Analytic Models of Workload Behavior and Pipeline Performance

Mark S. Squillante
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
mss@watson.ibm.com

David R. Kaeli, Himanshu Sinha
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115
{kaeli,sinha}@ece.neu.edu

Abstract

The evaluation of pipeline performance and the analysis of different design alternatives and cost/performance tradeoffs are a fundamental aspect of high-performance computer system design. This performance evaluation process requires accurate models of both the pipeline organization and the characteristics of the workload being executed. In this paper we derive general mathematical models and analyses of workload behavior and pipeline performance that can provide measures as accurate as detailed trace-driven simulations with the computational efficiency of analytic methods.

1 Introduction

The use of pipelining, wherein the execution of multiple instructions is performed concurrently, has been and continues to be a fundamental and important technique in the design of high-performance computer systems (e.g., see [5]). This technique consists of breaking the processing of instructions into steps, or *stages*, each performing a different piece of the entire instruction, and simultaneously executing instructions in different stages of processing. Although pipelining typically does not reduce the execution time of an individual instruction, the number of instructions completed per unit time is increased by overlapping the processing of one instruction with that of others.

The evaluation of pipeline performance is a critical aspect of computer system design that, together with performance comparisons of many different implementation strategies and alternatives, has received vast attention in the research literature (e.g., see [5]). An important metric for evaluating the performance of pipelined processors is the number of instructions executed per processor cycle. The maximum, or *ideal*, throughput of one instruction (or possibly more in the case of superscalar architectures) departing the pipeline on each cycle is often not realized in practice due to control, data and structural hazards in the instruction execution stream. These hazards reduce pipeline performance by causing stalls, or *bubbles*, that interrupt the flow of instructions through the pipeline. An important objective of computer system design consists of reducing the impact of these pipeline penalties to achieve pipeline throughputs as close to the ideal as possible.

The performance evaluation of computer pipelines and performance comparisons of different design alternatives depend upon the behavior of the workload being executed and the pipeline organization and implementation techniques. Hence, the pipeline design and workload behavior are inseparable, and the performance evaluation of these computer systems requires accurate models of each component and an accurate representation of their interactions. Prior to our present work, it has been argued that there is no sufficiently accurate mathematical model to characterize aspects of program behavior with respect to pipeline performance [7], with the possible exception of memory reference behavior models (see [11]). For this reason, instruction-level traces are commonly used to represent the workload behavior and to drive simulation models of different pipeline organizations to evaluate the performance of these designs.

A detailed, cycle-based, trace-driven simulation of a computer pipeline can be used to obtain very accurate performance measures of the pipeline design under a particular workload. However, the time and resources necessary to perform such simulations can be very expensive due to the length of the workload instruction traces and the complexity of the cycle-based pipeline model. An alternative approach at the opposite end of the methodological spectrum consists of obtaining instruction frequency measures and related data from instruction traces or hardware profiling, and then using these measurements to parameterize a simple static model of pipeline performance (e.g., see [4, 8]). This static modeling method can be used to obtain results very quickly, at the expense of a significant reduction in the model accuracy.

Our primary objectives in this paper are to derive general mathematical models that completely and accurately capture the workload behavior, and that provide pipeline performance measures as accurate as detailed, cycle-based simulations with the efficiency of simple static modeling methods. Our approach is based in part on the use of stochastic marked point processes to completely characterize the occurrences of workload events which affect pipeline performance and obtain distributions for such measures. This detailed workload model and analysis is then used to parameterize a model of pipeline performance that is more detailed than static models and captures the dynamics of instruction execution in the pipeline. We then use this modeling framework to study the performance impact of branch instructions on different pipeline organizations. Branches are chosen because they can dominate the overall performance of a processor pipeline, and thus continue to be the focus of processor design research (e.g., see [3, 6, 13]). In the interest of space, we only present here a small subset of our modeling and performance results, and we refer the interested reader to an expanded version of this paper [11] for additional background material, references, technical details, results and applications of our modeling methodology.

2 Mathematical Models and Analysis

In this section we derive analytic models and a mathematical analysis of workload behavior and pipeline performance. Our approach is very general and applies to all control, data and structural hazards, as well as any other workload events that impact pipeline performance. However, the description in this section will be in terms of taken branch instructions. The analysis for other workload events follows in a straightforward manner, as is discussed in [11].

We first present our modeling analysis of workload behavior, which yields a set of distributions characterizing this behavior. We then describe our use of these distributions to evaluate pipeline performance for all organizations of interest. Throughout this paper we use $\mathbb{N}$ to denote the set of positive integers.

2.1 Workload Behavior

Our model of workload behavior is based on a mathematical characterization of the contents of a sequential segment of the instruction execution stream, which we call a *pipeline window*, whose length is equal to the number of stages L in the pipeline being evaluated. The occurrences of taken branch instructions are represented as points in time scattered over a (infinite) discrete-time instruction execution stream in some random manner, where the unit of time is a single instruction in the execution stream. Without loss of generality, we assume the instruction execution stream starts at time 1. Our analysis then determines distribution measures of the charateristics of taken branches active in a pipeline window of length L, $1 \leq L < \infty$. This formulation makes it possible to directly compute the distribution measures of active events in any finite-length pipeline window, and thus eliminates the need to separately model workload behavior for each pipeline organization.

More formally, let $t_n \in \mathbb{N}$ denote the *point* in time at which the n^{th} taken branch occurs in the instruction execution stream, $n \geq 1$, where $t_0 \equiv 0$ by convention, $t_0 < t_1 < t_2 < \ldots$, and $\lim_{n\to\infty} t_n = \infty$. The interbranch time between the $n-1^{\text{st}}$ and n^{th} taken branch is given by $\tau_n \equiv t_n - t_{n-1}$, $n \geq 1$. Associated with the n^{th} taken branch is a *mark* $k_n \in \mathbb{K}$, where k_n is used to represent all important and useful information for the n^{th} taken branch and $\mathbb{K}$ is the space of all such marks, called the mark space. Examples of possible marks include information on the type of taken branch and the pipeline delay associated with the branch. The occurrences of taken branches in the instruction execution stream are then modeled as a stochastic marked point process $\psi = \{(t_n, k_n)\ ;\ n \geq 0\}$ defined over the state space $\mathcal{S} \equiv \mathbb{N} \times \mathbb{K}$, where we define $k_0 \equiv \{\}$. In other words, ψ is a stochastic sequence of points $\{t_1, t_2, \ldots\}$ in (discrete) time at which taken branch instructions occur, together with a corresponding sequence of general marks $\{k_1, k_2, \ldots\}$ containing information about these taken branches.

A counting process $N(t)$ is associated with the marked point process ψ which represents the number of points that lie in the time interval $[1, t]$, and is given by

$$N(t) \;\equiv\; \max\{n \;:\; t_n \leq t\}, \quad t \geq 1. \qquad (1)$$

We also define

$$N(s,t) \;\equiv\; N(t) \;-\; N(s-1), \quad 1 \leq s \leq t, \qquad (2)$$

to be the number of taken branch instructions that occur during the interval from time s to time t, inclusively.

It is very convenient for our purposes to view ψ as the set function on the subsets of $\mathcal{S}$ defined by [10]

$$\psi(A \times K) \;=\; \sum_{n=1}^{\infty} I\{t_n \in A, k_n \in K\}, \qquad (3)$$

for $A \subset \mathbb{N}$ and $K \subset \mathbb{K}$, where $A \in \mathcal{B}(\mathbb{N})$, $\mathcal{B}(\mathbb{N})$ denotes the Borel sets of $\mathbb{N}$ [2], $K \in \mathcal{B}(\mathbb{K})$, $\mathcal{B}(\mathbb{K})$ denotes the Borel sets of $\mathbb{K}$, and $I\{\cdot\}$ denotes the indicator function [2]. Intuitively, $\psi(A \times K) \in \{0\} \cup \mathbb{N}$ denotes the number of taken branches during the time interval A that have marks in the mark set K. The set function ψ in (3) clearly satisfies the conditions for a *measure*, which is infinite (since $\psi(\mathcal{S}) = \infty$) and σ-finite (since $\psi(A \times \mathbb{K}) < \infty$, $A \subset \mathbb{N}$) [2]. The corresponding counting measure can be defined as

$$N(A) \;\equiv\; \sum_{n=1}^{\infty} I\{t_n \in A\}, \quad A \subset \mathbb{N}. \qquad (4)$$

Hence, $N(A)$ is equivalent to $\psi(A \times \mathbb{K})$, $A \subset \mathbb{N}$.

Let $\mathbb{M}$ be the set of all marked point processes ψ with point space $\mathbb{N}$ and mark space $\mathbb{K}$, and let $\mathcal{B}(\mathbb{M})$ denote the Borel sets of $\mathbb{M}$. We then consider a random element Ψ of $\mathbb{M}$ defined on the probability measure space $(\mathcal{S}, \mathcal{F}, P)$ [2]. The sample paths of Ψ are the marked point processes $\psi \in \mathbb{M}$ defined above, and the distribution $\mathrm{P}[\,\Psi \in G\,]$ defines the probability that the realizations of Ψ take on values in $G \in \mathcal{B}(\mathbb{M})$. Ψ is called a random marked point process, which can also be viewed as a random measure $\Psi(\omega, A \times K)$ defined analogous to equation (3) for each $\omega \in \mathcal{S}$ [10]. As a specific example, suppose that $G = \{\psi \in \mathbb{M} : t_1 = 1, k_1 = (\text{"unconditional"}, \text{delay} = 1), t_2 = L, 2 \leq L < \infty, k_2 = (\text{"conditional"}, \text{delay} = L-1), t_3 > L\}$ for a given marked point process space $\mathbb{M}$. Then $\mathrm{P}[\,\Psi \in G\,]$ defines the probability that the realizations of the corresponding instruction execution stream have an unconditional branch that occurs at time 1 causing a pipeline stall of 1 cycle, a conditional branch that occurs at time L, $2 \leq L < \infty$, causing a pipeline stall of $L-1$ cycles, and all other branch occurrences take place beyond the time interval $[1, L]$.

We define Ψ_s to be the random marked point process obtained from the original process Ψ shifted by time s with its origin redefined to be s. An important component of our approach is the property of time stationarity.

Definition 2.1 *A random marked point process Ψ is* time stationary *if*

$$\mathrm{P}[\,\Psi_s \in G\,] \;=\; \mathrm{P}[\,\Psi \in G\,] \tag{5}$$

for all $G \in \mathcal{B}(\mathbb{M})$ and $s \in \mathbb{N}$.

We then have the following fundamental result.

Proposition 2.2 *Consider a random marked point process Ψ and define the random marked point process Ψ^* by the limiting distribution*

$$\mathrm{P}[\,\Psi^* \in G\,] \;\equiv\; \lim_{t\to\infty} \frac{1}{t} \sum_{s=1}^{t} \mathrm{P}[\,\Psi_s \in G\,] \tag{6}$$

for all $G \in \mathcal{B}(\mathbb{M})$. Assuming the limiting distribution in (6) exists, then Ψ^* is time stationary.

Proof. Let Ψ^*_s denote the random marked point process obtained by shifting Ψ^* to the left by time s with its origin redefined to be s. Since the probability measure of each event $G \in \mathcal{B}(\mathbb{M})$ is preserved under these shift mappings, it then follows from equation (6) that

$$\begin{aligned}
\mathrm{P}[\,\Psi^*_s \in G\,] \;&=\; \lim_{t\to\infty} \frac{1}{t} \sum_{u=1}^{t} \mathrm{P}[\,\Psi_{s+u} \in G\,] \\
&=\; \lim_{t\to\infty} \frac{1}{t} \sum_{v=1}^{s+t} \mathrm{P}[\,\Psi_v \in G\,] \\
&\qquad -\; \lim_{t\to\infty} \frac{1}{t} \sum_{v=1}^{s} \mathrm{P}[\,\Psi_v \in G\,] \\
&=\; \lim_{t\to\infty} \frac{1}{t} \sum_{v=1}^{s+t} \mathrm{P}[\,\Psi_v \in G\,]. \qquad (7)
\end{aligned}$$

From the definition in (6), this final limit tends to $\mathrm{P}[\,\Psi^* \in G\,]$, which together with equation (5) completes the proof. □

The above formulation together with Propostition 2.2 provide the basis for our workload behavior model and analysis. This is a powerful and rich mathematical approach with a number of important and easily established properties and results [10], which can be exploited to model and analyze various aspects of workload behavior, pipeline performance and their interactions. Intuitively, our analysis yields time average distributions of any measure of the characteristics of taken branches in a (infinite) discrete-time instruction execution stream with respect to a (finite) pipeline window. As a simple example, $\mathrm{P}[\,\Psi^* \in G\,]$ with $G = \{\psi \in \mathbb{M} : \psi([1, L] \times \mathbb{K}) = \ell\}$ provides the long-run proportion of time that a pipeline window of length L contains ℓ taken branches of any type, $1 \le \ell \le L$, under the instruction execution stream being considered. The technical details depend in part on the distributions for the sequence of interbranch times $\{\tau_n\,;\, n \ge 1\}$ and marks $\{k_n\,;\, n \ge 1\}$. In the remainder of this section we present some of these details for a very simple instance of our general modeling analysis.

2.1.1 Renewal Process

Under the assumption that the interbranch times $\tau_n \in \mathbb{N}$, $n \ge 1$, are independent and identically distributed with a common distribution function F, the counting process $N(t)$ associated with the branch times $\{t_n\,;\, n \ge 1\}$ is a renewal process [9]. We define $\mathcal{T}_n \equiv \sum_{j=1}^{n} \tau_j$, $n \ge 1$, whose distribution is given by the n-fold convolution of F with itself, denoted by F_n. The probability distribution of $N(t)$ can then be obtained as [9]

$$\begin{aligned}
\mathrm{P}[\,N(t) = \ell\,] \;&=\; \mathrm{P}[\,N(t) \ge \ell\,] \;-\; \mathrm{P}[\,N(t) \ge \ell + 1\,] \\
&=\; \mathrm{P}[\,\mathcal{T}_\ell \le t\,] \;-\; \mathrm{P}[\,\mathcal{T}_{\ell+1} \le t\,] \\
&=\; F_\ell(t) \;-\; F_{\ell+1}(t). \qquad (8)
\end{aligned}$$

We now consider a particular measure for this renewal process, namely the time average distribution of the number of taken branch instructions active in a pipeline window of length L, denoted by $p^{\mathrm{L}}_{\mathrm{B}}(\ell) \equiv \mathrm{P}[\,N^*(L) = \ell\,]$, $0 \le \ell \le L$. From equation (6), this can be expressed as

$$\mathrm{P}[\,N^*(L) = \ell\,] = \lim_{t\to\infty} \frac{1}{t} \sum_{s=1}^{t} \mathrm{P}[\,N(s, s+L-1) = \ell\,], \tag{9}$$

where $N^*(t)$ is the time stationary version of the original renewal process $N(t)$, and $N^*(0) \equiv 0$.

The equilibrium renewal process $N^*(t)$ is by definition identical to $N(t)$ with the exception that the first interbranch time (i.e., τ_1) has distribution F_e, whereas all other interbranch times (i.e., τ_n, $n > 1$) continue to have distribution F. One can easily establish that this equilibrium distribution of F is given by [9]

$$F_e(x) \;=\; \frac{1}{\mathrm{E}[\,X\,]} \sum_{\ell=1}^{x} \mathrm{P}[\,X \ge \ell\,], \quad x \ge 1, \tag{10}$$

where $X \sim F$. It then follows from its definition and equation (10) that the probability distribution of $N^*(t)$ can be obtained as

$$\mathrm{P}[\,N^*(t) = \ell\,] \;=\; H_\ell(t) \;-\; H_{\ell+1}(t) \tag{11}$$

where H_ℓ is the convolution of F_e with $F_{\ell-1}$. Substituting this expression in equation (9) yields the desired probability distribution of the number of taken branch instructions contained in a pipeline window of length L. Specifically, we have

$$p^{\mathrm{L}}_{\mathrm{B}}(\ell) \;=\; \mathrm{P}[\,N^*(L) = \ell\,] \;=\; H_\ell(L) \;-\; H_{\ell+1}(L). \tag{12}$$

Based on the renewal assumptions of this section, the vectors (τ_n, k_n), $n \ge 1$, are independent and identically distributed where k_n can be dependent on τ_n. The above results can be used to accommodate any such distribution for the marks k_n, $n \ge 1$ (recall that $N(A) = \psi(A \times \mathbb{K})$, $A \subset \mathbb{N}$).

2.2 Pipeline Performance

The mathematical model and analysis of the previous section, which completely characterizes the occurrences of taken branches in the instruction execution stream, is now used to parameterize a model of pipeline performance that captures the dynamics of instruction execution in the pipeline. Several performance measures are of interest, but in the interest of space our focus here is on two particular measures, and a particular form for the taken-branch penalty function upon which these measures are based. The interested reader is referred to [11] for definitions of other performance measures and more general penalty functions.

One important measure of pipeline performance is the average number of machine cycles executed per instruction in an L-stage pipeline, denoted by $CPI(L)$. Throughout this section we assume the ideal cycles-per-instruction (CPI) to be 1. The computer system realizes this ideal CPI when the pipeline window contains no taken branches, which occurs with probability $p_{\mathrm{B}}^{\mathrm{L}}(0)$. Otherwise, with probability $1-p_{\mathrm{B}}^{\mathrm{L}}(0)$, the pipeline CPI equals the ideal plus the number of cycles lost when there are k branches in the window, summed over all k in an appropriately weighted manner. We therefore can express the function $CPI(L)$ as

$$CPI(L) = p_{\mathrm{B}}^{\mathrm{L}}(0)+(1-p_{\mathrm{B}}^{\mathrm{L}}(0))\left[1+\frac{1}{L}\sum_{k=1}^{L} p_{\mathrm{B}}^{\mathrm{L}}(k)P_{\mathrm{B}}^{\mathrm{L}}(k)\right], \tag{13}$$

where $P_{\mathrm{B}}^{\mathrm{L}}(k)$ is the performance penalty incurred when the L-stage pipeline encounters k taken branch instructions within a pipeline window of length L.

Another important performance measure concerns the processor speedup obtained with pipelining. The relative speedup for an L-stage pipeline, $\mathrm{S}_{\mathrm{R}}(L)$, is the ratio of $CPI(1)$ to $CPI(L)$. Since the ideal CPI is $CPI(1)/L$ for our purposes, and given the assumption of an ideal CPI of 1, we have

$$\mathrm{S}_{\mathrm{R}}(L) \;=\; \frac{L}{CPI(L)}. \tag{14}$$

Observe that both of these performance measures are based on the branch penalty function $P_{\mathrm{B}}^{\mathrm{L}}$, which in practice is highly dependent upon many aspects of the pipeline design and implementation. In this paper we consider a particular form for this function based in part on our DLX implementation [11], although our approach is quite general and can be used to consider more complex pipeline penalty functions.

At the time when a branch is determined to be taken, all instructions that have entered the pipeline after the branch instruction are voided. The resulting empty pipeline stages are considered to generate cycles of delay, since they do produce any useful work. They must wait for the new stream of instructions to refill the pipeline.

In our model of pipeline delay due to taken branches we can say that in an L-stage pipeline, the number of cycles lost due to a taken branch is $L-m$, where m is an integer constant and is typically much smaller than L. While it is true that there are a myriad of possible pipeline implementations (and branch prediction schemes) upon which the exact penalty of a specific branch depends, we do not try to characterize any specific pipeline implementation as this is not one of the objectives of this paper. However, for any implementation, the penalty will be proportional to L, and this is sufficient for our model. Similarly, the assumption that m is a constant may not be strictly true in practice, but it is a reasonable approximation and is sufficient for our purposes here. We therefore have

$$P_{\mathrm{B}}^{\mathrm{L}}(k) \;=\; k\,(\,L\,-\,m\,). \tag{15}$$

3 Results

Next we consider the performance characteristics of different pipeline organizations within the context of our workload and performance models presented in Section 2. We will assume in this analysis that the interbranch times are independent and identically distributed (see Section 2.1.1), the marks are not used (i.e., each taken branch point has the same mark $k_n = \{\}$, $n \geq 1$), and m is set to 1 (see equation (15)). We address relaxing these assumptions in future work.

3.1 Accuracy of Models

We begin by studying a set of benchmark programs, which was specifically chosen because we want to also feed it to a VHDL model of our target CPU (running larger programs is not feasible). The program we study include: (1) *Acker*, an Ackerman merge and split function; (2) *Bubsort*, the bubblesort algorithm on a 100-element table; (3) *Fib*, the Fibonacci number sequence algorithm; (4) *Prime*, the Erostathenes prime number sieve; and (5) *Puzzle*, the Tower of Hanoi problem. Benchmarks taken from the SPEC suite were also studied and yielded results comparable to those obtained for the above list of benchmarks.

Instruction traces are captured from benchmark execution and a single pass is made through each trace to obtain the distribution F of the the number of instructions between successive taken branch instructions. Then the distributions F_e, H_ℓ and $p_{\mathrm{B}}^{\mathrm{L}}$ are all computed from the equations in Section 2.1. In Table 1 we present for each trace the probability distribution $p_{\mathrm{B}}^{\mathrm{L}}$ obtained from our model (MPP) together with the corresponding distribution (WW) measured by computing the RHS of equation (6). Pipeline lengths of 4 and 8 are provided, noting that the results for $L=2$ are (essentially) identical as in the $L=4$ case.

We find that the results from our simple renewal point process model are essentially identical to those measured from Bubsort and Puzzle. The same holds true for all of the other benchmarks when $L \leq 4$. This is rather surprising because we know that there are dependencies between some of the taken branch instructions, and these dependencies are not reflected in the model due to the assumption of independent and identically distributed interbranch times. However, most (if not all) of these dependencies have a span that is larger than L for the cases noted above (as we have verified), and thus with respect to the contents of a pipeline window of length L, any errors resulting from the simple renewal assumption are negligible.

Some small discrepancies appear at a couple of points for the Fib benchmark when $L=8$, with somewhat larger differences occurring for the same case of the Acker and Prime benchmarks. The primary cause of

Trace	$B=0$	$B=1$	$B=2$	$B=3$	$B=4$
Ac(4)	.45/.45	.50/.50	.05/.05	.00/.00	.00/.00
Bu(4)	.67/.67	.33/.33	.00/.00	.00/.00	.00/.00
Fi(4)	.39/.39	.57/.57	.04/.04	.00/.00	.00/.00
Pr(4)	.25/.25	.61/.61	.14/.14	.00/.00	.00/.00
Pu(4)	.54/.54	.46/.46	.00/.00	.00/.00	.00/.00
Ac(8)	.15/.15	.54/.50	.26/.35	.04/.00	.00/.00
Fi(8)	.08/.08	.55/.53	.34/.38	.02/.00	.00/.00
Bu(8)	.49/.49	.35/.35	.15/.15	.00/.00	.00/.00
Pr(8)	.00/.00	.37/.30	.47/.63	.15/.07	.00/.00
Pu(8)	.20/.20	.67/.67	.12/.12	.00/.00	.00/.00

Table 1: Probability distributions of having B branches in 4- and 8-stage pipelines (MPP / WW).

these errors is due to *tight loops*, where the dependencies between some of the taken branch instructions have spans less than or not significantly larger than $L = 8$.

We compare our results with a cycle-by-cycle VHDL simulation of the 4-stage DLX processor described in Section **??** in Table 2. Equation (13) is used to compute $CPI(L)$ based on the workload distribution $p^{\mathrm{L}}_{\mathrm{B}}$ and $P^{\mathrm{L}}_{\mathrm{B}}(k) = k(L-1)$, $L = 4$, for each of the workloads considered. In our VHDL model we are able to eliminate all data dependency stalls, so that we can study the effects of taken branches in isolation.

Trace	VHDL	Models	Relative Error
Fib	1.298101	1.299876	0.001
Acker	1.251793	1.247549	0.003
Puzzle	1.150341	1.157624	0.006
Bubsort	1.042752	1.080815	0.037
Prime	1.341794	1.497353	0.116

Table 2: Comparison of CPI measures from our mathematical models and a VHDL simulation.

We observe that the results from our simple performance model, parameterized by our simple workload model, are in excellent agreement with those obtained from the detailed VHDL simulation for the Fib, Acker and Puzzle benchmarks, each exhibiting a relative error of considerably less than 1%. A somewhat larger error of 3.6% is observed for Bubsort, with a relative error of 11.5% exhibited for the benchmark Prime. It is important to note, however, that these discrepancies are not due to errors in characterizing the workload behavior, since the renewal workload model results for Bubsort are essentially identical to those measured from the trace and the same is true for Prime with $L = 4$ (see Table 1). The errors for Bubsort and Prime in Table 2 are instead due to the branch penalty function defined in equation (15) with $m = 1$. In our simple pipeline performance model we treat all taken branches as incurring the same latency, whereas in our VHDL model branch execution times will vary to some degree due to the fact that not all branch states are equal (branch state is the defined as the state of the pipeline when the branch is decoded). Many of these special cases are implementation dependent, and are not determined by the underlying architecture. These effects have a fairly negligible impact on the $CPI(L)$ values for Fib, Acker and Puzzle. However, these special cases represent a non-negligible fraction of the branch penalties incurred under the Bubsort and Prime workloads, which in turn results in relative errors of 3.7% and 11.6%, respectively.

In

addition to the performance measures just presented, we can also provide the measures $CPI(L)$,$\mathbf{X}_{\mathrm{R}}(L)$ and $\mathbf{S}_{\mathrm{R}}(L)$ for each of the benchmark programs considered and $1 \leq L \leq 10$. These functions are computed from equations (13), (**??**) and (14), respectively, and shown in Figure 1. Recall that the ideal CPI is 1, which represents no pipeline delay. We do not provide plots of the $\mathbf{X}_{\mathrm{R}}(L)$ curves since they are simply the inverse of the $CPI(L)$ curves.

We can see that all of the CPI curves are monotonically increasing, i.e., the pipeline delays increase as the length of the pipeline increases. This result is not surprising since the cost to refill a longer pipeline is higher. The benefit of the deep pipeline is that we can use a faster clock rate since less work is performed in each stage. We will tradeoff the number of cycles of latency for the duration of a single processor clock cycle. This tradeoff can be highly dependent upon the workload characteristics, as shown by the significant differences in the curves for larger values of L.

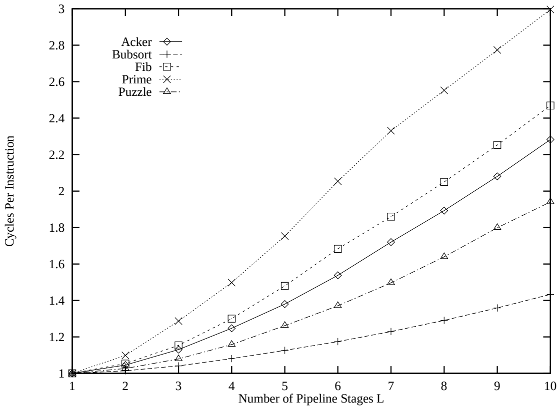

Figure 1: Pipeline CPI as a function of L.

It is also important to quantify the efficiency of our modeling methodology in comparison with that of a cycle-based VHDL simulation. Running the instruction traces through our VHDL model of a 4-stage pipeline took on average 12 hours of cpu time for each benchmark program, on a DEC Alpha 3000/800. This is in comparison to less than a second of cpu time required by our workload behavior and pipeline performance models to compute all of the performance measures presented in this section ($\forall L$, $1 \leq L \leq 10$) for each benchmark trace, given the interbranch distribution F, and on the order of tens of seconds to process each instruction trace to generate the distribution F. As we showed in this section, these results are obtained with essentially no sacrifice in the accuracy of performance measures for most of the cases considered. Furthermore, our methodology is very general, much more so than the simple model instances considered in this section, and this can be exploited to efficiently obtain performance as accurate as desired.

4 Discussion

The previous section demonstrates how very simple instances of our mathematical models and analysis can be used to efficiently obtain accurate performance measures for different pipeline organizations. Our methodology, however, can be used to obtain significantly more detailed measures of both workload behavior and pipeline performance. In particular, the derivation in Section 2.1 can be used to completely characterize any workload event that impacts performance (recall that the marks associated with each point can contain any relevant and useful information for the corresponding event), as well as arbitrary mixtures of such workload events. Note that our only assumption is that a time stationary version of the random marked point process exists, which tends not to be a problem in practice.

One important area that we plan to investigate with our modeling analysis is the study of branch prediction techniques, which have received considerable attention in the research literature (e.g., see [3, 6, 7, 13]). Many current approaches to branch prediction are based on the patterns of taken and non-taken branches leading up to the current branch. Our approach makes it possible to probabilistically analyze these patterns for various workloads to gain fundamental insights into the problem of branch prediction under different types of workloads. This can potentially be used to increase the accuracy of hardware-based, compiler-based and hybrid (combining hardware and software methods) branch prediction schemes, as well as to determine optimal parameter values for existing techniques (e.g., [13]) under different types of workloads.

The evaluation of pipeline performance is a critical aspect of computer system design that requires an analysis of the cost-performance tradeoffs of a large set of different design and implementation alternatives. The efficiency and accuracy of our modeling methodology makes it an effective mechanism for analyzing these many design issues in detail, which can be encapsulated in a performance modeling tool so that system designers are isolated from the mathematical details to whatever degree they desire. Such a tool would be useful in many aspects of system design, which is a potential area of future work. For example, a detailed analysis of workload behavior can be used to design general compiler optimizations, including approaches to reduce data dependencies and scheduling techniques to reduce taken branch penalties. Similarly, the hardware designer can use a tool based on our workload behavior and pipeline performance models to efficiently determine the most cost-effective way to achieve a certain level of performance. Of course, this can be used together with detailed, cycle-by-cycle simulations based on the insights gained from the more interactive analysis provided by our methodology. Moreover, by extending the performance model presented in this paper, our approach can be used to consider more advanced pipelining and superscalar architectures [5].

As we noted in Section 3, somewhat more sophisticated methods than the renewal instances of our models can be used to address any accuracy issues found in the results presented in this paper. For example, the tight-loop characteristics of taken branches in some of the workload benchmarks can be handled by dividing the taken branches into two types, namely those due to tight loops and all the other cases, and the methods described in [11] can then be used represent the corresponding behavior of each of these types of taken branches as well as their interactions. Alternatively, the methods of Whitt [12] and Albin [1] can be used to construct a renewal process that approximates the behavior of these tight loops, to which the original analysis of Section 2.1.1 is applied. Our methodology also supports the large and diverse class of regenerative processes, which is another useful approach for handling tight loops in the workload because they include processes that stochastically regenerate themselves at random points in time.

5 Concluding Remarks

In this paper we developed a general methodology for the quantitative evaluation of pipeline performance that can provide measures as accurate as detailed trace-driven simulations with the computational efficiency of analytic methods. Our approach is based in part on stochastic marked point processes that completely capture the workload behavior which affects pipeline performance and yield accurate distributions for characterizing the instruction execution stream contents of different pipeline organizations. These distributions are then used to parameterize a model of pipeline performance that captures the dynamics of instruction execution in the pipeline. Our results show that even simple instances of our general models provide performance measures in excellent agreement with those from a detailed trace-driven simulation for most of the benchmarks considered. We then used these models to study pipeline performance for different workloads and pipeline organizations.

References

[1] S. L. Albin. On poisson approximations for superposition arrival processes in queues. *Management Science*, 28(2):126–139, February 1982.

[2] P. Billingsley. *Probability and Measure.* John Wiley and Sons, second edition, 1986.

[3] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–11, May 1994.

[4] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, 36(6):859–876, June 1987.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 1995.

[6] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the International Symposium on Computer Architecture*, pages 34–42, May 1991.

[7] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984.

[8] B. L. Peuto and L. J. Shustek. An instruction timing model of cpu performance. In *Proceedings of the International Symposium on Computer Architecture*, pages 165–178, May 1977.

[9] S. M. Ross. *Stochastic Processes.* John Wiley and Sons, New York, 1983.

[10] K. Sigman. *Stationary Marked Point Processes: An Intuitive Approach.* Chapman and Hall, 1995.

[11] M. S. Squillante, D. R. Kaeli, and H. Sinha. Analytic models of workload behavior and pipeline performance. Technical report, IBM Research Division, August 1996.

[12] W. Whitt. Approximating a point process by a renewal process, I: Two basic methods. *Operations Research*, 30(1):125–147, January-February 1982.

[13] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 276–286, May 1995.