

Dynamic Schemes for Speculative Execution of Code *

Prabhakar Raghavan[†]

Hadas Shachnai[‡]

Mira Yaniv[§]

Abstract

Speculative execution of code is becoming a key technique for enhancing the performance of pipeline processors. In this work we study schemes that predict the execution path of a program based on the history of branch executions. Building on previous work, we present a model for analyzing the effective speedup from pipelining, when speculative execution is employed. We follow this with stochastic analyses of several schemes for speculative execution.

A main result of our study is that if we can predict branch resolution with high probability (as in the Pentium Pro processor, for example) the Single Path scheme commonly used on modern processors is within factor of 2 from the optimal. We conclude with simulations covering several of the settings that we study.

Keywords. pipeline processors, speculative execution, branch prediction, on-line algorithms

1 Introduction

1.1 Pipeline Execution of Programs

We consider the problem of on-line instruction fetch on pipeline processors. The *pipelining* technique is commonly used on modern processors for speeding up the execution of programs. Typically, the execution of a single instruction in the code can be partitioned to steps, starting with *instruction fetch*. While traditional processors allow the execution of one instruction at a time, a pipeline processor starts the execution of an instruction as soon as the previous instruction completes the first of these steps (see Figure 1).

The throughput of a processor is the rate at which instructions complete their execution. The speedup factor is the ratio between the throughput of a pipeline processor and the throughput of a processor that does not use pipelining. Ideally, this ratio should equal the length of the pipeline.

* A preliminary version of this paper appeared in the Proceedings of *the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[†]Verity, Inc. 892 Ross Drive, Sunnyvale, CA 94089, USA. e-mail: pragh@verity.com. This work done while the author was at IBM Almaden Research Center.

[‡]The Department of Computer Science, Technion, Haifa 32000, Israel. e-mail: hadas@cs.technion.ac.il. Author supported in part by B. and G. Greenberg Research Fund (Ottawa) and by the Fund for the Promotion of Research at the Technion.

[§]The Department of Computer Science, Technion, Haifa 32000, Israel. e-mail: ymira@cs.technion.ac.il.

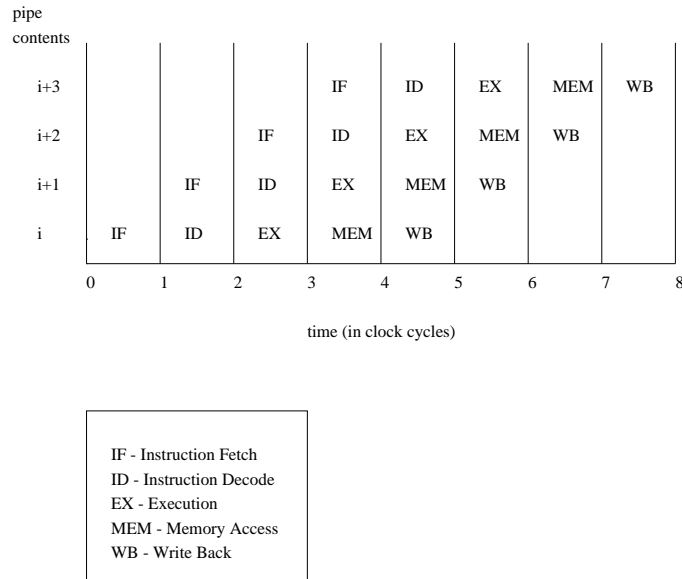


Figure 1: Pipeline of five stages

In practice, the speedup achieved by using a pipeline is much smaller, due to delays incurred when executing the program code. A major cause for delays is the execution of *branches*. Indeed, when the program starts executing a branch, the next instruction to be executed is known only when that branch is resolved. However, delaying the pipeline until the branch execution is completed is undesirable. Therefore, mechanisms for speculative execution of code are used to forecast the instruction path that will be taken by the program after each branch, with the objective of keeping the pipeline as full of useful instructions as possible. Alternatively, our goal is to minimize the overall execution time of the program, by minimizing the delays caused by ‘late fetches’ of some instructions (that relate to the execution of branches). In Section 2 we formalize this as a path selection problem in a graph, where the objective is to maximize the benefit from correct decisions.

1.2 Related Work

There is a wide literature on speculative execution of code, most notably in the area of architecture design for pipeline processors. However, as our focus here is on evaluating the performance of the proposed mechanisms, rather than on implementation issues, we summarize below the main performance results obtained for the schemes that we study.¹

Most generally, schemes for speculative execution of code are classified into two categories: *static* and *dynamic*. Static schemes fetch a predetermined set of instructions to the pipe, regardless of the actual runtime behavior of the program. For example, the Predict Taken (PT) scheme always continues the execution of the program after a branch instruction as if the branch was *taken*; the

¹A comprehensive survey of other related work is given in [11].

Predict Not Taken (PNT) scheme fetches to the pipe after each branch the next instruction in the program, assuming the branch was *not taken* [4].

The eager execution (EE) approach defines another static scheme, in which speculative execution continues along multiple paths of a *decision tree* representing the program. Each node in this tree corresponds to a branch instruction while each edge represents a series of non-branch instructions that define *basic blocks* in the program. After fetching a branch instruction into the pipe, the EE scheme proceeds with a parallel execution of the two paths emanating from the corresponding node in the tree. Indeed, this simple scheme always guarantees a correct prediction; however, EE is impractical as the number of processing elements it uses is exponential in the length of the pipeline [9, 10].

In contrast, dynamic schemes choose the path that will be executed speculatively using the accumulated history of branch execution in each specific run. Typically, dynamic schemes improve on the performance of static schemes, due to their usage of statistics on program behavior [4].

A scheme that is commonly implemented on modern processors (e.g., Intel’s Pentium Pro [1], IBM’s PowerPC 604 [8]) is the single-path (SP) scheme: at each branch *one* of the possible outcomes is chosen, and the corresponding instruction path is executed (until the next branch is encountered). This approach has been shown empirically to be efficient for the speculative execution of *easy* branches, whose resolution can be predicted correctly with probability higher than $(1 - \alpha)$ for $\alpha < 0.1$, using a standard predictor [7]. However, the SP approach performs poorly when used for branches that are harder to predict.

Uht and Sindagi [9] introduced the disjoint eager execution (DEE) scheme, which fetches into the pipe at any time the set of *most probable* instructions (as determined by the branch predictor). Each node in the tree of branches is associated with a prediction probability (this is made precise in the following section). DEE proceeds by always fetching the node that is most likely to be executed next. While the SP scheme selects a single path, proceeding from the current “frontier” node to its more probably child (as given by the branch predictor), DEE selects one or more execution paths for the program, based on the corresponding prediction probabilities. Thus, DEE belongs to a class of schemes that use the *multiple path (MP)* approach. A simulation study [9] showed that DEE can improve the performance of SP when implemented on the Levo machine, using a 2-bit counter as a branch predictor.

1.3 Our Contribution

In this paper we study the performance of dynamic speculative execution schemes based on the multiple path approach. In particular, we quantify the superiority of DEE and other MP schemes over SP. A main contribution of the paper is in modeling the speculative execution of a program as the problem of selecting a ‘good’ path in a graph: we show that the *branch tree* model introduced in [9], combined with results from Markov decision theory, enable us to derive analytic performance bounds for an important class of schemes. To the best of our knowledge, all of the previous performance results for these schemes were empirical.

The *branch tree* model and associated definitions are given in Section 2. Section 3 presents our analysis of the *single-phase problem*; this is a sanitized version of our analysis, in which we

consider a fixed branch-tree given in advance. We show that DEE is optimal in this setting. The more realistic *adaptive* setting is the subject of Section 4; we show that DEE is no longer necessarily optimal, and argue (using Markov decision theory) that at least in principle, it is possible to give an optimal algorithm. For the special case where the average prediction probability is high (as reported, e.g., in performance studies of the Pentium Pro [7, 9]), we show that the SP scheme is within a factor of 2 from the optimal. Section 5 gives the results of our simulations on traces, using a number of branch tree generation models and a two level branch predictor [12], that relates our mathematical models to real program behavior.

2 Preliminaries

2.1 The Branch Tree Model

Given a program consisting of N conditional branch instructions, the set of the execution paths of the program can be represented by a binary tree T of size $\hat{N} \geq N$; we call this the branch tree for the program. Each internal node i represents an instance of a conditional branch; an edge (i, j) represents a basic block in the program, i.e., a sequence of non-branch instructions, that will be executed after branch i and before branch j . The leaves of T represent termination instructions of the program. Using a branch predictor we associate with each node i a *prediction probability*² $p(i) \geq 1/2$; this is the probability that the predictor makes the correct guess in node i . Note that many internal nodes in T may be associated with the same branch (i.e., a single instruction in the program). These nodes will generally have the same prediction probabilities, even though they are represented by different branch tree nodes.

Each node in T has also an *accumulated* probability $p_a(i)$. Let u be the parent of node i in T , then

$$p_a(i) = \begin{cases} 1 & \text{if } i \text{ is the root of } T, \\ p_a(u)p(u) & \text{if } i \text{ is a left child of } u, \\ p_a(u)(1 - p(u)) & \text{otherwise.} \end{cases}$$

An *online prefetching algorithm* bases its decisions only on the prediction probabilities of those branch nodes that it has speculatively prefetched. Initially, it has “prefetched” only the root and thus knows $p(1)$. At any subsequent instant, it can have at most l prefetched nodes in its pipeline. At all times, it can only prefetch a node that is a direct child of one of the k nodes currently in its pipeline, where $1 \leq k \leq l$.

We wish to compare such an online algorithm with a prescient algorithm that knows how every branch will be resolved. We henceforth refer to the prescient algorithm as the *offline algorithm*. Given its complete knowledge of the future, the offline algorithm will choose the (optimal) path in T and “speculatively” prefetch and execute only the code along this path.

In the following we define the two path selection problems studied in this paper.

²Throughout the paper we sometimes use *local* probability when referring to the prediction probability of a node.

2.2 Single-Phase Path Selection

For any $l > 1$ let $Off_path(l)$ denote a path of length l chosen by the offline algorithm, starting from r , the root of T . Let T_r^l denote a connected subtree of T of l nodes containing r . In general, the set of nodes of T speculatively executed by an online algorithm is such a connected subtree of T containing r . We define the expected benefit of T_r^l as

$$E[B_{T_r^l}] = \sum_{i \in T_r^l} Prob(i \in Off_path(l)) .$$

(This is the expected number of “useful” instructions in T_r^l .)

The **Single-Phase Path Selection (S_PS)** problem can be stated as follows:

Given a branch tree $T = (V, E)$ rooted in r , with $|V| = \hat{N}$, the set of local probabilities $(p(1), \dots, p(\hat{N}))$ and some integer $l > 1$, select a connected subtree T_r^l of T of l nodes, such that $E[B_{T_r^l}]$ is maximized:

$$E[B_{T_r^l}] = \max_{\tau_r^l \in T} E[B_{\tau_r^l}] ,$$

where τ_r^l is any connected subtree of T rooted at r containing l nodes.

2.3 Adaptive Online Path Selection

In our study of speculative execution schemes, we refer to the following extension of the S_PS problem. We are initially given a branch tree T rooted at r , and the parameters $n \geq l > 1$. At time $t = 0$, the offline algorithm selects a path of length n , given by $Off_path(n) = (r, v_1, \dots, v_{n-1})$ in $T = T(0)$. Each node on that path requires l time units of processing. At time $t = l$, the identity of v_1 is revealed to the online algorithm and v_1 becomes the root of the updated tree $T(l)$. Thus, every node to which there is no directed path from v_1 (together with its incident edges) is omitted from the tree and the online algorithm no longer considers these nodes.

We seek an algorithm \mathcal{A} that constructs the path (r, v_1, \dots, v_{n-1}) as follows:

- (i) In each time unit \mathcal{A} can add a single node to the selected path;
- (ii) Exactly l time units after \mathcal{A} adds v_{j-1} to its path, the identity of v_j is revealed to \mathcal{A} ;
- (iii) \mathcal{A} will modify its selected path $\mathcal{A_path}$ as the nodes on Off_path are revealed to it, until $\mathcal{A_path}(n) = Off_path(n)$.

Since each node requires l time-units for processing, an online algorithm aims to select each of the nodes on $Off_path(n)$ as early as possible.

Formally, let $d_j^{\mathcal{A}}$ denote the cost incurred by \mathcal{A} for processing v_j , $0 \leq j \leq n - 1$, where $v_0 = r$. The identity of v_j is revealed at time t_j , where $l + j \leq t_j \leq j \cdot l$, $j \geq 1$. If v_j has not been selected

by \mathcal{A} by time t_j , then \mathcal{A} incurs a fault on v_j and $d_j^{\mathcal{A}} = l$; otherwise, if \mathcal{A} selects v_j at time $t_j^{\mathcal{A}} < t_j$, then $d_j^{\mathcal{A}} = l - (t_j - t_j^{\mathcal{A}})$. Thus, the total cost incurred by an on-line algorithm \mathcal{A} for the processing of a path of length n is $D_n^{\mathcal{A}} = \sum_{j=0}^{n-1} d_j^{\mathcal{A}}$.

We now define the **Adaptive Path Selection (A_PS)** problem:

Given a branch tree $T = (V, E)$ with $|V| = \hat{N}$, the set of local probabilities $(p(1), \dots, p(\hat{N}))$ and the integers $n \geq l > 1$, adaptively select a connected subtree T_r^k of T for some $k \geq n$, using an online algorithm \mathcal{A} , such that $E[D_n^{\mathcal{A}}]$ is minimized.

3 Algorithms for Single-Phase Path Selection

3.1 Optimality of the DEE Scheme

In the following we show that algorithm DEE, which selects the l nodes in T with maximal accumulated probabilities, outperforms any other on-line algorithm for single-phase path selection.

Let $T_r^l(\mathcal{A})$ denote the connected subtree selected by algorithm \mathcal{A} . For brevity we define $E[B_l^{\mathcal{A}}] \equiv E[B_{T_r^l(\mathcal{A})}]$ to be the expected benefit of \mathcal{A} on a path of length l .

Theorem 1 *Given a branch-tree T and $l \geq 1$, for any on-line algorithm \mathcal{A} ,*

$$E[B_l^{\mathcal{DEE}}] \geq E[B_l^{\mathcal{A}}] . \quad (1)$$

Proof: For any $v \in V$, let

$$w(v) = p_a(v) \quad (2)$$

denote the weight of v . The proof uses the following *marking problem*: given the tree $T = (V, E)$, with weights as defined in (2), choose a subset of l nodes $\{v_{i_1}, \dots, v_{i_l}\}$, such that $\sum_{j=1}^l w(v_{i_j})$ is maximized. Indeed, the marking problem is a version of the S_PS problem, in which we relax the requirement that the set of selected nodes is a connected subtree of T . Now, renumber the nodes in non-increasing order by weights, and let $S = \{v_1, \dots, v_l\}$. Denote by \mathcal{A}_M the algorithm that chooses the nodes in S . Clearly, \mathcal{A}_M is optimal. Note that by the definition of the weights of the nodes, if u is an ancestor of v in T and $v \in S$, then $u \in S$. Hence, \mathcal{A}_M also gives an optimal solution for the S_PS problem.

Finally, the optimality of DEE follows from the fact that it selects a set of nodes S' that is a connected subtree of T , such that $\sum_{v \in S'} w(v) = \sum_{i=1}^l w(v_i)$.

□

In practice, an algorithm known as Single Path (SP) is commonly used for speculative execution of code; SP constructs a path of length l starting from the root, based on the predictor's decision for each branch. Algorithm SP is easy to implement, since at each branch it only needs to know

which child of the current node has the largest probability (rather than the exact value of $p(i)$). However, by Theorem 1, DEE performs better. In the following we quantify the improvement in the expected benefit obtained by using the DEE algorithm, compared to SP. The analysis of DEE entails an interesting process akin to the generation of Fibonacci numbers.

Theorem 2 *For any $l \geq 1$ and a branch tree $T = (V, E)$, if $p(i) = p \quad \forall 1 \leq i \leq |V|$, with p satisfying $1 < k \equiv \log_p(1 - p) < l$, then*

$$E[B_l^{\mathcal{D}\mathcal{E}\mathcal{E}}] = \sum_{n=0}^y f_k(n)p^n + p^{y+1}(l - b) , \quad (3)$$

where

$$f_k(n) = \sum_{r=0}^{\lfloor n/(k-1) \rfloor} \binom{n - r(k-1)}{r} \quad \forall n \geq 0 \quad (4)$$

and y is the largest integer satisfying

$$\sum_{n=1}^y f_k(n) = b \leq l . \quad (5)$$

Proof: Let

$$f_k(n) = \begin{cases} 1 & 0 \leq n \leq k - 1 \\ f_k(n-1) + f_k(n-k) & n \geq k. \end{cases}$$

We first show that if $p^k = 1 - p$ for some $1 \leq k < l$ then $f_k(n)$ is the number of paths in T with probability p^n , $1 \leq n \leq l$. We handle separately the two ranges of n (for simplicity of exposition we assume here that k is an integer):

- (i) For $1 \leq n < k$, since $p^n > 1 - p$, there is a single path of n left_child nodes.
- (ii) For $n \geq k$ we construct a path with probability p^n by adding a node to a shorter path with probability $p^{n'}$, $n' < n$. Specifically, we can either add a left_child node to a path with probability p^{n-1} or a right_child node to a path with probability p^{n-k} , which gives the required relation for $f_k(n)$.

It remains to show that $f_k(n)$ satisfies equation (4). This can be done by induction on n , using the relation $\binom{m}{r-1} + \binom{m}{r} = \binom{m+1}{r}$, that holds for any integers $m \geq r \geq 0$. \square

While DEE performs the same as SP when p is large, namely, whenever $p^l > 1 - p$, when p gets close to $1/2$, DEE becomes significantly better, and $E[B_l^{\mathcal{D}\mathcal{E}\mathcal{E}}]/E[B_l^{SP}] \rightarrow \log l$. The above result can easily be extended to the more general case where the branch tree T is generated as follows. Given a vector $\{p_1, \dots, p_s\}$ with $p_j \in [1/2, 1]$ for $1 \leq j \leq s$, and a vector $\hat{q} = (q_1, \dots, q_s)$, $0 \leq q_i \leq 1$, such

that $\sum_{i=1}^s q_i = 1$, $p(i)$ assumes the value p_j with probability q_j , independently of $p(k)$ for $k \neq i$. Let

$$\hat{p} = \sum_{j=1}^s p_j q_j . \quad (6)$$

By linearity of expectation, for such a randomly generated tree, the expected benefit for a path of length t , composed of r left edges and $(t-r)$ right edges is given by $\hat{p}^r (1-\hat{p})^{t-r}$, for any $1 \leq t \leq l$ and $0 \leq r \leq t$. Hence, for obtaining the expected benefit averaged over all these random trees we can replace T with the homogeneous tree $T' = (V', E')$ with $V' = V, E' = E$, in which $p(i) = \hat{p} \forall 1 \leq i \leq |V'|$. Thus, we have

Corollary 3 *If $\hat{p}^l > 1 - \hat{p}$ then*

$$E[B_l^{\mathcal{D}\mathcal{E}\mathcal{E}}] = E[B_l^{\mathcal{S}\mathcal{P}}]. \quad (7)$$

3.2 The Constrained Path Selection Problem

DEE as described above can be impractical: if the prediction probabilities are close to $1/2$, it continues prefetching from both sub-trees of many branch nodes. This results in an exponentially-growing number of “parallel” executions, with a concomitant growing demand on the number of functional units. In practice, however, there is a fixed number F of functional units in a processor. This begs the question: given F, l and a branch tree in which the prediction probability at every node is $p > 1/2$, what is the expected benefit of the best online algorithm? We now provide a characterization of the answer to this question.

Consider the set S of points (a, b) in the positive quadrant of the plane, where a and b are non-negative integers. Each node v of a binary branch tree can be mapped (many-to-one) to a point in S by the following mapping f : if the path from the root to v follows a left-branches and b right-branches, then f maps v to (a, b) . The root is mapped to the origin, and in general f maps $\binom{a+b}{a}$ points of the branch tree to (a, b) . Consider now any line-segment from $(x, 0)$ to $(0, y)$, for positive reals x, y . Let $\Delta(x, y)$ denote the lattice (i.e., integer coordinate) points of S in the triangle bounded by this line-segment and the axes. Let $n(x, y)$ denote the number of branch-tree nodes mapped by f into the points of $\Delta(x, y)$. Now consider a pair of integers x, y such that $n(x, y) = l$. The pre-image (in the branch-tree) of any point of $\Delta(x, y)$ is a feasible solution to the path-selection problem for pipe length l . Indeed, let

$$R(l) = \bigcup_{x, y | n(x, y) = l} \Delta(x, y)$$

denote the union of all such feasible points (although we have allowed x and y to be reals, the number of such feasible points is in fact clearly finite; thus the pairs (x, y) fall into equivalence classes for which $\Delta(x, y)$ is the same). Then, the pre-images for all $x, y \in R(l)$ are precisely the set of all feasible solutions to the path-selection problem for pipe length l . (As special cases, these feasible solutions include SP and DEE.) Each such feasible solution has an associated benefit, as well as a number of leaves. We pick the feasible solution with maximum benefit among all solutions having at most F leaves.

The optimality of this rule follows from two observations: (1) Any solution to the path-selection problem for pipe length l maps (under f) to a point in S such that $n(x, y) \leq l$, and is thus in $R(l)$. Hence, the set of solutions we consider is precisely the set of all feasible solutions. (2) Amongst all of the feasible solutions, we are choosing the one with maximum expected benefit.

4 Algorithms for Adaptive Path Selection

4.1 The Optimal Scheme

While DEE is optimal for the S_PS problem, we now show its suboptimality for the A_PS problem.

Example 4.1 Consider the branch tree $T = (V, E)$, where

$$V = \{r, v_1, v_2, v_3, v_4, v_5, v_6\}$$

and

$$E = \{(r, v_1), (r, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_2, v_6)\} .$$

Let $n = l = 3$, and suppose that we start fetching r at time $t = 0$. Then at $t = 3$ the execution of r is terminated and we know whether the offline algorithm chose v_1 or v_2 .

Assume that the local probabilities of the nodes $\{r, v_1, v_2\}$ are $\{p, p, 1\}$ respectively, where $p > 1/2$ and satisfies $p^2 < 1 - p$.

At time $t = 1$ DEE chooses the node v_1 and at $t = 2$ the node v_2 . Thus,

$$E[D_3^{\mathcal{D}\mathcal{E}\mathcal{E}}] = p(6p + 7(1 - p)) + 6(1 - p) .$$

Consider an algorithm \mathcal{A} , which chooses the node v_1 at $t = 1$, and the node v_3 at $t = 2$, then

$$E[D_3^{\mathcal{A}}] = p(5p + 6(1 - p)) + 7(1 - p) .$$

and $E[D_3^{\mathcal{A}}] < E[D_3^{\mathcal{D}\mathcal{E}\mathcal{E}}]$ for any $1/2 < p < \frac{\sqrt{5}-1}{2}$.

We now argue that, in principle, it is possible to devise an algorithm for the A_PS problem. Unfortunately, this argument does not lead to an efficient (and certainly not a practical) algorithm. The state of an online algorithm at time j consists of a connected subtree of T of size $\leq l$ rooted at v_j . Given this state, the online algorithm faces a set of (up to $l/2$) choices of which node to prefetch next. The prediction probabilities $p(i)$ give the expected benefit for each choice. Thus, the problem of adaptive path selection for the online algorithm may be viewed as a Markov decision process [2]. By standard results in Markov decision processes [2], it follows that the online algorithm's policy is deterministic (i.e., whenever in the course of execution it encounters the same subtree with the same probabilities, its prefetching policy is the same). Moreover, this policy can be computed, in general, using a linear program: there is a variable $x_{(T', v)}$ for each combination (T', v) , where T' is a connected subtree of size $\leq l$ rooted at v_j , and v is one of (upto $l/2$) choices of the node v to be prefetched next. The variable (which is constrained to assume values in $[0, 1]$) specifies the

probability with which to select v to prefetch, given that its current state (connected subtree of prefetched nodes) is T' . The constraints in the linear program ensure that $\sum_v x_{(T',v)} = 1$, the summation being over the possible v that can be prefetched from T' (thus ensuring that some node v will be chosen). The objective function seeks to minimize the expected delay $E[D_n^A]$, which is a linear function of the values $x_{(T',v)}$. Thus, we have:

Theorem 4 *For a given branch tree T , and any $n \geq l > 1$, the problem of computing the optimal on-line algorithm for adaptive path selection is a problem of linear programming in $\Omega(nb_l)$ variables,*

where b_l is the l -th Catalan number, given by $b_l = \frac{1}{l+1} \binom{2l}{l}$.

4.2 Application to Speculative Execution of Code

In this section we present results that apply to branch trees with high prediction probabilities. The following implies that if only a small fraction of the branches in the program are hard to predict, i.e., the average prediction probability is relatively high, then the SP scheme is close to the optimal for a wide range of values for l and n .

Theorem 5 *Let T be a branch tree such that for any $v \in V$, $p(v) \in \{p_1, \dots, p_s\}$, with the corresponding frequency vector $\{q_1, \dots, q_s\}$. Let \hat{p} be defined as in (6). If*

$$\hat{p}^l > 1 - \hat{p}, \tag{8}$$

then SP is within a factor of 2 from the optimal online algorithm for the A_SP problem.

Proof: Consider first the expected benefit of SP in solving the single-phase path selection problem. By Corollary 3, for any value of \hat{p} satisfying (8), SP is optimal. For obtaining the bound for A_SP, we consider a ‘lazy’ variant of SP, denoted by SP_L. We partition the execution of the program to iterations, each of length $2l$ time units. In the first l time units of the i -th iteration, SP_L selects a path of l vertices, starting from the first missing vertex. Then, SP_L stalls for l time units, after which the next missing vertex is revealed; at this time the iteration ends and SP_L starts iteration $(i + 1)$. Clearly, SP outperforms SP_L, since SP never stalls if it can fetch a new vertex. Hence, we can use SP_L to obtain a bound for SP. Note that in each iteration, any algorithm (and in particular an optimal one) has an expected benefit of at most twice the expected benefit of SP_L. This is due to the optimality of SP_L in the first l steps. This yields the bound in the theorem. \square

The next lemma will be useful in our empirical study of SP in the random model. The proof is straightforward, and is thus omitted.

Lemma 6 *For any $n > 1$ and $\hat{p} \in [1/2, 1]$,*

$$E[D_n^{SP}] = l + (n - 1)[\hat{p} + l(1 - \hat{p})] \tag{9}$$

5 Experimental Results

5.1 Model Description

We simulated the behavior of different dynamic schemes for speculative execution on conditional branch commands, in a set of benchmarks from the SPECint95 suite (including Compress, Gcc, Go, Lisp and Perl). We used trace files of branch commands in those programs, which provided for each branch the following information: Branch address, branch type (call, return, unconditional/conditional branch) and program behavior for this branch (taken/not taken).

The offline algorithm’s behavior was determined by information from the trace files: given a trace, the offline algorithm knows in advance how every branch will be resolved, so it can select and execute only the code along this (optimal) path.

Given an online algorithm \mathcal{A} , we experimentally compared the throughput of \mathcal{A} to the throughput of the offline algorithm, i.e., we studied the ratio

$$\frac{\text{offline algorithm's time to execute } N \text{ branches}}{\mathcal{A}'\text{s time to execute } N \text{ branches}}.$$

We simulated three dynamic schemes: SP, DEE and the following hybrid scheme (HYB): Given the *threshold parameter* $0.5 < p' < 1$, let $L(T_r^s)$ denote the set of leaves in the subtree of T chosen by HYB in the first s time units. Then HYB visits each of the leaves in $L(T_r^s)$ from left to right and for each node $i \in T_r^s$, if $p(i) < p'$, HYB fetches the two children of i in T , otherwise it fetches only the left child.

We examined three possible program structure models:

- The HOMOGENEOUS (FIXED) model: All the conditional branches in the program have the same prediction probability.

In the next two models the program contains *hard* and *easy-to-predict* branches.

- The RANDOM model: The hard branches are distributed randomly and uniformly along the program. Instances of the same branch have the same probability.
- The DENSITIES model: Hard branches are grouped together to *hard branch regions*, i.e., these branches appear consecutively in the program. For a program containing N hard branch instructions, the number of such regions is given by the parameter *density*. When *density* = i there are i regions, each containing N/i hard branches. The starting points of these regions are distributed uniformly along the program.

5.2 Performance Results

We first describe the set of base parameter values used in the simulation. For each program we obtained results of 100 runs, each on 1000 conditional branch commands. The results displayed in

the figures are the average of those 100 runs on the various programs. The threshold parameter used for HYB was $p' = 0.7$.

The pipe length is the maximal number of conditional branches that can be simultaneously in the pipe at any time. This length can take different values, as indicated in the graphs. In Figures 2-10 we describe the main results of our experiments. In all the plots, as the length of the pipe increases, the performance of the three schemes decreases relative to the optimum. This is due to the fact that the cost of faults in long pipelines is higher. As expected, for a given pipe length, an improvement in prediction probability improves the schemes' performance.

Figures 2-7 present the results obtained when we ran our simulator using a synthetic predictor, for which we could tune the prediction probability in each branch. This enabled an accurate implementation of our three models. In these experiments, the percentage of conditional branches was set based on the required prediction probability and program structure model. Unless otherwise indicated, the average prediction probability used for each program was 0.94 (where the average is taken over all branches – conditional as well as unconditional). The prediction probability of hard branches was set to 0.65, and for easy branches 0.95 (with the exception that when the required average prediction probability was smaller than 0.8, the prediction probability for hard branches was 0.5 and for easy branches 0.8). In the HOMOGENEOUS model, all branches were assumed to be easy and associated with the same prediction probability.

In Figures 8-10 we plotted the results obtained when the simulator used a Two Level Adaptive Branch Predictor (his standard predictor is implemented, for instance, on the Pentium Pro [7]).

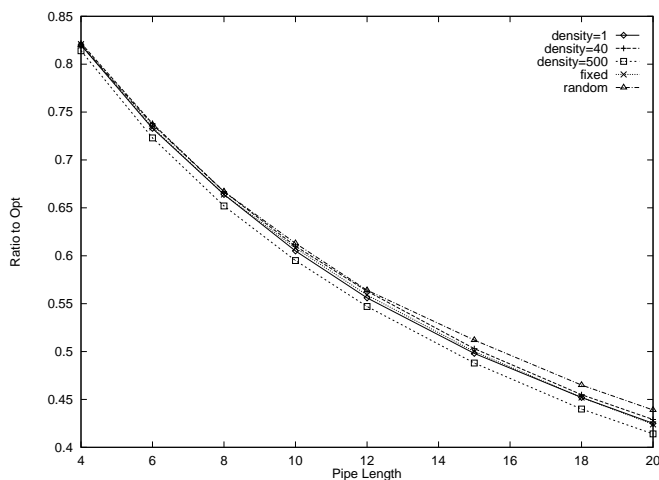


Figure 2: Performance of SP

In Figure 2 we show the performance of SP for the three program structure models. Note that SP is not affected by program structure. Indeed, SP's behavior at each branch is determined only by information about that branch, regardless of the branch environment. Later we show that for programs structured as in the DENSITIES model, the HYB scheme achieves similar performance for all densities of hard branches (see Figure 6). These two facts enable us to compare the performance of the SP and HYB schemes for structured programs, using *any* density: we chose $density = 1$, as shown in Figure 3.

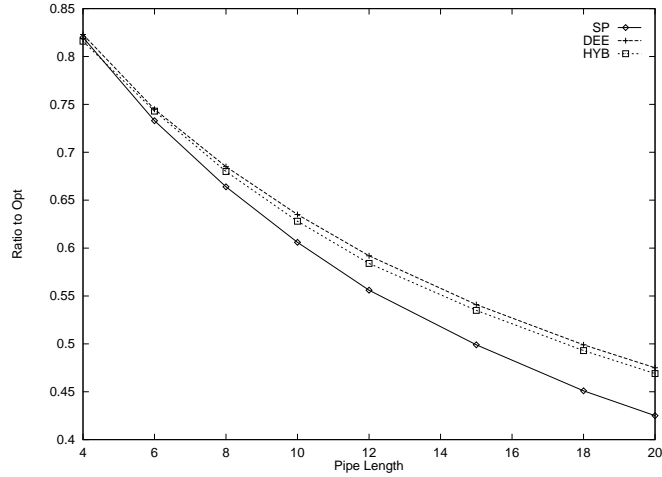


Figure 3: Comparison of SP, DEE and HYB in the DENSITIES model (density=1)

Figure 3 compares the performance of the MP schemes to the SP scheme for varying pipe lengths and a structured program. Note that the MP schemes (DEE and HYB) improve on SP's performance for sufficiently long pipelines (e.g., for pipe length 20, MP schemes increase the performance by about 5%). Clearly, when DEE and HYB perform similarly, the HYB scheme is preferable, since it is much easier to implement. Later we show that DEE's performance improves when the hard branches appear in a large number of small blocks along the program (see Figure 7). This means that for less structured programs and long pipelines DEE is a better choice.

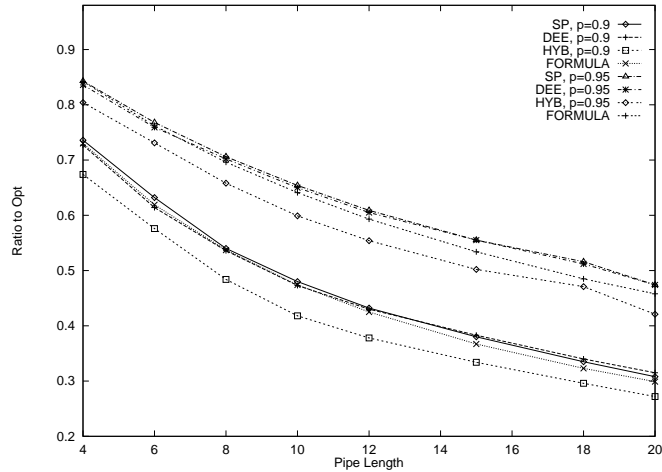


Figure 4: Comparison of SP, DEE and HYB in the RANDOM model

In Figure 4 we present the performance of the various schemes in the RANDOM model, for high prediction probabilities and varying pipe lengths. We also plotted the expected runtime of SP, as computed in (9). In this model, DEE and SP are almost identical. This is due to the high prediction probabilities and the fact that there are no large regions of hard branches; thus DEE is not eager

in most of the branches. As for the HYB scheme, we see that its performance in the RANDOM model with high average prediction probabilities is worse than the performances of SP and DEE. This is due to the wastefulness of HYB caused by its unlimited eagerness whenever a hard branch is encountered.

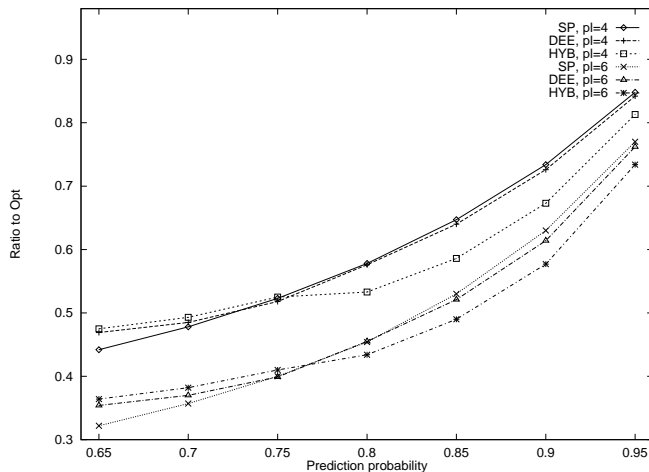


Figure 5: Comparison of SP, DEE and HYB in the RANDOM model, with varying prediction probabilities

Next we study the performance in the RANDOM model for short pipelines and varying prediction probabilities (Figure 5). We conclude that when the average prediction probability is low, HYB and DEE are similar and yield higher throughput compared to SP (HYB becomes less wasteful, since there is a greater chance for large regions of hard branches).

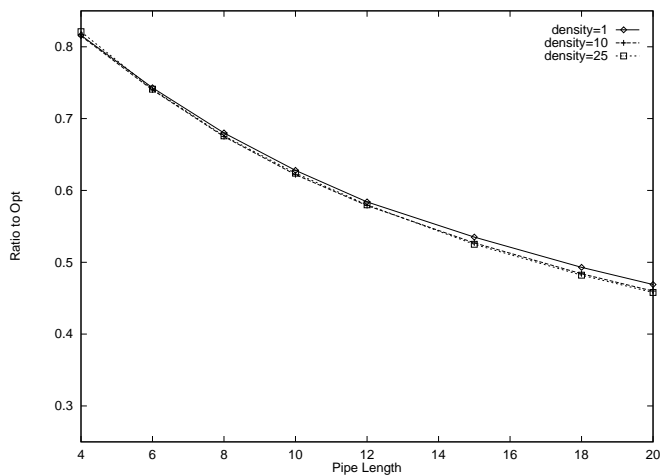


Figure 6: Performance of HYB in the DENSITIES model

Figures 6 and 7 provide two conclusions (mentioned above) on the performance of HYB and DEE in the DENSITIES model: Figure 6 shows that for structured programs, the density of hard branches almost does not affect the performance of HYB. Indeed, like SP, HYB handles each branch

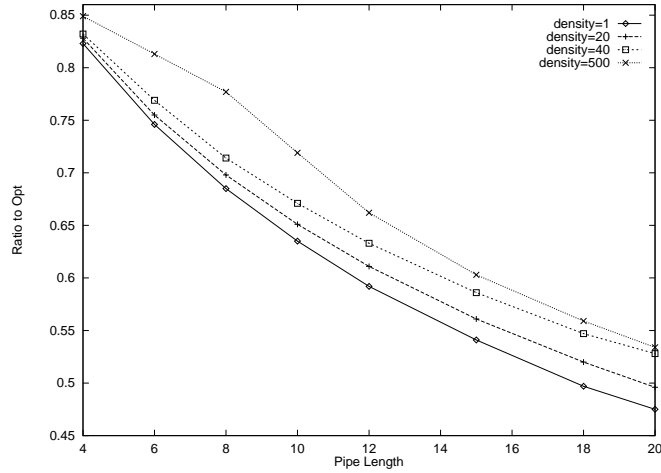


Figure 7: Performance of DEE in the DENSITIES model

locally, not taking into account the branch environment. In contrast, DEE is affected by the branch environment and thus, the performance changes with the change in density (see in Figure 7). As the density of hard branches decreases (more regions of hard branches), DEE is less eager and performs better. Note that the improvement in performance increases with the length of the pipe.

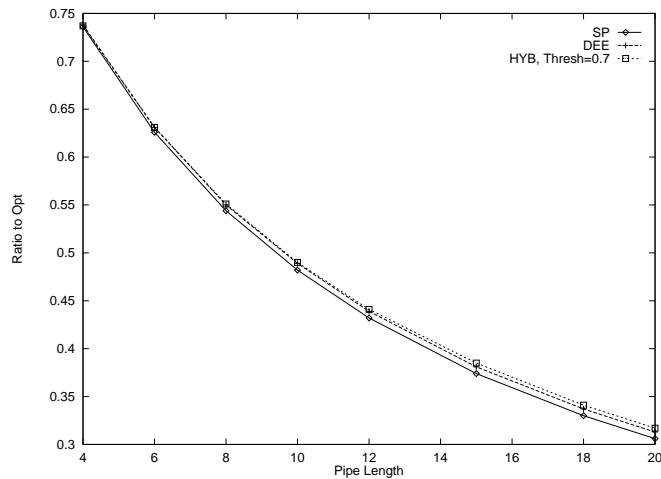


Figure 8: Performance of SP, DEE and HYB on Compress, with two level predictor ($\hat{p} = 0.9$)

In Figures 8-10 we compared the performances of the three schemes on several programs from the SPECint95 benchmark, using a two level branch predictor. We give the results for three programs (Compress, Lisp and Perl): the results for other programs were similar. From these figures we note that the simulation results with a standard predictor are very similar to the results obtained for the RANDOM model. In addition, the average prediction probabilities in all programs satisfied condition (8), for any pipe length in the range $[4, 20]$. We conclude that Theorem 5 applies for these programs.

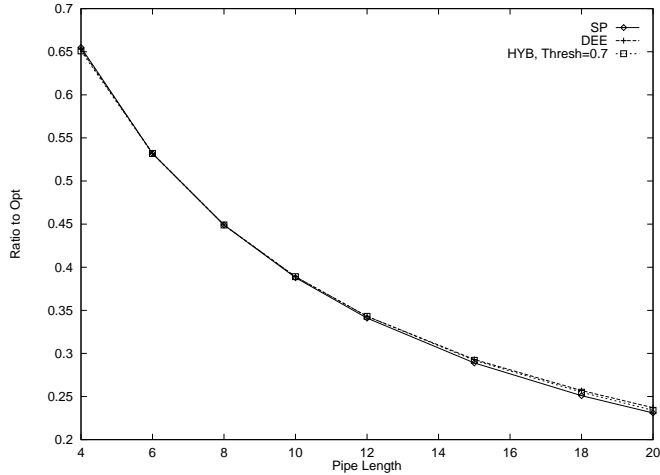


Figure 9: Performance of SP, DEE and HYB on Lisp with two level predictor ($\hat{p} = 0.86$)

6 Discussion

We have studied analytically and empirically the performance of three speculative execution schemes. Our analytic results show that SP, commonly used in modern processors, outperforms any other scheme when the average prediction probability is high and the RANDOM model applies. Our experimental study shows that the RANDOM model can provide a close approximation to the distribution of branches in real programs.

New developments. Recently, Gaysinsky et al. [3] considered the problem of *caching with locality and pipelined prefetching (CLPP)*, which generalizes the A_PS problem. The results in [3] for the Markovian CLPP can be applied to improve the result of Theorem 5. Specifically, it can be shown that when \hat{p} satisfies (8), SP is within factor of $1 + o(1)$ from the optimal in the set of online algorithms for the A_PS. Also, the paper [3] shows that DEE is optimal to within factor of 2 in solving the A_PS problem with *arbitrary* prediction probabilities, thus answering a question posed in an earlier version of the present paper.

Future work. We mention below several possible directions for future work.

- We considered here three models of program structure. It would be interesting to extend this study to other models (e.g., a RANDOM model in which the distribution of the hard branches along the program is *non-uniform*).
- We measured the performance of the various schemes based on the prediction probability parameter. Other factors, such as prefetch time (which is lower for instructions that are available from the cache) need to be considered.

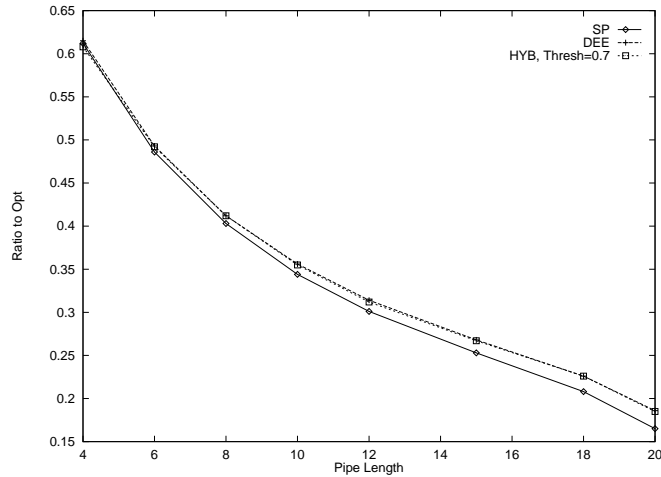


Figure 10: Performance of SP, DEE and HYB on Perl with two level predictor ($\hat{p} = 0.86$)

- Finally, we have studied an execution model, in which there is a single instruction in each stage of the pipeline. Using parallelism can significantly improve the speedup obtained by pipelining alone.

Acknowledgments

We thank Oded Lempel and Ilan Spillinger for many stimulating discussions, throughout the research work that led to the results presented in this paper.

References

- [1] D. Bhandarkar and J. Ding. Performance Characterization of the Pentium Pro Processor. *Proc. of the 3rd International Symposium on High Performance Computer Architecture*, San Antonio, 1997.
- [2] C. Derman. *Finite State Markov Decision Processes*. Academic Press, New York, 1970.
- [3] A. Gaysinsky, A. Itai and H. Shachnai. Strongly Competitive Algorithms for Caching with Pipelined Prefetching. *Proc. of the 9th Annual European Symposium on Algorithms*, 2001.
- [4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.
- [5] A. R. Karlin, S. Phillips and P. Raghavan. Markov Paging. *Proc. of the 33rd Symposium on Foundations Of Computer Science*, Pittsburgh, PA, 208–216, 1992.
- [6] D. J. Lilja. Reducing the Branch Penalty in Pipelined Processors. *IEEE Computer*, July 1988, pp.47–55.

- [7] G. Linley. New Algorithm Improves Branch Prediction; Better Accuracy Required for Highly Superscalar Design. *Microprocessor Report Journal*, March 1995.
- [8] D. A. Patterson and J. L. Hennessy. Computer Organization & Design, the Hardware/Software Interface. Morgan Kaufmann Publishers, INC., 1997.
- [9] A. K. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. *Proceedings of MICRO-28*, 1995.
- [10] S. S. H. Wang and A. K. Uht. Ideograph/Ideogram: Framework/Architecture for Eager Evaluation. *Proceedings of MICRO-23*, 1990.
- [11] M. Yaniv. Dynamic Schemes for Speculative Execution of Code. M.Sc. Thesis, Dept. of Computer Science, The Technion, 1998.
- [12] T-Y Yeh and Y. N. Patt. Two-Level Adaptive Branch Prediction. *Proceedings MICRO-24*, 1991.