelism can be separated, as shown in Figure 1.3. It does not, however, use the state vector concept described earlier. State vectors are introduced with the more advanced model presented in the next chapter.

The model is an abstract representation of the instruction flow through a processor. It has five stages (see Figure 2.2). Program parallelism and machine parallelism stages are interleaved. The parallelism at each stage represents the potential instruction flow through that stage, ignoring the effects of subsequent stages. The potential flow through a particular stage is a function of the potential flow through the previous stage. Each stage thus represents a reduction in the potential instruction flow. The flow through the last stage is a measure of the overall performance. The next several paragraphs define each of the five model stages.

Program parallelism is the degree to which instructions can be executed in parallel, given only the constraints in the object code [Jou89]. The program parallelism stages in the model represent the effects of program dependences on instruction flow. This includes

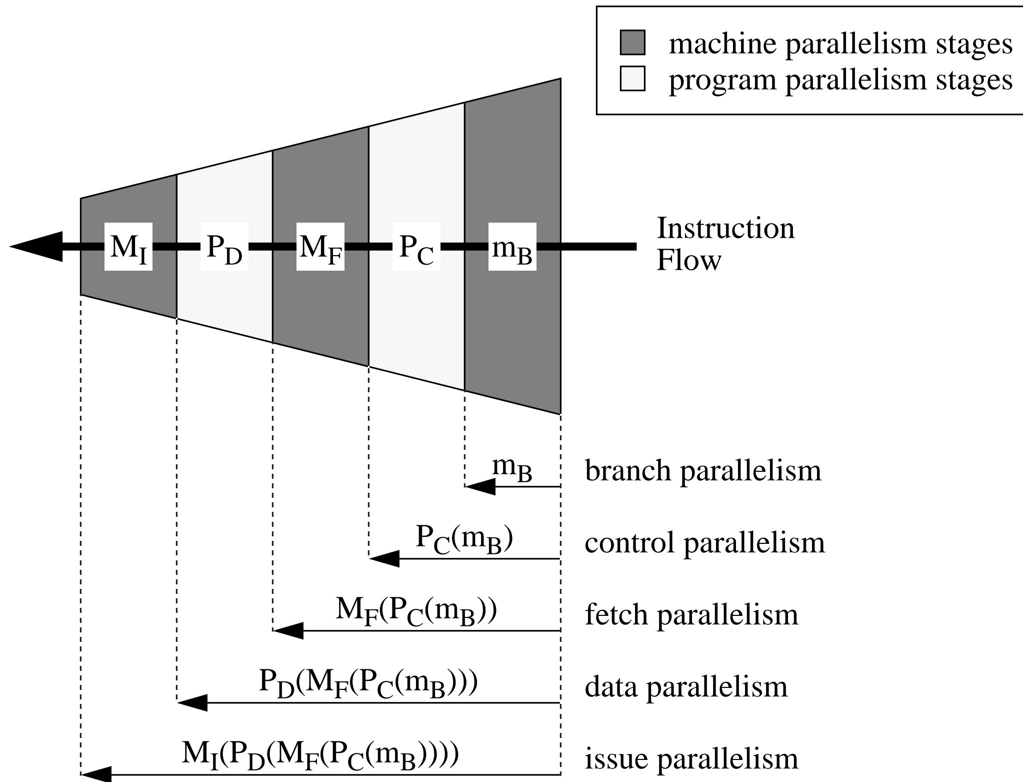


Figure 2.2: The five stages of the matrix multiplication model. The boxes represent the restrictions imposed by each stage. Instructions flow to the left; each stage decreases the available parallelism or potential throughput.

effects due to the source language, compiler, ISA, and the particular data set used to generate the trace, as well as properties inherent to the program. Program dependences take two forms: control dependences and data dependences [HP90]. Consequently, program parallelism is broken into two corresponding stages:

- **Control parallelism** is a measure of the number of useful instructions which are visible to the processor's instruction-fetch mechanism. Control parallelism is limited by the control dependences caused by branch instructions. For this reason, it is a function of the branch parallelism. In a processor which does not support speculative execution, instructions immediately following a branch instruction cannot be executed until the outcome of the branch is known. All instructions after a branch must wait for it to be resolved. A processor can, however, extract control parallelism from beyond branches using branch prediction [RF93]. With branch prediction, potentially useful instructions can be fetched from beyond branches. These instructions are considered useful only if the branch was *correctly* predicted.

- **Data parallelism** is a measure of the number of instructions in a processor which are ready to be executed, as a function of the number which are available to be examined. Data parallelism is limited by the data dependences caused by register and memory references. Instructions which use a value (register or memory word) must wait until the instruction which generates that value has executed. An instruction can be executed only if it is independent of all other instructions which are still executing.

In addition to control and data parallelism, two other program parameters need to be considered. Both of these are used as auxiliary inputs to the machine parallelism functions. The dynamic instruction mix is needed because processors handle various instruction types differently, e.g., result latencies may be different for integer and floating point instructions. The dynamic branch characteristics of the code, including the branch type mix and

a characterization of the behavior of conditional branches, are used as inputs to the branch parallelism function to determine the predictability of branches.

Machine parallelism is the degree to which instructions can be executed simultaneously by the processor, given only its inherent implementation constraints [Jou89]. Machine parallelism is the overall capacity of the machine for processing instructions. The machine parallelism stages of the model represent the effects of limited machine resources on instruction flow.

For a scalar pipeline with all the pipeline stages operating in lockstep and no additional buffering other than the single buffers between adjacent stages, a single function can characterize this capacity. For superscalar processors, there are constraints that limit the capacity at various intermediate points of instruction processing. These constraints are known as structural dependences [HP90]. In the model, structural dependences are divided into three categories: branch constraints, fetch/decode constraints, and issue constraints, corresponding to the three main stages of instruction processing prior to execution. Superscalar processors often have buffers between these stages, e.g., issue buffers and reservation stations. This separates the stages, allowing some decoupling between them. The model accounts for this by separating machine parallelism into three stages:

- **Branch parallelism** is a measure of the number of basic blocks which are "visible" to the fetching mechanism. This is determined by the processor's branch prediction and speculative execution capabilities. Branch parallelism is a function of the branching characteristics of the program.

- **Fetch parallelism** is a measure of the number of instructions fetched and decoded by the processor per cycle. This is determined by the maximum fetch bandwidth and/or the size of the issue buffer, depending on the specific processor implementation. Fetch parallelism is a function of the control parallelism, i.e., the number of

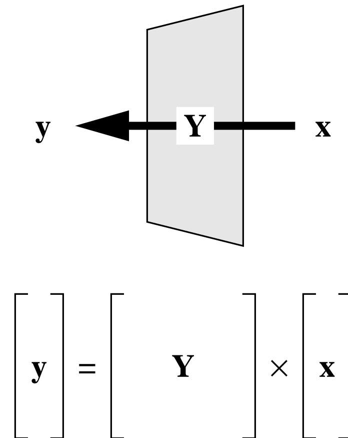


Figure 2.3: A parallelism distribution function in graphical form (top) and matrix-vector multiplication form (bottom). The **x** vector (lowercase) is the input and the **y** vector (lowercase) is the output. The **Y** matrix (uppercase) is the function. Each stage of the model is a parallelism distribution function. potential throughput.

instructions available to be fetched.

- **Issue parallelism** is a measure of the number of instructions issued to functional units (or reservation stations) per cycle. This is determined by the number and types of functional units, their issue and result latencies, and the number of reservation stations associated with each. Issue parallelism is a function of the data parallelism, i.e., the number of independent instructions available to be issued.

2.4.1 Specifics

This section applies the matrix multiplication model to a simple processor. The modeled processor issues up to two instructions per cycle to three functional units: integer, floating point, and memory. Each functional unit has a one-cycle latency. Branch prediction is backward taken, forward not taken. The DEC Alpha instruction set [Dig92] is used.

Each stage of the model is a function whose input and output are parallelism probabil-

ity distributions (Figure 2.3, top). The first stage of the model, branch parallelism, is not a function but simply a distribution. A **parallelism distribution function** is represented by a matrix. An application of such a function to an input distribution is a matrix-vector multiplication (Figure 2.3, bottom). Matrices (functions) are indicated by uppercase boldface letters; vectors (distributions) are indicated by lowercase boldface letters. The element $\mathbf{Y}_{ij}$ of the function matrix is the probability that the output parallelism is i, given that the input parallelism is j. Column j of $\mathbf{Y}$, denoted $\mathbf{Y}(j)$, is the parallelism distribution that would result if the input parallelism were always j. The output parallelism distribution $\mathbf{y}$ is obtained by weighting each column of $\mathbf{Y}$ by the associated value in the input distribution $\mathbf{x}$ and summing the resulting vectors. This is just a matrix-vector multiplication:

$$\begin{aligned}\mathbf{y} &= \mathbf{x}_0\mathbf{Y}(0) + \mathbf{x}_1\mathbf{Y}(1) + \cdots + \mathbf{x}_n\mathbf{Y}(n) \\ &= \mathbf{Y} \times \mathbf{x}\end{aligned}$$

To illustrate this matrix-vector multiplication, consider a single stage of the model. Let the input parallelism, i.e., the instruction flow through the previous model stage, be:

$$\mathbf{x} = \begin{bmatrix} 0.20 \\ 0.50 \\ 0.30 \end{bmatrix}$$

This means that in 20% of the cycles, zero instructions flow; in 50% of the cycles, one instruction flows; and in 30% of the cycles, two instructions flow. Consider the following

parallelism distribution function:

$$\mathbf{Y} = \begin{bmatrix} 1.00 & 0.10 & 0.10 \\ 0.00 & 0.90 & 0.30 \\ 0.00 & 0.00 & 0.60 \end{bmatrix}$$

If the input parallelism is two (the last column) then the output parallelism has a 10% probability of being zero, a 30% probability of being one, and a 60% probability of being two, and similarly for the other columns. The overall output parallelism distribution is the product of this matrix and the input parallelism vector:

$$\begin{aligned}
\mathbf{y} = \mathbf{Y} \times \mathbf{x} &= \begin{bmatrix} 1.00 & 0.10 & 0.10 \\ 0.00 & 0.90 & 0.30 \\ 0.00 & 0.00 & 0.60 \end{bmatrix} \times \begin{bmatrix} 0.20 \\ 0.50 \\ 0.30 \end{bmatrix} \\
&= 0.20 \times \begin{bmatrix} 1.00 \\ 0.00 \\ 0.00 \end{bmatrix} + 0.50 \times \begin{bmatrix} 0.10 \\ 0.90 \\ 0.00 \end{bmatrix} + 0.30 \times \begin{bmatrix} 0.10 \\ 0.30 \\ 0.60 \end{bmatrix} \\
&= \begin{bmatrix} 0.28 \\ 0.54 \\ 0.18 \end{bmatrix}
\end{aligned}$$

This is the instruction flow through the model stage: there is a 28% probability of zero instructions flowing, a 54% probability of one instruction flowing, and an 18% change of two instructions flowing.

Each of the function applications in Figure 2.2 is a matrix-vector multiplication. The remainder of this section explains how the $\mathbf{m}_B$ vector and the $\mathbf{P}_C$, $\mathbf{M}_F$, $\mathbf{P}_D$, and $\mathbf{M}_I$ matrices are computed.

The branch parallelism, $\mathbf{m}_B$, of a processor is the distribution of the number of branches past which the processor can fetch. $\mathbf{m}_B$ is the first stage of the model, so it is a distribution (vector), not a distribution function (matrix). $\mathbf{m}_B$ is actually a function of the program branch characteristics described above, but this is not explicitly represented. If the processor does not perform branch prediction, it can always see up to the next branch and no further, and the branch parallelism is always one:

$$\mathbf{m}_B = [\ 0 \quad 1 \quad 0 \quad \ldots \quad 0\ ]^T$$

If, for example, the processor can speculate past one branch 60% of the time, the branch parallelism would be:

$$\mathbf{m}_B = [\ 0 \quad 0.4 \quad 0.6 \quad 0 \quad \ldots \quad 0\ ]^T$$

It is possible to have a non-zero first element: if the processor mispredicts and is fetching useless instructions, the branch parallelism will be zero. This accounts for the misprediction penalty.

The control parallelism of a program describes the number of visible useful instructions as a function of the number of branches visible to the processor, i.e., the branch parallelism. That is, $(\mathbf{P}_C)_{ij}$ is the probability that i useful instructions are visible, given that the processor can see up to the jth next branch. The control parallelism distribution is the result of applying the control parallelism function to the branch parallelism distribution:

$$\mathbf{p}_C = \mathbf{P}_C \times \mathbf{m}_B$$

The $\mathbf{P}_C$ matrix is automatically extracted from the trace. $(\mathbf{P}_C)_{ij}$ is the probability that there are i instructions visible up to and including the jth next branch. This can be measured

by scanning the trace, considering each instruction as a starting point, and counting the number of instructions visible up to each of the next n branches.

This process assumes that each instruction in the trace is equally likely to be a starting point for fetching during any given cycle. However, the processor spends less time in areas with more parallelism, and hence, instructions in these areas are less likely to be fetch starting points. To account for this, the ith element of each distribution (column) in $\mathbf{P}_C$ is divided by i, and the distribution is rescaled so that it sums to one:

$$(\mathbf{P}_C)_{ij} = \frac{\frac{1}{i} \cdot (\mathbf{P}_C^{\text{orig}})_{ij}}{\sum_{k=0}^{n} \frac{1}{k} \cdot (\mathbf{P}_C^{\text{orig}})_{kj}}$$

The fetch parallelism function describes the number of instructions fetched and decoded by the processor per cycle as a function of the number of useful instructions which are visible to the fetch unit, i.e., the control parallelism. $(\mathbf{M}_F)_{ij}$ is the probability that i instructions can be fetched if j are visible. The fetch parallelism distribution is the result of applying the fetch parallelism function to the control parallelism distribution:

$$\mathbf{m}_F = \mathbf{M}_F \times \mathbf{p}_C$$

For the processor modeled here, fetch parallelism is just a simple bandwidth limit, representing the finite fetching capabilities of the processor. Since it can fetch up to two

instructions per cycle, the fetch parallelism matrix looks like this:

$$\mathbf{M}_F = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ & & \vdots & & & \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

If zero instructions are visible, zero can be fetched; if one instruction is visible, one can be fetched; if two or more are visible, two can be fetched.

The data parallelism function describes the number of available data-ready instructions as a function of the number of instructions which are available to be examined and issued, i.e., the fetch parallelism. $(\mathbf{P}_D)_{ij}$ is the probability that i instructions are data-ready, given that j instructions have been fetched. The data parallelism distribution is the result of applying the data parallelism function to the fetch parallelism distribution:

$$\mathbf{p}_D = \mathbf{P}_D \times \mathbf{m}_F$$

Finally, the issue parallelism function describes the number of instructions issued to functional units per cycle as a function of the number and types of instructions in the issue buffer, i.e., the data parallelism and instruction mix. The issue parallelism function is determined by the number and types of functional units and their issue latency.

The $\mathbf{M}_I$ matrix is computed based on the instruction mix of the program. The instruction mix information extracted from the trace specifies the probability of every possible mix of instructions. More specifically, r_{uvw} is the probability of getting u integer instructions, v floating point instructions, and w memory instructions in an arbitrary sequence of

$u+v+w$ instructions. The processor can issue at most one instruction of each type, so the total number issued is $\min\{1,u\}+\min\{1,v\}+\min\{1,w\}$. Then:

$$(\mathbf{M}_I)_{ij} = \sum_{\substack{(u,v,w)\in \\ \text{issue}(i,j)}} r_{uvw}$$

where:

$$\text{issue}(i,j) = \{(u,v,w) \mid u+v+w=j \text{ and } \min\{1,u\}+\min\{1,v\}+\min\{1,w\}=i\}$$

Issue parallelism is the final stage of the model — it provides the desired IPC distribution:

$$\begin{aligned}\mathbf{IPC} = \mathbf{m}_I &= \mathbf{M}_I \times \mathbf{p}_D \\ &= \mathbf{M}_I \times \mathbf{P}_D \times \mathbf{M}_F \times \mathbf{P}_C \times \mathbf{m}_B\end{aligned}$$

2.4.2 Benchmarks

Throughout this dissertation, the SPEC95 benchmarks [SPE95] are used. This suite consists of eight integer benchmarks (SPECint95) and ten floating point benchmarks (SPECfp95). Reduced input sets are used in most cases. Table 2.1 lists the SPEC95 benchmarks and their instructions counts with the reduced input sets. All benchmarks are compiled for the DEC Alpha, using the DEC C and FORTRAN compilers with '-O4' optimization.

2.4.3 Experimental Results

Figure 2.4 compares the average IPC computed by the matrix multiplication model to that measured by a timing simulator for the SPEC95 benchmarks.

benchmark	instructions (millions)
compress	55
gcc	148
go	78
ijpeg	88
li	56
m88ksim	99
perl	48
vortex	76

SPECint95

benchmark	instructions (millions)
applu	30
apsi	81
fpppp	35
hydro2d	630
mgrid	95
su2cor	47
swim	28
tomcatv	88
turb3d	2,598
wave5	78

SPECfp95

Table 2.1: The SPEC95 benchmarks and their instruction counts.

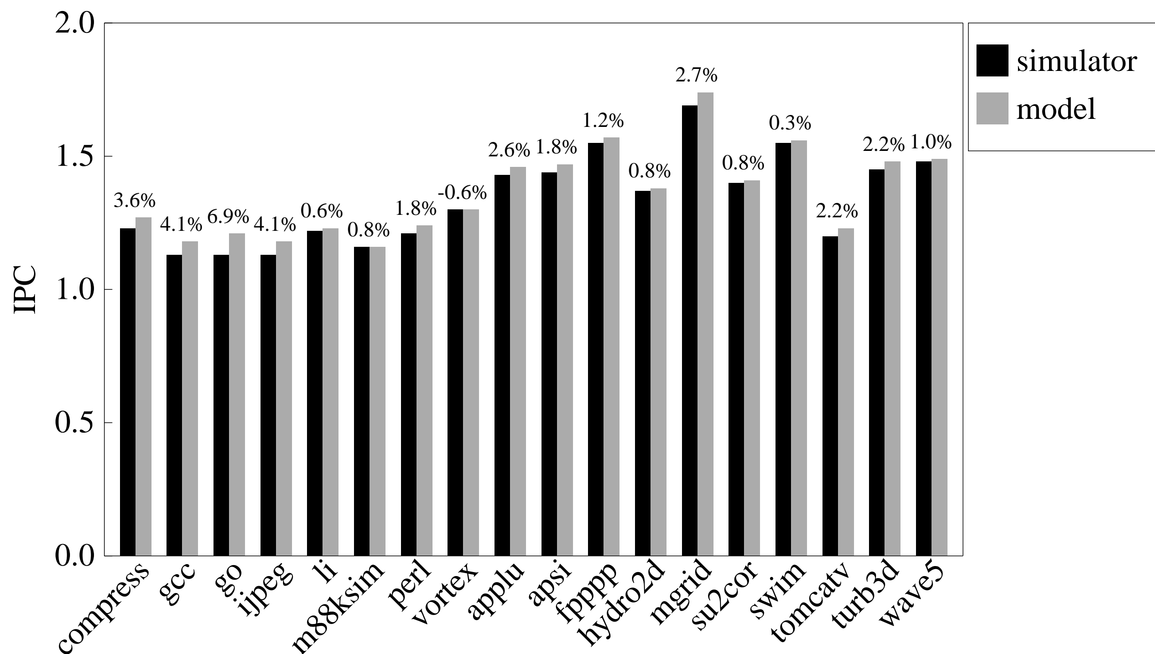


Figure 2.4: The average IPCs for the SPEC95 benchmarks, as measured by a simulator (black bars) and computed by the matrix multiplication model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

The maximum error across all eighteen benchmarks is 6.9%. This fairly simple model is very accurate because the modeled processor is also quite simple. The small error that is present is due to assumptions made in constructing the model. For example, it is implicitly assume that control parallelism is independent of data parallelism. In general, this is not a valid assumption. For example, it may be the case for some benchmarks, that sections of code with a high control parallelism (branches relatively far apart) have a low data parallelism (long data dependence chains).

The matrix multiplication model has the advantages of simplicity and elegance. Each of the five model stages corresponds intuitively to a part of the processor or a property of the program. The elements of the five matrices are straightforward probabilities.

The disadvantage is that this model is *too* simple. It suffers from the dependent probability problem mentioned above. It does not model buffers between stages, which can make up for some of the lack of cycle-to-cycle smoothness in parallelism. There is also no reasonable way to model functional units with multi-cycle dependences.

The next chapter uses more complex statistical techniques to produce a model which overcomes some of these problems.

Chapter 3

Using Markov Models

Markov models are a statistical construct used to analyze systems which can be modeled by a progression of states. This chapter explains how Markov models are applied to processor performance evaluation. Statistical models are presented for a simple one-pipeline processor and for a three-pipeline processor similar to the one used in the previous chapter.

3.1 Markov Models

This section gives a brief overview of basic notions concerning Markov chains. This is followed by a description of how a Markov chain is used to model a processor.

A **Markov chain** is a stochastic process whose next state probability distribution depends only on the current state [Ros83, Ste94]. That is, it is a sequence of random variables:

$$X_0, X_1, \ldots, X_t, X_{t+1}, \ldots$$

The variable X_t is called the **state** of the system at time t. The set of all possible states, i.e., all possible values of X_t (for any t), is called the **state space**.

The Markov property states that the probability of the next state, X_{t+1}, is dependent only on the current state, X_t. More formally:

$$\text{Prob}\left[X_{t+1} = x_{t+1} \,|\, X_0 = x_0, X_1 = x_1, \ldots, X_t = x_t\right] = \text{Prob}\left[X_{t+1} = x_{t+1} \,|\, X_t = x_t\right]$$

These probabilities are called the **transition probabilities**:

$$\mathbf{P}_{ij} = \text{Prob}\left[X_{t+1} = j \,|\, X_t = i\right]$$

For a state space of size n, the transition probabilities form an $n \times n$ matrix, $\mathbf{P}$.

It is useful to consider the probability of the Markov chain being in a particular state i at a certain time t:

$$\mathbf{d}_i(t) = \text{Prob}\left[X_t = i\right]$$

These values form a state probability distribution $\mathbf{d}(t)$. Given the state probability distribution at time t and the transition matrix $\mathbf{P}$, the state distribution for time $t+1$ can be computed:

$$\mathbf{d}(t+1) = \mathbf{P}^T \mathbf{d}(t)$$

A Markov chain (assuming it is irreducible, aperiodic, and positive recurrent [Ros83]) has a **stationary distribution**. The stationary distribution is the state distribution $\mathbf{d}$ for which:

$$\mathbf{P}^T \mathbf{d} = \mathbf{d}$$

In steady state behavior, this distribution gives the probability of finding the Markov chain in a particular state at any given time. This is equivalent to the fraction of time spent in that state, as with the probability distributions described in Section 2.1.

There are three steps involved in using a Markov chain to model a superscalar processor. It has already been noted that a statistical model uses a notion of state similar to that used by a processor simulator. Time, as used by the Markov chain, is measured in cycles. In each cycle, the processor is in some state and is about to transition to another state. The first step in constructing a Markov chain is designing its state space. The state space is a function of the microarchitecture and is independent of the programs which are to be run on the processor. Each "Processor Model" box in Figure 1.3 is a state space for the corresponding processor. A simulator can afford to be generous with its state space — it only has to store the state vector for the current cycle and perhaps another for the next cycle. If there are n states, the required memory is on the order of $\log_2 n$ bits. The Markov chain transition matrix, however, holds 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 state space design is a critical issue.

The second step is computing the state transition matrix. This requires information from the program trace (the "Program Params" boxes in Figure 1.3) and is obviously also dependent on the state space.

Once the transition matrix is available, the stationary distribution is computed. The result is a distribution specifying the fraction of time spent in each state (see Figure 1.1).

As with the matrix-based model (Section 2.4), the program trace is analyzed only once. Enough information is extracted from each trace to be able to construct a transition matrix for any Markov chain, i.e., for any processor model. The following sections describe exactly what data is extracted from the trace and how it is used to build the transition matrix.

3.2 Instruction Characteristics

This section presents concepts and definitions which are necessary to understand the processor models which follow.

An instruction in a program is called a **static instruction**. A static instruction is uniquely identified by its address. A **dynamic instruction** is an instance of a static instruction within the execution of the program. A single static instruction may be executed many times, e.g., in a loop, resulting in multiple dynamic instructions. An instruction trace is a list of dynamic instructions.

Dynamic instructions are characterized by their type and data dependences. The **instruction type**, y, places an instruction into a category. In the models presented here, instructions are divided into four types:

- integer
- floating point
- memory
- branch

The instruction type for all dynamic instances of a static instruction will obviously be the same.

Data dependences, on the other hand, may be different for each dynamic instance of an instruction. A dynamic instruction's data dependence is characterized by the **dependent instruction distance**, s. This is a measure of the distance from the instruction to its most recent source instruction. A source instruction is one which produces a data value needed by this instruction, and the distance is the number of intervening instructions. A dynamic instruction dependent on the immediately preceding instruction would thus have a dependent instruction distance of zero.

The following example should clarify the definition of dependent instruction distance. Throughout this dissertation, dynamic instructions taken from a trace are given labels of the form "Txxx". These labels have no significance other than their use in referring to specific instructions in the text, and should not be confused with instruction addresses or with the instruction numbers used in later chapters.

Consider this sequence of dynamic instructions.

```
T100:  ld   0(r1)   --> r2    ; y = mem
T101:  ld   8(r1)   --> r3    ; y = mem
T102:  add  r2, r3  --> r4    ; y = int,  s = 0
T103:  sub  r2, #2  --> r5    ; y = int,  s = 2
T104:  mul  r2, r5  --> r6    ; y = int,  s = 0
T105:  br   L1                ; y = br,   s = ∞
```

Instruction T102's closest dependence is on the immediately preceding instruction, number T101, so its dependent instruction distance is 0. Instruction T103's only dependence is on T100, so the distance is 2. Instruction T104 is similar to T102. Instruction T105 is an unconditional branch and has no data dependences, so its dependent instruction distance is ∞. The distances for instructions T100 and T101 cannot be determined without seeing previous instructions in the trace.

The concept of dependent instruction distance can be extended to dependences on instructions of multiple types. Instead of looking at just the closest dependence, the distances from an instruction to the most recent source instruction of each type are measured. That is, for any type t, s_t is the number of intervening type-t instructions between the instruction under consideration and its most recent type-t source. In the processor models used here, branches are executed by the integer unit, so they are grouped with integer instructions for purposes of data dependences. Consider the same code sequence again:

```
T100:  ld   0(r1)   --> r2   ; y = mem
```

```
T101:  ld   8(r1)   --> r3   ; y = mem
T102:  add  r2, r3  --> r4   ; y = int, s_int = ∞, s_fp = ∞, s_mem = 0
T103:  sub  r2, #2  --> r5   ; y = int, s_int = ∞, s_fp = ∞, s_mem = 1
T104:  mul  r2, r5  --> r6   ; y = int, s_int = 0, s_fp = ∞, s_mem = 1
T105:  br   L1               ; y = br, s_int = ∞, s_fp = ∞, s_mem = ∞
```

Instruction T102 depends on the immediately previous memory instruction, so $s_{\text{mem}} = 0$. Since it does not depend on any integer or floating point instructions, it has $s_{\text{int}} = s_{\text{fp}} = \infty$. Instruction T103 depends only on instruction T100, which is a memory instruction. Type-specific dependent instruction distances are measured by counting intervening instructions *of that type*, so $s_{\text{mem}} = 1$. Instruction T104 depends on both the previous integer instruction ($s_{\text{int}} = 0$) and the second previous memory instruction ($s_{\text{mem}} = 1$). Finally, instruction T105 has no data dependences, so $s_{\text{int}} = s_{\text{fp}} = s_{\text{mem}} = \infty$.

For practical reasons, dependent instruction distances are limited to an arbitrary maximum, s_{max}. Instructions with distances greater than s_{max}, including instructions with no dependences, are assigned $s = s_{\text{max}}$:

$$s = \begin{cases} s_{\text{actual}} & \text{if } s_{\text{actual}} < s_{\text{max}} \\ s_{\text{max}} & \text{if } s_{\text{actual}} \geq s_{\text{max}} \end{cases}$$

Instructions with dependent instruction distances greater than the depth of the longest pipeline cannot possibly be data dependent on any instruction still in flight, and s_{max} is chosen based on this.

Subsequent sections occasionally refer to a "y/s instruction". This is shorthand for "an instruction of type y with dependent instruction distance s".

In characterizing the instructions this way, the fundamental assumption is that all instructions with the same y and s parameters, when executed in the same context, will behave identically. This is analogous to electrons behaving identically in a circuit. (Of course,

there is only one kind of electron, in contrast to the many different types of instructions.)

The instruction characterization described here is not entirely independent of the processor — the ideal goal of a complete separation between program analysis and processor analysis is not quite achieved. Specifically, the set of instruction types and the value of s_{max} are both chosen based on the range of microarchitectures to be modeled. However, this should not be a major limitation for most uses of the statistical model.

3.3 Processor Models

A microarchitecture is broken down into **components** of various types, such as issue buffers and pipelines. See Figure 3.1. Instructions flow from one component to the next along **connections**. The components and connections form a block diagram of the processor.

The transition from one state (i.e., cycle) to the next consists of instructions flowing between components. The number of instructions that flow across a particular connection is determined by three things: the push, pull, and bandwidth. Consider connection-1 in Figure 3.1. Its source is the issue buffer, and its destination is pipe-1. Clearly, only instructions that are currently in the issue buffer can flow across connection-1. The number of instructions available in the source component in a particular cycle is called the **push out** of the source or the **push in** to the destination.

Second, some of those instructions may be blocked by the destination component due to control, data, or structural hazards. In the example, an instruction that has a data dependence on another not-yet-completed instruction will be blocked. Also, if pipe-1 is stalled and not accepting any instructions, then flow into it will be blocked (this is a structural hazard). The number of instructions which can be accepted by the destination component in this cycle is called the **pull in** to the destination or the **pull out** of the source. The pull out of a connection can be greater than the push into that connection. For example, if the

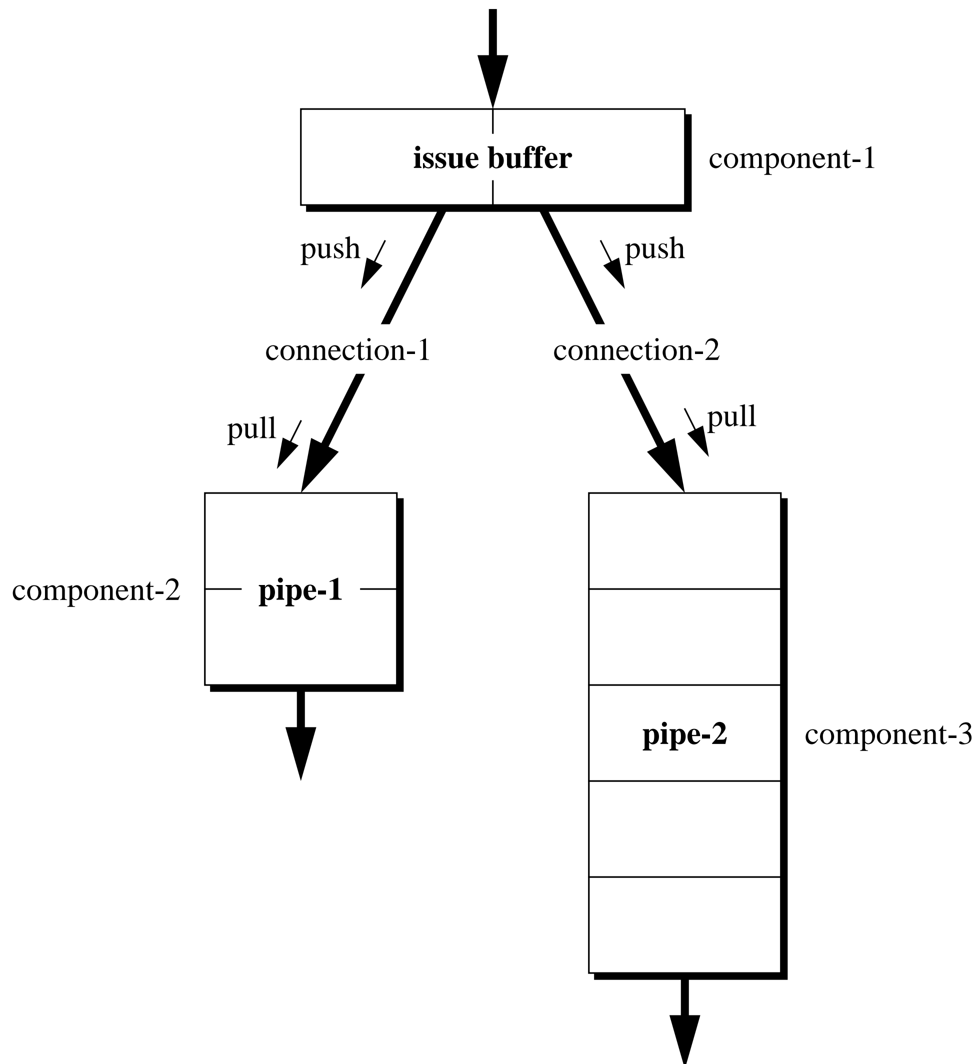


Figure 3.1: Microarchitecture components and connections.

issue buffer and pipes are all empty, the push into connection-1 would be zero, since there are no instructions available to be issued, and the pull out of connection-1 would be one, since pipe-1 is free of all dependences and so can accept a new instruction.

Finally, the flow through a connection is limited by the bandwidth of that connection. Consider two buffers, each capable of holding eight instructions, connected by a bus which can transfer up to two instructions per cycle. If the source buffer is full, the destination buffer is empty, and there are no other hazards, the push and pull are both eight, while the connection bandwidth is only two.

The flow, f, across a connection in a given cycle is equal to the minimum of these three quantities, push p, pull q, and bandwidth m:

$$f = \min\{p, q, m\}$$

This is just a formalized version of the computation done by a timing simulator to determine the state of the processor in the next cycle. Up to this point, the statistical approach is very similar to the simulation approach.

3.4 A One-Pipeline Processor

This section presents a Markov chain-based model of a single-pipeline processor. An example is used to illustrate the statistical modeling process, from trace analysis to IPC computation.

3.4.1 The Model

The one-pipeline processor is shown in Figure 3.2. It consists of three components: a fetch buffer, an issue buffer, and the execution pipeline. The fetch and issue buffers each hold one

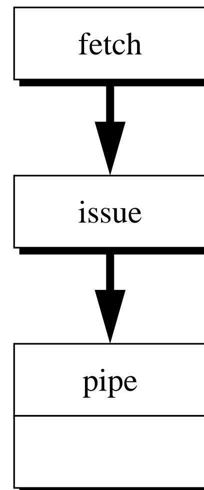


Figure 3.2: Components and connections of the one-pipe processor.

instruction. The fetch buffer has two variants: one which does perfect branch prediction (no stalls), and one which does no branch prediction (fetching stalls after a branch is fetched, until the branch is issued). The issue buffer does not cause any hazards. It does, however, introduce a one-cycle bubble after every branch when the no-prediction fetch buffer is used. The pipeline executes all instructions with a latency of two cycles and writeback is assumed not to generate stalls. 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 one cycle.

There are two instruction types: branches and non-branches. Since the single execution pipeline handles all instruction types, there are no differences in the way integer, floating point, and memory instructions are executed. Because branches are resolved at issue, instruction type information is not needed for instructions in the pipe. Once an instruction issues, it moves through the pipe and leaves the machine without stalling.

The state vector consists of the following elements:

- x_F = the number of instructions in the fetch buffer ($0 \leq x_F \leq 1$).
- x_I = the number of instructions in the issue buffer ($0 \leq x_I \leq 1$).
- x_P = the occupancy bit vector for the pipeline — one bit per stage: an occupied stage is indicated by a 1 bit and an empty stage is indicated by a 0 bit ($0 \leq x_P < 2^2$).
- y_i = the type of the ith next instruction to be issued: 0 means non-branch, 1 means branch ($y_i \in \{0,1\}$ for $0 \leq i \leq 1$).
- s_i = the dependent instruction distance of the ith next instruction to be issued ($0 \leq s_i \leq s_{\max}$ for $0 \leq i \leq 1$).

The pipe occupancy (x_P) element is a bit vector, with one bit per pipe stage. x_P values are written in binary format with angle brackets. For example, $x_P = \langle 10 \rangle$ means there is an instruction in the first stage and none in the second stage.

The type (y_i) and distance (s_i) values are for the next two instruction 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.

In this processor, an instruction with a dependent instruction distance of at least one, i.e., independent of the immediately preceding instruction, can always issue immediately (using a forwarded result if necessary). Thus $s_{\max} = 1$ could be used. However, the s values are also used for instruction sequencing (see below), which is made more accurate by using a higher value of $s_{\max}$. The value $s_{\max} = 5$ is chosen to match the three-pipe models presented later.

The size of the state space is computed by considering all possible states, i.e., all possible combinations of the state vector elements. x_F and x_I can each take on two values and x_P can take on $2^2 = 4$ values. y_0 and y_1 can each take on 2 values. s_0 and s_1 can each take on $s_{\max} + 1 = 6$ values. The state space consists of all possible combinations of these values. There are thus $2 \times 2 \times 4 \times 2^2 \times 6^2 = 2,304$ states.

The trace analyzer is responsible for extracting from each benchmark trace all information necessary to construct a transition matrix. For the one-pipe processor model, the only necessary data is the table of instruction sequencing probabilities:

$$\text{instrSeq}[y_0, s_0, y_1, s_1, y_2, s_2] = \text{Prob}\left[\text{next instruction is } y_2/s_2 \,|\, \text{current instructions are } y_0/s_0 \text{ and } y_1/s_1\right]$$

That is, for every consecutive pair of instructions in the trace of types y_0 and y_1 and dependent instruction distances s_0 and s_1, the trace analyzer looks at the next instruction and updates the appropriate counter. The instrSeq table entries are the probabilities:

$$\text{instrSeq}[y_0, s_0, y_1, s_1, y_2, s_2] = \frac{\text{number of occurrences of } (y_0/s_0),\ (y_1/s_1) \text{ followed by } (y_2/s_2)}{\text{number of occurrences of } (y_0/s_0),\ (y_1/s_1)}$$

3.4.2 Transition Matrix Computation

The example presented here starts with the trace from a small program and shows how the statistical model is used:

- the Markov chain transition matrix is constructed
- the stationary distribution is computed

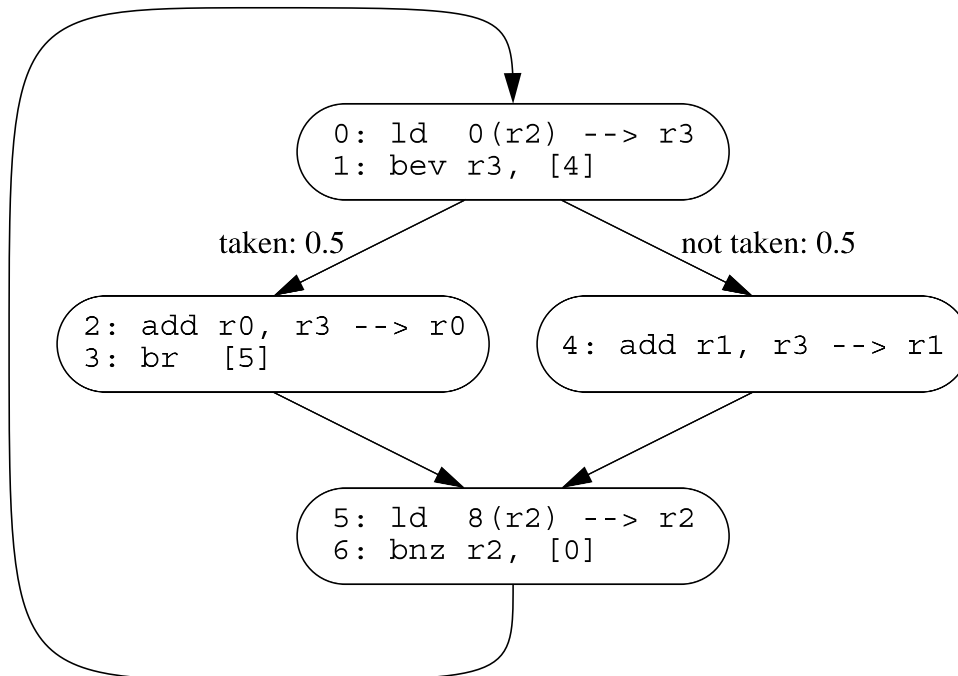


Figure 3.3: Control flow graph with assembly code for a simple loop. The arc weights indicate the taken/not-taken probabilities for the conditional branch. (`bev` is "branch if even" and `bnz` is "branch if not zero".)

- the IPC is computed from the stationary distribution

The results of statistical modeling are compared with a manual simulation of the program.

Figure 3.3 shows the control flow graph for a small piece of assembly code. This loop traverses a linked list, examining the integer at each node, and adds it to `r0` if even or `r1` if odd. The linked list is constructed to contain sequential integers so that the trace alternates between the two possible ("even" and "odd") paths of execution. This figure shows the static code. Instructions are identified by their static instruction addresses, as opposed to the "Txxx" trace labels used earlier.

A section of the resulting trace is shown in Figure 3.4, which lists the instruction numbers along with the corresponding instruction types and dependent instruction distances.

instruction:	0	1	4	5	6	0	1	2	3	5	6	...
type (y):	0	1	0	0	1	0	1	0	1	0	1	...
distance (s):	1	0	1	4	0	1	0	1	5	5	0	...

Figure 3.4: A section of the trace for the loop shown in Figure 3.3. The example analysis assumes that this trace fragment is repeated, and no other code executes.

y_0	s_0	y_1	s_1	y_2	s_2	probability
0	1	1	0	0	1	2/2 = 1.00
1	0	0	1	0	4	1/4 = 0.25
				1	0	2/4 = 0.50
				1	5	1/4 = 0.25
0	1	0	4	1	0	1/1 = 1.00
0	4	1	0	0	1	1/1 = 1.00
0	1	1	5	0	5	1/1 = 1.00
1	5	0	5	1	0	1/1 = 1.00
0	5	1	0	0	1	1/1 = 1.00

Table 3.1: Instruction sequencing probabilities for the trace shown in Figure 3.4.

This trace section includes an execution of the "even" path, followed by an execution of the "odd" path. For this example, it is assumed that the processor executes iterations of this loop and nothing else.

The instruction sequencing probabilities extracted by the trace analyzer are shown in Table 3.1. Consider the three entries for $y_0 = 1, s_0 = 0, y_1 = 0, s_1 = 1$. These correspond to instruction sequences:

- 1,4,5: $y_2 = 0,\ s_2 = 4$
- 6,0,1: $y_2 = 1,\ s_2 = 0$ — at the end of the even path
- 1,2,3: $y_2 = 1,\ s_2 = 5$

- 6,0,1: $y_2 = 1,\ s_2 = 0$ — at the end of the odd path (this "wraps around" the end of the trace)

From this, it can be seen that there are three possible combinations of y_2 and s_2, with probabilities 1/4, 2/4, and 1/4, as shown in Table 3.1. The other entries in the table are computed similarly.

The transition matrix is constructed by considering all possible transitions from each state. In a given state, the flow across each connection is computed as follows. For the issue-pipe ($I-P$) connection the pull (q_{I-P}) is determined by data dependence and the push (p_{I-P}) is determined by the issue occupancy:

$$q_{I-P} = \begin{cases} 0 & \text{if } x_P = \langle \text{1x} \rangle \text{ and } s_0 = 0 \\ 1 & \text{otherwise} \end{cases}$$

$$p_{I-P} = x_I$$

$$f_{I-P} = \min\{p_{I-P}, q_{I-P}, 1\}$$

For the fetch-issue ($F-I$) connection, the pull is the number of empty slots in the issue buffer after flow out to the pipe, and the push is the fetch occupancy:

$$q_{F-I} = 1 - (x_I - f_{I-P})$$

$$p_{F-I} = x_F$$

$$f_{F-I} = \min\{p_{F-I}, q_{F-I}, 1\}$$

If the no-branch-prediction fetch buffer is used and there is a branch remaining in fetch or issue after the flow to the pipe, the pull into fetch is zero. Otherwise, the pull into fetch

is the number of empty slots in fetch. The push into fetch is always one.

$$q_F = \begin{cases} 0 & \text{if a branch remains in fetch or issue} \\ 1 - (x_F - f_{F-I}) & \text{otherwise} \end{cases}$$

$$p_F = 1$$

$$f_F = \min\{p_F, q_F, 1\}$$

The current state consists of the elements x_F, x_I, x_P, y, and s. The elements of the next state are denoted by prime marks: x'_F, x'_I, x'_P, y', and s'. Given the flows computed above, most of the next state vector can be computed:

$$x'_P = \langle (f_{I-P}) : (x_P \gg 1) \rangle$$

$$x'_I = x_I - f_{I-P} + f_{F-I}$$

$$x'_F = x_F - f_{F-I} + f_F$$

$$y'_0/s'_0 = \begin{cases} y_0/s_0 & \text{if } f_{I-P} = 0 \\ y_1/s_1 & \text{if } f_{I-P} = 1 \end{cases}$$

The notation for x'_P means that the x_P bit vector is shifted right, removing any instruction in the last stage, and a new instruction may enter the first stage as indicated by f_{I-P}.

If $f_{I-P} = 0$ then $y'_1/s'_1 = y_1/s_1$, but if $f_{I-P} = 1$, a new instruction is needed. In this case, there may be multiple possible y'_1/s'_1 values. The probability of a particular y'_1/s'_1 is given by

$$\text{Prob}\left[y'_1/s'_1\right] = \text{instrSeq}[y_0, s_0, y_1, s_1, y'_1, s'_1]$$

This determines all possible state transitions and their probabilities.

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	
state vector	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, s_0	0,1	1,0	1,0	0,1	0,1	0,4	1,0	1,0	0,1	0,1	1,0	1,0	0,1	0,1	1,5	0,5	0,5	1,0	1,0	0,1
	y_1, s_1	1,0	0,1	0,1	0,4	0,4	1,0	0,1	0,1	1,0	1,0	0,1	0,1	1,5	1,5	0,5	1,0	1,0	0,1	0,1	1,0

Table 3.2: Model state at each cycle as the processor executes the trace shown in Figure 3.4. For clarity, the upper half shows the instructions in each component (fetch buffer, issue buffer, and the two pipe stages) while the lower half shows the actual state vectors.

Table 3.2 shows the processor state at every cycle, based on hand simulation. The upper half of the table shows the instructions which are in each component, using the instruction numbers from Figure 3.3. This part is not actually part of the state and is shown only for clarity. The lower half shows the state vector as used by the statistical model for each cycle. In this example, the processor is doing no branch prediction, i.e., fetch stalls after every branch.

In cycles 2, 7, 11, and 18, the processor is in the same state: $x_F = 0, x_I = 1, x_P = \langle 01 \rangle, y_0 = 1, s_0 = 0, y_1 = 0, s_1 = 1$. In cycle 2, the next two instructions to be issued are numbers 1 (which is in the issue buffer) and 4 (which has yet to be fetched). Instruction 1 is a branch ($y_0 = 1$) and instruction 4 is not ($y_1 = 0$).

The left side of Figure 3.5 shows the processor in this state. The right side shows the processor after the transition to the next state. When the processor is in the state described

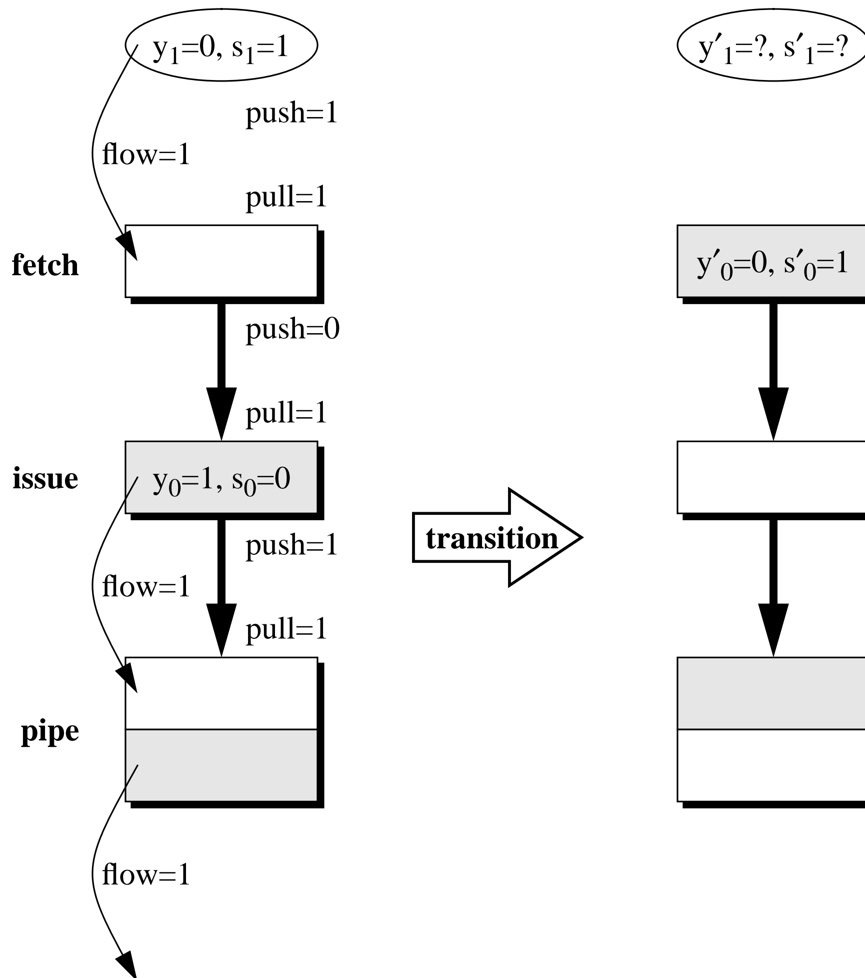


Figure 3.5: An example state transition in the one-pipe processor. The gray boxes show instructions in components; white boxes are empty buffers or pipe stages. The ovals at the top show the next instruction which will enter fetch.

above (the left side of the figure), the potential next states are computed as follows:

- The instruction in the last pipe stage leaves the pipe.
- The pull into the pipe is 1 since there is nothing for the next instruction to be dependent on. If there had been an instruction in the first stage of the pipe, rather than the last stage, the pull would be 0, since $s_0 = 0$ means that the next instruction would be dependent on this one. However, the instruction was in the last stage, and so its result is forwarded in time for the next instruction to issue.
- The push out of the issue buffer into the pipe is 1, since there is an instruction in the issue buffer.
- The push and pull are both 1, so the flow from issue to the pipe is $\min\{1,1,1\} = 1$.
- The pull into the issue buffer is 1, since it will be empty after the instruction flows into the pipe.
- Since the fetch buffer is empty, the push out of fetch into issue is 0.
- The flow from fetch to issue is $\min\{0,1,1\} = 0$.
- The pull into the fetch buffer is 1, since it is empty and there are no branches remaining in fetch or issue. If there were a branch in fetch, or a branch in issue that could not flow into the pipe, then the pull into fetch would have been 0 because fetch stalls after every branch, until the branch issues.
- The push into the fetch buffer is always 1. (There is a perfect instruction cache feeding the fetch buffer.)
- The flow into issue is $\min\{1,1,1\} = 1$.

At this point, most of the next state vector can be computed. The instruction in the last stage of the pipe flows out and a new instruction flows into the first stage, so $x'_P = 10$. The instruction in issue flows into the pipe and no new instruction flows into issue, so $x'_I = 0$. A new instruction does flow into the fetch buffer, so $x'_F = 1$. An instruction issued, so the y and s values shift by one, that is, $y'_0 = y_1 = 0$ and $s'_0 = s_1 = 1$. The instruction sequencing probabilities are needed to determine the possible values of y'_1 and s'_1.

Looking at Table 3.1, it can be seen that there are three possible values for the y'_1/s'_1 pair. There are thus three possible next states:

- $x'_F = 1,\ x'_I = 0,\ x'_P = 10,\ y'_0 = 0,\ s'_0 = 1,\ y'_1 = 0,\ s'_1 = 4$ with probability 0.25;
- $x'_F = 1,\ x'_I = 0,\ x'_P = 10,\ y'_0 = 0,\ s'_0 = 1,\ y'_1 = 1,\ s'_1 = 0$ with probability 0.50; and
- $x'_F = 1,\ x'_I = 0,\ x'_P = 10,\ y'_0 = 0,\ s'_0 = 1,\ y'_1 = 1,\ s'_1 = 5$ with probability 0.25.

These three states correspond to cycle 3, cycles 8 and 19, and cycle 12, respectively.

Note that the next states and their probabilities were computed using only the current state vector and the instruction sequencing probabilities. The instruction sequencing data extracted from the trace is sufficient to construct the entire transition matrix.

Figure 3.6 shows the complete transition matrix. This is a square matrix with one row and column for each state. Only states with non-zero probabilities are shown. The state vectors are shown along the left side of the matrix.

Table 3.3 shows the stationary distribution computed from the transition matrix of Figure 3.6. Each state is listed along with the corresponding cycle(s) and the stationary probability, i.e., fraction of time spent in the state. As expected, the probabilities match the fraction of cycles spent in each state. For example, the state corresponding to cycles 1, 10, and 17 has a probability of 3/20 = 0.15.

The stationary distribution can be used to compute the issue rate (in instructions per cycle, or IPC) of the processor. Each state-to-state transition has an associated number of

$$
\begin{array}{l}
1,1,01,0/1,1/0 \\
0,1,10,1/0,0/1 \\
0,1,01,1/0,0/1 \\
1,0,10,0/1,0/4 \\
1,1,01,0/1,0/4 \\
1,1,10,0/4,1/0 \\
0,1,11,1/0,0/1 \\
1,0,10,0/1,1/0 \\
1,0,10,0/1,1/5 \\
1,1,01,0/1,1/5 \\
0,1,10,1/5,0/5 \\
1,0,11,0/5,1/0 \\
1,1,01,0/5,1/0
\end{array}
\begin{bmatrix}
0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0.50 & 0.25 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.00 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.00 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.00 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.00 \\
0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Figure 3.6: The complete transition matrix for the simple loop on the one-pipe processor. The state descriptions along the left side are of the form $x_F, x_I, x_P, y_0/s_0, y_1/s_1$.

state	cycle(s)	probability
1, 1, 01, 0/1, 1/0	0,9	0.10
0, 1, 10, 1/0, 0/1	1,10,17	0.15
0, 1, 01, 1/0, 0/1	2,7,11,18	0.20
1, 0, 10, 0/1, 0/4	3	0.05
1, 1, 01, 0/1, 0/4	4	0.05
1, 1, 10, 0/4, 1/0	5	0.05
0, 1, 11, 1/0, 0/1	6	0.05
1, 0, 10, 0/1, 1/0	8,19	0.10
1, 0, 10, 0/1, 1/5	12	0.05
1, 1, 01, 0/1, 1/5	13	0.05
0, 1, 10, 1/5, 0/5	14	0.05
1, 0, 11, 0/5, 1/0	15	0.05
1, 1, 01, 0/5, 1/0	16	0.05

Table 3.3: The stationary distribution for the simple loop on the one-pipe processor. The states are written $x_F, x_I, x_P, y_0/s_0, y_1/s_1$ as in Figure 3.6.

issued instructions. In the case of this particular processor, instruction issue depends only on the current state, and not on which transition is taken out of that state. For example, in the $0,1,01,1/0,0/1$ state (this is the same state used above, corresponding to cycles 2, 7, 11, and 18), one instruction is issued. In the $1,0,10,0/1,0/4$ state (cycle 3), zero instructions are issued. To get the IPC distribution, the stationary probabilities corresponding to each possible issue value (0 and 1) are summed:

$$\mathbf{IPC} = \begin{bmatrix} \sum_{\substack{\text{states } x \text{ in} \\ \text{which zero} \\ \text{instr's issue}}} d_x \\ \sum_{\substack{\text{states } x \text{ in} \\ \text{which one} \\ \text{instr issues}}} d_x \end{bmatrix} = \begin{bmatrix} 0.45 \\ 0.55 \end{bmatrix}$$

The average IPC is easily computed:

$$\begin{aligned} IPC_{\text{avg}} &= \mathbf{IPC}_0 \cdot 0 + \mathbf{IPC}_1 \cdot 1 \\ &= 0.45 \cdot 0 + 0.55 \cdot 1 \\ &= 0.55 \end{aligned}$$

Computing the IPC from the simulation shown in Table 3.2 produces the same value:

$$IPC_{\text{avg}} = \frac{11 \text{ instructions}}{20 \text{ cycles}} = 0.55$$

This simple example is chosen to illustrate the modeling process. The resulting state probabilities and IPC exactly match the simulation results. The results of applying this model to the SPEC95 benchmark suite are presented next.

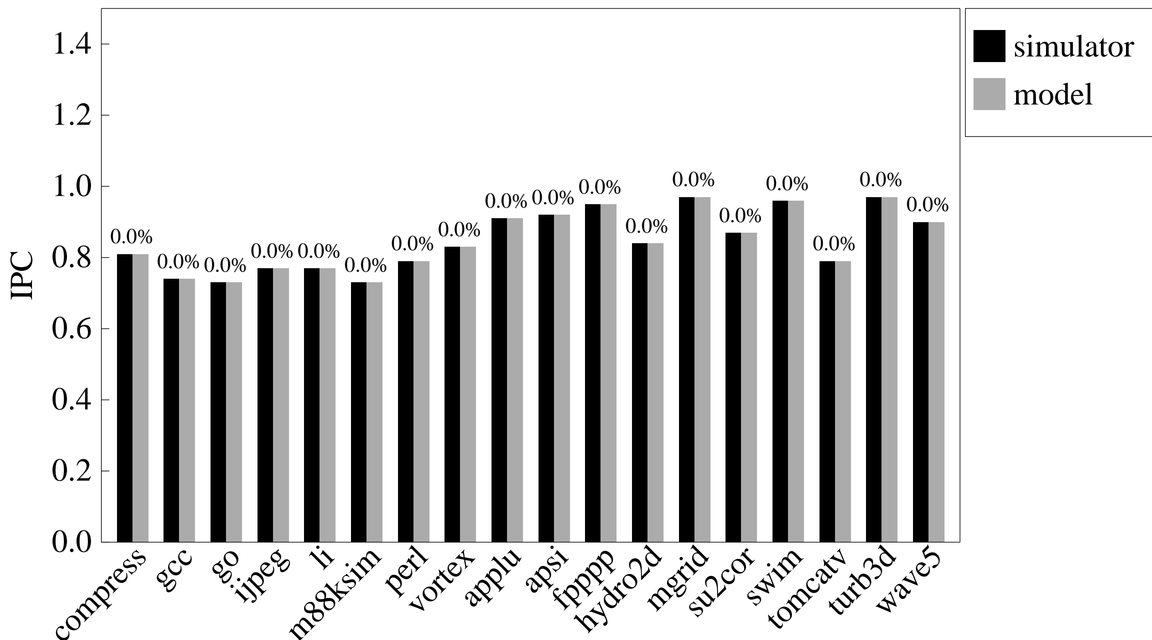


Figure 3.7: The average IPCs for the SPEC95 benchmarks, as measured by a simulator (black bars) and computed by the statistical model (gray bars) for the one-pipeline processor with perfect branch prediction. The figure above each pair of bars is the percent difference between the two IPC values.

3.4.3 Experimental Results

Figures 3.7 and 3.8 show the modeled performance numbers with perfect branch prediction and no branch prediction, respectively. Each figure shows the average IPCs for each benchmark, as computed by the statistical model and as measured by a simulator. (These numbers should not be compared to those from Chapter 2 as the processor used there is somewhat different, with a one-stage pipe and a different branch predictor.)

The statistical model is able to completely capture the behavior of this simple processor. That is, the probabilities extracted from the trace (Table 3.1) contain sufficient information to exactly predict the next states and their probabilities. Thus the statistical model exactly matches the simulator in Figures 3.7 and 3.8. These results verify that the statistical tech-

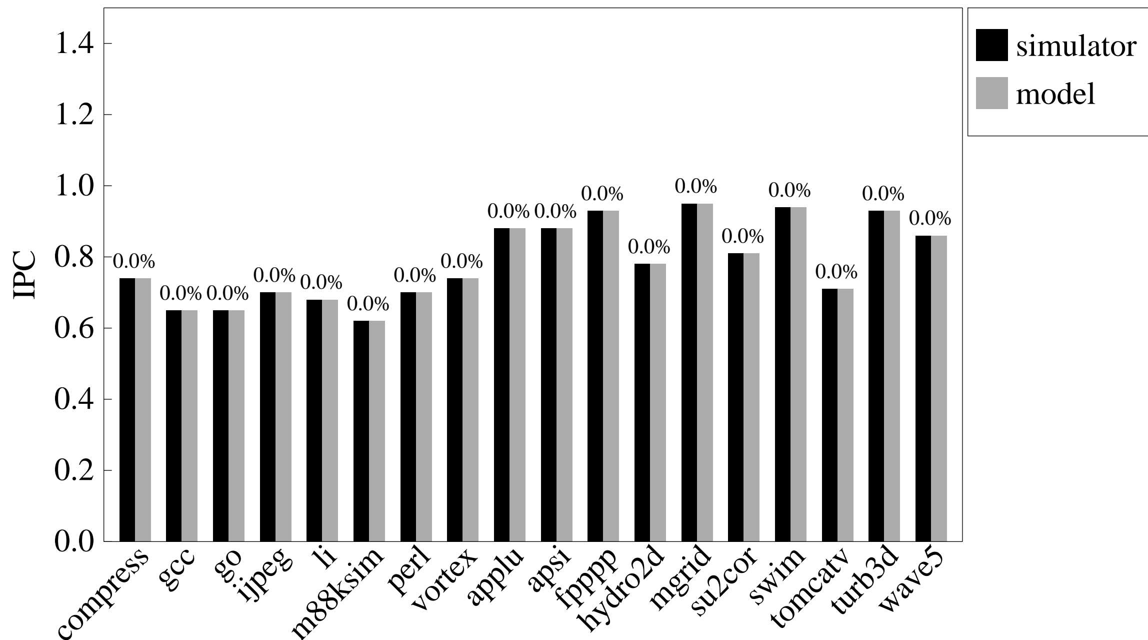


Figure 3.8: The average IPCs for the SPEC95 benchmarks, as measured by a simulator (black bars) and computed by the statistical model (gray bars) for the one-pipeline processor with no branch prediction. The figure above each pair of bars is the percent difference between the two IPC values.

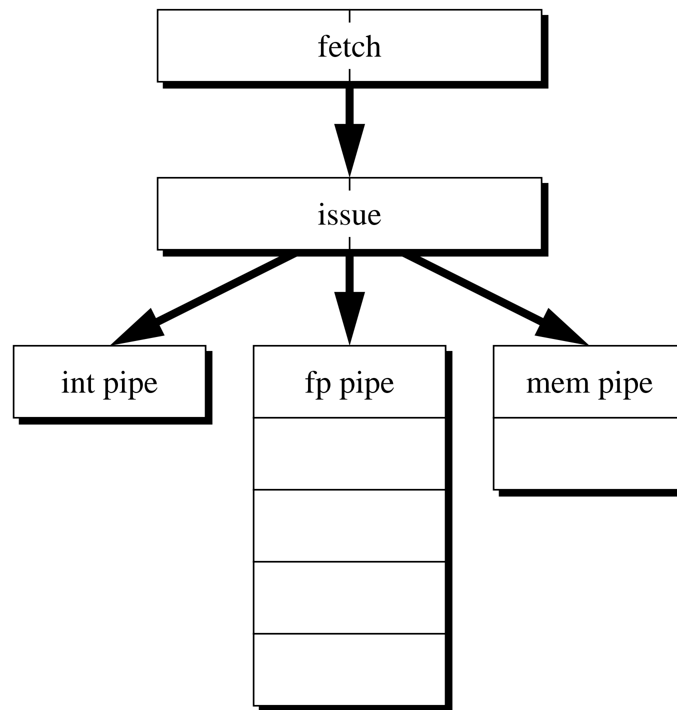


Figure 3.9: Components and connections of the three-pipe processor.

niques presented above work as expected.

The next section and subsequent chapters apply the statistical modeling techniques to more complex processor microarchitectures. With these more complex models, the extracted data lose some of the information present in the trace, and this introduces error into the predictions made by the statistical model.

3.5 A Three-Pipeline Processor

This section extends the ideas presented above to model a more complex processor. The three-pipeline processor is shown in Figure 3.9. The fetch and issue buffers work as in the one-pipe processor, but now each buffer holds two instructions. The issue policy is in-

order. Instructions are issued to three pipelines. These execute, respectively, integer (and branch) instructions with a latency of one cycle, floating point instructions with a latency of five cycles, and memory instructions with a latency of two cycles. There is no register renaming, so issue is stalled both by read-after-write and write-after-write dependences.

The state vector used for the one-pipe processor can be extended in a straightforward way to the three-pipe processor. There are now four different instruction types (integer, floating point, memory, and branch), and each instruction has three dependent instruction distances (see Section 3.2). Because there can be up to two instructions in each of the fetch and issue buffers, instruction type and distance information has to be kept for four instructions. The resulting state vector looks like this:

- x_F = the number of instructions in the fetch buffer ($0 \leq x_F \leq 2$).
- x_I = the number of instructions in the issue buffer ($0 \leq x_I \leq 2$).
- $x_{P\text{int}}$ = the occupancy bit vector for the integer pipeline — one bit per stage: an occupied stage is indicated by a 1 bit and an empty stage is indicated by a 0 bit ($0 \leq x_{P\text{int}} < 2^1$).
- $x_{P\text{fp}}$ = the occupancy bit vector for the floating point pipeline ($0 \leq x_{P\text{fp}} < 2^5$).
- $x_{P\text{mem}}$ = the occupancy bit vector for the memory pipeline ($0 \leq x_{P\text{mem}} < 2^2$).
- y_i = the type of the ith next instruction to be issued ($y_i \in \{0, 1, 2, 3\}$ for $0 \leq i \leq 3$).
- $s_i^{\text{int}}, s_i^{\text{fp}}, s_i^{\text{mem}}$ = the dependent instruction distances of the ith next instruction to be issued ($0 \leq s_i^{\text{int}}, s_i^{\text{fp}}, s_i^{\text{mem}} \leq s_{\max}$ for $0 \leq i \leq 1$).

The floating point pipe is deepest and thus determines the value of $s_{\max}$. (A single $s_{\max}$ value is used for simplicity, but a different value could be used 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 s values from 0 to 4 are needed, plus one more to indicate a longer dependence or no dependence. Thus $s_{\max} = 5$ is chosen.

This state vector results in a state space of size:

$$3 \times 3 \times 2^1 \times 2^5 \times 2^2 \times 4^4 \times ((5+1) \times (5+1) \times (5+1))^4 = 1.28 \times 10^{15}$$

This is clearly far too large to work with.

One way to reduce the state space size is to remove the distance values from the state. For each state, the probability of each possible set of s values can be computed using information extracted from the trace. To increase the accuracy, the types of the two most recently issued instructions are added to the state vector:

- y_{pi} = the type of the $(1-i)^{\text{th}}$ previous instruction issued ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).

The trace analyzer extracts the instruction distance probabilities, conditional on the six y values:

$$\begin{aligned}
&\text{instrDist}^i[y_{p0}, y_{p1}, y_0, y_1, y_2, y_3, s_0^i, s_1^i] = \\
&\quad \text{Prob}\left[\text{next instructions to be issued have distances } s_0^i, s_1^i \;\middle|\right. \\
&\qquad\qquad \left.\text{next instructions are types } y_0, y_1, y_2, y_3 \text{ and previous instructions are } y_{p0}, y_{p1}\right]
\end{aligned}$$

There is a separate instrDist table for each pipe. The instruction sequencing probabilities

are modified to include instruction types only:

$$\text{instrSeq}[y_{p0}, y_{p1}, y_0, y_1, y_2, y_3, y_{n0}, y_{n1}] = \text{Prob}\,[\text{next instructions are types } y_{n0}, y_{n1} \mid \text{current instructions are } y_{p0}, y_{p1}, y_0, y_1, y_2, y_3]$$

The state space now has

$$3 \times 3 \times 2^1 \times 2^5 \times 2^2 \times 4^6 = 9.44 \times 10^6 \text{ states}$$

This is still too large to solve. The next chapter presents a technique developed to reduce the size of the Markov chains. The state vector elements described above are still used, but they are split across several smaller Markov chains. That is, the state vector is partitioned into several smaller pieces so that the resulting individual Markov chains are each small enough to solve.

Chapter 4

Partitioned Markov Models

Using a single, monolithic Markov model is unreasonable for a processor like the three-pipe microarchitecture of Figure 3.9. This chapter shows how such a model can be partitioned into several smaller models. These smaller sub-models are easier to work with, and can be solved much more quickly and using a reasonable amount of memory. The key problem is handling the dependences between the sub-models and obtaining a simultaneous solution for all of the models.

4.1 Partitioning

The large, monolithic Markov chains described in Chapter 3 can be partitioned into smaller, mostly independent chains. The block diagram of a processor (e.g., Figure 3.9) provides a convenient partitioning guide. A separate Markov chain is used for each component. Each component has its own state vector and its own state space. These state spaces are individually much smaller than the previous monolithic state space.

This technique takes advantage of the fact that the components of the processor operate somewhat independently of each other. However, there are dependences between compo-

nents. In the processor with no branch prediction, for example, no new instructions can be fetched while a branch is in the issue buffer. These dependences between component models are handled by overlapping their state spaces. That is, the state vector for one component may contain some information which is also part of another component's state vector. This allows the state spaces to be tailored for the dependences which are present, while taking advantage of the partly-independent nature of the components. If there was no overlap between components, each element of the monolithic state would be assigned to exactly one component, and the product of the component state space sizes would be exactly equal to the size of the original, unpartitioned state space. With overlapping state spaces, the product of the component state space sizes is greater than the size of the monolithic state space. Even with the overlap, the individual state spaces are significantly smaller than the single, monolithic state space.

4.2 A Three-Pipeline Processor

This section reexamines the three-pipeline processor described in Section 3.5. The partitioning technique described above is applied to generate a five-component model. An example is used to show how the transition matrices are constructed and how all five Markov chains are solved simultaneously.

4.2.1 The Model

The monolithic Markov chain from Section 3.5 is first partitioned into five smaller, non-overlapping chains. One chain is used for each component: the fetch buffer, the issue buffer, and the three pipes. The elements of the original monolithic state vector are reassigned accordingly.

The fetch buffer state consists of:

- x_F = the number of instructions in the fetch buffer.

The issue buffer state consists of:

- y_i = the type of the ith next instruction to be issued ($y_i \in \{0,1,2,3\}$ for $0 \leq i \leq 3$).
- y_{pi} = the type of the $(1-i)^{\text{th}}$ previous instruction issued ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).
- x_I = the number of instructions in the issue buffer.

The instruction types (y) are part of the issue buffer state vector because they affect the push out of issue (this is described below).

The state for pipeline-i consists of:

- x_{Pi} = the occupancy bit vector for the pipeline — one bit per stage.

There are now five Markov chains. Solving for all five stationary distributions is effectively solving a set of simultaneous equations. The unknowns are the push (p) out of, and the pull (q) into, each component.

For the fetch buffer, the unknown is the pull into the issue buffer:

$$q_{F-I} = \text{the number of instructions that the issue buffer can accept}$$

For the issue buffer, the unknowns are the push out of fetch and the pulls into the three pipes:

$$\begin{aligned} p_{F-I} =& \text{the number of instructions that are ready to leave the fetch buffer} \\ q_{I-Pi(Pj)} =& \text{the number of type-}i\text{ instructions that are} \\ & \text{independent of all instructions in pipe-}j \end{aligned}$$

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 example, $q_{I-P\text{mem}(P\text{fp})} = 1$ means that the next memory instruction is independent of any instructions currently in the floating point pipe. For an instruction to flow from issue to a pipe, it must be independent of the instructions in *all* of the pipes. For a memory instruction to issue, it must be the case that $q_{I-P\text{mem}(P\text{int})} = q_{I-P\text{mem}(P\text{fp})} = q_{I-P\text{mem}(P\text{mem})} = 1$, i.e., that the memory instruction is independent of all instructions currently in the pipes. If $q_{I-P\text{mem}(P\text{mem})} = 0$, then the next memory instruction is dependent on a previous instruction still in the memory pipe. In this case, the new instruction cannot issue, even if $q_{I-P\text{mem}(P\text{int})} = q_{I-P\text{mem}(P\text{fp})} = 1$.

For pipe-i, the unknowns are the push out of the issue buffer and the pulls into the *other* pipes:

$$
\begin{aligned}
p_{I-Pi} =& \text{the number of type-}i\text{ instructions ready to leave the issue buffer} \\
q_{I-Pi(Pj)} =& \text{the number of type-}i\text{ instructions that are} \\
& \quad \text{independent of all instructions in pipe-}j \quad (j \neq i)
\end{aligned}
$$

These are the same pull values needed by the issue buffer, except that the pull into pipe-i is obviously not an unknown to pipe-i.

The push-out and pull-in values for a particular component depend on the current state of that component. Since a component does not “know” the states of the other components,

push and pull probability distributions are used instead:

$$\mathbf{p}_k^{F-I} = \text{Prob}\left[k \text{ instructions are ready to leave fetch}\right]$$
$$\mathbf{q}_k^{F-I} = \text{Prob}\left[\text{the issue buffer can accept } k \text{ instructions}\right]$$
$$\mathbf{p}_k^{I-Pi} = \text{Prob}\left[k \text{ type-}i \text{ instructions are ready to leave the issue buffer}\right]$$
$$\mathbf{q}_k^{I-Pi(Pj)} = \text{Prob}\left[k \text{ type-}i \text{ instructions are independent of all instructions in pipe-}j\right]$$

Building the transition matrix for a particular component requires the push-out from its source (if any), and the pull-in to its destinations (if any). Given these probability distributions, the transition matrix can be constructed and the Markov chain can be solved for its stationary distribution. Given the stationary distribution, the component's own pull-in and push-out distributions can be computed and supplied to its source and destination, respectively. An iterative relaxation technique is used to generate a simultaneous solution for all five Markov chains.

This technique relies on an implicit assumption. The component states must be statistically independent. More specifically, the probabilities of push into and pull out of a component must be independent of the current state of that component, since the component sees only a probability distribution. For example, when computing the transition matrix for the issue buffer, the push-in, $\mathbf{p}^F$, and pulls-out, $\mathbf{q}^{PjPi}$, must be independent of the current issue state.

This assumption turns out to be inaccurate. For example, consider the following piece of code:

```
T100:  fadd   f1, f2    --> f3
T101:  fadd   f4, f5    --> r6
T102:  st     f3, 0(r2)            ; s_fp = 1
T103:  st     f6, 8(r2)            ; s_fp = 0
T104:  fsub   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 (T100 and T102) are separated by another floating point add. The second pair (T101 and T103) 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 memory instructions (T102 and T103) in issue, the dependence distance of the next instruction to issue is 1. When there are one memory instruction and one floating point instruction in issue, the dependence distance is 0. Thus the pull by the floating point pipe is dependent on the issue state, which violates the independence assumption. Experiments indicate that this type of correlation is a real problem which affects the accuracy of the model

The solution is to drop the assumption that the fetch, issue, and pipe components are entirely independent. The model must allow for some correlation between them. To do this, part of the state is shared across multiple components. This is the overlap mentioned earlier. The fetch state is augmented with six instruction types and a branch-in-issue flag:

- y_i^F = the type of the ith next instruction to leave the fetch buffer ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).
- y_{pi}^F = the type of the $(1-i)^{\text{th}}$ previous instruction to leave the fetch buffer ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).
- b_I = 1 if there is a mispredicted branch in the issue buffer, 0 if not.
- x_F = the number of instructions in the fetch buffer.

The b_I flag is needed by the fetch component because fetching stops until a mispredicted branch is issued. The fetch buffer now has $4^4 \times 4^2 \times 2 \times 3 = 24,576$ states.

The issue buffer state vector is unchanged:

- y_i^I = the type of the ith next instruction to be issued ($y_i \in \{0,1,2,3\}$ for $0 \leq i \leq 3$).
- y_{pi}^I = the type of the $(1-i)^{\text{th}}$ previous instruction issued ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).
- x_I = the number of instructions in the issue buffer.

The issue buffer now has $4^4 \times 4^2 \times 3 = 12,288$ states.

The instruction type information is also added to the pipe-i state vector:

- y_i^I = the type of the ith next instruction to be issued ($y_i \in \{0,1,2,3\}$ for $0 \leq i \leq 3$).
- y_{pi}^I = the type of the $(1-i)^{\text{th}}$ previous instruction issued ($y_{pi} \in \{0,1,2,3\}$ for $0 \leq i \leq 1$).
- x_{Pi} = the occupancy bit vector for the pipeline — one bit per stage.

The pipes now have $4^4 \times 4^2 \times 2^n$ states: 8192 for the integer pipe, 131,072 for the floating point pipe, and 16,384 for the memory pipe.

The y^I and y_p^I values in the pipe state correlate with the value in the other pipe states and with the values in the issue state. That is, if the issue Markov chain is in a particular state, it "knows" that each pipe is in a state with matching y^I and y_p^I values, and vice versa. Similarly, the y^F and y_p^F values in the fetch state correlate with the y^I and y_p^I values in the issue state. (Fetch and issue are offset by the number of instructions in the issue buffer so y^F/y_p^F and y^I/y_p^I do not overlap completely).

Because the instruction types are shared among all five state spaces, the push and pull probabilities can be made conditional on instruction types and the mispredicted branch flag:

$$\mathbf{p}_k^{F-I}(y^F, y_p^F, b_I) = \text{Prob}\left[k \text{ instructions are ready to leave fetch} \,\middle|\, y^F, y_p^F, b_I\right]$$

$$\mathbf{q}_k^{F-I}(y^F, y_p^F, b_I) = \text{Prob}\left[\text{the issue buffer can accept } k \text{ instructions} \,\middle|\, y^F, y_p^F, b_I\right]$$

$$\mathbf{p}_k^{I-Pi}(y^I, y_p^I) = \text{Prob}\left[k \text{ type-}i \text{ instr's are ready to leave the issue buffer} \,\middle|\, y^I, y_p^I\right]$$

$$\mathbf{q}_k^{I-Pi(Pj)}(y^I, y_p^I) = \text{Prob}\left[k \text{ type-}i \text{ instr's are independent of all instr's in pipe-}j \,\middle|\, y^I, y_p^I\right]$$

4.2.2 Transition Matrix Computation

A separate transition matrix is constructed for each component of the model. A brief outline of the transition matrix construction algorithm is given for each component. This is followed by a detailed example.

The push out of fetch is x_F. The pull out of fetch may have several possible values, with probabilities given by $\mathbf{q}^I$. If there is a branch remaining in issue, as indicated by $b_I = 1$ and $q^I = 0$, or in fetch, the pull into fetch will be zero. Otherwise, the pull into fetch is the number of empty slots left after flow out to issue. The push into fetch is always two. If one or two instructions leave fetch, the instrSeq table is consulted for the probabilities of the new instruction types. For any given state, there are several possible transitions. Each transition probability is the product of a pull-out probability and an instrSeq probability.

There is a separate push out of issue into each pipe. These push values depend on x_I and y^I. For example, if there are an integer and a floating point instruction in issue ($x_I = 2$ and $y^I = [01\ldots]$), then the push out to the integer pipe is one, the push out to the floating point pipe is one, and the push out to the memory pipe is zero. If the instructions are both integer ($y^I = [00\ldots]$), the push out to the integer pipe is two (but the flow is limited to one

by bandwidth), and the pushes out to the other two pipes are zero. As described above, there is a pull by each pipe into each pipe. The flow from issue to a pipe is one if the push out to that pipe is at least one and the pulls by all pipes into that pipe are one. There may be several combinations of pull values, with probabilities given by $\mathbf{q}^{I-Pi(Pj)}$. If a branch remains in issue, the pull into issue is zero. This signals the fetch buffer that the branch in issue has not yet left. If there is no branch, or if the branch issues in this cycle, the pull into issue is the number of empty buffer slots. The push from fetch into issue can have multiple possible values, as indicated by the probabilities, $\mathbf{p}^{F-I}$. As with the fetch buffer, if instructions leave issue, new instructions enter with probabilities according to the instrSeq table. A transition probability is thus the product of a push-in probability, a set of pull-out probabilities (one per pipe), and an instruction type sequencing probability.

The pipelines never stall, so the push out and flow out are identical and are determined simply by the presence of an instruction in the last stage. Each pipe is responsible for computing the pulls *by* itself *into* all of the pipes. Given the set of instruction types in issue, there are potentially several possible sets of dependent instruction distance values. The distances and their probabilities are given by the instrDist for this pipe. This type and distance information is used to compute the pull values. In order to compute the flow into this pipe, the push-in is needed, along with the pulls *into* this pipe *by* all of the pipes. Finally, the instrSeq table is used to look up the probabilities of new instruction types. Therefore, each transition probability is the product of the push probability from issue, the pull probability by each other pipe, a dependent instruction distance probability, and an instruction type sequencing probability.

The following example uses the same program and trace as used with the one-pipe example (Figure 3.3). Figure 4.1 shows this trace annotated with the instruction types and dependent instruction distances. As with the previous example, the processor does no branch prediction. Table 4.1 shows the component states at each cycle, based on manual

cycle		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
instr's in processor	fetch	01	xx	xx	xx	45	6x	xx	xx	01	xx	xx	xx	23	xx	xx	56	xx	xx	xx
	issue	xx	01	1x	1x	xx	45	6x	6x	xx	01	1x	1x	xx	23	3x	xx	56	6x	6x
	int pipe	6	x	x	x	1	x	4	x	6	x	x	x	1	x	2	3	x	x	x
	mem pipe	xx	xx	0x	x0	xx	xx	5x	x5	xx	xx	0x	x0	xx	xx	xx	xx	xx	5x	x5
fetch state	$y^F_{0\text{-}3}$	23 02	02 32	02 32	02 32	02 32	32 30	23 03	23 03	23 03	03 23	03 23	03 23	03 23	23 23	23 23	23 23	23 02	23 02	23 02
	$y^F_{p0\text{-}1}$	23	23	23	23	23	02	23	23	23	23	23	23	23	03	03	03	23	23	23
	b_I	0	1	1	1	0	0	1	1	0	1	1	1	0	1	1	0	1	1	1
	x_F	2	0	0	0	2	1	0	0	2	0	0	0	2	0	0	2	0	0	0
issue state	$y^I_{0\text{-}3}$	23 02	23 02	30 23	30 23	02 32	02 32	32 30	32 30	23 03	23 03	30 32	30 32	03 23	03 23	32 32	23 23	23 23	32 30	32 30
	$y^I_{p0\text{-}1}$	23	23	32	32	23	23	02	02	23	23	32	32	23	23	30	03	03	32	32
	x_I	0	2	1	1	0	2	1	1	0	2	1	1	0	2	1	0	2	1	1
int pipe state	$y^I_{0\text{-}3}$	23 02	23 02	30 23	30 23	02 32	02 32	32 30	32 30	23 03	23 03	30 32	30 32	03 23	03 23	32 32	23 23	23 23	32 30	32 30
	$y^I_{p0\text{-}3}$	23	23	32	32	23	23	02	02	23	23	32	32	23	23	30	03	03	32	32
	x_{Pint}	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0
mem pipe state	$y^I_{0\text{-}3}$	23 02	23 02	30 23	30 23	02 32	02 32	32 30	32 30	23 03	23 03	30 32	30 32	03 23	03 23	32 32	23 23	23 23	32 30	32 30
	$y^I_{p0\text{-}3}$	23	23	32	32	23	23	02	02	23	23	32	32	23	23	30	03	03	32	32
	x_{Pmem}	00	00	10	01	00	00	10	01	00	00	10	01	00	00	00	00	00	10	01

Table 4.1: Model state at each cycle as the processor executes the trace shown in Figure 4.1. The top section shows the instructions in each component: fetch buffer, issue buffer, integer pipe, and memory pipe. The floating point pipe is omitted because it is not used by this code. The other four sections show the state vectors at each cycle for the four components.

instruction:	0	1	4	5	6	0	1	2	3	5	6	...
type (y):	2	3	0	2	3	2	3	0	3	2	3	...
distance (s^{int}):	5	5	1	5	5	5	5	1	5	5	5	...
distance (s^{fp}):	5	5	5	5	5	5	5	5	5	5	5	...
distance (s^{mem}):	0	0	0	1	0	0	0	0	5	1	0	...

Figure 4.1: A section of the trace for the loop shown in Figure 3.3. The example analysis assumes that this trace fragment is repeated, and no other code executes.

simulation.

In cycles 2 and 3 (from Table 4.1), the fetch buffer is in the state:

$$y^F = [0232], \quad y_p^F = [23], \quad b_I = 1, \quad x_F = 0$$

the issue buffer is in the state:

$$y^I = [3023], \quad y_p^I = [32], \quad x_I = 1$$

and the integer pipe is in the state:

$$y^I = [3023], \quad y_p^I = [32], \quad x_{P\text{int}} = \langle 0 \rangle$$

In cycle 2, the memory pipe is in the state:

$$y^I = [3023], \quad y_p^I = [32], \quad x_{P\text{mem}} = \langle 10 \rangle$$

and in cycle 3, it is in the state:

$$y^I = [3023], \quad y_p^I = [32], \quad x_{P\text{mem}} = \langle 01 \rangle$$

First, consider the transition probabilities from the current integer pipe state. The next instruction to be issued is a branch ($y_0^I = 3$), and all branches are executed by the integer pipe. In order to compute the flow from issue to the integer pipe, the integer instruction pull by each pipe is needed. The pull by the integer pipe is always 1 (since the integer dependence distances are all greater than 0) 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:

$$\begin{aligned}
&\mathbf{q}_0^{I-P\text{int}(P\text{mem})}(3,2,3,0,2,3) \\
&\qquad = \text{Prob}\left[\text{pull of int instr by mem pipe} = 0 \,\middle|\, y^I = [3023], y_p^I = [32]\right] \\
&\qquad = 0.5 \\
&\mathbf{q}_1^{I-P\text{int}(P\text{mem})}(3,2,3,0,2,3) \\
&\qquad = \text{Prob}\left[\text{pull of int instr by mem pipe} = 1 \,\middle|\, y^I = [3023], y_p^I = [32]\right] \\
&\qquad = 0.5
\end{aligned}$$

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

In the case when the pull is 1, the instruction sequencing information is needed to determine the new instruction type ${y^I}'_3$. The information extracted from the trace shows that there is only one possible sequence of instruction types:

$$\text{Prob}\left[{y^I}'_3 = 2 \,\middle|\, y^I = [3023], y_p^I = [32]\right] = \text{instrSeq}[3,2,3,0,2,3,2,3] = 1$$

(This also implies that the instruction type after ${y^I}'_3$ will be 3, but this information is not needed here.)

Given all of this, there are two possible state transitions for the integer pipe:

- if pull $= 0$: $y^{I'} = [3023], y^{I'}_p = [32], x'_{P\text{int}} = 0$ (no change in state), with probability 0.5 (this corresponds to the cycle 2 $\rightarrow$ cycle 3 transition)

- if pull $= 1$: $y^{I'} = [0232], y^{I'}_p = [23], x'_{P\text{int}} = 1$ (an instruction issues to the integer pipe), with probability 0.5 (this corresponds to the cycle 3 $\rightarrow$ cycle 4 transition)

Next, consider the memory pipe transitions. In cycle 2, there is an instruction in the first stage of the memory pipe ($x_{P\text{mem}} = \langle 10 \rangle$). As with the integer pipe, the integer pull by each pipe is needed. As noted above, the pulls by the integer and floating point pipes are always 1, given the current y^I and y^I_p values. To compute the pull by the memory pipe, the dependent instruction distance probabilities, which were extracted from the trace, are needed:

$$\text{Prob}\left[s_0^{\text{mem}} = 0, s_1^{\text{mem}} = 0 \,\middle|\, y^I = [3023], y^I_p = [32]\right] = \text{instrDist}[3,2,3,0,2,3;0,0] = 1$$

Since the next instruction has dependent instruction distance $s_0^{\text{mem}} = 0$ and there is a instruction in the memory pipe, the new instruction is data dependent and cannot issue. Therefore, the integer pull by the memory pipe is 0. There is only one possible state transition:

- $y^{I'} = [3023], y^{I'}_p = [32], x'_{P\text{mem}} = 01$ — no new instructions issue, the instruction in the pipe moves to the next stage

In cycle 3, the y^I and y^I_p values are the same, but the instruction has moved into the last stage of the pipe. Since this instruction will produce its result this cycle, the instruction in issue can now issue. Using the instrSeq table as with the integer pipe, the one possible state transition is computed:

- $y^{I'} = [0232], y^{I'}_p = [23], x'_{P\text{mem}} = 0$, with probability 1 — the instruction leaves the memory pipe and a new instruction issues to the integer pipe

Now the probability distribution for the pull by the memory pipe can be computed. The two states discussed above (corresponding to cycles 2 and 3) share y^I and y^I_p values, and so they share an entry in the pull probability vector. In the first state, the integer pull was 0, and in the second state, the integer pull was 1. The pull by the memory pipe for this y^I, y^I_p configuration looks like this:

$$\mathbf{q}_0^{I-P\text{int}(P\text{mem})}(3,2,3,0,2,3) = 0.5$$
$$\mathbf{q}_1^{I-P\text{int}(P\text{mem})}(3,2,3,0,2,3) = 0.5$$

Note that these are the pull probabilities needed by the integer pipe model above.

The issue buffer uses these same pull probabilities, along with the push probability from fetch (the push is zero because fetch is empty), to compute its transitions:

- $y^{I'} = [3023], y^{I'}{}_p = [32], x'_I = 1$ (no change in state), with probability 0.5 (this corresponds to the cycle 2 $\rightarrow$ cycle 3 transition)
- $y^{I'} = [0232], y^{I'}{}_p = [23], x'_I = 0$ (an instruction issues), with probability 0.5 (this corresponds to the cycle 3 $\rightarrow$ cycle 4 transition)

In the case of the first transition, the pull into issue is 0 since there is a branch remaining in the issue buffer. In the case of the second transition, the pull is 2 because the issued instruction will leave the buffer completely empty. The issue pull-in distribution is thus:

$$\mathbf{q}_0^{F-I}(3,2,3,0,2,3) = 0.5$$
$$\mathbf{q}_1^{F-I}(3,2,3,0,2,3) = 0$$
$$\mathbf{q}_2^{F-I}(3,2,3,0,2,3) = 0.5$$

Finally, the fetch buffer is considered. Since the fetch buffer is empty in both cycles,

the push out is zero. There is a probability of 0.5 that the pull out of fetch into issue is 0. This combined with the branch flag, $b_I = 1$, in that state vector, indicates that there is a branch remaining in issue, and that no instructions can be fetched. The other possibility is that the pull into issue is 2, also with probability 0.5. In this case, the branch is leaving issue and so two new instructions can be fetched.

The two fetch state transitions are:

- $y^{F\prime} = [0232], y^{F\prime}{}_p = [23], b_I' = 1, x_F' = 0$ (no change in state), with probability 0.5 (this corresponds to the cycle 2 $\rightarrow$ cycle 3 transition)
- $y^{F\prime} = [0232], y^{F\prime}{}_p = [23], b_I' = 0, x_F' = 2$ (the branch issues and two new instructions are fetched), with probability 0.5 (this corresponds to the cycle 3 $\rightarrow$ cycle 4 transition)

The push and pull probabilities form a set of interdependences among the component Markov chains. A simultaneous solution is reached by an iterative process. At each iteration, the transition matrices are constructed using the push and pull distributions from the previous iteration. Then the stationary distributions are computed, which in turn allows new push and pull values to be computed. This process is repeated until it converges on a solution.

Computing the IPC for the three-pipe processor is a little more complicated than for the one-pipe processor because the per-cycle issue rate depends not just on the current state, but also on the transition out of that state. Table 4.2 lists each state with its stationary probability, along with all possible transitions out of that state and the associated issue rates and transition probabilities. The probability of issuing i instructions is the sum of the products of the state probabilities and the probabilities of the transitions which cause i

state	cycles	stationary probability	next state	transition probability	issue rate
2302,23,0	0	0.053	2302,23,2	1.0	0
2302,23,2	1	0.053	2302,23,2	1.0	1
3023,32,1	2,3	0.105	3023,32,1	0.5	0
			0232,23,0	0.5	1
0232,23,0	4	0.053	0232,23,2	1.0	0
0232,23,2	5	0.053	3230,02,1	1.0	2
3230,02,1	6,7	0.105	3230,02,1	0.5	0
			2303,23,0	0.5	1
2303,23,0	8	0.053	2303,23,2	1.0	0
2303,23,2	9	0.053	2303,23,2	1.0	1
3032,32,1	10,11	0.105	3032,32,1	0.5	0
			0323,23,0	0.5	1
0323,23,0	12	0.053	0323,23,2	1.0	0
0323,23,2	13	0.053	3232,30,1	1.0	1
3232,30,1	14	0.053	3232,30,1	1.0	1
2323,03,0	15	0.053	2323,03,2	1.0	0
2323,03,2	16	0.053	3230,32,1	1.0	1
3230,32,1	17,18	0.105	3230,32,1	0.5	0
			2302,23,0	0.5	1

Table 4.2: All states of the issue buffer in the three-pipe model, listed with their stationary probabilities. States are shown in the format: y^I, y^I_p, x_I. The "cycles" column lists the cycles in which the issue buffer is in that state (see Table 4.1). In addition, all transitions out of each state are listed, with their probability and issue rate.

instruction to issue, i.e.,

$$\mathbf{IPC}_i = \sum_{\text{states } x} \mathbf{d}_x \cdot \sum_{\substack{\text{transitions } x \to y \\ \text{that issue } i \text{ instrs}}} \mathbf{P}_{x \to y}$$

where $\mathbf{d}$ is the stationary issue state distribution and $\mathbf{P}$ is the issue transition matrix. For the example,

$$\begin{aligned}
\mathbf{IPC} &= \begin{bmatrix} 0.053 \cdot 1.0 + 0.105 \cdot 0.5 + 0.053 \cdot 1.0 + 0.105 \cdot 0.5 + 0.053 \cdot 1.0 \\ \quad + 0.105 \cdot 0.5 + 0.053 \cdot 1.0 + 0.053 \cdot 1.0 + 0.105 \cdot 0.5 \\ 0.053 \cdot 1.0 + 0.105 \cdot 0.5 + 0.105 \cdot 0.5 + 0.053 \cdot 1.0 + 0.105 \cdot 0.5 \\ \quad + 0.053 \cdot 1.0 + 0.053 \cdot 1.0 + 0.053 \cdot 1.0 + 0.105 \cdot 0.5 \\ 0.053 \cdot 1.0 \end{bmatrix} \\
&= \begin{bmatrix} 0.474 \\ 0.474 \\ 0.053 \end{bmatrix}
\end{aligned}$$

The average IPC is then:

$$IPC_{\text{avg}} = 0.474 \cdot 0 + 0.474 \cdot 1 + 0.053 \cdot 2 = 0.579$$

which matches the expected value of $11/19 = 0.579$.

4.2.3 Experimental Results

Figures 4.2 and 4.3 show the model results with perfect branch prediction and no branch prediction, respectively. Each figure shows the average IPCs computed by the statistical model compared to the simulated figures.

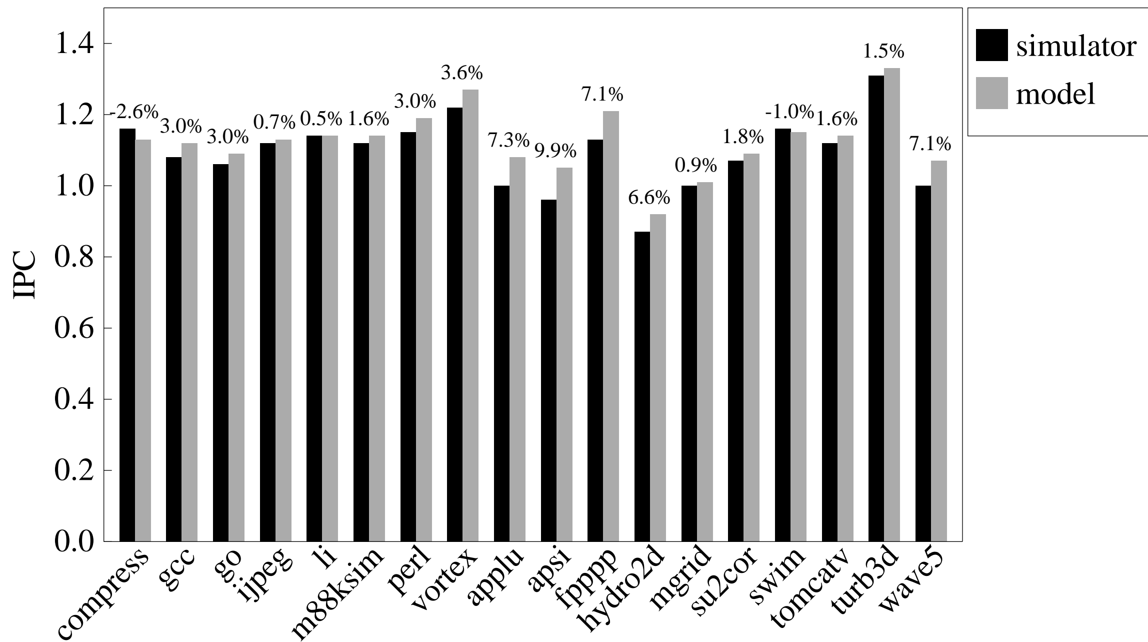


Figure 4.2: The average IPC for the SPEC95 benchmarks, with perfect branch prediction, as measured by a simulator (black bars) and computed by the partitioned Markov chain model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

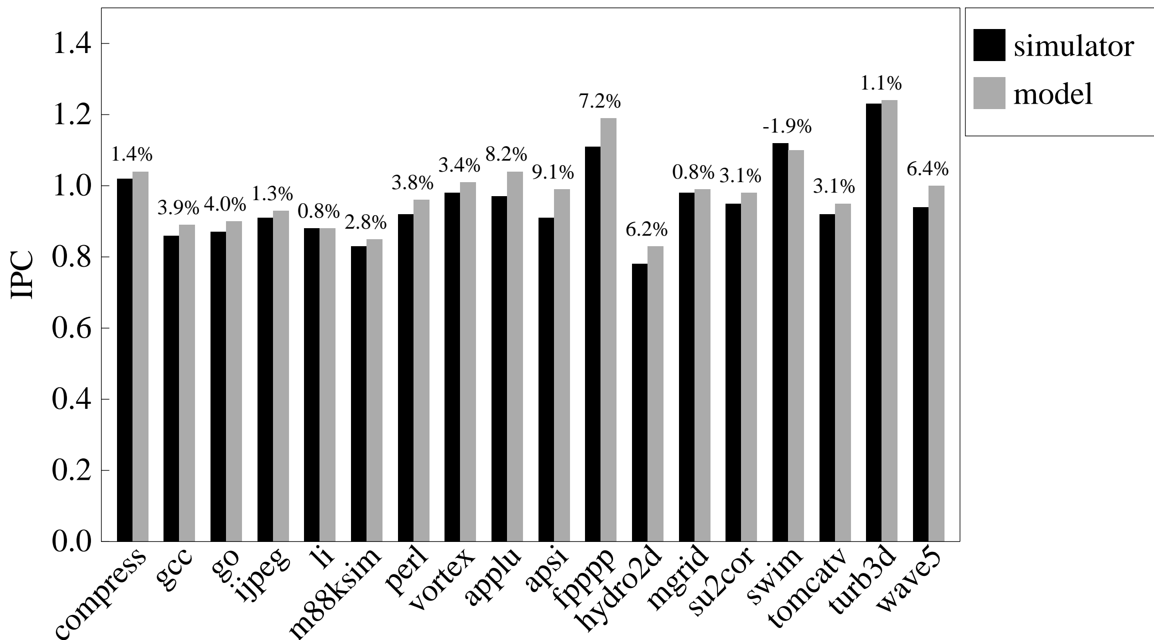


Figure 4.3: The average IPCs for the SPEC95 benchmarks, with no branch prediction, as measured by a simulator (black bars) and computed by the partitioned Markov chain model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

The maximum error for either branch predictor is 9.9%, which is for `apsi` with perfect branch prediction. It also has the worst error figure with no branch prediction: 9.1%. The average errors are 3.5% and 3.8% with perfect and no branch prediction, respectively. The main disadvantage of this model is that the state vector does not accurately identify a position in the trace. This leads to the aliasing problem described in the next section.

4.3 Aliasing

Computation of the pipeline transition matrices makes use of the dependent instruction distance probabilities extracted from the trace:

$$\begin{aligned}&\text{instrDist}^i[y_{p0}, y_{p1}, y_0, y_1, y_2, y_3; s_0^i, s_1^i] = \\ &\quad \text{Prob}\left[\text{next instructions to be issued have distances } s_0^i, s_1^i \;\middle|\right. \\ &\qquad\qquad \left.\text{next instructions are types } y_0, y_1, y_2, y_3 \text{ and previous instructions were } y_{p0}, y_{p1}\right]\end{aligned}$$

There is an inherent **aliasing** problem here. It is implicitly assumed in the conditional probability notation above that the probabilities of the distances, Prob $\left[s_0^i, s_1^i\right]$, depend *only* on the instruction types, $y_{p0}, y_{p1}, y_0, y_1, y_2, y_3$. This seems at first to be a reasonable assumption, but in fact it is problematic. The pipe state vector contains more information than just the instruction types. In particular, it is possible for the distance probabilities to depend on the pipe occupancy, x_{Pi}.

The following example demonstrates the aliasing problem. In one state, the instruction types match several aliased positions in the trace. The pipeline state, $x_{P\text{fp}}$, indicates one specific position in the trace, but $x_{P\text{fp}}$ is not used in looking up the Prob $\left[s_0^i, s_1^i\right]$ values.

Consider a trace that contains the following sequence of instructions, repeated many times:

y_{p0}	y_{p1}	y_0	y_1	y_2	y_3	s_0^{int}	s_0^{fp}	s_1^{int}	s_1^{fp}	probability
1	0	1	0	1	0	∞	∞	∞	∞	0.33
1	0	1	0	1	0	∞	0	∞	∞	0.33
1	0	1	0	1	0	∞	1	∞	∞	0.33
0	1	0	1	0	1	∞	∞	∞	∞	0.33
0	1	0	1	0	1	∞	∞	∞	0	0.33
0	1	0	1	0	1	∞	∞	∞	1	0.33

Table 4.3: The instruction distance (instrDist) table for the aliasing example.

```
T100:  fadd   f1, f2   --> f3    ; y = 1 (fp),   s_int = ∞,  s_fp = ∞
T101:  add    r10, r11 --> r12   ; y = 0 (int),  s_int = ∞,  s_fp = ∞
T102:  fadd   f3, f4   --> f5    ; y = 1 (fp),   s_int = ∞,  s_fp = 0
T103:  add    r13, r14 --> r15   ; y = 0 (int),  s_int = ∞,  s_fp = ∞
T104:  fadd   f3, f6   --> f7    ; y = 1 (fp),   s_int = ∞,  s_fp = 1
T105:  add    r16, r17 --> r18   ; y = 0 (int),  s_int = ∞,  s_fp = ∞
```

Instruction T102 is dependent on T100, so it will have to wait for the floating point pipe to produce T100's result. Instruction T104 is also dependent on T100, but it will issue after T102, so there will be no additional stalling.

The instrDist table extracted from the trace is shown in Table 4.3. (There are no dependences on memory instructions, so s^{mem} is not shown.) The first three lines represent the case where the next two instructions to issue are floating point and integer, respectively ($y_0 = 1,\ y_1 = 0$). The floating point instruction is either independent ($s_0^{\text{fp}} = \infty$), dependent with distance zero ($s_0^{\text{fp}} = 0$), or dependent with distance one ($s_0^{\text{fp}} = 1$) with equal probabilities of 0.33 each. The integer instruction is always independent ($s_1^{\text{int}} = s_1^{\text{fp}} = \infty$). This corresponds to the three different floating point instructions in the trace. The last three lines represent the other case, where an integer instruction is ahead of a floating point instruction. The probabilities in this case are the same, but the dependence is marked by s_1^{fp} instead of s_0^{fp}, since the floating point instruction is next to issue.

If, in a given state, the floating point pipe has instructions in each of its first three

stages, they must be T102 and T104 from one iteration, followed by T100 from the next iteration. These are the only three consecutive, mutually-independent floating point instructions. However, this information is not explicit in the state vector. When the floating point pipe transition probabilities for this state are computed, there will be a 0.33 probability that the next floating point instruction is independent, even though it is clear from the state vector that it must be instruction T102, which is not independent and cannot issue.

Aliasing happens because the component states do not accurately identify a position within the trace, and because not all of the information the states contain is used to index the lookup tables extracted from the trace. Another example is correlation between the instrSeq and instrDist tables. Two separate tables are used, but it is very likely that the probability of the next two instruction types is not independent of the probability of the dependence distances of the current two instructions. The next chapter presents a technique for avoiding the aliasing problem.

Chapter 5

The Statistical Flow Graph

The state vectors described in the last two chapters are similar, but not identical, to the processor state maintained by a simulator. The statistical model state includes enough information to determine how the instructions currently in flight in the processor will behave, but unlike the simulator state, does not refer to *specific instructions* within the trace. A simulator keeps track of the instruction address and/or binary image of every instruction in flight. The statistical model keeps track only of instruction types and possibly dependent instruction distances. The statistical model's more abstract state leads to the aliasing problem described in Section 4.3. The statistical flow graph concept presented in this chapter is a method of making the statistical model state more closely related to a position within the program in order to decrease aliasing.

5.1 The Statistical Flow Graph

A dynamic instruction is identified by its type (y) and dependent instruction distances ($s^{\text{int}}, s^{\text{fp}}, s^{\text{mem}}$). Every instruction in the trace with identical y and s parameters is assigned the same **instruction number** by the trace analyzer. In general, many dynamic instruc-

	y	s^{int}	s^{fp}	s^{mem}	instr#	
`T200: ld  0(r1)  --> r2`	2	∞	∞	∞	30	sequence #10
`T201: add r2, #1 --> r3`	0	∞	∞	0	31	sequence #11
`T202: ld  8(r1)  --> r4`	2	∞	∞	∞	30	sequence #10
`T203: add r4, #1 --> r5`	0	∞	∞	0	31	sequence #11
`T204: ld  16(r1) --> r6`	2	∞	∞	∞	30	sequence #12
`T205: add r3, r6 --> r7`	0	1	∞	0	32	
⋮						
`T500: ld  0(r1)  --> r8`	2	∞	∞	∞	30	sequence #10
`T501: sub r8, #2 --> r9`	0	∞	∞	0	31	sequence #13
`T502: br`	3	∞	∞	∞	33	

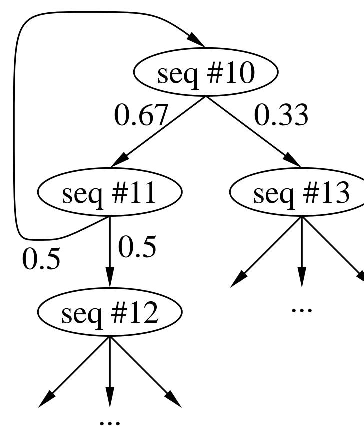
(a)

instruction number	y	s^{int}	s^{fp}	s^{mem}
30	2	∞	∞	∞
31	0	∞	∞	0
32	0	1	∞	0
33	3	∞	∞	∞

(b)

sequence number	instr 0	instr 1
10	30	31
11	31	30
12	30	32
13	31	33

(c)



(d)

Figure 5.1: (a) A section of a trace with instruction types and dependent instruction distances, instruction numbers, and instruction sequence numbers. (b) The distinct instructions, listed by instruction number. (c) The instruction sequences. (d) The statistical flow graph for the trace.

tions, including instructions with different static instruction addresses, are assigned the same instruction number.

Figure 5.1(a) shows a section of a trace. Each instruction is marked with its instruction number. (The “Txxx” numbers along the left side of the trace are trace indexes, not to be confused with instruction numbers.) Instructions T200, T202, T204, and T500 have the same *y* and *s* parameters, so they are all assigned the same instruction number: 30. Instructions T201, T203, and T501 are similarly given instruction number 31. Instructions T205 and T502 do not match any other instructions, so they are assigned instruction numbers 32 and 33, respectively. The complete list of distinct instructions is given in Figure 5.1(b).

Consecutive dynamic instructions are grouped into **instruction sequences**. Here, instruction sequences of length two are used (because the fetch and issue buffers hold two instructions; the sequence length would have to be increased to accommodate processors with larger buffers). Each dynamic instruction in the trace is the start of a sequence — consecutive sequences overlap. An instruction sequence consists of two instruction numbers. Each unique instruction sequence is assigned an instruction sequence number.

Instruction sequence numbers are marked along the right side of Figure 5.1(a). Instruction pairs T200-T201, T202-T203, and T500-T501 all consist of instruction numbers 30 and 31, so they are assigned the same sequence number: 10. Similarly, instruction pairs T201-T202 and T203-T204 are assigned sequence number 11. The pairs T204-T205 and T501-T502 are distinct from all of the other pairs, so they are given sequence numbers 12 and 13. Figure 5.1(c) shows all of the instruction sequences found in the trace section.

A sequence and its successor overlap. For example, in Figure 5.1(a), the first two sequences are 10 and 11. Sequence 11, which is the successor of sequence 10 in this instance, overlaps it by one instruction. Sequence 10 ends with instruction 31; sequence 11 begins with instruction 31.

For each sequence, a record is kept of its successors and how many times each successor

followed it. From these counts, a set of next-sequence probabilities is computed. The **statistical flow graph** is the graph of instruction sequences. Arcs connect each sequence to all of its successors, and these arcs are annotated with the next-sequence probabilities. The statistical flow graph is similar in concept to a control flow graph annotated with branch outcome probabilities, as used in code profiling, except that the statistical flow graph nodes are overlapping instruction sequences rather than basic blocks.

Figure 5.1(d) is the portion of the statistical flow graph corresponding to the portion of the trace shown in Figure 5.1(a). (For purposes of the example, it is assumed that these instruction sequences do not appear elsewhere in the trace.) Sequence 10 is followed twice by sequence 11 and once by sequence 13, so its successors are sequence 11 with probability 0.67 and sequence 13 with probability 0.33. Sequence 11 is followed once by sequence 12 and once by sequence 10, so the successor probabilities are 0.5 and 0.5.

The trace analyzer is responsible for building the statistical flow graph. This takes the form of two tables. The first, shown in Figure 5.1(b) contains information on each instruction. The table entries for instruction number *instr* are:

$$\begin{aligned}
\text{instrInfo}[\textit{instr}].y &= \text{the instruction type} \\
\text{instrInfo}[\textit{instr}].s^{\text{int}} &= \text{the integer dependent instruction distance} \\
\text{instrInfo}[\textit{instr}].s^{\text{fp}} &= \text{the floating point dependent instruction distance} \\
\text{instrInfo}[\textit{instr}].s^{\text{mem}} &= \text{the memory dependent instruction distance}
\end{aligned}$$

The second table contains the sequence information shown in Figure 5.1(c,d). The table

entries for sequence number *seq* are:

$$\text{seqInfo}[seq].instr_0 = \text{the number of the first instruction in the sequence}$$
$$\text{seqInfo}[seq].instr_1 = \text{the number of the second instruction in the sequence}$$
$$\text{seqInfo}[seq].n_{\text{next}} = \text{the number of possible next sequences}$$
$$\text{seqInfo}[seq].next_i = \text{the } i^{\text{th}} \text{ next sequence number } (0 \le i < n_{\text{next}})$$
$$\text{seqInfo}[seq].prob_i = \text{Prob}\left[\text{sequence } seq \text{ is followed by sequence } next_i\right]$$

Sequence *seq* has a number of successors, given by $\text{seqInfo}[seq].n_{\text{next}}$, in the statistical flow graph. The list of successors is given by the $next_i$ entries in the seqInfo table. The probabilities of the successors are given by the corresponding $prob_i$ entries.

The instrInfo and seqInfo tables are used by the modeler to construct the transition matrices. The instrSeq table used in Chapter 4 is replaced by the next-sequence probabilities from the seqInfo table. The instrDist probability table is no longer needed because the dependent instruction distances are part of the instrInfo table.

In addition to identifying the instruction types and distances, a sequence number also identifies, in a probabilistic way, a position in the trace. This helps solve the aliasing problem, especially with the further modifications described below in Section 5.5.

5.2 The Three-Pipeline Processor

This section looks at the same three-pipe processor (Figure 3.9) examined in Sections 3.5 and 4.2. The state vectors no longer explicitly encode the instruction type information (y and y_p). Instead, they include the instruction sequence number, which serves as an index into the seqInfo table.

The new fetch buffer state consists of:

- seq_F = the sequence number of the sequence whose first instruction will next leave fetch. If fetch is non-empty, this is the sequence that starts with the first instruction currently in fetch. If fetch is empty, this is the next sequence which will enter fetch. The instructions in this sequence are the same instructions which were described by y_0^F and y_1^F in the previous model.

- b_I = 1 if there is a mispredicted branch in the issue buffer, 0 if not.

- x_F = the number of instructions in the fetch buffer.

The seq_F value replaces the y^F and y_p^F values used in the model of Section 4.2.

The issue buffer state is modified similarly:

- seq_I = the sequence number of the sequence whose first instruction will be next to issue. If issue is non-empty, this is the sequence starting with the first instruction currently in issue. If issue is empty, this is the next sequence which will enter issue.

- x_I = the number of instructions in the issue buffer.

And finally, the pipeline states look like this:

- seq_I = the sequence number to be issued next. This is the same seq_I value used in the issue buffer state.

- x_{Pi} = the occupancy bit vector for the pipeline — one bit per stage.

Here, the seq_I value replaces the y^F and y_p^F values and also serves as a table index to get the distance (s) values, thus removing the need for a separate instrDist table.

There are still parts of the state vectors which are shared between the components. Instead of y and y_p instruction type values, the shared state is now the sequence numbers, seq_F

and seq_I. The push and pull probabilities, which were conditional on the instruction types (and mispredicted branch flag), are now made conditional on the instruction sequences:

$$\begin{aligned}
\mathbf{p}_k^{F-I}(seq_F, b_I) &= \text{Prob}\left[k \text{ instructions are ready to leave fetch} \,|\, seq_F, b_I\right] \\
\mathbf{q}_k^{F-I}(seq_F, b_I) &= \text{Prob}\left[\text{the issue buffer can accept } k \text{ instructions} \,|\, seq_F, b_I\right] \\
\mathbf{p}_k^{I-Pi}(seq_I) &= \text{Prob}\left[k \text{ type-}i \text{ instr's are ready to leave the issue buffer} \,|\, seq_I\right] \\
\mathbf{q}_k^{I-Pi(Pj)}(seq_I) &= \text{Prob}\left[k \text{ type-}i \text{ instr's are independent of all instr's in pipe-}j \,| \right. \\
&\qquad \left. seq_I\right]
\end{aligned}$$

The number of distinct instructions found in a trace is denoted n_{instr}. The maximum possible n_{instr} value is determined by the number of instruction types and the maximum dependent instruction distance. In order to reduce the number of instructions, integer dependence distances are limited to $s_{\text{max}}^{\text{int}} = 1$. (This is sufficient as long as the integer pipe has only one stage). A trace would contain the maximum possible number of distinct instructions if every instruction type were present with every possible combination of dependent instruction distances:

$$\begin{aligned}
n_{\text{instr}}^{\text{max}} &= n_{\text{types}} \times (s_{\text{max}}^{\text{int}} + 1) \times (s_{\text{max}}^{\text{fp}} + 1) \times (s_{\text{max}}^{\text{mem}} + 1) \\
&= 4 \times 2 \times 6 \times 6 \\
&= 288
\end{aligned}$$

The state space sizes depend on the number of instruction sequences in the trace, n_{seq}. Since each instruction sequence consists of two instructions, the maximum possible n_{seq}

value is given by:

$$n_{\text{seq}}^{\text{max}} = (n_{\text{instr}}^{\text{max}})^2$$
$$= 82,944$$

In practice, the number of sequences extracted from real traces is significantly smaller than the theoretical maximum. For the SPEC95 benchmarks, n_{seq} ranges from 357 to 1654.

For example, if $n_{\text{seq}} = 1000$, the state space sizes are as follows:

- fetch: $n_{\text{seq}} \times 2 \times 3 = 6,000$.
- issue: $n_{\text{seq}} \times 3 = 3,000$.
- integer pipe: $n_{\text{seq}} \times 2^1 = 2,000$.
- floating point pipe: $n_{\text{seq}} \times 2^5 = 32,000$.
- memory pipe: $n_{\text{seq}} \times 2^2 = 4,000$.

The transition matrices are constructed in basically the same way described in Section 4.2.2. Instead of using the instrSeq table to look up all of the possible succeeding instruction types and their probabilities, the statistical flow graph is used to determine all possible succeeding instruction sequences and their probabilities. The previous model did not include the dependent instruction distances in the state vectors. For any given state, there were multiple possible combinations of distances, each with a probability of occurrence. The distances and their probabilities were given by the instrDist table. With the statistical flow graph, the distances are now effectively part of the state, via the s entries in the instrInfo table.

	y	s^{int}	s^{fp}	s^{mem}	
`T100: fadd f1, f2 --> f3`	1	∞	∞	∞	sequence #10
`T101: add  r10, r11 --> r12`	0	∞	∞	∞	sequence #11
`T102: fadd f3, f4 --> f5`	1	∞	∞	0	sequence #12
`T103: add  r13, r14 --> r15`	0	∞	∞	∞	sequence #13
`T104: fadd f3, f6 --> f7`	1	∞	∞	1	sequence #14
`T105: add  r16, r17 --> r18`	0	∞	∞	∞	sequence #15
`T100: fadd f1, f2 --> f3`	1	∞	∞	∞	sequence #10
`T101: add  r10, r11 --> r12`	0	∞	∞	∞	
⋮					

Figure 5.2: The aliasing example from Section 4.3 with instruction sequences marked. The six instructions repeat indefinitely; the figure repeats the first two (T100 and T101) to show the instruction sequence numbers.

5.3 Aliasing

Using instruction sequences and the statistical flow graph significantly reduces the effects of the aliasing problem. Figure 5.2 shows the same piece of code used in Section 4.3 to demonstrate the aliasing problem. Here, the code is annotated with the assigned instruction sequence numbers. Recall that if there are three instructions in the floating point pipe, they must be instructions T102 and T104 from one iteration and instruction T100 from the next, because of data dependence constraints.

With the previous model, the only "position" information contained in the state vectors is the instruction types, which are always either [0 1 0 1 0 1] or [1 0 1 0 1 0], i.e., an alternating series of integer and floating point instructions. This is not sufficient information to look up the correct dependent instruction distances, which depend on the state of the floating point pipe in addition to the instruction types.

With instruction sequences, there are six unique sequence numbers. The floating point

pipe cycles through the following states:

1. $seq_I = 14$, $x_{P\text{fp}} = \langle 10000 \rangle$: instruction T102 has just issued, after waiting for the floating point pipe to empty out (since it is dependent on the previous floating point instruction, T100). Sequence 14, i.e., instructions T104 and T105 are next in line to issue.

2. $seq_I = 10$, $x_{P\text{fp}} = \langle 11000 \rangle$: instruction T104, which is independent of the instruction currently in the pipe (since $s^{\text{fp}} = 1$), has just issued.

3. $seq_I = 12$, $x_{P\text{fp}} = \langle 11100 \rangle$: instruction T100, which is completely independent ($s^{\text{fp}} = \infty$) has just issued.

4. $seq_I = 12$, $x_{P\text{fp}} = \langle 01110 \rangle$: instruction T102 cannot issue yet because it is dependent on the latest instruction in the pipe ($s^{\text{fp}} = 0$).

5. $seq_I = 12$, $x_{P\text{fp}} = \langle 00111 \rangle$

6. $seq_I = 12$, $x_{P\text{fp}} = \langle 00011 \rangle$

7. $seq_I = 12$, $x_{P\text{fp}} = \langle 00001 \rangle$

8. $seq_I = 14$, $x_{P\text{fp}} = \langle 10000 \rangle$: the pipe finally drains, and instruction T102 issues. This is the same state as the first one in this list — each iteration takes seven cycles to execute.

If there are three instructions in the floating point pipe, i.e., the pipe is in one of the third through seventh states listed above, then the current sequence number must be 12. Given the sequence number, the correct dependent instruction distances are found in the instrInfo table, via the instruction numbers from the seqInfo table.

Although the statistical flow graph significantly improves the aliasing problem, they do not solve it entirely. Multiple instruction pairs map to the same sequence number, and there

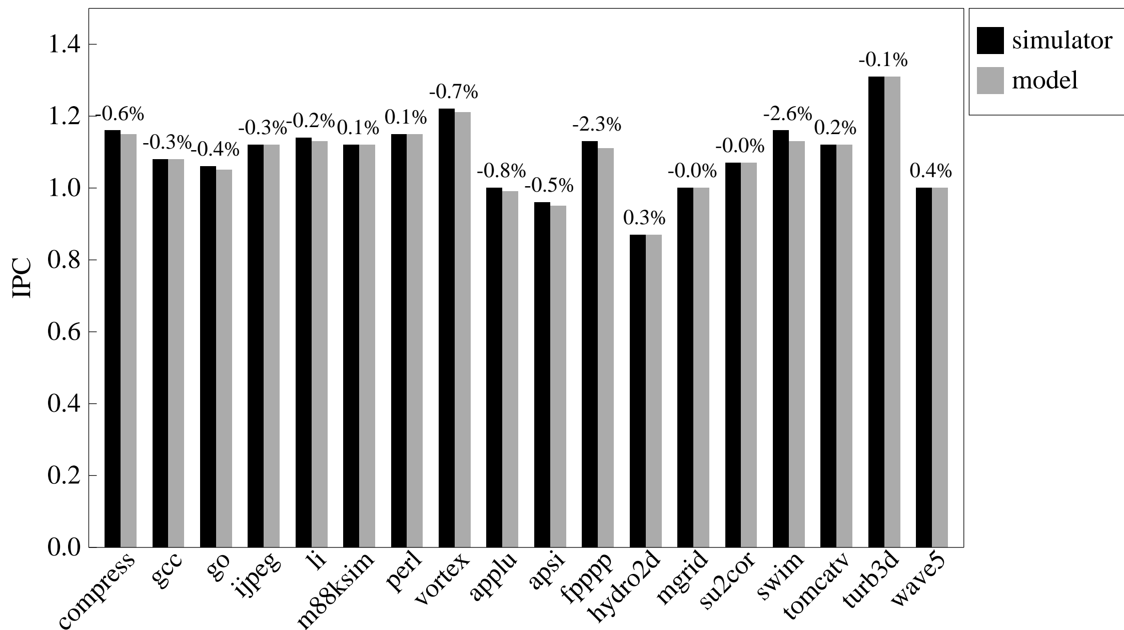


Figure 5.3: The average IPCs for the SPEC95 benchmarks, with perfect branch prediction, as measured by a simulator (black bars) and computed by the instruction sequence-based model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

can be aliasing problems between these multiple pairs. Section 5.5 augments the instruction sequences with additional information which further reduces the aliasing problem.

5.4 Experimental Results

Figures 5.3 and 5.4 show the results of applying the instruction sequence model to the SPEC95 benchmarks with perfect branch prediction and no branch prediction, respectively. The accuracy has improved over the original partitioned model (Figures 4.2 and 4.3) for nearly all of the benchmarks. (There are a few cases where the error is slightly larger.) The worst case errors with the instruction sequence model are 2.2% with perfect branch prediction and 2.6% with no branch prediction, compared to 9.9% and 9.1% with the previous

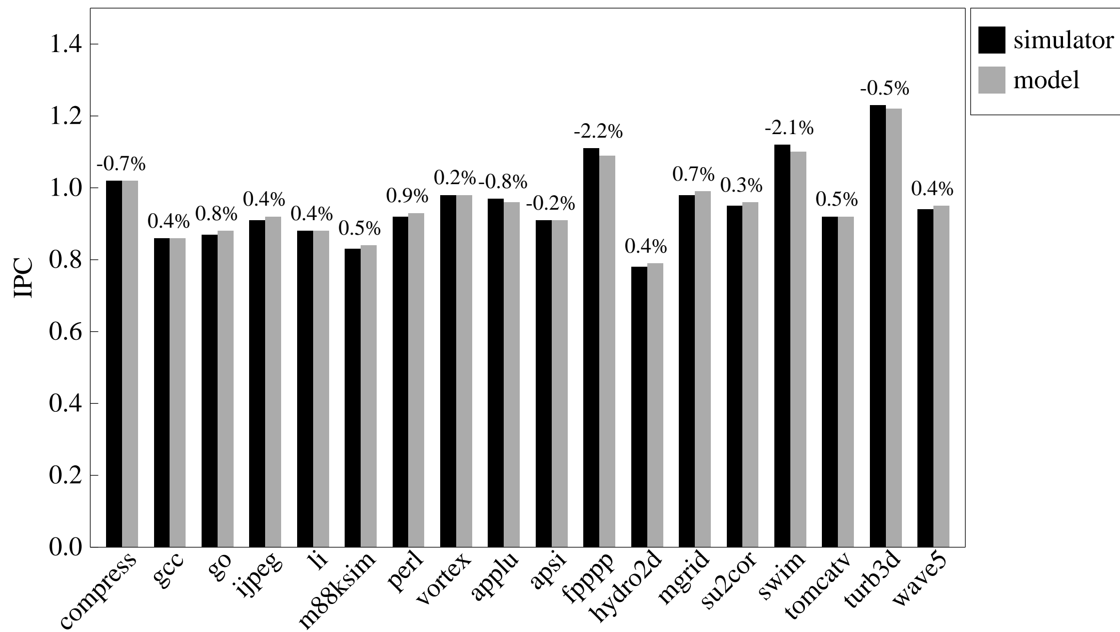


Figure 5.4: The average IPCs for the SPEC95 benchmarks, with no branch prediction, as measured by a simulator (black bars) and computed by the instruction sequence-based model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

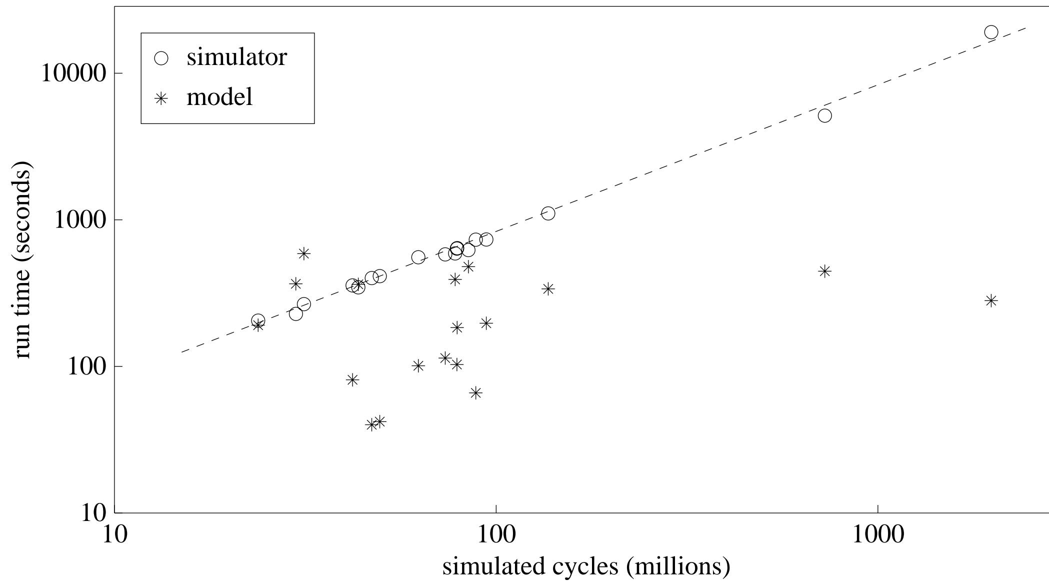


Figure 5.5: Run times (in CPU-seconds) of the simulator and statistical model for each benchmark, plotted against the number of simulated cycles for that benchmark. The dashed line has a slope of 120,000 cycles per second, which is the average simulation speed. Note that this is a log-log plot.

model. The average errors are down to 0.5% and 0.7%, from 3.5% and 3.8% previously.

Figure 5.5 shows the run times of the simulator and statistical model for the processor model and benchmarks in Figure 5.3. The simulator consistently performs at approximately 120 thousand cycles per second (the dashed line). This linear relation is the expected result for a timing simulator.

As shown by Figure 5.6, the statistical model run times are linearly dependent on the size of the statistical flow graph, i.e., n_{seq}. The statistical flow graph sizes are in turn dependent on the number of unique instructions executed, and are independent of the trace length. The model run times fall into a range between 40 and 600 seconds. There are two benchmarks for which the model is actually slower than the simulation. These are both cases in which the statistical flow graph is relatively large, and the trace is relatively short.

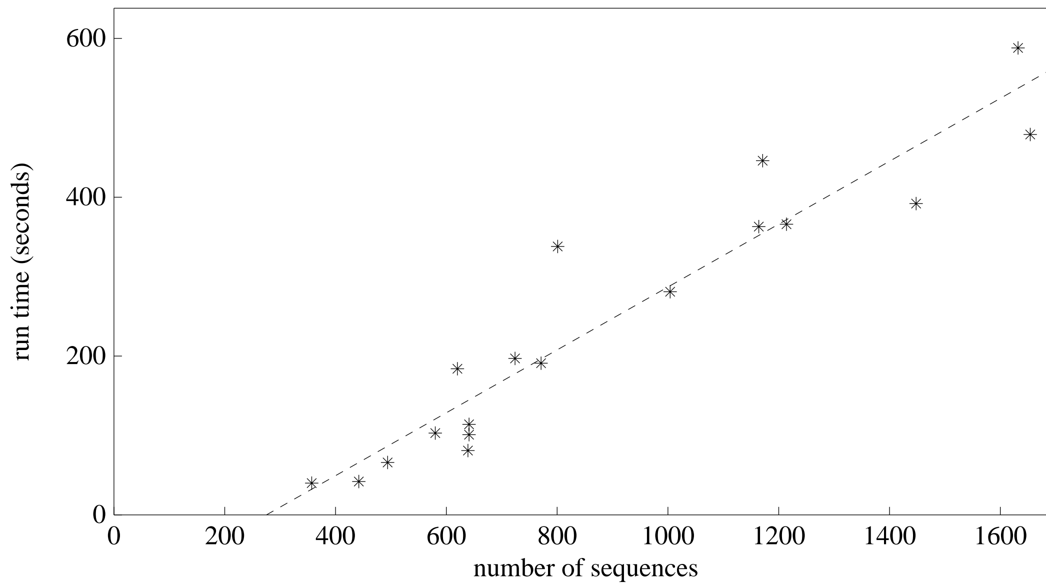


Figure 5.6: Run times (in CPU-seconds) of the statistical model, plotted against the number of sequences, i.e., nodes in the statistical flow graph.

The most important feature of the graph in Figure 5.5 is that model run time is independent of trace length. Even the longest traces have model run times in the same 40–600 second range.

Parameters other than branch prediction can also be varied. As an example, Figure 5.7 shows what happens for one floating point benchmark, `fpppp`, as the depth of the floating point pipeline is decreased (with perfect branch prediction). The error decreases as pipe depth decreases because aliasing is less of a problem with fewer instructions in flight to cause data dependence hazards. Chapters 6, 7, and 8 present additional examples of varying machine characteristics.

As is pointed out in Chapter 2, parallelism distributions are often more useful than single average values. Figure 5.8 shows several interesting parallelism distributions for the `fpppp` benchmark (with no branch prediction). The buffer and pipe occupancy distributions show the fraction of time during which the buffer or pipe contains each possible

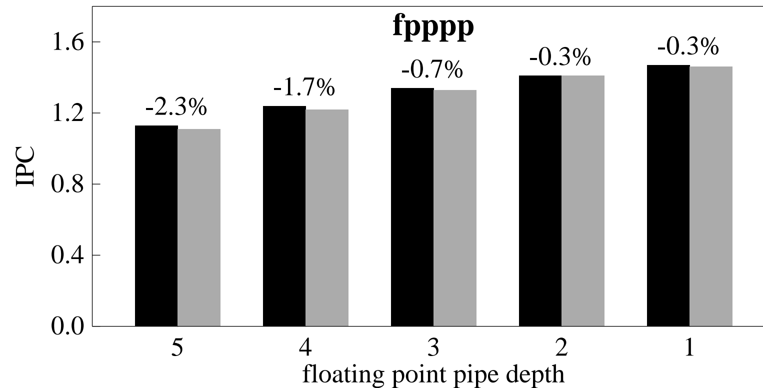


Figure 5.7: The average IPC for the fpppp benchmark as the number of stages in the floating point is decreased from five to one. The black bars are the simulated IPCs, the gray bars are the modeled IPC, and the number above each pair of bars is the percent error.

number of instructions. The pipe issue rate distributions show the fractions of cycles in which zero instructions are issued and the fraction in which one instruction is issued to each pipe. The total issue rate distribution gives the overall issue rate out of the issue buffer into all of the pipes — the average computed from this distribution is the IPC shown in Figure 5.4. The accuracy of the distribution values is similar to that of the average IPC figures.

5.5 Adding Instruction Addresses

If different static instructions were not allowed to map to the same instruction number, i.e., if instructions had to have identical static instruction addresses in addition to matching types and dependent instruction distances, there would be no aliasing. With the aliasing problem removed, the statistical model would produce results nearly identical to simulation. However, for anything other than the most trivial benchmarks, the statistical flow graphs would be enormous, and the resulting state spaces would be too large to work with.

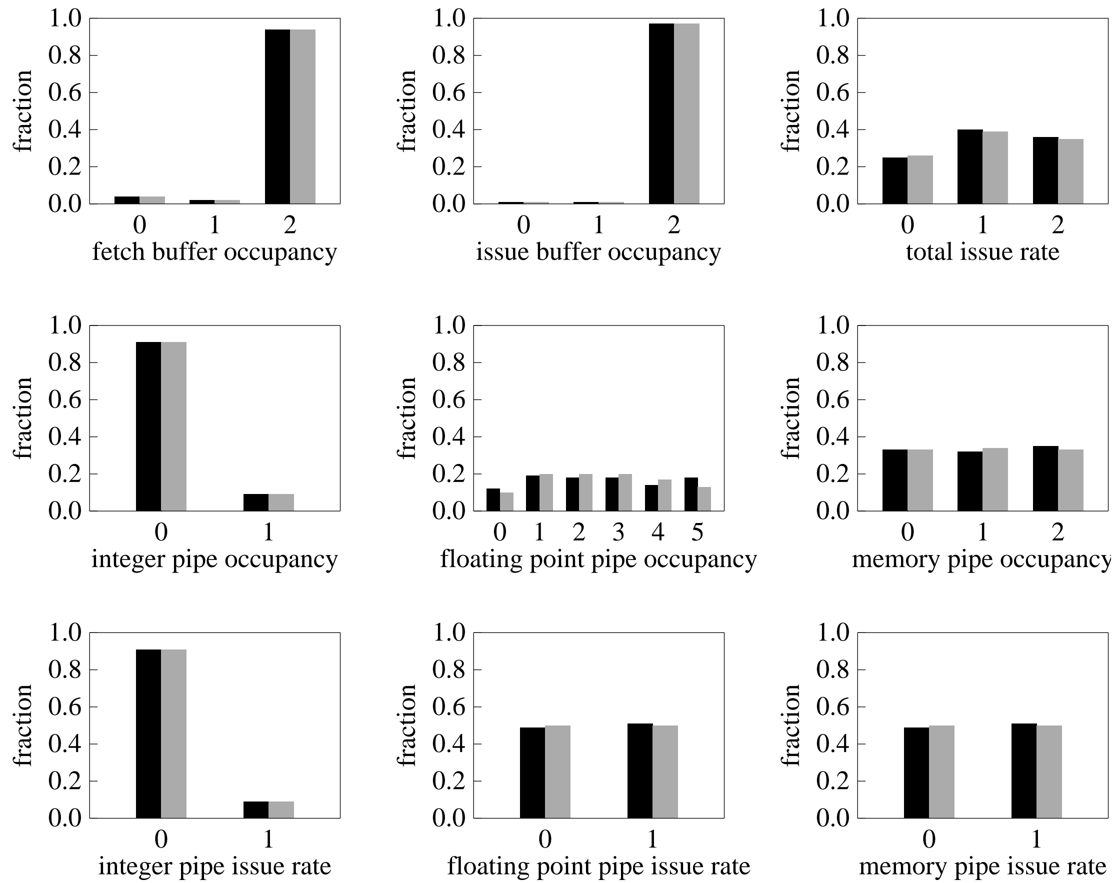


Figure 5.8: Parallelism distributions for several processor parameters for the `fpppp` benchmark with no branch prediction. The black bars are the simulated values and the gray bars are the modeled values.

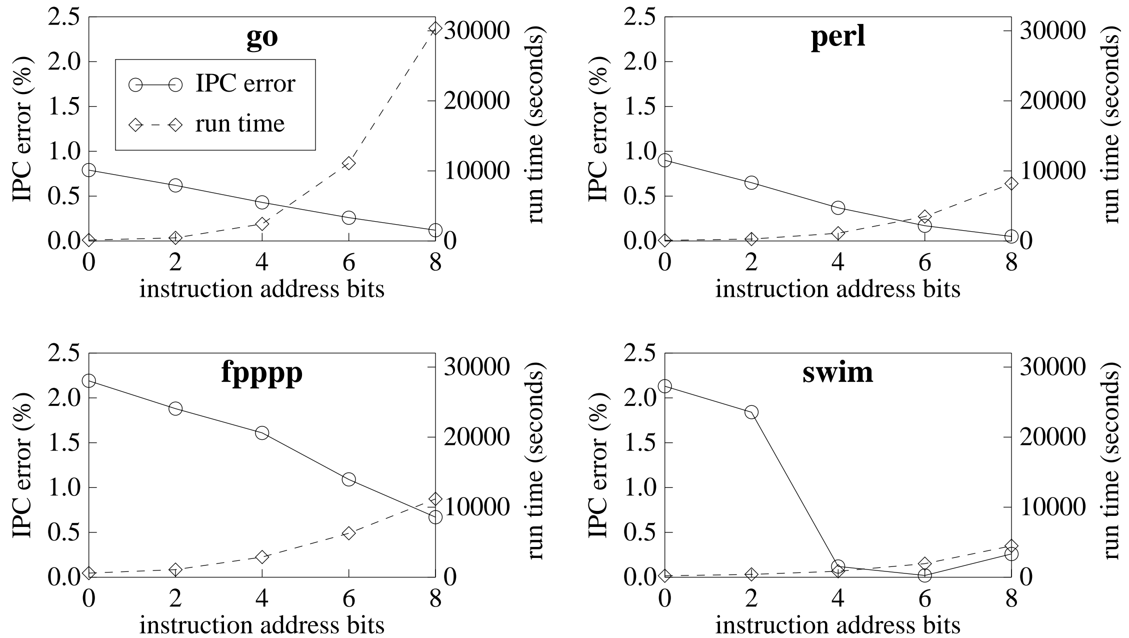


Figure 5.9: The results of adding 2, 4, 6, or 8 low-order instruction address bits to the instruction sequences for the two SPECint and two SPECfp benchmarks with the worst errors. The solid curve shows the errors (modeled IPC compared to simulated IPC) and the dashed curve shows the modeling run times.

While it is not possible to add full instruction address information to the instructions and instruction sequences, it is possible to add partial address information. Specifically, a few bits from the static address of the instruction can be added. That is, a single instruction number is shared by all instructions with the same:

- address bits: the n_{addr} low-order bits of the static address of the instruction;
- instruction type: y; and
- dependent instruction distances, $s^{\text{int}}, s^{\text{fp}}, s^{\text{mem}}$.

This trades an increase in the number of instructions and instruction sequences and a corresponding increase in state space sizes for a reduction of aliasing.

Figure 5.9 shows the results of adding instruction address bits to the instruction sequences. The two SPECint and two SPECfp benchmarks with the highest errors in Figure 5.4 are selected. Each of the four traces are reanalyzed to collect data with instructions sequences augmented with $n_{\text{addr}} = 2$, 4, 6, and 8 address bits. The graphs plot the model error (the decreasing solid curve) and the model run time (the increasing dashed curve) for each benchmark. Note that the model error decreases significantly in each case.

The tradeoffs of adding instruction address bits to the instruction sequences need to be further explored. As can be seen from Figure 5.9, the model run time increases significantly. In addition, the memory usage increases at a similar rate. Several of the benchmarks with larger n_{seq} values could not be modeled with $n_{\text{addr}} > 0$ due to memory constraints. The data in the remainder of this dissertation uses unaugmented instruction sequences, i.e., statistical flow graphs with $n_{\text{addr}} = 0$.

Chapter 6

Modeling Branch Prediction

This chapter describes the addition of branch prediction to the statistical model. The model is extended to handle branch misprediction statistics, in addition to the always-correct and always-incorrect predictions used previously. The trace analyzer is used to collect data for a range of branch predictors for each benchmark.

6.1 Branch Predictors

Four types of branch prediction are modeled:

- **None**: No branch prediction is done. Every branch is considered mispredicted and causes a corresponding delay. This is the "no prediction" model from the previous chapters.
- **Loop**: Backward branches are predicted taken and forward branches are predicted not taken. This is a standard static prediction technique which works well with loop-terminating branches [Smi81].

- **Bimodal**: The processor maintains a table of saturating two-bit counters. The table is indexed by the low-order bits of the branch instruction address. When a branch is encountered in the instruction stream, its counter is examined. If the counter value is 0 or 1, the branch is predicted not taken; if it is 2 or 3, the branch is predicted taken. When the branch is executed, the counter is incremented (saturating at 3) if the branch is taken and decremented (saturating at 0) if the branch is not taken. The size of the table can be varied. This is a standard dynamic prediction technique [Smi81].

- **Perfect**: All branches are correctly predicted. This is an "oracle"-based technique often used in limit studies [LW92]. This is the "perfect prediction" model from previous chapters.

The predictors are listed above in order of expected performance. Other branch predictors can also be modeled using the process described below.

As described earlier, whenever the processor encounters a mispredicted branch, it stalls until the branch is issued, i.e., branches are resolved in the decode/issue stage. This means that there is a one-cycle "bubble" behind a mispredicted branch, during which no instructions are fetched.

6.2 Statistical Modeling

In order to model branch prediction, the state spaces of the fetch and issue buffers are modified. Previously, the fetch buffer state contained enough information to tell if there was a mispredicted branch in fetch. With no branch prediction, if there is a branch in fetch, as indicated by the buffer occupancy (x_F) and instruction types (via the sequence number, seq_F), it is always mispredicted. With perfect prediction, there is never a mispredicted

branch in fetch. To model more general branch prediction techniques, the fetch buffer must keep track of whether any branch it currently holds is mispredicted. For this purpose, a new element is added to the fetch state vector:

- $b_F = 1$ if there is a mispredicted branch in the fetch buffer, 0 if not.

The fetch buffer state's b_I element is used as before to indicate a mispredicted branch in the issue buffer.

The issue buffer state vector is augmented for the same reason:

- $b_I = 1$ if there is a mispredicted branch in the issue buffer, 0 if not.

The pipeline state vectors are not modified.

The trace analyzer is also modified. As a trace is processed, the analyzer simulates the loop predictor and bimodal predictors with several table sizes. Every distinct instruction (as defined in Section 5.1) that is a branch has a set of counters: for each branch predictor, there is a correct prediction counter and misprediction counter. Every time a branch is encountered in the trace, each branch predictor is simulated, and the appropriate counter (based on the instruction number of the branch and whether the prediction is correct or not) is incremented. The misprediction rate (misprediction count divided by total count) associated with a particular instruction, then, is the fraction of all dynamic branches that are mapped to that instruction number and are mispredicted. These misprediction rates are made part of the instrInfo table:

$$\text{instrInfo}[instr].\,\text{mispred}_{bp} = \text{Prob}\left[\text{branch is mispredicted by predictor } bp\right]$$

The mispred fields in the table are only valid for branch instructions. This misprediction rate information is used in building the Markov chain transition matrices for the fetch and issue buffers.

The push out of fetch and pull into fetch are computed as before (see Sections 4.2.2 and 5.2), with the modification of using the b_F flag to check whether there is a mispredicted branch in fetch. As before, if there is a mispredicted branch in fetch, no new instructions are fetched until the branch issues. If an instruction enters fetch, the seqInfo table entries, $\mathrm{seqInfo}[seq_F].\mathrm{next}$ and $\mathrm{seqInfo}[seq_F].\mathrm{prob}$, are checked to find all possible next sequences. Each possible next sequence generates one of two possible next states. If, for a particular next sequence number, the new instruction is a branch, there are two possible next states with that sequence number. One has $b_F = 1$ and probability $\mathrm{seqInfo}[seq].\mathrm{mispred}_{bp}$, the other has $b_F = 0$ and probability $1 - \mathrm{seqInfo}[seq].\mathrm{mispred}_{bp}$ (where bp is the branch predictor in use). These two states correspond to the branch being incorrectly and correctly predicted, respectively. The issue buffer transitions are computed similarly.

6.3 Experimental Results

Figure 6.1 shows the average IPCs computed by the model with each of the four branch predictors. The bimodal branch predictor uses a 512-entry table. The maximum error is 6.9% for the loop predictor and 4.4% with the bimodal predictor, compared to 2.6% with the perfect predictor and 2.2% with no predictor. The average errors are 2.6% (loop) and 1.6% (bimodal), which are also noticeably higher than the previous 0.5% (perfect) and 0.7% (none).

An important observation is that the IPC trend "across" the branch predictors is consistent. That is, for every benchmark:

$$IPC^{\mathrm{sim}}_{\mathrm{none}} \leq IPC^{\mathrm{sim}}_{\mathrm{loop}} \leq IPC^{\mathrm{sim}}_{\mathrm{bimodal}} \leq IPC^{\mathrm{sim}}_{\mathrm{perfect}}$$

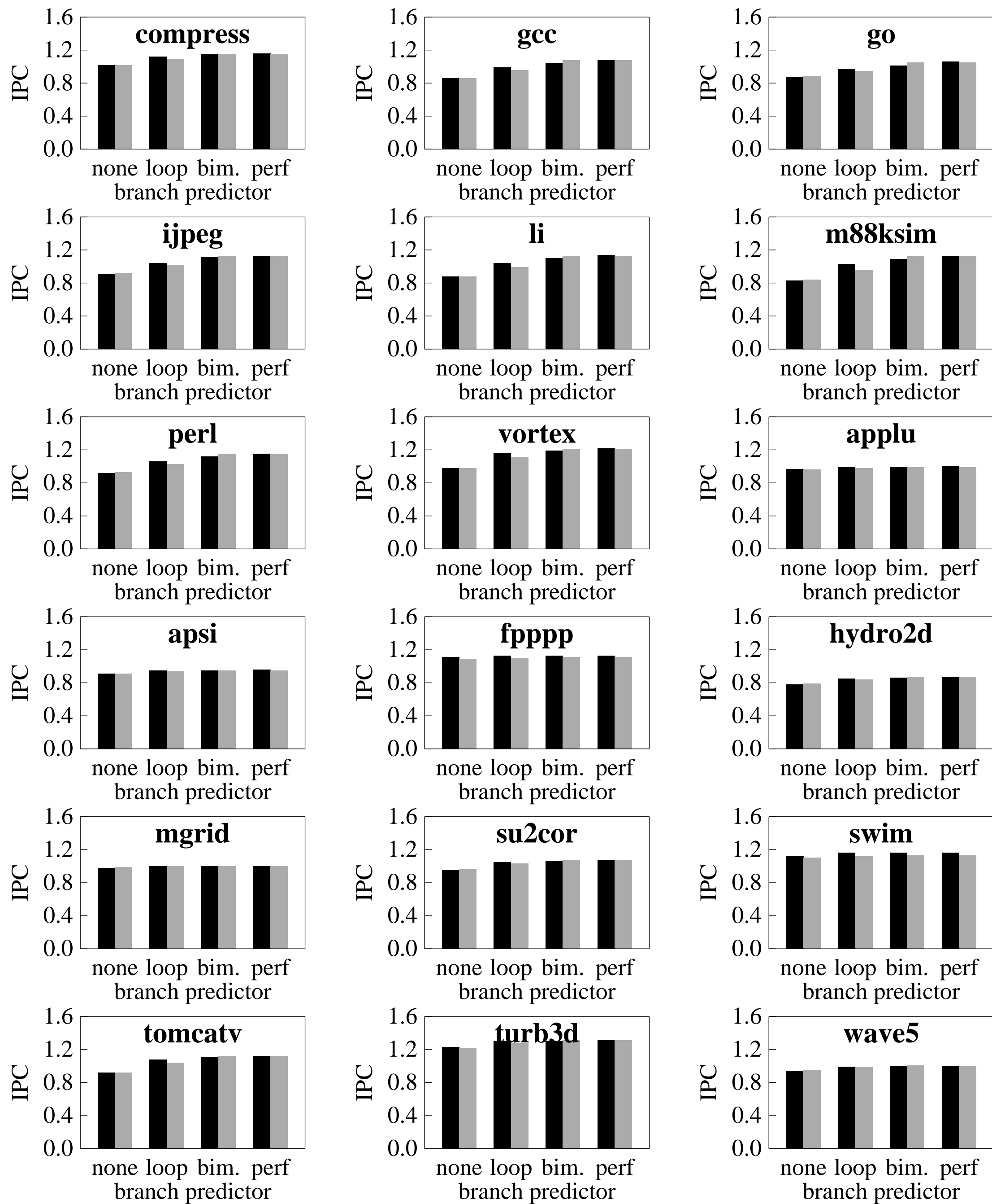


Figure 6.1: The average IPCs for the SPEC95 benchmarks with four different branch predictors, as measured by a simulator (black bars) and computed by the model (gray bars).

and:

$$IPC^{\text{model}}_{\text{none}} \leq IPC^{\text{model}}_{\text{loop}} \leq IPC^{\text{model}}_{\text{bimodal}} \leq IPC^{\text{model}}_{\text{perfect}}$$

It is also interesting to look at the qualitative nature of these trends. In all of the benchmarks, the bimodal branch predictor gets all or nearly all of the potential benefit of branch prediction (i.e., its performance is very close to that of the perfect predictor). The modeled IPCs correctly show this trend. In benchmarks where even perfect branch prediction does not appreciably affect performance (`applu`, `fpppp`, `mgrid`), the model again captures this behavior.

Because the trace analyzer actually simulates each of the branch predictors, the modeled branch misprediction rates are nearly identical to the simulated rates. Incorporating branch misprediction information in the model facilitates analysis of the effect of mispredictions on the IPC, something which is not possible with a branch prediction simulator by itself.

Chapter 7

Modeling Caches

The models presented so far assume an ideal memory system. Instruction fetch never stalls due to a cache miss. Loads and stores always finish with the two-cycle latency expected from the two-stage memory pipeline. This chapter adds an instruction cache and data cache to the statistical model. The technique used is very similar to the branch prediction model in Chapter 6. The trace analyzer collects miss rate information for a range of I-cache and D-cache sizes for each benchmark. The component state spaces are extended to handle cache stall cycles, and the statistical miss rate information is used to compute the state transitions.

7.1 Instruction Cache

The modeled I-cache has the following parameters:

- **Fully associative**. Full associativity simplifies the trace analysis (see the description of stack distance below), however set-associative caches could also be modeled using a similar technique.
- **Size**. The number of cache blocks is adjustable.

cycle	seq_F	b_F	b_I	x_F	$stall_F$	
0	50	0	0	2	0	not stalled
1	51	0	0	2	0	cache hit; a new instruction is fetched
2	52	0	0	1	4	cache miss; new instruction is not yet fetched
3	52	0	0	0	3	the last instruction in fetch moves to issue
4	52	0	0	0	2	
5	52	0	0	0	1	
6	52	0	0	2	0	cache access completes; two instr's are fetched

Table 7.1: An example sequence of fetch buffer states showing the effect of an I-cache miss. Cycles 2–5 are the 4-cycle cache stall.

- **Latency**. The length of the stall for a cache miss is also adjustable.
- **16-byte blocks**. Other block sizes could be modeled if the necessary data were extracted from the traces.

Only the fetch buffer state vector must be changed to model an I-cache. When an I-cache miss occurs, the fetch buffer stalls for a number of cycles (this number is the miss latency parameter). The state vector is augmented with a counter which is used to keep track of stalls:

- $stall_F$ = the number of cycles remaining in an I-cache stall. If $stall_F > 0$, it is a count of the cycles remaining before the current cache access completes and the next instruction enters the fetch buffer. If $stall_F = 0$, there is no currently pending I-cache stall.

A typical sequence of fetch buffer states, with a four-cycle cache latency, is shown in Table 7.1. While the $stall_F$ counter is non-zero, fetching is stalled. On a cache miss, $stall_F$ is set to the miss latency. The $stall_F$ counter is decremented on each cycle. While $stall_F > 0$, any instructions already in fetch can move to issue, but no new instructions enter the fetch buffer.

I-cache miss rates are collected from the trace for several different cache sizes. Trace analysis is described in Section 7.3.

7.2 Data Cache

The data cache is modeled similarly. Like the I-cache, it is fully associative with 16-byte blocks. The size and miss latency are adjustable parameters.

A memory load/store instruction stalls in the first stage of the memory pipe until its cache access completes. An instruction below it in the pipe can continue, but no new instructions can enter the pipe during a cache stall. A stall counter is added to the memory pipeline state vector:

- $stall_{P_{\mathrm{mem}}}$ = the number of cycles remaining in a D-cache stall. If $stall_{P_{\mathrm{mem}}} > 0$, it is a count of the cycles remaining before the current cache access completes and the instruction in the first pipe stage can move. If $stall_{P_{\mathrm{mem}}} = 0$, there is no currently pending D-cache stall.

The $stall_{P_{\mathrm{mem}}}$ counter operates in the same way as the $stall_F$ counter.

D-cache miss rates are collected from the trace as described below.

7.3 Trace Analysis

The concept of **stack distance** [JCSM96] is introduced in Chapter 1. The definition is repeated here with slight modifications to refer to memory blocks rather than words. Memory is divided into blocks the size of cache blocks. The ith reference (load or store instruction)

to memory block A is denoted A_i. Each reference, A_i, has a stack distance:

$$\text{stack distance of } A_i = \text{the number of distinct memory blocks referenced between } A_{i-1} \text{ and } A_i$$

An example is given in Section 1.3.4.

For a fully associative cache with LRU replacement, the stack distance of A_i is the largest cache size for which reference A_i would be a miss. If the cache has this many or fewer blocks, block A will have been replaced by the time reference A_i is made, and A_i will be a miss. If the cache has at least one more block than this, the copy of A loaded by A_{i-1} will still be in the cache, and reference A_i will be a hit. This property applies to instruction caches as well as data caches. For an I-cache, instruction fetches, rather than load/store instructions, are the pertinent memory references.

The trace analyzer measures the stack distance for each load/store instruction and for each instruction fetch. The details of this algorithm are given in Appendix B. Instruction distances are classified into bins, according to a set of "interesting" cache sizes:

- stack distance $< 2^6$
- $2^6 \leq$ stack distance $< 2^7$

  ⋮

- $2^{13} \leq$ stack distance $< 2^{14}$
- $2^{14} \leq$ stack distance

These bins are sufficient to model cache sizes from $2^6 = 64$ blocks (1kB) to $2^{14} = 16$k blocks (256kB).

For each instruction number, the trace analyzer counts the number of instruction fetch stack distances which fall into each bin. This is equivalent to simulating an I-cache of each size and counting the number of hits. The size-c I-cache hit rate for an instruction is given by summing all bins up to bin c and dividing by the total number of references to that instruction. The result is a set of statistical I-cache hit rates, which are added to the instrInfo table:

$$\text{instrInfo}[instr].\text{Ihit}_c = \text{Prob}[\text{instruction would be a hit in an I-cache of size } c]$$

A similar set of statistics is kept for load/store instructions:

$$\text{instrInfo}[instr].\text{Dhit}_c = \text{Prob}[\text{instruction would cause a hit in a D-cache of size } c]$$

Note that the trace analyzer keeps track of hits and misses caused by a particular load/store *instruction*, not hits and misses to a particular *data address*.

The statistical miss rate described here is very similar in nature to the branch misprediction rates of Chapter 6. For branch predictors, a misprediction rate is kept for each predictor for each instruction. For caches, a hit rate is kept for each cache size for each instruction. Like branch predictors, this technique requires collecting data for every cache size to be modeled.

7.4 Experimental Results

Figure 7.1 shows the average IPCs computed by the model with 4kB, 8kB, 16kB, and perfect instruction and data caches. (The I-cache and D-cache are the same size.) The miss latency is set to four cycles. The perfect cache results are the same numbers reported in Chapter 6. The 512-entry bimodal branch predictor is used in all of the runs.

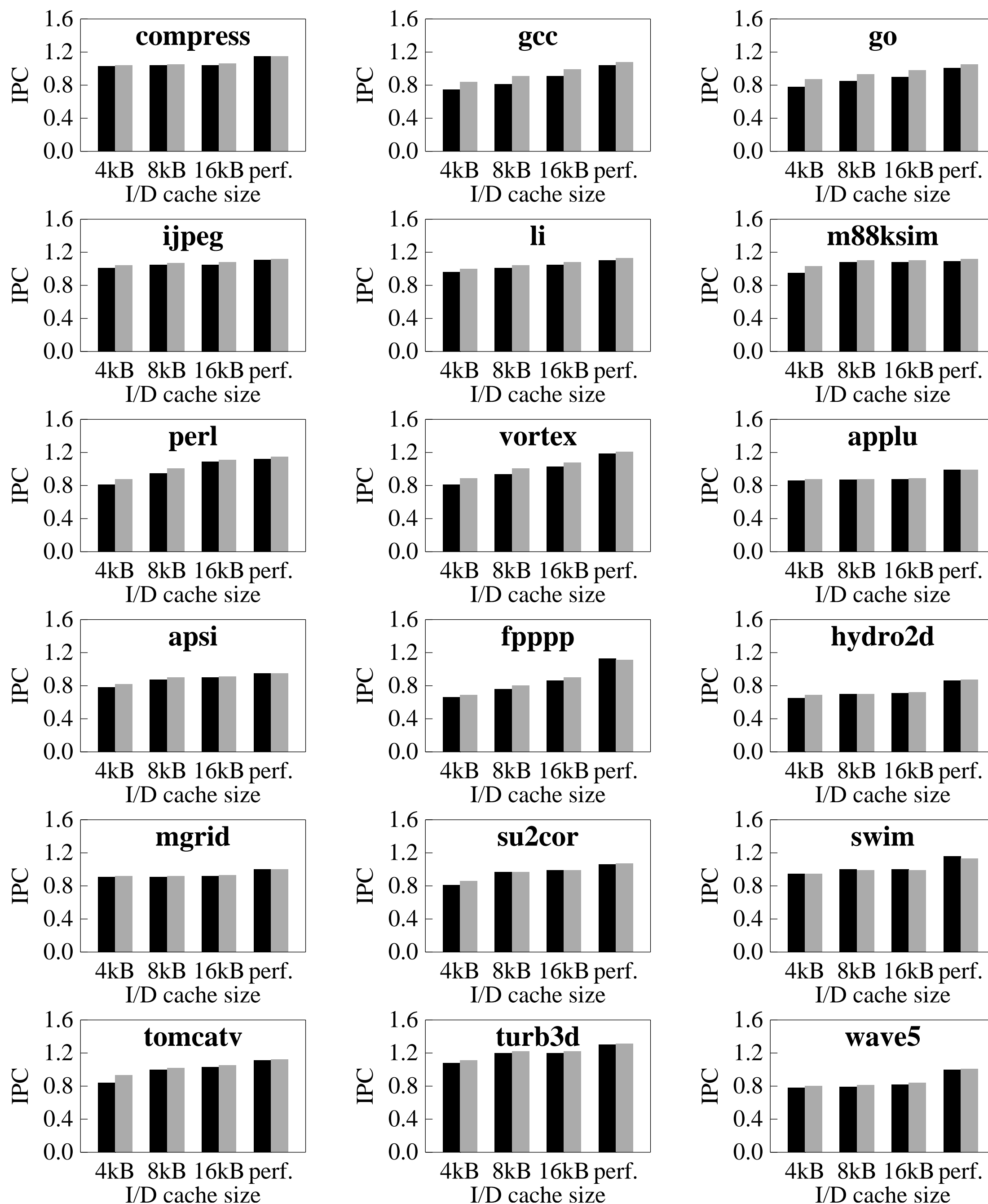


Figure 7.1: The average IPCs for the SPEC95 benchmarks with 4kB, 8kB, 16kB, and perfect instruction and data caches, as measured by a simulator (black bars) and computed by the model (gray bars).

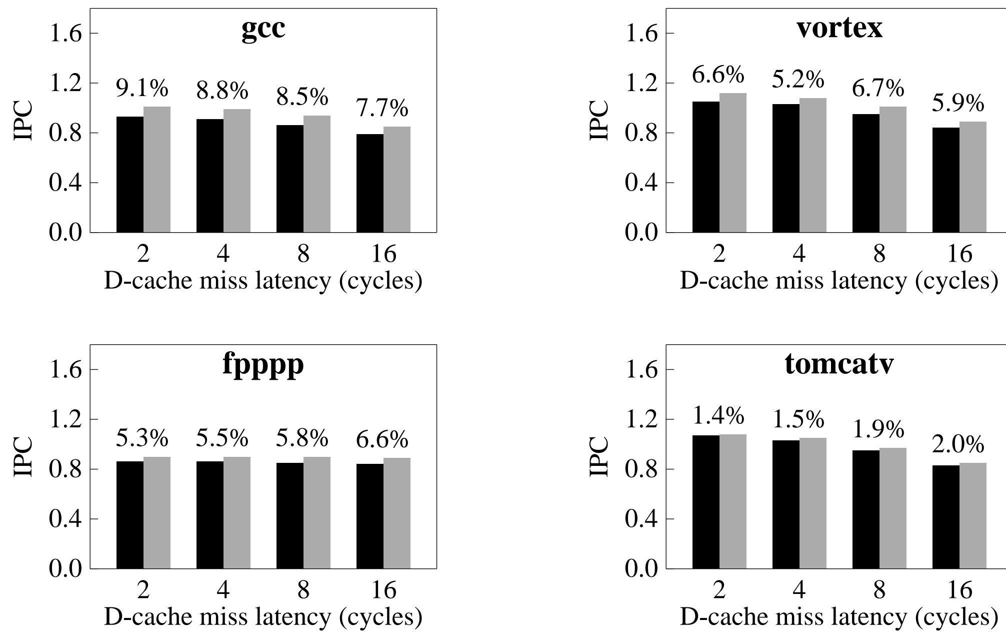


Figure 7.2: The average IPCs for the SPEC95 benchmarks with 16kB caches and D-cache miss latencies of 2, 4, 8, and 16 cycles, as measured by a simulator (black bars) and computed by the model (gray bars). The figure above each pair of bars is the percent difference between the two IPC values.

The maximum errors with 4kB, 8kB, and 16kB caches are 12.6%, 12.1%, and 8.8%. The average errors are 5.6%, 3.5%, and 2.7%. The model error rate is related to the miss rate, and thus to the cache size. This is another type of aliasing effect. For a given D-cache size, each load/store instruction has a certain miss rate, given by the instrInfo table extracted from the trace. The assumption is that the misses are uniformly distributed over all instances of that instruction number. In general, however, the likelihood of a miss is related to other factors, e.g., the preceding instruction sequence. Adding instruction address bits to produce more "specific" instruction numbers (see Section 5.5) would probably improve the accuracy of the cache models.

Size is not the only adjustable cache parameter. Figure 7.2 shows the effect of varying the D-cache miss latency on two SPECint and two SPECfp benchmarks. Note that changing

the cache latency does not significantly affect the accuracy of the model.

As with the branch prediction model, the trace analyzer effectively simulates caches of each possible size. Thus the cache miss rates from the simulator and statistical model are virtually identical. The benefit of adding caches to the statistical model is not in measuring miss rates, but in measuring their effect on the IPC. It is also important to note that, while the trace analyzer must simulate each branch predictor and caches of each size, the branch predictor, I-cache, and D-cache simulations are independent. The interactions between these three are handled entirely by the statistical model and its transition matrices.

Chapter 8

Modeling Value Prediction

This chapter shows how the statistical model can be extended to support value prediction [LS96]. Chapters 6 and 7 add branch prediction and caches — standard, well-understood microarchitectural features — to the statistical model. In contrast, value prediction is a new, experimental feature. Value prediction attempts to bypass the dataflow limit by predicting the results of instructions before they execute. While value prediction is much more effective on wider machines than on the two-issue processor used here, the results presented in this chapter indicate that the statistical model can be effectively used to model prototype microarchitectural features.

8.1 Value Prediction

This chapter adds a simplified version of value prediction, as described by Lipasti and Shen [LS96], to the statistical model. Value prediction requires that two tables be added to the processor. Both tables are indexed by the low-order bits of the PC, similar to the bimodal branch predictor described in Chapter 6, except that there is an entry for every instruction. The value prediction table (VPT) entry for an instruction is a prediction of

the result of that instruction. The classification table (CT) contains a saturating two-bit counter for each instruction. If the CT counter value is 2 or 3, the instruction's result is predicted, and the predicted value is used by any subsequent instructions which depend on this instruction. If the CT counter value is 0 or 1, the result is not predicted, and dependent instructions have to wait for the computed result as usual. If a prediction is made and turns out to be incorrect, the dependent instructions must wait for the result and then reissue with an additional one-cycle penalty. The CT counters are incremented on correct value predictions and decremented on incorrect predictions. The VPT entry is updated after an incorrect prediction, but only if the CT counter is at 0, 1, or 2.

8.2 Statistical Modeling

The trace analyzer simulates value prediction for several different VPT and CT sizes, just as it does with branch predictors and caches. However, instead of adding misprediction rate statistics to the instrInfo table, the instructions themselves are modified. A value prediction flag for each dependence type is added, so that each instruction now consists of:

- y: the instruction type.
- $s^{\text{int}}, s^{\text{fp}}, s^{\text{mem}}$: the dependent instruction distances.
- $v^{\text{int}}, v^{\text{fp}}, v^{\text{mem}}$: the value prediction flags.

The v flags can take on values of "unpredicted", "correctly predicted", or "mispredicted". The combination of s^t and v^t describes instruction i's most recent dependence on a type-t instruction. For example, if $s^{\text{fp}} = 2$ and $v^{\text{fp}} =$ "correctly predicted", then the instruction is dependent on the second previous floating point instruction, and the result of that instruction is correctly predicted.

Adding the value prediction flags increases the number of instructions and instruction sequences in a benchmark's statistical flow graph. The trace analyzer must simulate VPTs and CTs of several different sizes and construct a separate statistical flow graph for each size.

The pipeline state vectors are not modified, aside from having more instruction sequences. The transition matrix computation is, however, affected by the value prediction flags in the seqInfo table. An instruction whose inputs are all computed or correctly predicted is allowed to issue the cycle immediately following the cycle in which its last correctly predicted source issued. An instruction with one or more incorrectly predicted sources must wait until the last result is produced, plus the one-cycle penalty mentioned above. Unpredicted sources are treated as usual: the instruction must wait for the results to be produced.

8.3 Experimental Results

Figure 8.1 shows the average IPCs computed by the model with 128-entry, 256-entry, and 512-entry tables. (The VPT and CT are the same size.) The maximum errors are 13.0%, 13.0%, and 12.4%. The average errors are 4.9%, 5.0%, and 5.1%.

The statistical model appears to match the trends reasonably well, at least for those benchmarks where value prediction makes any difference in performance. For example, the performance on `hydro2d` increases significantly with the 128-entry table, but larger table sizes have essentially no effect. The statistical model matches this behavior, although it overestimates the gain due to the 128-entry table. Value prediction is expected to produce a much larger performance improvement on wider processors. A more thorough analysis of statistical modeling of value prediction will have to wait for a statistical model that can handle these wider processors.

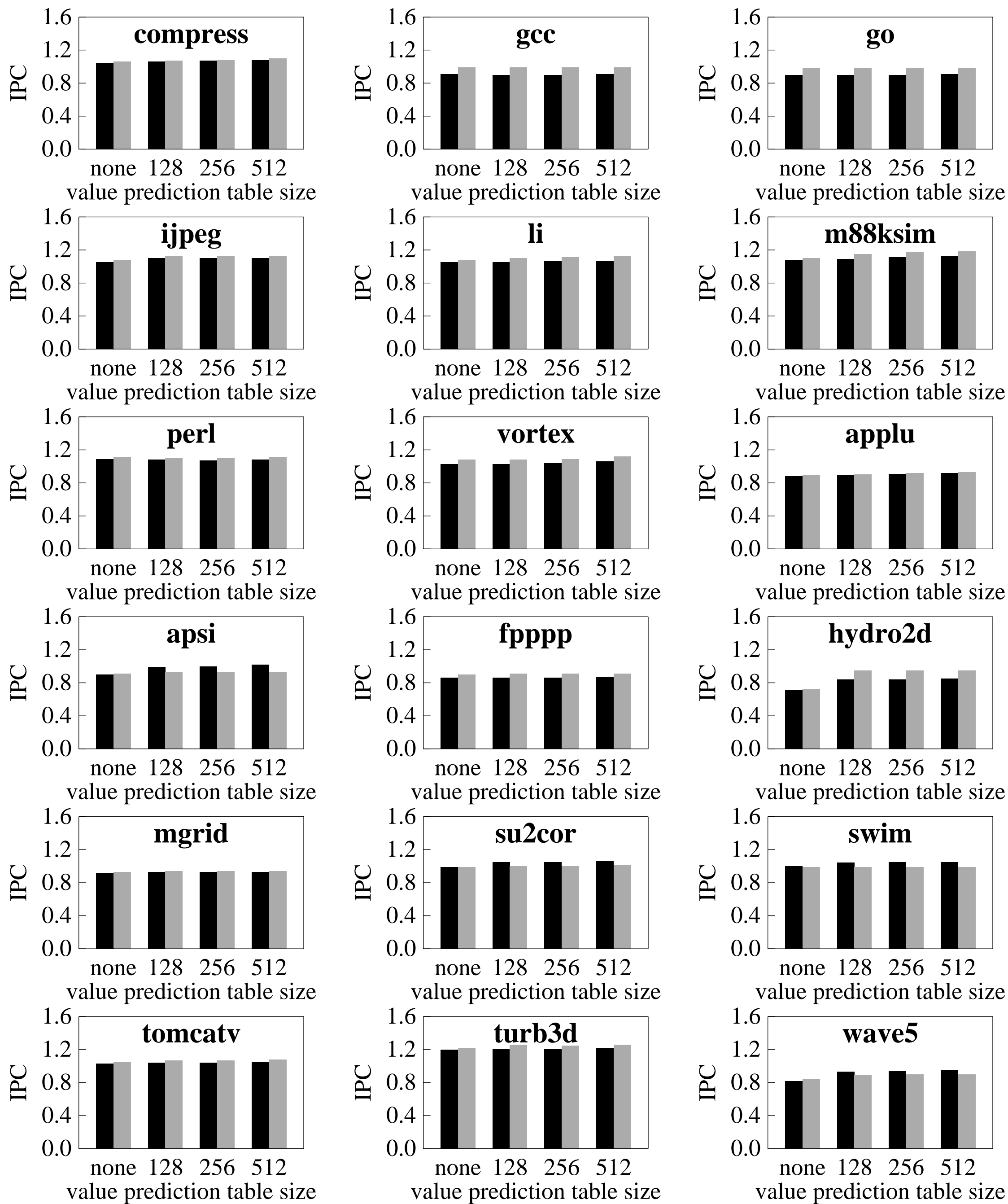


Figure 8.1: The average IPCs for the SPEC95 benchmarks with 128, 256, and 512-entry VPT and CT, as measured by a simulator (black bars) and computed by the model (gray bars).

Chapter 9

Conclusions

This chapter presents a summary of the contributions made by this dissertation. This is followed by a description of the limitations of the statistical modeling approach and some interesting areas for future research.

9.1 Contributions

Statistical Modeling Framework The key contribution of this research is a framework for statistical processor performance modeling. In contrast to a simulation-based model, a statistical model allows a much faster exploration of a large design space. The framework presented here incorporates several concepts.

Parallelism Distributions The parallelism distribution is used as the basis for the statistical modeling techniques. Traditional performance models work with single average values, e.g., the average issue rate in instructions per cycle (IPC). The models presented here instead use a probability distribution vector. This vector consists of the probability of issuing each possible number of instructions, or equivalently, the fraction of cycles in which any

given number of instructions issue. The benefit of using parallelism distributions appears when the parallelism is not smooth. A parallelism distribution can show a microarchitect where more resources are needed, e.g., more issue slots than the average IPC alone would indicate.

Separate Program and Machine Models As shown in Figure 1.3, the statistical modeling approach explicitly separates the program model from the machine model. This has a fundamental impact on modeling. The time-consuming part of the modeling process — the part which takes time proportional to the length of the benchmark traces — is done only once per benchmark. Enough information is extracted from the trace to allow modeling of a large family of processors. In contrast, a simulator must reread the trace (or regenerate it via execution-driven simulation) for every run, i.e., for every machine model.

Markov Chains Markov chains are a well-known statistical technique. This dissertation shows that they can be applied to superscalar processor performance modeling. The key challenge is to design a state space such that the Markov chain is small enough to solve, but detailed enough to generate accurate performance information.

Partitioned Markov Chains In order to reduce the size of the state space (and thus the Markov chain transition matrix), the single, monolithic Markov chain for the processor is partitioned. Each new Markov chain models just one component of the processor. This dissertation shows how the separate state spaces can be overlapped to account for the dependences between the components of the processor. An iterative relaxation technique is used to obtain a simultaneous solution for the multiple, overlapping Markov chains.

Statistical Flow Graph The statistical flow graph is a compact statistical representation for program traces. The nodes represent sequences of similar instructions and the arcs

represent the ordering of those sequences within the trace. A statistical flow graph can also incorporate branch prediction and cache hit rate statistics.

Adding Microarchitectural Features Finally, several microarchitectural features — branch prediction, data and instruction caches, and value prediction — are added to the processor model. This demonstrates that statistical modeling can accommodate processors more complex than just a simple pipeline.

9.2 Summary of Results

The partitioned Markov chain-based model was used to predict the performance of the 3-pipeline processor running the SPEC95 benchmark suite. The statistical model results have an average error of 3.5% compared to the simulated performance. Using the statistical flow graph, the average error is reduced to 0.5%.

Adding microarchitectural features increases the modeled performance error. With a bimodal branch predictor, the average error is 1.6%. Adding 16kB instruction and data caches increases the average error to 2.7%. Finally, adding a 512-entry value predictor increases the average error to 5.1%.

These errors are low enough to indicate that this model could be useful to computer architects. In addition, the statistical model qualitatively matches the simulation results in showing trends across different branch predictors and cache sizes. The increase in errors as features are added is due to the statistical assumptions in the branch predictor, cache, and value predictor models. The branch/cache/value miss/misprediction rates are averaged over each instruction number, which introduces error into these models.

processor	SFG size	fetch states		fp pipe states		modeling time (s)	sim. time (s)
		unpruned	pruned	unpruned	pruned		
3-pipe	1654	19848	1930	52928	37370	493	724
+ branch pred.	1654	19848	1654	52928	37450	423	626
+ caches	1654	99240	14312	52928	42046	778	1108
+ value pred.	4114	246840	35379	131648	99830	3677	1131

Table 9.1: The statistical flow graph size (number of sequence nodes), sizes of the fetch and floating point pipe state spaces, and modeling and simulation run times for the `apsi` benchmark as various processor features are added.

9.3 Limitations

The examples presented in this dissertation show how statistical modeling techniques can be applied to relatively simple superscalar processors. It is not clear that these techniques can be directly applied to state-of-the-art processors. Modeling processors which are more complex than those shown here will likely require significant additional research on statistical modeling techniques. Some ideas for this future research are given in Section 9.4.

Table 9.1 shows several characteristics of the statistical model as various features are added to the processor. The `apsi` benchmark is used as a worst-case example because it has the largest statistical flow graph, and thus the largest state spaces. The first column describes the processor being modeled. Each line adds another processor feature. The second column is the size of the statistical flow graph. Value prediction is modeled by adding nodes to the statistical flow graph; for the other processors, the size stays the same. The next four columns show the state space sizes for two components, the fetch buffer and the floating point pipe. For each component, the state space size is shown before and after pruning. Pruning removes states which are not reached, or are reached only by transitions with probabilities below a certain threshold (set to 10^{-10}). The last two columns show the run time of the statistical model and simulator, respectively. Note that the floating point

pipe Markov chain for the most complex processor has just under 100,000 states. Markov chains of this size are essentially the largest that can be handled by the software used in this research. This means that changes which tend to further increase state space sizes, e.g., increasing cache miss latencies or adding instruction address bits to the statistical flow graph nodes, are impractical. However, adding new Markov chains, further partitioning the model, is a viable alternative.

Wider machines also present a problem. The most complex processor modeled in this dissertation was a two-issue machine. Current processors can issue four and even six instructions. Modeling these processors would require enlarging the state spaces, either by using longer instruction sequences (which increases the size of the statistical flow graph, and thus the state spaces) or by having multiple sequence numbers in the state vectors.

The statistical flow graph cannot be used, at least without major changes, to model out-of-order processors. Because instructions can be arbitrarily reordered inside an out-of-order processor, the notion of instruction sequences as n adjacent instructions will not suffice. Section 9.4 presents some ideas for getting around this problem.

A question that comes up when considering the practical application of statistical modeling is ease of use. While the core of a statistical model is similar in many ways to a simulator, it is clearly somewhat harder to implement and debug. Debugging a simulator involves single-stepping through a trace and examining the processor state at each cycle. A statistical model does not provide this option. Instead, the user must run the model over a trace consisting mostly of iterations of a single small loop, and then examine the resulting states and transition matrices. This is more difficult, but certainly not impossible. Also, during the processor design process, it is desirable to track the actual RTL-level design with a performance model. A statistical model would be more difficult to keep up-to-date with the RTL model than would be a simulation-based model. Future improvements may help to make statistical models easier to use.

9.4 Future Work

There are at least three directions in which further research can proceed from the work presented here. One is analyzing the modeling error, i.e., the difference between the modeled and simulated performance numbers, and investigating ways of increasing the model's accuracy. The second is experimenting with statistical models of more complex processors, as described in the previous section. Third, there are many other potential uses for the statistical modeling techniques introduced in this dissertation.

9.4.1 Increasing Accuracy.

There is some discussion in Chapter 5 of adding instruction address bits to the instruction sequences. The results indicate that this can significantly improve the accuracy of the model, but at the cost of a correspondingly significant increase in run time and memory usage. One possible approach to this problem involves using something other than the lowest-order bits from the instruction address. Another possibility is to use the instruction address, or part of it, as the sequence number, and to obtain the instruction type and dependent instruction distance information from a table of probabilities indexed by instruction address. This effectively moves the statistical model's state space even closer to a simulator's state space. However, as shown by the run times in Figure 5.9, any increase in the size of the statistical flow graphs may be impractical.

Another interesting research area is trace "splitting". It may be the case that some benchmark traces can be split into several sections, with consistent behavior within a section, but relatively different behavior among different sections. If the changes in behavior can be detected automatically by the trace analyzer, it could produce a separate statistical flow graph for each section of the trace. These sections would be modeled essentially as separate programs, thus reducing the aliasing problem. In addition, this might result in

somewhat smaller statistical flow graphs.

9.4.2 Increasing Processor Complexity.

Section 9.3 discussed the problems inherent in statistical modeling of more complex processors. The main problem will clearly be designing models that have small enough state spaces to be practical, which still accurately predicting the performance of complex superscalar processors.

One potential way of decreasing the statistical flow graph size is pruning of the graph itself. Subgraphs with low occurrence probabilities can be removed entirely. This can be done during trace analysis, thus reducing the memory and time required by the modeler. The effect of this pruning on the prediction accuracy would need to be studied.

One significant feature missing from the processors modeled here is out-of-order issue. One possible approach is to abandon the statistical flow graph altogether. Instead of keeping track of a sequence number, each state vector could include several instruction numbers. With in-order issue, the net result *after state space pruning* would be the same, i.e., the statistical flow graph contains all pairs of instruction numbers which actually occur in the trace, in effect doing part of the state space pruning during the trace analysis phase. With out-of-order issue, however, the state space sizes would likely blow up. Because instructions can be reordered arbitrarily within the processor, the number of possible combinations of instruction numbers grows exponentially with the number of instruction numbers.

Another approach to this problem is to look at a different sort of statistical flow graph. Instead of grouping pairs of instructions which are adjacent in the trace, the graph could group data dependence chains, i.e., sequences of instructions which are each data dependent on the previous one. Since these instructions cannot be reordered without violating data

dependences, it may be possible to use these sequence numbers as state vector elements.

Another shortcoming of the models presented here is that the program parameters are not completely independent of the machine model. Some characteristics of the processor family to be modeled, e.g., instruction types, determine the form of the data extracted by the trace analyzer. It should be possible to divide the instructions into a larger number of narrower classes and provide a means for combining classes as necessary for the processor being modeled.

The trace analyzer currently has to simulate all possible branch predictors and all possible caches. Statistical models of branch prediction and caches would allow the trace analyzer to extract a more abstract set of information from the trace. It may be possible, for example, to invent something similar to dependent instruction distance for branches, i.e., for control dependences instead of data dependences.

9.4.3 Other Uses

The full utility of statistical modeling is not explored in this dissertation. The performance results from the model can be used to pinpoint the component or components causing a performance bottleneck. Parallelism distributions can be used to find resources which need to be increased in size or which can be reduced in size. The model could be used to find areas of the state space which are not well exercised by benchmarks, and which might contain unexpected performance problems. It may be possible to obtain information about a processor by characterizing the Markov chain transition matrices directly, before computing the stationary distributions.

Another direction that is not explored here is evaluation of software. Statistical modeling could be useful to software developers for optimizing software, and also to compiler designers interested in the effects of various optimizations.

The statistical flow graph itself may be an interesting tool. The statistical flow graph that is extracted from a trace characterizes the program sufficiently well to serve as input to an accurate statistical machine model. It would be interesting to explore the information present in this graph. For example, it may be possible to generate a trace from a statistical flow graph, in effect reversing the trace analysis process. Synthetic traces generated in this way might be used in the same way that sampled traces are used: to facilitate much faster simulation of long benchmarks. It might also be possible to combine statistical flow graphs, allowing the generation of a synthetic trace that somehow combines the characteristics of several benchmarks.

Appendix A

Markov Model Mathematics

This appendix describes the numerical methods used to solve individual Markov chains and the system of partitioned Markov chains.

A.1 Markov Chains

The transition matrix, **P**, contains an element for each possible state transition, i.e.,

$$\mathbf{P}_{ij} = \mathrm{Prob}\left[X_{t+1} = j \mid X_t = i\right]$$

For the Markov chains used here, **P** is very sparse, and a matrix representation which stores only non-zero elements is used.

The stationary distribution is the state distribution **x** which satisfies:

$$\mathbf{P}^T \mathbf{x} = \mathbf{x}$$

This equation can be rewritten:

$$\mathbf{Qx} = \mathbf{0}$$

with

$$\mathbf{Q} = \mathbf{P}^T - \mathbf{I}$$

This equation may be solved iteratively using the Jacobi method [Ste94]. The **Q** matrix is first split into a diagonal matrix **D** and strictly lower and upper triangular matrices **L** and **U** such that:

$$\mathbf{Q} = \mathbf{D} - (\mathbf{L} + \mathbf{U})$$

The stationary distribution equation becomes:

$$\begin{aligned}
[\mathbf{D} - (\mathbf{L} + \mathbf{U})]\,\mathbf{x} &= \mathbf{x} \\
\mathbf{Dx} &= (\mathbf{L} + \mathbf{U})\,\mathbf{x} \\
\mathbf{x} &= \mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})\,\mathbf{x}
\end{aligned}$$

The last equation can be iterated:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})\,\mathbf{x}^{(k)}$$

where $\mathbf{x}^{(k)}$ represents the kth approximation of **x**. The individual row computations look

like this:

$$\mathbf{x}_i^{(k+1)} = \frac{1}{\mathbf{D}_{ii}} \left[\sum_{j \neq i} \left(\mathbf{L}_{ij} + \mathbf{U}_{ij} \right) \mathbf{x}_j^{(k)} \right] \qquad \text{for } 1 \leq i \leq n$$

The Gauss-Seidel method [Ste94] is similar to the Jacobi method, but uses already-generated $\mathbf{x}^{(k+1)}$ values in the iteration:

$$\mathbf{x}_i^{(k+1)} = \frac{1}{\mathbf{D}_{ii}} \left[\sum_{j=1}^{i-1} \mathbf{L}_{ij} \mathbf{x}_j^{(k+1)} + \sum_{j=i+1}^{n} \mathbf{U}_{ij} \mathbf{x}_j^{(k)} \right] \qquad \text{for } 1 \leq i \leq n$$

More details can be found in Stewart [Ste94]. The Gauss-Seidel method is used here to solve the individual Markov chains.

A.2 Partitioned Markov Chains

An iterative relaxation technique is used to solve the system of partitioned Markov chains. Figure A.1 shows the process for the three-pipe model of Chapter 4. The model is split into five separate Markov chains. Each Markov chain is solved for its standard distribution, from which the push-out and pull-in probabilities are computed. In the next iteration, each component uses the push-out probabilities from its predecessor and the pull-in probabilities from its successor(s). In addition, the pipes need the pull-in probabilities for the other pipes, as explained in Chapter 4. In each iteration, a new transition matrix is constructed for each component, using the push/pull results from the previous iteration, and a new stationary distribution is computed. This process is repeated until the push and pull probabilities converge.

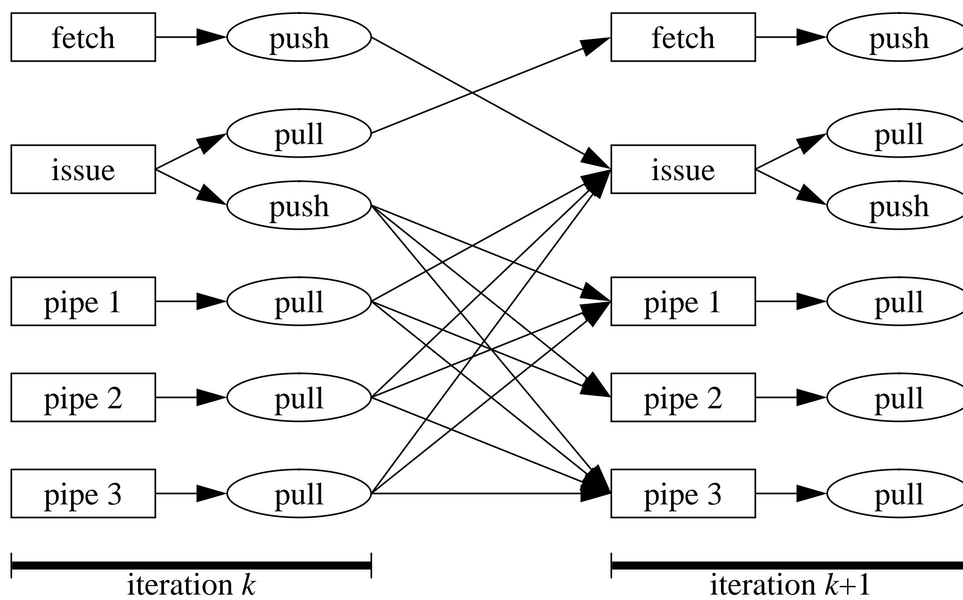


Figure A.1: The system of partitioned Markov chains is solved iteratively. Each of the five component Markov chains is solved, resulting in a set of push-out probabilities (if there is a successor component) and a set of pull-in probabilities (if there is a predecessor). These push and pull probabilities are used as the inputs to the next iteration.

Appendix B

Implementation

This appendix gives a detailed description of the code used to construct the transition matrices for the three-pipe processor model described in Chapter 5 with the branch predictor and cache models described in Chapters 6 and 7, respectively.

B.1 Overview

Each transition matrix is constructed by considering each state, in turn, as the initial state, and considering all possible "actions" that can occur from that state. Each action has an associated probability. Each action causes a transition to some state. The action probability is added to the appropriate element in the transition matrix. The basic algorithm looks like this:

```
set transition matrix P to all zeros
for each state x
  for each action a
    p := Prob[action a | current state is x]
    y := new state resulting from action a
    P[x,y] := P[x,y] + p
```

Most of the potential transition have probability zero, so the transition matrices use a sparse representation that only stores non-zero elements. The algorithm is modified to account for this:

```
set transition matrix P to all zeros
for each state x
  for each action a
    p := Prob[action a | current state is x]
    if (p != 0)
      y := new state resulting from action a
      P[x,y] := P[x,y] + p
```

For clarity, this zero-checking is omitted from the code presented below.

The following sections give the detailed algorithms for each component type: the fetch buffer, issue buffer, and pipelines.

B.2 Fetch Buffer

The following parameters are used by the fetch buffer model:

- `brPred` = branch predictor — one of ‘none’, ‘loop’, ‘bimodal’, ‘perfect’.
- `iCacheSize` = the size of the fully associative I-cache, in blocks; -1 for perfect cache
- `iCacheStall` = the number of stall cycles on an I-cache miss

The fetch buffer state consists of the following elements:

- `seqf` = the sequence number of the sequence whose first instruction will next leave fetch.
- `bf` = 1 if there is a mispredicted branch in the fetch buffer, 0 if not.

- `bi` = 1 if there is a mispredicted branch in the issue buffer, 0 if not.
- `xf` = the number of instructions in the fetch buffer.
- `stallf` = the number of cycles remaining in an I-cache stall.

The transition matrix is computed as follows. In addition to computing the transition matrix, this code also computes per-state push-out probabilities, which are combined later with the standard distribution to compute the component push-out probabilities.

```
// initial state
for each state1 = (seqf1, bf1, bi1, xf1, stallf1)

  // push out to issue
  pOut := xf1
  statePushOut[state1, seqf1, bi1, pOut] := 1

  // pull out to issue
  for qOut := 0 .. 2

    // Prob[qOut]
    p1 := issue.pullIn[seqf1, bi1, qOut]

    // flow out to issue
    fOut := min{pOut, qOut}

    // new bi value
    if bi1
      // mispred branch in issue -- does it stay there?
      bi2 := (qOut > 0) ? 0 : 1
    else if bf1 & fOut=xf1
      // mispred branch in fetch -- does it flow to issue?
      bi2 := 1
    else
      bi2 := 0

    // empty slots in fetch
    if (bf1 | bi2)
      // mispred branch in fetch/issue
```

```
    ef := 0
  else
    ef := 2 - (xf1 - fOut)

  // push into fetch is always 2
  pIn := 2

  // for each possible pair of instr seqs
  for nextSeq1Idx := 0 .. seqf1.nnext-1
    nextSeq1 := seqf1.next[nextSeq1Idx]
    nextSeq1Prob := seqf1.prob[nextSeq1Idx]
    for nextSeq2Idx := 0 .. nextSeq1.nnext-1
      nextSeq2 := nextSeq1.next[nextSeq2Idx]
      nextSeq2Prob := nextSeq1.prob[nextSeq2Idx]

      // Prob[nextSeq1, nextSeq2]
      p2 := nextSeq1Prob * nextSeq2Prob

      // new seqf value
      if fOut = 0
        seqf2 := seqf1
      else if fOut = 1
        seqf2 := nextSeq1
      else
        seqf2 := nextSeq2

      // concatenate InstrInfos
      info[0] := seqf1.instr[0]
      info[1] := seqf1.instr[1]     // = nextSeq1.instr[0]
      info[2] := nextSeq2.instr[0]  // = nextSeq1.instr[1]
      info[3] := nextSeq2.instr[1]

      // p3a = Prob[instrs up to here are not mispred
      //            branches]
      // (running product over mispred loop)
      p3a := 1

      // each possible mispred branch position
      for i := 0 .. 3
        bf[i] := 0
      if bf1
        bf[xf1-1] := 1
```

```
for mispred := xf1 .. xf1+ef

  // mispred = xf1+ef means no new mispred branch
  if mispred = xf1+ef
    p3 := p3a
    qIn1 := ef

  // mispred branch at specific index
  else

    // set mispred branch flag
    bf[mispred] := 1

    // Prob[mispred]
    if info[mispred].y != 3  // not a branch
      p3 := 0
    else if brPred = perfect
      p3 := 0
    else if brPred = none
      p3 := p3a
      p3a := 0
    else // loop or bimodal
      p3 := p3a * info[mispred].mispred[brPred]
      p3a := p3a *
             (1 - info[mispred].mispred[brPred])

    // max pull in
    qIn1 := mispred - xf1 + 1

  // p4a = Prob[cache hits for 0 .. qIn-1]
  // (running product over qIn loop)
  p4a := 1

  // pull in
  for qIn := 0 .. qIn1

    // compute Prob[qIn] and stallf2

    // perfect cache ==> qIn=qIn1
    if (cacheSize < 0)
      if (qIn = qIn1)
        p4 := 1
```

```
  else
    p4 := 0
  stallf2 := stallf1

// currently stalled for cache miss ==> qIn=0
else if stallf1 > 1
  if (qIn = 0)
    p4 := 1
  else
    p4 := 0
  stallf2 := stallf1 - 1

// cache miss just completed ==> first instr
// must now be a hit, i.e., qIn > 0
else if stallf1 = 1
  if qIn = qIn1
    // all pulled instrs are hits
    p4 := p4a
    stallf2 := 0
  else if qIn = 0
    // first instr must be a hit, so qIn != 0
    p4 := 0
    stallf2 := 0
  else
    // p4 := p4a * Prob[next instr is a miss]
    // p4a := p4a * Prob[next instr is a hit]
    p4 := p4a *
          (1 - info[xf1+qIn].iHit[iCacheSize])
    p4a := p4a * info[xf1+qIn].iHit[iCacheSize]
    stallf2 := iCacheStall

// otherwise ==> any instr may be a miss
else
  if qIn = qIn1
    // all pulled instrs are hits
    p4 := p4a
    stallf2 := 0
  else
    // p4 := p4a * Prob[next instr is a miss]
    // p4a := p4a * Prob[next instr is a hit]
    p4 := p4a *
          (1 - info[xf1+qIn].iHit[iCacheSize])
```

```
            p4a := p4a * info[xf1+qIn].iHit[iCacheSize]
            stallf2 := iCacheStall

        // flow in
        fIn := qIn

        // new xf value
        xf2 := xf1 - fOut + fIn

        // new bf value
        if xf2 > 0
          bf2 := bf[xf1 + fIn - 1]
        else
          bf2 := 0

        // transition probability
        state2 = (seqf2, bf2, bi2, xf2, stallf2)
        P[state1, state2] += p1*p2*p3*p4

      // reset bf flag
      if mispred < xf1+ef
        bf[mispred] := 0
```

B.3 Issue Buffer

The following parameters are used by the issue buffer model:

- `brPred` = branch predictor — one of ‘none’, ‘loop’, ‘bimodal’, ‘perfect’.

The issue buffer state consists of the following elements:

- `seqi` = the sequence number of the sequence whose first instruction will be next to issue.
- `bi` = 1 if there is a mispredicted branch in the issue buffer, 0 if not.
- `xi` = the number of instructions in the fetch buffer.

The transition matrix is computed as follows. In addition to computing the transition matrix, this code also computes per-state push-out and pull-in probabilities, which are combined later with the standard distribution to compute the component push-out and pull-in probabilities.

```
// initial state
for each state1 = (seqi1, bi1, xi1)

  // push out to pipes
  // (pOut is treated as both a bit vector and an array of
  // bits)
  for t := 0 .. 2
    if there exists 0 <= i < xi1 s.t. seqi1.instr[i].y = t
      pOut[t] := 1
    else
      pOut[t] := 0
  statePushOut[state1, seqi1, pOut] := 1

  // pull out to pipes
  // qOut[t][i] = pull out by pipe-t of pipe-i instr
  // (qOut[t] is treated as both a bit vector and an array
  // of bits)
  for qOut := <000,000,000> .. <111,111,111>

    // Prob[qOut]
    p1 := 1
    for t := 0 .. 2
      p1 := p1 * pipe[t].pullIn[seqi1, qOut[t]]

    // flow out to pipes
    // (binary-and of push and pull bit vectors)
    fOut := pOut & qOut[0] & qOut[1] & qOut[2]
    fOutSum := number of ones in fOut

    // empty slots in issue
    if (bi1 & fOutSum < xi1)
      // mispred branch left in issue
      ei := 0
    else
```

```
  ei := 2 - (xi1 - fOutSum)

// for each possible pair of instr seqs
for nextSeq1Idx := 0 .. seqi1.nnext-1
  nextSeq1 := seqi1.next[nextSeq1Idx]
  nextSeq1Prob := seqi1.prob[nextSeq1Idx]
  for nextSeq2Idx := 0 .. nextSeq1.nnext-1
    nextSeq2 := nextSeq1.next[nextSeq2Idx]
    nextSeq2Prob := nextSeq1.prob[nextSeq2Idx]

    // Prob[nextSeq1, nextSeq2]
    p2 := nextSeq1Prob * nextSeq2Prob

    // get seqf
    if xi1 = 0
      seqf := seqi1
    else if xi1 = 1
      seqf := nextSeq1
    else
      seqf := nextSeq2

    // new seqi value
    if fOutSum = 0
      seqi2 := seqi1
    else if fOutSum = 1
      seqi2 := nextSeq1
    else
      seqi2 := nextSeq2

    // concatenate InstrInfos
    info[0] := seqi1.instr[0]
    info[1] := seqi1.instr[1]     // = nextSeq1.instr[0]
    info[2] := nextSeq2.instr[0]  // = nextSeq1.instr[1]
    info[3] := nextSeq2.instr[1]

    // p3a = Prob[instrs up to here are not mispred
    //            branches]
    // (running product over mispred loop)
    p3a := 1

    // each possible mispred branch position
    for i := 0 .. 3
```

```
    bi[i] := 0
  if bi1
    bi[xi1-1] := 1
  for mispred := xi1 .. xi1+ei

    // mispred = xi1+ei means no new mispred branch
    if mispred = xi1+ei
      p3 := p3a
      qIn := ei

    // mispred branch at specific index
    else

      // set mispred branch flag
      bi[mispred] := 1

      // Prob[mispred]
      if info[mispred].y != 3  // not a branch
        p3 := 0
      else if brPred = perfect
        p3 := 0
      else if brPred = none
        p3 := p3a
        p3a := 0
      else // loop or bimodal
        p3 := p3a * info[mispred].mispred[brPred]
        p3a := p3a *
               (1 - info[mispred].mispred[brPred])

      // pull in from fetch
      qIn := mispred - xi1 + 1

    // pull in from fetch
    statePullIn[state1, seqf, bi1, qIn] += p1*p2*p3

    // push in from fetch
    for pIn := 0 .. 2

      // Prob[pIn]
      p4 := fetch.pushOut[seqf, bi1, pIn]

      // flow in from fetch
```

```
fIn := min{pIn, qIn}

// new xi value
xi2 := xi1 - fOutSum + fIn

// new bf value
if xf2 > 0
  bi2 := bi[xi1 + fIn - 1]
else
  bi2 := 0

// transition probability
state2 := (seqi2, bi2, xi2)
P[state1, state2] += p1*p2*p3*p4
```

B.4 Pipeline

The following parameters are used by the pipeline model:

- `pipeNum` = the number (0, 1, or 2) of this pipe
- `depth` = pipe depth
- `dCacheSize` = the size of the fully associative D-cache, in blocks; -1 for perfect cache
- `dCacheStall` = the number of stall cycles on a D-cache miss

The issue buffer state consists of the following elements:

- `seqi` = the sequence number of the sequence whose first instruction will be next to issue.
- `xp` = the occupancy bit vector for the pipe.
- `stallp` = the number of cycles remaining in a D-cache stall.

The transition matrix is computed as follows. In addition to computing the transition matrix, this code also computes per-state pull-in probabilities, which are combined later with the standard distribution to compute the component pull-in probabilities.

```
// for each initial state
for each state1 = (seqi1, xp1, stallp1)

  // push out of pipe
  if depth = 1 && stallp1 > 0
    pOut := 0
  else
    pOut := xp1 & 1
  update pushOut[pOut] := 1

  // compute 'before' vector:
  // before[t] = # type-<pipeNum> instrs before first
  // type-<t> instr
  for t := 0 .. 2
    before[t] := 0
    for i := 0 .. 1
      if seqi1.instr[i].y = 3 // branches flow into int pipe
        t2 := 0
      else
        t2 := seqi1.instr[i].y
      if t2 = t
        break
      if t2 = pipeNum
        before[t] += 1

  // pull out of pipe is always 1
  qOut := 1

  // compute number of instrs left in pipe
  if stallp1 > 0
    k := number of ones in xp1
  else
    k := number of ones in (xp1 >> 1)

  // for each possible push into pipes
  for pIn := <000> .. <111>
```

```
// Prob[pIn]
p1 := issue.pushOut[seqi1, pIn]

// compute pull into pipes, constrained by dependences
// on this pipe
qIn := 0
for i := 0 .. 1
  if seqi1.instr[i].y = 3 // branches flow into int pipe
    t2 := 0
  else
    t2 := seqi1.instr[i].y
  if qIn & bit[t2]
    break
  if t2 = pipeNum
    // instr is headed for this pipe and this pipe is
    // stalled or the instr is dependent on an instr
    // left in this pipe
    if (stallp1 > 0 ||
        (seqi1.instr[i].s[pipeNum] < maxDepDist &&
         seqi1.instr[i].s[pipeNum] < k))
      break
  else
    // instr is headed for another pipe and is dependent
    // on an instr ahead of it in issue or an instr left
    // in this pipe
    if (seqi1.instr[i].s[pipeNum] < maxDepDist &&
        seqi1.instr[i].s[pipeNum] < k + before[t2])
      break
  qIn |= bit[t2]
statePullIn[state1, seqi1, qIn] += p1

// for all possible pulls into pipes
// qPipes[t][i] = pull in by pipe-t of pipe-i instr
for qPipes := <000,000,000> .. <111,111,111>
              except qPipes[pipeNum] = qIn

  // Prob[pulls into pipes]
  p2 := 1
  for t := 0 .. 2
    if t != pipeNum
      p2 *= pipe[t].pullIn[seqi1, qPipes[t]]
```

```
// flow into pipes
fIn = pIn
for t := 0 .. 2
  fIn &= qPipes[t]
fInSum := number of ones in fIn

// for each possible pair of instr seqs
for nextSeq1Idx := 0 .. seqi1.nnext-1
  nextSeq1 := seqi1.next[nextSeq1Idx]
  nextSeq1Prob := seqi1.prob[nextSeq1Idx]
  for nextSeq2Idx := 0 .. nextSeq1.nnext-1
    nextSeq2 := nextSeq1.next[nextSeq2Idx]
    nextSeq2Prob := nextSeq1.prob[nextSeq2Idx]

    // Prob[nextSeq1, nextSeq2]
    p3 := nextSeq1Prob * nextSeq2Prob

    // next sequence index
    if fInSum = 0
      seqi2 := seqi1
    else if fInSum = 1
      seqi2 := nextSeq1
    else
      seqi2 := nextSeq2

    // new xp value
    if stallp1 > 0
      // stalled for cache miss
      xp2 := (first bit of xp1) | (rest of xp1 >> 1)
    else
      // not stalled
      xp2 = xp1 >> 1
      if fIn & bit[pipeNum]
        xp2 |= bit[0]

    // compute two possible stall2 values and their
    // probabilities:
    //   Prob[stall2a] = p4
    //   Prob[stall2b] = 1-p4
    if stall1 > 0
      // currently stalled
```

```
  stall2a = stall1 - 1
  stall2b = 0
  p4 = 1

else if (this is the memory pipe) &&
        (fIn & bit[pipeNum])
  // new mem instr is entering pipe
  i = first instr of type pipeNum
  stall2a = 0
  stall2b = dCacheStall
  p4 = seqi1.instrs[i].dHit[cacheSize]

else
  stall2a = 0
  stall2b = 0
  p4 = 1

// update transition matrix for stall2a
state2a := (seqi2, xp2, stall2a)
update P[state1,state2a] += p1 * p2 * p3 * p4

// update transition matrix for stall2b
state2b = (seqi2, xp2, stall2b)
update P[state1,state2b] += p1 * p2 * p3 * (1-p4)
```

Bibliography

[AS92] Todd M. Austin and Gurindar S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proc. ISCA 19*, pages 342–351, 1992.

[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.

[BS97] Bryan Black and John Paul Shen. Rigorous Validation of Superscalar Performance Models. In *Performance Analysis and its Impact on Design (PAID)*, pages 64–70, 1997. (workshop at ISCA).

[BKW90] Anita Borg, R. E. Kessler, and David W. Wall. Generation and Analysis of Very Long Address Traces. In *Proc. ISCA 17*, pages 270–279, 1990.

[CE85] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[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.

[CHM96] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *Proc. International Conference on Computer Design*, 1996.

[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.

[DS95] Trung A. Diep and John Paul Shen. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer*, 28(12):57–64, 1995.

[Dig92] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[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.

[DN95] Pradeep K. Dubey and Ravi Nair. Profile-driven Sampled Trace Generation. Technical Report RC 20041, IBM Research Division, 1995.

[FP94] John W. C. Fu and Janak H. Patel. Trace Driven Simulation using Sampled Traces. In *Proc. 27th Annual Hawaii International Conference on System Sciences*, pages 211–220, 1994.

[GAR+93] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson. BACH: a hardware monitor for tracing microprocessor-based systems. *Microprocessors and Microsystems*, 17(8):443–459, October 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[JCSM96] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An Analytical Model for Designing Memory Hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, October 1996.

[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.

[LPI88] Subhasis Laha, Janak H. Patel, and Ravishankar K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proc. ISCA 19*, pages 46–57, 1992.

[Lau94] Gary Lauterbach. Accelerating Architectural Simulation by Parallel Execution of Trace Samples. In *Proc. 27th Annual Hawaii International Conference on System Sciences*, pages 205–210, 1994.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proc. MICRO-29*, 1996.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993. Special Issue on Instruction-Level Parallelism.

[RF96] Charlton D. Rose and J. Kelly Flanagan. Constructing Instruction Traces from Cache-filtered Address Traces (CITCAT). *Computer Architecture News*, 24(5):1–8, December 1996.

[Ros83] Sheldon M. Ross. *Stochastic Processes*. John Wiley & Sons, 1983.

[Smi81] James E. Smith. A Study of Branch Prediction Strategies. In *Proc. ISCA 8*, pages 135–148, 1981.

[SF91] Gurindar S. Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proc. ASPLOS-IV*, pages 53–62, 1991.

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

[SPE95] Standard Performance Evaluation Corporation. *SPEC CPU95 benchmark suite*, 1995.

[SCH+91] Chriss Stephens, Bryce Cogswell, John Heinlein, Gregory Palmer, and John P. Shen. Instruction Level Profiling and Evaluation of the IBM RS/6000. In *Proc. ISCA 18*, pages 180–189, 1991.

[Ste94] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[SF89] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Proc. International Conference on Measurement and Modeling of Computer Systems*, pages 70–78, 1989.

[TGH92] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the Limits of Program Parallelism and its Smoothability. In *Proc. MICRO-25*, pages 10–19, 1992.

[Wal91] David W. Wall. Limits of Instruction-Level Parallelism. In *Proc. ASPLOS-IV*, pages 176–188, 1991.