Statistical Modeling of Control Flow for Speculative Helper Thread Optimization

TOR M. AAMODT, University of British Columbia
PEDRO MARCUELLO, Intel Barcelona Research Center, Intel Labs—UPC
PAUL CHOW, University of Toronto
ANTONIO GONZÁLEZ, Intel Barcelona Research Center, Intel Labs—UPC
PER HAMMARLUND, Digital Enterprise Group, Intel Corp.
HONG WANG, Microarchitecture Research Lab, Intel Corp.
and
JOHN P. SHEN, Nokia Corp.

This paper describes a framework for modeling program behavior and applies it to optimizing prescient instruction prefetch—a novel technique that uses helper threads to improve single-threaded application performance. Spawn-target pair selection is optimized by modeling program behavior with Markov chains and analyzing them with path expression mappings. Mappings for reaching, and posteriori probability; path length mean, and variance; and path footprint are presented. A limit study demonstrates speedups of 4.8% to 17% for benchmarks with high I-cache miss rates. The framework has been applied to data prefetch helper threads and is potentially applicable to other thread speculation techniques.

Categories and Subject Descriptors: C.0 [General]: Modeling of computer architecture
General Terms: Algorithms, Measurement, Performance, Design, Theory
Additional Key Words and Phrases: Analytical Modeling, Helper Threads, Optimization

1Extension of Conference Paper: An earlier version of the work in this paper appeared in the proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems [Aamodt et al. 2003]. The new material in this paper consists of: (1) An extension of the case study of control flow statistics in a large function in 186.crafty in Section 5.2 to include (a) data demonstrating that higher-order control flow graphs [Aamodt 2006; Aamodt and Chow 2007] can accurately model the effects of branch outcome correlation, and (b) data showing the sensitivity of reaching probability to program inputs; (2) Detailed derivations of the path expression operators for expected path length and variance in Appendix A.

Contact author: Tor M. Aamodt, Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, Vancouver, B.C. V6T 1Z4, CANADA. E-mail: aamodt@ece.ubc.ca; This research was conducted while Tor and John were at Intel Corporation. Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

As the gap between processor and memory speed continues to widen, performance is increasingly determined by the effectiveness of the cache hierarchy. Prefetching is a well-known technique for improving the effectiveness of the cache hierarchy. We investigate the use of spare simultaneous multithreading (SMT) [Tulsen et al. 1995; Emer 1999; Intel Corporation] thread contexts for prefetching instructions. On an SMT machine, a thread context may go unused if the operating system cannot find useful work for it. Even though SMT has been shown to be an effective way to boost throughput performance with limited impact on processor die area [Hinton and Shen 2001] the performance of many single-threaded applications does not benefit from SMT. A number of proposals have been put forth to exploit SMT resources to improve the latency of single-threaded applications [Dubois and Song 1998; Chappell et al. 1999; Zilles and Sohi 2000; Roth and Sohi 2001; Moshovos et al. 2001; Zilles and Sohi 2001; Annavaram et al. 2001; Luk 2001; Collins et al. 2001; Collins et al. 2001; Chappell et al. 2002; Liao et al. 2002; Kim et al. 2004; Wang et al. 2004; Zhang et al. 2007]. In particular, several studies have investigated using helper threads in the form of program-slice based precomputation for reducing latency due to load instructions tending to miss in the cache, and branches that mispredict. Prior to this work [Aamodt et al. 2003], there had been little if any published work that focused specifically on improving I-cache performance by exploiting helper threads. This paper describes a novel framework for modeling such helper threads so as to optimize their impact on performance. The framework we describe uses path expressions [Aamodt et al. 2003], and while it has been shown to also be applicable to data prefetch helper threads [Aamodt 2006; Aamodt and Chow 2007], we focus here on laying sound theoretical underpinnings for using path expressions to analyze program behavior using instruction prefetch helper threads as the motivation. Elsewhere, we have explored hardware support for realistic implementations of prescient instruction prefetch helper threads [Aamodt et al. 2004].

One task common to both helper thread approaches as well as to speculative multithreading [Sohi et al. 1995; Steffan and Mowry 1998; Marcuello et al. 1998; Akkary and Driscoll 1998] is the selection of a trigger point where a helper thread should be spawned off. As a matter of convenience, we call this trigger the spawn point. In contrast to helper threads for loads and branches, for instruction prefetch the target identifies not just a single instruction, but rather the beginning of an entire region of code that the main thread is likely to execute soon. A similar notion of a spawn-target pair can be found in speculative multithreading, where a thread is forked to speculatively execute a future portion of the program with the goal of quickly reusing the architected state it generates when the main thread catches up with it. Traditionally, control flow idioms such as loops and procedure calls have been exploited for identifying spawn-target pairs. Generalizing the near control equivalence between spawn and target common to such idioms, Marcuello and Gonzalez [Marcuello and Gonzalez 2002] proposed using a reaching probability threshold to define a far larger set of candidate spawn-target pairs from which thread-level parallelism may be harvested. They considered refining the selection on the basis of maximal spawn-target distance, minimum inter-thread dependencies, and value predictability.

ACM Journal Name, Vol. V, No. N, Month 20YY.
By using a spare thread context to speculatively execute a future, probabilistically control-equivalent region of the program, far ahead of the main thread, prescient instruction prefetch can bring most instructions into the first-level cache likely to be encountered soon by the main thread, before it needs them. As prescient instruction prefetch speeds up the main thread by hiding fetch latency, rather than updating the main thread’s architected state, it needs to accurately execute only those instructions pertaining to control-flow.

In this paper, we present a framework for analyzing traditional control-flow and I-cache profile information to judiciously select spawn-target pairs for instruction prefetch. The key ideas, are: (1) estimating the execution time benefit of a helper thread defined by a particular spawn-target pair to guide the overall spawn-target pair selection process, by (2) using a Markov model of control flow to enable the evaluation of statistical properties of program execution over the set of all paths between any pair of points in the program. Novel path expression mappings are applied to leverage Tarjan’s fast-path algorithm [Tarjan 1981b]. Figure 1 depicts the overall framework for optimization. Using this framework, we propose and evaluate a simple optimization algorithm. Furthermore, we demonstrate that instruction prefetch helper threads constructed for the selected spawn-target pairs can achieve significant performance improvement.

The rest of this paper is organized as follows: Section 2 introduces the prescient instruction prefetch paradigm. Section 3 describes the framework for characterizing program behavior, formulates a set of key statistical quantities related to helper thread execution, and details a novel approach to use path expressions to compute these statistical quantities. Section 4 describes an algorithm using this...
framework to perform spawn-target pair selection for prescient instruction prefetch helper threads. Section 5 presents simulation results, and performance analysis. Section 6 summarizes related work. Finally, Section 7 concludes.

2. PRESCIENT INSTRUCTION PREFETCH

Prescient instruction prefetch uses helper threads that perform instruction prefetch to reduce stall cycles due to I-cache misses for single-threaded applications. Here the term prescient carries two connotations: One, that the helper threads are initiated in a timely and judicious manner based upon a profile-guided analysis of program’s global behavior, and two, that the instructions prefetched are useful as the helper thread follows the same control-flow path through the program that the main thread will follow when it reaches the code that the helper thread has prefetched.

Profile information is used to identify code regions that incur significant I-cache misses in the single-threaded application; candidate target points are then identified as instructions closely preceding the basic blocks that incur the I-cache misses. For each candidate target identified in the single-threaded application, a set of corresponding spawn points are found that can serve as trigger points for initiating the execution of a helper thread for prefetching instructions beginning at the target. Once a spawn-target pair is identified, a helper thread is generated and attached to the original program binary (i.e., the main thread). At run time, when a spawn point is encountered in the main thread, a helper thread can be spawned to begin execution in an idle thread context. The execution of the helper thread effectively prefetches for anticipated I-cache misses along a control flow path subsequent to the target.

Figure 2 illustrates these concepts by highlighting a program fragment containing three distinct control-flow regions. In particular, the region following the target is called the postfix region, and is assumed to suffer significant instruction cache misses. The region before the spawn is called the prefix region, and the region between the spawn and target the infix region. The postfix region is limited to instructions within a short distance from the target.

A register, or memory location is said to be ‘live-in’ to a region, if its value is read before being written in that region. In general, the helper thread may need to precompute some live-in values before starting to execute the program code in the postfix region [Aamodt et al. 2003; Aamodt et al. 2004; Quiñones et al. 2005]. Since effective instruction prefetch requires accurate resolution of branches in the postfix region, the precomputation consists of the backward slice of these branches.

As shown in Figure 2(b), helper thread execution consists of two phases. The first phase, live-in precomputation, reproduces the effect of the code skipped over in the infix that relates to the resolution of branches in the postfix. We refer to these precomputation instructions as the infix slice. Instructions in the infix region that are not related to any branch instruction in the postfix region are not executed. This is the key aspect that allows the helper thread to run faster and reach the target point earlier than the main thread does. In the second phase, the helper thread executes the same code in the postfix region that the main thread will when the main thread reaches the target. The computation in the second phase both resolves control-flow for the helper thread in the postfix region, and prefetches
instructions for the main thread. The helper thread is terminated after it finishes executing the postfix region, or when the main thread catches up with it.

The prescient instruction prefetch mechanism differs from traditional branch predictor based instruction prefetch mechanisms such as [Reinman et al. 1999], in that it uses a global view of control-flow gained from a program profile and is thus able to anticipate potential performance degrading regions of code from a long-range (e.g., the distance between spawn and target could be thousands of dynamic instructions). The key challenges are two-fold: One, identification of an appropriate set of distant yet strongly control-flow correlated pairs of spawn-target points; and two, accurate resolution of branches in the postfix region.

To highlight the challenges and motivate intuitive insights, we consider a brief example.

**Example: Spawn-Target Selection Tradeoffs**

Figure 3(a) depicts a control-flow graph fragment. Nodes represent basic blocks, and edges represent potential control flow transfer between basic blocks. The shaded block labeled x is known from cache profiling to suffer many instruction cache misses. Each edge is labeled with the probability that the main program will make the respective control flow transfer unless the value is exactly one. Block e calls subroutine foo(). The questions of interest are: (1) which locations are the best to spawn a helper thread to prefetch x, and (2) what should the target be?

Note that, starting from any location, the program is very likely to reach x, because every iteration the probability of exiting the loop (i.e., from block d) is much smaller than the probability of transitioning to x (i.e., from node b). Even though any choice of spawn is roughly as “good” as any other in this particular sense, as we show next, not all spawn points are necessarily as effective as others.

For instance, consider the impact the size of subroutine foo() has on spawn-target pair selection. If foo(), together with blocks b, d, e, and x fit inside the instruction cache, then initiating a prefetch of block x starting at block a is effective, because the loop will likely iterate several times before the main thread branches from b to access x. On the other hand, if foo() together with all of its callees (if any) require more instruction storage than the capacity of the cache, then there is no point in spawning a prefetch thread to target x from any block. The reason is...
that instructions prefetched by the helper thread will almost certainly be evicted by an intervening call to `foo()` before the main thread can access them due to the low probability of branch `b` transitioning to `x`. A better target in this situation would be block `b`, because evaluating the branch at the end of block `b`, would cause the helper thread to prefetch `x` only when it is about to be executed by the main thread. A good set of spawn points targeting `b` in this case might include the beginning of blocks `a`, and `d`.

In the latter case, if we can choose only one spawn location from `a` or `d`, due to resource limitations, the more profitable choice is `d`, because `a` would only cover misses at `x` in the event control goes directly from `a` to `b` to `x`, while `d` would cover the cache misses at `x` in all other cases.

As this example demonstrates, it is important to accurately predict the run time properties of helper threads when selecting spawn-target pairs. To help tackle such challenges, Section 3 formulates an analytical framework for rigorously quantifying the tradeoffs involved in spawn-target pair selection.

3. ANALYTICAL FRAMEWORK

To be effective, spawn-target pairs should meet the following necessary conditions: One, the corresponding helper thread must run ahead of the main thread, but not so far ahead as to evict instructions soon to be used by the main thread. Two, the spawn and the target should be highly control-flow correlated during the main thread’s execution. In other words, when the main thread reaches the spawn point, it will very likely see the target soon thereafter, and correspondingly, whenever the main thread executes the target, it should have recently been at the spawn point. Three, the helper thread should follow the same postfix path that the main thread will execute after it reaches the target.

In this section an analytical framework that models program behavior as a Markov chain is introduced. This framework consists of a set of key statistical quantities characterizing the first two necessary conditions, and is thus essential to the selection of effective spawn-target pairs. In addition, a novel technique for transforming the computation of these quantities to Tarjan’s classic path expression algorithm [Tarjan 1981b] is described, thus leveraging the latter’s efficiency. The resulting framework is used as the foundation for a simple spawn-target pair selection algorithm described in Section 4 (see Figure 5).

3.1 Single Helper Thread Instance

To characterize the distance that a helper thread runs ahead of the main thread, we model the prefetch slack of an instance of instruction `i`, targeted by a helper thread spawned at `s` with target `t`, using the following expression (illustrated in Figure 3(b)):

\[
\text{slack}(i, s, t) = \frac{A}{B}
\]

where

\[
A = (\text{CPI}_m(s, t) \cdot d(s, t) + \text{CPI}_m(t, i) \cdot d(t, i))
\]

\[
B = \text{CPI}_h(t, i) \cdot d(t, i) - o(s, t)
\]

(1)
where $CPI_m(s, t)$ represents the average cycles per fetched instruction for the main thread when traversing the infix region between this particular instance of $s$ and $t$; $CPI_m(t, i)$ represents the average cycles per fetched instruction for the main thread in the postfix region between $t$ and $i$; $CPI_h(t, i)$ represents the average cycles per fetched instruction for the helper thread between $t$ and $i$; $d(x, y)$ denotes the distance between instructions $x$ and $y$, measured in number of instructions executed; and $o(s, t)$ represents the overhead, in cycles, incurred by the helper thread of spawning at $s$ and performing precomputation for live-ins at $t$. A given instance of a helper thread can reduce the I-cache miss latency of an instruction access in the main thread if the prefetch slack for this instruction is positive.

The amount of prefetching any helper thread can perform, and therefore the maximum extent of the postfix region it can target, are limited by two factors. First, prefetching will improve the average IPC of the main thread, but, since it executes the same instructions as the main thread in the postfix region, the IPC of helper thread does not improve, leading to an upper bound on how far ahead a helper thread can run before the main thread will catch up to it. In particular, the main thread will catch up with the helper thread when

$$
\text{slack}(i, s, t) = 0
$$

in Equation 1 or,

$$
\begin{align*}
d(t, i)_{\text{max}} &= \frac{CPI_m(s, t) \cdot d(s, t) - o(s, t)}{CPI_h(t, i) - CPI_m(t, i)} \\
&= d(t, i)\left(1 - \frac{CPI_h(t, i)}{CPI_m(t, i)} \right) + \frac{o(s, t)}{CPI_m(t, i)}
\end{align*}
$$

Figure 3(b) portrays a graphical representation of this concept by plotting instructions executed versus time for a portion of a program’s execution trace. The solid line indicates the progress of the main thread, and the dotted line indicates the progress of the helper thread after overhead $o(s, t)$ due to thread invocation and live-in precomputation. The slack of one particular instruction $i$ is shown. The helper thread ceases to provide useful prefetch slack when the main thread catches up to it. This is indicated by the point where the dotted line intersects the solid line. The distance computed in Equation 2 corresponds to the height of the shaded box in Figure 3(b).

The other factor limiting the postfix region is the size of the infix slice required for target live-in precomputation. The size of the infix slice depends upon the amount and type of code skipped between the spawn and target, and the number of branches in the postfix region. The properties of such infix slices have been explored in related work [Aamodt et al. 2004; Quiñones et al. 2005]. As the distance between spawn and target grows, the number of operations in the infix that may affect the outcome of a branch in the postfix increases. On the other hand, given a fixed spawn-target distance, increasing the size of the postfix requires additional branches to be precomputed; some of these may depend upon computation in the infix that is unrelated to earlier postfix branches.

It is interesting to note that Equation 2 implies a lower bound on the number of concurrent helper threads required to prefetch all instructions. Assuming negligible precomputation overhead (i.e., $o(s, t) = 0$), the number of helper threads to ensure there is a helper thread to prefetch every instruction is

$$
geq \left\lceil \frac{CPI_h}{CPI_m} \right\rceil
$$

where $CPI_m$, and $CPI_h$ are the values assuming full coverage is achieved. Lower $CPI_h$ reduces the number of concurrent helper threads required. Achieving $CPI_h = CPI_m$ (implying only one helper thread is required) can be challenging since prescient instruction
prefetch helper threads typically trigger more cache misses than the main thread (in addition to incurring the overhead of infix precomputation). An technique explored in related work for improving the effectiveness of precient instruction prefetch, called \textit{finite state machine recall}, decreases CPI$_h$ significantly by executing only those instructions required to resolve weakly biased postfix branches and leveraging instruction set support for prefetching [Aamodt et al. 2004].

3.2 A Statistical Model of Program Execution

For a control flow graph, where nodes represent basic blocks, and edges represent transitions between basic blocks, we model the intra-procedural program execution as a discrete Markov chain [Hammerstrom and Davidson 1977]. A Markov chain is defined by a set of states and transitions. The basic blocks in a procedure represent the states of the Markov chain, and transitions are defined by the probabilities of branch outcomes in the control flow graph. These probabilities can be gleaned from traditional edge profiling [Chang et al. 1991], which measures the frequency that one block flows to another. For inter-procedural control flow, we summarize the effect of procedure calls when necessary by computing summaries for subroutines, or sets of mutually recursive functions. This is equivalent to restricting transitions into and out of procedures so that a callee must return to its caller. As it is based upon using traditional edge profile data, this model ignores correlation between branch outcomes.

To model the effects of instruction cache access, we assume a two-level memory hierarchy composed of a finite-sized fully associative instruction cache with LRU replacement, and an infinite sized main memory so that all misses are either cold misses or capacity misses. Note that the control-flow path of program execution entirely determines the contents of this cache independent of the program’s static code layout; hence, by considering the probability of taking each possible path through the program it is possible to determine the probability with which a given instruction resides in the cache at any given point in the program execution.

The \textit{necessity} of instruction prefetches can be assessed by analyzing the instruction footprint distribution along paths between the target and the spawn. The \textit{timeliness} of prefetches can be quantified by analyzing the distribution of slack values in relation to the latency of the memory hierarchy by viewing the distances in Equation 1 as statistical quantities. Finally, the \textit{accuracy}, and \textit{coverage} of prefetches can be gauged by analyzing the control flow correlation quantified in the next section by the reaching probability, and posteriori probability.

3.3 Computing Statistics via Path Expressions

In this section we define statistical quantities related to spawn-target selection. These are the reaching probability, posteriori probability, expected path length and variance, and expected path footprint. We show how to transform the evaluation of these quantities to the classic path expression problem (also known as an algebraic path problem), which can be solved efficiently using Tarjan’s path expression algorithm [Tarjan 1981a; 1981b].

Given a sequence of branch outcomes, the \textit{path length} between the starting block $x$ and end block $y$ is the number of instructions executed along the associated path through the program. Given the underlying statistical model of control flow
described above, the path length takes on a distribution of values resembling that seen by the actual program. Determining the mean and variance of this path length distribution enables the estimation of the probability that the path length is within a given range.

To avoid selecting spawn points that never perform useful instruction prefetching because the instructions they prefetch are either evicted before they are needed, or are likely to reside in the cache already, the concept of a path’s instruction cache footprint is useful. The instruction cache footprint is defined as the capacity required to store all the instructions along a given path assuming full associativity and a cache block size equal to a single instruction. The expected path footprint between two points is determined by computing the average instruction cache footprint between two points in the program.

The reaching probability, \( RP(x, y) \), between basic blocks \( x \) and \( y \) is defined as the probability that \( y \) will be encountered at some time in the future given the processor is at \( x \). In prior work [Marcuello and Gonzalez 2002], the point \( y \) is said to be control quasi-independent of the point \( x \) if the reaching probability from \( x \) to \( y \) is above a given threshold (for example 95%). Here we introduce the posteriori probability, which is defined as the probability of having previously visited state \( x \) since the last occurrence of \( y \) (if any), given the current state is \( y \).

3.3.1 Path Expressions. A path expression [Tarjan 1981a] is simply a regular expression summarizing all paths between two points in a graph. For example, in Figure 3(a) the set of all paths from block \( a \) to block \( x \), start by following edge \( A \), then going around the loop any number of iterations along either control path inside the loop, before finally taking edge \( B \). This can be summarized by the path expression:

\[
P(a,x) = A \cdot (((B \cdot C) \cup (D \cdot E)) \cdot F)^* \cdot B
\]

Here the symbols \( \cup \), \( \cdot \), and \( ^* \) denote the regular expression operators union, concatenation, and closure, respectively, and parentheses are used to enforce order of evaluation. These operators have the following interpretation: The union operator is used to combine two distinct paths that start and end at the same point, the concatenation operator joins one path ending at a particular point with another one that starts there, and the closure operator represents the union of all paths that make any number of iterations around a loop. As the union operators defined later in this paper are not idempotent, it is vital that the path expressions we use are unambiguous in the sense that no path can be enumerated in two, or more ways (informally, there is no way to further simplify the path expression). For example, “\( A \cup A \)” enumerates \( A \) twice so this path expression is ambiguous (for a more formal definition, see Tarjan [Tarjan 1981b, Appendix B]).

We apply path expressions by interpreting each edge as having some type of value, and the regular expression operators (union, concatenation, and closure) as functions that combine and transform these values. We give an example of this process in the next section. Tarjan’s fast path algorithm produces unambiguous path expressions very efficiently. In particular Tarjan’s algorithm solves the single source path problem (one source many destinations) in \( O(ma(m, n)) \) time on reducible flow graphs, where \( n \) is the number of nodes and \( m \) is the number of edges.
Table I. Path Expression Mappings

<table>
<thead>
<tr>
<th></th>
<th>Reach. Prob.</th>
<th>Exp. Path Length</th>
<th>Path Length Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>concat. ([R_1 \cdot R_2])</td>
<td>(pq)</td>
<td>(X + Y)</td>
<td>(v + w)</td>
</tr>
<tr>
<td>union ([R_1 \cup R_2])</td>
<td>(p + q)</td>
<td>(pX + qY)</td>
<td>(p + q)</td>
</tr>
<tr>
<td>closure ([R_1^*])</td>
<td>(1 - p)</td>
<td>(pX)</td>
<td>(\frac{p(v + X^2)}{1 - p} + \left(\frac{pX}{1 - p}\right)^2)</td>
</tr>
</tbody>
</table>

and \(\alpha\) is a functional inverse of of Ackermann’s function [Tarjan 1981b] (which grows very slowly). This means we get a path expression for every pair of basic blocks in a procedure in \(O(n\alpha(m, n))\) time. To better utilize Tarjan’s algorithm, a tree-like representation of path expressions is used to support on-demand updating when an edge weight is set to zero (a requirement described later).

The mappings used for computing reaching probability, expected path length and path length variance are summarized in Table I and justified in the following sections. These mappings are not arbitrary, but rather arise from a rigorous analysis of the underlying probabilistic model. To the best of our knowledge, the mappings for expected path length, and path length variance are novel to this work. The mapping used for reaching probability also arises, for example, in the solution of systems of linear equations [Tarjan 1981a], and dataflow frequency analysis [Ramalingam 1996]. However our method of zeroing edges to ensure the implied summation is over disjoint events appears to be novel. Note that the calculation of path length variance, requires that of expected path length, which in turn requires the reaching probability.

3.3.2 Reaching Probability. The reaching probability, \(RP(x, y)\), from \(x\) to \(y\), for \(x \neq y\), may be computed as follows: Label all transitions in the Markov model with their respective transition probability. Set the probability of edges leaving \(y\) to zero, so that paths through \(y\) are ignored (because setting these edges to zero means these paths have zero probability). Then evaluate \(RP(x, y)\) by recursively replacing the path expression in square braces in the first column in Table I, with the value computed by the expression in the second column.

The validity of the mapping used for reaching probabilities arises from several facts: First, probability theory states that the probability of one event occurring out of a set of disjoint events is the sum of the individual probabilities of each event. Thus, the probability of taking any path encoded by the union of two path expressions is the sum because the path expressions formed by Tarjan’s algorithm are unambiguous. Second, if we make the assumption (which a simple extension of this work can eliminate [Aamodt and Chow 2007]) of independent branch outcomes implies that the probability of taking a particular path is the product of the probabilities of all branch outcomes along the path. This, combined with the fact that multiplication distributes over addition, allows us to evaluate the concatenation of two path expressions simply by multiplying their individual values because this is the same as adding the probabilities of each individual path enumerated by the combined path expression. It is well known that branch outcomes are correlated [Yeh and Patt 1991]. It is possible to incorporate the effects of branch
correlation using the our path expression framework by applying the analysis to a higher-order control flow graph where nodes represent limited sequences of basic blocks [Aamodt and Chow 2007]. In Section 5.2, we demonstrate how this improves the accuracy of the path expression framework.

The closure mapping for a loop with probability $p$ of returning from the loop header back to itself can be found by viewing the set of paths encoded by the closure operator as an infinite sum of products, and applying the well known formula for the geometric series:

$$\sum_{i=0}^{\infty} p^i = \frac{1}{1 - p}, \text{ if } |p| < 1$$

Note that the result of the closure mapping does not represent a probability, because the paths it combines are not disjoint events. The path expression analysis is valid if the profile data represents a program run to completion so that there are no loops with a loop probability of one.

**Example: The reaching probability from a to x.**

Each path from $a$ to $x$ in Figure 3(a) must start with edge $A$ and end with edge $B$. In between, there can be any number of iterations around the loop taking the path segment composed of edges $DEF$. The result of applying the procedure outlined above is:

$$P(a,x) = A \cdot (((B \cdot C) \cup (D \cdot E)) \cdot F)^* \cdot B$$

$$[P(a,x)] = 0.98 \cdot \left( \frac{1}{1.0 - (0.1(0.0) + 0.9(1.0)) \cdot (0.999)} \right) \cdot 0.10 \approx 0.97$$

The value underlined in the denominator represents edge $C$ and, as explained earlier must be set to zero to ensure paths going through $x$ are ignored.

This example provides an important and perhaps counter-intuitive insight: The probability of reaching a block can only be accurately (or reliably) determined by examining global behavior. The probability of taking the path directly from $a$ to $x$ is only 0.098, yet the probability of reaching $x$ at least once before control flow exits the region along the edges marked $x$ or $y$ in Figure 3(a), is 0.97, which is much higher. An intuitive explanation of this result is that the probability of exiting the loop each iteration, 0.001, is much lower than the probability of executing $x$ each iteration, 0.1, so that it is very likely for the program to execute $x$ at least once before the loop exits.

3.3.3 **Posteriori Probability.** To evaluate the posteriori probability that $x$ precedes $y$, reverse the direction of control flow edges, and re-label them with the frequency a block precedes, rather than follows another block. Then, setting the probability of edges from $x$ to successors of $x$, and from predecessors of $y$ to $y$ to zero (referring to the new edge orientation), apply the mapping for reaching probability.

3.3.4 **Expected Path Length and Variance.** Using path expressions, the expected path length, and the path length variance from $x$ to $y$ (assuming the current state is $x$), can be computed as follows. With each edge we associate a 3-tuple. The first element represents the probability of branching from the predecessor to the
successor (set to zero for edges emanating from y), the second element represents the length of the predecessor basic block, and the third element represents the path length variance of the edge, and is thus zero. Similarly, for path expressions $R_1$ and $R_2$ we associate 3-tuples $< p, X, v > \text{ and } < q, Y, w >$. The rules for computing the first, second, and, third element are listed in the second, third, and fourth columns of Table I, respectively. In a manner similar to the relationship between reaching probability and posteriori probability, we may define the posteriori expected path length and variance.

These mappings arise by analyzing the expected value, and variance of the path length given the probability of following a particular path, as determined by the edge profile data. It can be verified that the mappings for concatenation and union in Table I satisfy the distributive law, as required. The mapping derivations for the expected path length is described first, followed by a sketch of the derivation for the path length variance (detailed derivations are provided in Appendix A).

The correctness of the expected path length mapping for concatenation follows immediately from the fact that the expected value of the sum of a set of independent random variables equals the sum of the expected values of those random variables. For a given path expression $R$, let $\sigma(R)$ represent the set of all paths enumerated by $R$. The correctness of the formulation for the union operator follows immediately from the fact that $R_1$, $R_2$ and $R_1 \cup R_2$ are unambiguous and therefore:

$$
\sigma(R_1 \cup R_2) = E[|R_1|] \cdot Pr(\text{follow path in } \sigma(R_1) \text{ follow path in } \sigma(R_1 \cup R_2)) + E[|R_2|] \cdot Pr(\text{follow path in } \sigma(R_2) \text{ follow path in } \sigma(R_1 \cup R_2))
$$

In the above, the notation $|R|$, where $R$ is a path expression, represents the expected path length. In general square brackets around a path expression means the path expression is replaced by a mapping such as one of those in Table I.

To derive the formulation for closure we exploit the fact that the expected value of the sum of a random set of independent random variables is the weighted sum of the expected values (weighted by probability of occurrence) and focus on examining a loop with expected path length per iteration of $X$ and backedge probability of $p$ (see Figure 4). For the closure operator we are interested in the expected path length upon entering the loop up to, but not including the edge exiting the loop.

$$
E[\text{closure length}] = \sum_{p_i \text{=path formed by iterating } t \text{ times}} E[\text{length}(p_i)] \cdot Pr(p_i)
$$

The reason each term carries a factor of $(1 - p)$ is that we are interested in the event that the loop iterates a specific number of times and then definitely exits.
The above summation can be expressed as:

\[ E[\text{closure length}] = p(1-p)X \sum_{i=0}^{\infty} ip^{i-1} = \frac{pX}{1-p} \]

The variance of the sum of two independent random variables is simply the sum of the variances, hence the concatenation operator for path length variance. As with the expected path length, the union and closure operators for path length variance are derived by evaluating the conditional expectation, marginalized over the actually taken path using the probabilities found with the reaching probability mapping. The details of the derivation are provided in Appendix A.

3.3.5 Expected Path Footprint. Assuming basic blocks \( x \) and \( y \) are in the same function, and ignoring storage requirements of subroutine calls, the expected path footprint between \( x \) and \( y \), denoted \( F(x, y) \), can be computed using the formula:

\[
F(x, y) = \frac{1}{RP(x, y)} \sum_v \text{size}(v) \cdot RP_\alpha(x, v|\neg y) \cdot RP_\beta(v, y)
\]  

(3)

where the summation runs over all basic blocks \( v \) on any path from \( x \) to \( y \) for which \( y \) only appears as an endpoint\(^2\), \( \text{size}(v) \) is the number of instructions in basic block \( v \), and \( RP_\alpha(x, v|\neg y) \) is the probability of traversing from \( x \) to \( v \) along any path except those passing through basic block \( y \),

\[
RP_\alpha(x, v|\neg y) = \begin{cases} 
RP(x, v) \text{ s.t. no path thru } y & \text{if } x \neq v, \\
1 & \text{if } x = v
\end{cases}
\]

(4)

This quantity is computed similar to \( RP(x, v) \), except that edges leading from \( y \) are evaluated as having zero probability (in addition to those leading from \( v \)) to ensure they are not included in the summation. The quantity \( RP_\beta(x, y) \) used in Equation 3 is defined as,

\[
RP_\beta(x, y) = \begin{cases} 
RP(x, y) & \text{if } x \neq y \\
0 & \text{if } x = y
\end{cases}
\]

(5)

Equation 3 is significant because it allows us to evaluate the expected path footprint in terms of values we know how to compute efficiently. To take into account the code footprint used by subroutine calls we weigh the size of each callee basic block by the probability it is reached at least once.

The derivation of Equation 3, in particular the interpretation of \( RP_\beta(x, y) \) and \( RP_\alpha(x, v|\neg y) \), is as follows. The expected path footprint from \( x \) to \( y \) is computed as the summation, over all paths from \( x \) to \( y \) such that \( y \) only appears as the endpoint of any particular path, of the relative frequency we traverse from \( x \) to \( y \) following path \( p \) times the footprint of path \( p \):

\[
F(x, y) = \sum_{p \in \sigma(P_\epsilon(x, y))} \text{freq}(p|x, y) \cdot f(p)
\]

\(^2\)This set can be found while evaluating the reaching probability.
where $P_o(x, y)$ is the path expression enumerating all ways of going from $x$ to $y$ such that $y$ is encountered only once (a by-product of the reaching probability computation), freq($p|x, y$) is the fraction of times starting from $x$ and ending at $y$ we followed path $p$, and $f(p)$ is the path footprint due to all blocks along $p$ except $y$. By expressing the fraction of times we traverse from $x$ to $y$ following path $p$ as the probability of taking path $p$ starting from $x$, $Pr(p)$, divided by the probability of reaching $y$ from $x$ (along any path), we can rewrite this as:

$$F(x, y) = \frac{1}{RP(x, y)} \cdot \sum_{p \in \sigma(P_o(x, y))} f(p) \cdot Pr(p)$$

By expanding $f(p)$ in terms of the sizes of the unique basic blocks encountered along the path $p$ we can express $F(x, y)$ as:

$$F(x, y) = \frac{1}{RP(x, y)} \cdot \sum_{p \in \sigma(P_o(x, y))} \left( \sum_{v \in p, v \neq y} \text{size}(v) \cdot Pr(p) \right)$$

We further transform this equation by exchanging orders of summation to obtain:

$$F(x, y) = \frac{1}{RP(x, y)} \cdot \sum_{v \in B} \text{size}(v) \cdot \sum_{p \in P_o^\prime} Pr(p) \tag{6}$$

where $B$ is the set of all blocks except $y$, which are passed through one or more times by at least one path in $\sigma(P_o(x, y))$, and $P_o^\prime$ is the subset of paths in $\sigma(P_o(x, y))$ passing through $v$. Each path from $x$ to $y$ passing through $v$, such that $y$ appears only as an endpoint, can be expressed as the concatenation of two disjoint paths: Assuming $x$, and $y$ are distinct from $v$, the first path goes from $x$ to $v$ without passing through $y$ such that $v$ appears only as an endpoint, and the second path goes from $v$ to $y$ such that $y$ appears only as an endpoint. If $x$ equals $v$, the first path consists of the single vertex $x$. When $v$ is $y$, the second path consists of the single vertex $y$. This decomposition defines two path sets. The sum of the probabilities of going from $x$ to $v$ over all paths in the first set is given by Equation 4, and the sum of the probabilities over the paths in the second set is given by Equation 5. Thus we can express the sum of the probabilities of all paths from $x$ to $y$ passing through $v$, such that $y$ only appears at the end of a path as the product of these factors:

$$\sum_{p \in P_o^\prime} Pr(p) = RP_o(x, v|\neg y) \cdot RP_o(v, y)$$

Substituting this result into Equation 6 yields Equation 3.

Note that $RP_o(x, v|\neg y) \leq RP(x, v)$. Thus, we may conservatively estimate path footprints by using $RP(x, v)$ rather than $RP_o(x, v|\neg y)$. This approximation was used when generating the spawn-target pairs evaluated in Section 5.3 and yields an $O(n)$ reduction in the number of edge weight modifications (e.g., setting an edge weight to zero) that need to be evaluated.

3.3.6 Eliminating Spawn-Point Redundancy. We way that a spawn-point $s_1$ implies another spawn-point $s_2$ for a given target $t$ if any path from $s_1$ to $t$ passes through $s_2$. In other words, if $s_2$ is reached along the path from $s_1$ to $t$, spawning
when the main thread reaches $s_2$ may be redundant. Two spawn-points are said to be independent with respect to a common target if neither spawn-point implies the other. By selecting a set of mutually independent spawn-points we are assured that only one will execute for a given dynamic instance of $t$. Furthermore, the reduction in execution time from helper threads with independent spawn-points is additive. Path expressions provide a convenient way to determine spawn-point independence: given $s_1$, $s_2$, and $t$, we can determine whether $s_1$ implies $s_2$ by checking whether any edge in the path expression from $s_1$ to $t$ starts or ends with $s_2$ after eliminating edges emanating from $t$. This can be performed efficiently while evaluating the reaching probability.

4. SPAWN-PAIR SELECTION ALGORITHM

This section describes one particular spawn-target selection algorithm based upon the framework presented in Section 3, which is useful for prescient instruction prefetch helper threads. We have implemented this algorithm for the Itanium® architecture and the effectiveness of this algorithm is demonstrated in Section 5. A high-level flow-chart of the algorithm is shown in Figure 5.

The inputs to the algorithm are edge frequency and instruction cache-miss profiles, and the estimated main and helper thread CPI. The output is a set of mutually independent spawn-target pairs, and associated postfix region sizes. Profile data is supplied via procedure control flow graphs annotated with basic-block and branch execution frequencies, and total instruction cache misses per basic block. Control flow graphs include information about procedure calls such as the frequency a given subroutine is called from a given site. For our purposes we consider a call site to be a basic block boundary, and augment the graph with an additional edge from the call instruction to the next instruction in the caller’s function body. This edge has probability one minus the probability the call is predicated false, and path length and variance computed by summarizing subroutines.

The algorithm first splits control flow graph nodes that represent large basic
blocks into linear sequences of nodes, each representing only a portion of the original basic block. This is done because spawn and target locations are constrained to the beginning of basic blocks to reduce complexity. Without considering spawn and target locations within large basic blocks the optimization algorithm would find some instructions could not be prefetched, when, for example, no admissible target immediately precedes them. Next, the algorithm computes procedure summary information. In particular, the expected path length and variance from entry to exit, and the reaching probabilities from the entry to each block are computed. The latter is used for computing the expected path footprint taking into account procedure calls.

Once summary information is computed, we rank the basic blocks by the absolute number of I-cache misses they generate. We target basic blocks that account for the top 95% of all instruction cache misses. In each procedure visited, the reaching probability, expected path length, path length variance, and posteriori probability, are computed between every pair of basic blocks according to the method described in Section 3.3. It may be possible to improve the algorithm’s performance by pruning some of these evaluations.

We maintain an estimated number of instruction cache misses remaining in each block given the spawn-target pairs already selected. The initial value is the number of I-cache miss found during profiling. Then, we iteratively select the block in the current procedure with the highest estimated remaining I-cache misses, determine a target and corresponding set of independent spawn points for this block, and update the estimated remaining cache misses in all blocks that may be prefetched by helper threads with these spawn-target pairs. The latter update is based on estimating the amount of coverage each block receives using the posteriori probability, and the probability the slack will be sufficient to hide the memory latency. Similarly, we maintain an estimated number of running helper threads at each point in the program such that a spawn-point is not selected if this selection leads to a high probability of attempting to concurrently run more helper threads than available thread contexts (in which case some of the helper threads would not be able to run).

For each selected block \( v \) having a high number of instruction cache misses remaining, potential targets are earlier blocks with high posteriori probability of being visited before \( v \). Thus, the target basic blocks do not necessarily suffer heavy I-cache misses themselves. It is often necessary to perform this step to find targets with high reaching probability from potential spawn points. The selection process for spawn and target are coupled in the following way: A set of potential targets with posteriori expected distance less than half the (preset) maximum postfix distance we wish to allow\(^3\) is generated, and ranked in descending order by distance from the selected block. By selecting the target to be an earlier block in this way, we reduce the selection of spawn-target pairs with overlapping postfix regions. For each potential target a set of independent spawn points is found using the following process.

For a given target \( t \), a set of mutually independent spawn points are selected among all blocks in the procedure with reaching probability to \( t \) greater than a

\(^3\)Set experimentally to 100 Itanium® bundles, each of which contains 2 to 3 instructions.
threshold\textsuperscript{4}, by computing a heuristic value indicating their effectiveness as spawn points for the target. In particular, the merit of a given spawn-point is computed as the product of the following factors: The first factor is the posteriori probability that the spawn precedes the target. This factor along with the expected path footprint quantify the fraction of I-cache misses at the target that are covered by instances of the helper thread. Furthermore, together with the restriction on reaching probability it ensures the spawn and target are control flow correlated. The second factor uses the expected path footprint to penalize those spawn points whose average target-to-spawn (i.e., paths leading from the prefetched region back to the spawn) footprints are less than the cache size, because this condition implies a greater likelihood the prefetched instructions are still in the I-cache and so do not need to be prefetched (for target-to-spawn footprints less than the cache size we reduce the value by a factor of the cache size divided by the expected footprint size). Similarly, spawn points with expected spawn-to-target path footprints larger than the instruction cache capacity are given a value of zero since instructions prefetched by the associated helper threads are likely to be evicted before helping the main thread. The third factor takes account of the size of the postfix region the helper thread can prefetch, which is determined by finding the maximum postfix length that always provides a minimum threshold probability\textsuperscript{5} that the slack of the last prefetch issued by the helper thread is still positive (for simplicity we assume path lengths take on a Gaussian distribution using the mean and variance found using the mappings in Table I). As spawn-target distance increases, this term increases until the number of instructions the helper thread can prefetch reaches the preset maximum postfix distance. The number of instructions the helper thread is likely to prefetch before getting caught by the main thread is also output (and used to terminate the helper thread).

5. SIMULATION RESULTS

This section presents both an evaluation of the accuracy of our modeling framework compared to statistics collected from actual program execution traces, and a performance analysis of prescient instruction prefetch based upon cycle accurate simulation. We examine the performance impact of prescient instruction prefetch threads defined by spawn-target pairs selected using the algorithm described in Section 4.

5.1 Methodology

We select four benchmarks from Spec 2000 and one from Spec 95 that incur significant instruction cache misses on the baseline processor model. The application binaries were built with the Intel Electron compiler, all, except for \texttt{fpppp}, with profile feedback enabled. We profile the branch frequencies and I-cache behavior by running the programs to completion. The spawn-target generation algorithm selected between 34 and 1348 static spawn-target pairs per benchmark as shown in Table II, which also quantifies the number of dynamic occurrences of these spawn target pairs, and the average distance between spawn and target (infix), and the

\textsuperscript{4}Experimentally set to 0.95

\textsuperscript{5}Experimentally set to 0.5

ACM Journal Name, Vol. V, No. N, Month 20YY.
average number of instructions prefetched per dynamic helper thread instance for a 4-way SMT (postfix). The small number of static pairs implies small instruction storage requirement for helper threads. The large infix size means long memory latencies can be tolerated, while small postfix (relative to the infix) indicates helper threads do run slower than the main thread (statistics for infix slices were studied in related work [Aamodt et al. 2004], where it was found they are small relative to the infix distance between spawn and target).

The baseline architecture is a cycle-accurate research in-order SMT processor model for the Itanium® architecture [Huck et al. 2000] based upon SMTSIM [Tullsen et al. 1995] with microarchitectural parameters as summarized in Table III. The baseline uses a hardware instruction prefetch mechanism that issues a prefetch for the next sequential line on a demand miss, and supports Itanium® instruction stream prefetch hints tagged onto branch instructions by the compiler. The rest of the memory hierarchy organization models latencies resembling an Itanium® 2 at 1.5GHz.

We evaluate the potential of prescient instruction prefetch assuming values for all live-in register, and memory operands to the postfix region are available at no cost as soon as the helper thread spawns. A helper thread is spawned when the main thread commits a spawn-point. If no free thread contexts are available the spawn-point is ignored. The helper thread may begin fetching from the target the following cycle and runs for its expected maximum postfix distance before exiting. If the main thread catches up with it before that point the helper thread is killed and stops prefetching instructions. Helper thread instructions contend for execution resources with the main thread, but have reduced fetch and issue priorities relative to the main thread. We assume store operations executed by helper threads do not commit to memory, but that they correctly forward their values to dependent loads within the same helper thread. Once a helper thread is stopped and its instructions drain from the pipeline, the thread context is available to run other helper threads. As this is a limit study we do not simulate the effect of a helper thread spawned when the target does not appear (only the last spawn-point seen before a given instance of a target-point initiates a helper thread) and we do not model off-path fetching by helper threads.

To evaluate the performance impact of prescient instruction prefetch we collect data for 100 million instructions after warming up the cache hierarchy while fast-forwarding past the first billion instructions.

### 5.2 Framework Accuracy: An Example

We assess the accuracy of the modeling framework by carrying out a detailed study for the `Evaluate()` subroutine in Spec 2000 benchmark `186.crafty`. This subroutine accounts for roughly one quarter of all I-cache misses in `crafty`, and moreover
its control flow graph is non-trivial—containing 230 basic blocks, 345 edges, and several loops.

We carry out our study in three steps. First, in Section 5.2.1 we measure the reaching probability, expected path length, path length variance, and expected path footprint for 25 spawn-target pairs chosen by the selection algorithm for subroutine `Evaluate()` in benchmark `crafty` by simulating a Markov model of the subroutine, and by collecting statistics from program traces. Second, in Section 5.2.2 we explore the impact of applying the path expression framework to a higher order control flow graph [Aamodt 2006; Aamodt and Chow 2007]. Third, in Section 5.2.3 we explore the impact of program input selection.

### 5.2.1 Prediction accuracy

Figure 6(a)-6(d) compare predicted statistics with a Monte Carlo simulation based upon an idealized Markov chain with the exact same edge probabilities used by the framework (“measured” in these figures represents averages from 1 million trials per spawn-target pair). The units in Figures 6(b), (c) and (d) are Itanium® bundles, while Figure 6(a) plots relative frequency versus probability both measured in percentage. Note that path variation is reported as the standard deviation—i.e., the square root of the variance (computed using the path expression in Table I). The expected footprint predictions are computed using Equation 3—i.e., without substituting $RP(x, v)$ for $RP_\alpha(x, v | \neg y)$. The strong correlation present in Figures 6(a)-6(d) underscores the robustness of the path expression based methodology described in Section 3, and Table I.

Figures 7(a)-7(d) show a similar comparison for the same spawn-target pairs, but with statistics from the 100 million instruction segment used for performance evaluation. Both the reaching probability, and path length variation exhibit several outlying data points. However, note that 16 of the 25 data points are clustered within 1% of the upper right corner in Figure 7(a) so that reaching probability is more correlated than casual observation may suggest. The outlying data points appear to result from two approximations in the modeling framework: First, tran-

<table>
<thead>
<tr>
<th>Threading</th>
<th>SMT processor with 2, 4, or 8 hardware threads.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelining</td>
<td>In-order: 12-stage pipeline.</td>
</tr>
<tr>
<td>Fetch</td>
<td>2 bundles from 1, or 1 bundle from 2 threads</td>
</tr>
<tr>
<td></td>
<td>prioritize main thread, helpers ICOUNT</td>
</tr>
<tr>
<td></td>
<td>[Tulsen et al. 1996].</td>
</tr>
<tr>
<td>Fetch</td>
<td>next line prefetch (triggered on miss)</td>
</tr>
<tr>
<td></td>
<td>stream prefetcher (triggered by compiler hints)</td>
</tr>
<tr>
<td></td>
<td>max. 4 outstanding prefetches per context.</td>
</tr>
<tr>
<td>Branch Pred.</td>
<td>2k-entry gshare. 256-entry 4-way associ. BTB.</td>
</tr>
<tr>
<td></td>
<td>helper threads: oracle branch prediction</td>
</tr>
<tr>
<td>Issue</td>
<td>2 bundles from 1, or 1 bundle from 2 threads</td>
</tr>
<tr>
<td></td>
<td>prioritize main thread, helpers: round-robin.</td>
</tr>
<tr>
<td>Function units</td>
<td>4 int., 2 fp., 3 br., and 2 data mem. ports</td>
</tr>
<tr>
<td>Register files</td>
<td>128 int, 128 fp, 64 predicate, 8 br.</td>
</tr>
<tr>
<td></td>
<td>128 control registers.</td>
</tr>
<tr>
<td>Caches</td>
<td>L1 (separate I&amp;D): 16KB (each). 4-way. 1-cyc.</td>
</tr>
<tr>
<td></td>
<td>L3 (shared cache): 3072KB. 12-way. 30-cycles.</td>
</tr>
<tr>
<td></td>
<td>Fill buffer: 16 entries. caches have 64B lines.</td>
</tr>
<tr>
<td></td>
<td>helper threads: infinite store buffer 1-cyc.</td>
</tr>
<tr>
<td>Memory</td>
<td>230-cycles. TLB miss penalty: 30 cyc.</td>
</tr>
</tbody>
</table>
_transition probabilities were determined by profiling over the whole program execution, which averages out behaviors unique to distinct phases of execution. Second, even within a single phase of program execution the correlation between branch outcomes is ignored in the model. Both of these approximations can lead to the existence of paths never executed by the program having a finite probability in the model. Similarly, some paths with a very small predicted probability of occurring in the model may actually have a high probability of occurring during execution due to strong branch correlation. Imprecise path probability estimates can lead a weighting of possible outcomes during path expression evaluation that differs from real program execution. By separately modeling program behavior in distinct program phases [Sherwood et al. 2002], and taking into account the impact of branch correlation (by extending the current approach to use higher order Markov chain models [Aamodt 2006; Aamodt and Chow 2007]) these quantities may be predicted with greater accuracy as we explore in the following section.

5.2.2 Modeling Branch Correlation with Higher Order Control Flow Graphs. Figure 8 plots the reaching probability for a wider selection of basic block pairs...
Fig. 8. Reaching probability prediction accuracy. These graphs show the impact of increasing control flow graph order (order $N$ means each vertex in the graph represents a sequence of $N$ basic blocks) for randomly selected basic block pairs in 186.crafty function Evaluate() for training inputs, profiling and measurement of 100M instructions after fast-forwarding past the first 100M instructions.

- (a) 1st order
- (b) 8th order
- (c) 24th order

Fig. 9. Reaching probability prediction accuracy and graph size as control flow graph order increases.

(not just those selected as spawn-target pairs) as the order of the control flow graph is increased. Figure 8(a) plots the reaching probability when each vertex in the graph we apply path expressions to represents a single basic block (as in a normal control flow graph). It can be seen that there are several “outlier” data points for which the predicted reaching probability is not the same as the measured value. One particular outlier has predicted reaching probability of 0.2, while the actual reaching probability is closer to 1.0. In Figure 8(b), the path expression framework is applied to an $8^{\text{th}}$ order control flow graph, in which each vertex represents a short path of eight consecutive basic blocks (in the order they are encountered during program execution). The details of such higher order control flow graphs are described elsewhere [Aamodt 2006; Aamodt and Chow 2007]. It can be seen in Figure 8(b) that the outliers have moved closer to the line $y = x$ (which represents perfect prediction accuracy). By the time the order of the graph is increased to $24^{\text{th}}$ order, the prediction accuracy is nearly perfect. Figure 9(a) shows the correlation coefficient for these basic block pairs as the control flow graph order is increased from first-order to $32^{\text{th}}$ order. Figure 9(b) tracks the increase in the number of vertices in the higher order graph. The growth in graph size is less than one would expect if all possible paths through the program were equally likely [Ball and Larus 1996]. The above data suggests a good trade-off in accuracy.
5.2.3 **Input Sensitivity.** Figure 10 plots the impact that program input has on the reaching probability for the same basic block pairs as in the previous section. The figure compares measured reaching probability for the training inputs versus reference inputs. While there are outliers, the correlation coefficient is 0.946 which is roughly in the range of prediction accuracies for the lower order control flow graphs above.

5.3 **Results**

Figure 11 illustrates the speedup of prescient instruction prefetch compared to the baseline model. The first bar on the left for each benchmark is the speedup if all instruction accesses hit in the first level cache, and shows speedups ranging from 7% to 25% with a harmonic mean of 18%. Thus the benchmarks we study do indeed suffer significantly from instruction cache misses. The bars labeled $2t$, $4t$, and $8t$ represent the speedup of idealized prescient instruction prefetch on a machine with 2, 4 and 8 hardware thread contexts respectively. In each case spawn-target pairs are generated assuming only one main thread will share processor resources with the helper threads. Speedup improved with increasing number of hardware thread contexts with harmonic mean speedups of 5.5%, 9.8%, and 13% respectively, with speedups of up to 4.8% to 17% on the 8 thread context configuration. As might be expected, increasing the number of concurrent helper threads lead to a reduction in the IPC of individual helper threads due to resource contention. The last two bars in this figure will be described below.

Most applications see a substantial reduction in the number of I-cache misses that improves as additional thread contexts are made available with a reduction in I-cache misses averaging 16% for $2t$, 33% for $4t$, and 42% for $8t$. To better understand how speedup is obtained, as well as the quality of the spawn-target selection algorithm, Figure 12 shows the source of remaining I-cache misses normalized to the number of baseline instruction cache misses. Each miss is classified into one of five categories: “no target” means that the cache miss is not preceded by an instruction selected as a target, within that target’s maximum postfix distance; “no spawn” means that although there is at least one target, none of them were preceded by a spawn since the last occurrence of the target; “no context” means
and spawn-target pairs selected with lower CPI, performance modeled so that helper threads do not block on an I-cache miss. “4t” and spawn-target pairs selected with lower reaching probability threshold (0.75).

Fig. 11. Limit study performance. “ideal”: speedup if all I-accesses hit. “4t”: prescient instruction prefetch speedup for 4t thread contexts. “F”: spawn-target pairs selected for 4t and fast helpers (low CPI), performance modeled so that helper threads do not block on an I-cache miss. “L”: 4t and spawn-target pairs selected with lower reaching probability threshold (0.75).

Fig. 12. Breakdown of remaining I-cache misses relative to baseline.

there was a spawn-target pair that could have prefetched the I-cache miss, but no SMT thread context was available when the spawn committed; “too slow” means a helper thread was running that could have prefetched the I-cache miss, but it was caught by the main thread before it could do so (note: partial misses are included in this chart); finally, “evicted” means a helper thread did prefetch the accessed line, but the line was evicted before being reached by the main thread.

Ideally, all components should be small, but especially the “too slow” and “no contexts” should be low if our framework is able to predict run time behavior well and the algorithm is properly tuned. On average, for 4t, 32% of I-cache misses remain because there was no target, 12% because there was no spawn, 7% due to lack of SMT thread contexts, 6% because the associated helper threads ran too slowly, and 9% because the prefetches were too aggressive and prefetched instructions were evicted. The large “no target” component results when the selection algorithm could not find a spawn-target pair to target the miss, either due to resource constraints, or because it could not find an early enough location that was control flow correlated because we constrained spawn-target pairs to be within the same function body. Figure 12 indicates that increasing the number of thread resources (and static spawn-target pairs) does not seem to significantly impact “no target”. So it is more likely the latter cause is the culprit. To further corroborate this view, we examine the bar labeled F in Figure 11 and Figure 12, which represents the impact when spawn-target pairs are selected assuming the helper thread progresses with the same CPI as the main thread. For this to make sense, the execution model is modified to allow helpers to prefetch instructions without executing them,
so that I-cache misses do not stall a helper thread (although fetch contention still does). This reduces the “no target” component to an average 7%, highlighting the importance of approaches such as finite state machine recall [Aamodt et al. 2004]. Note that, except for F, vortex shows significant speedups due to data prefetching.

For crafty, there was a large fraction of “no spawn”. To reduce this, a seemingly straightforward approach is to lower the threshold for reaching probability in hope of including more spawn candidates. To quantify this intuition, the bars labeled L are for spawn-target pairs selected with a reaching probability threshold of only 75%. However, this lowered threshold does not seem to help significantly. Detailed inspection shows many targets associated with this type of miss have spawns with low posteriori probability. Often these were the only remaining locations within the current procedure that provided sufficient slack. Thus, lowering selection threshold alone is not sufficient, further improvements may come from a better (i.e., non-greedy) selection algorithm.

6. RELATED WORK

The path expression framework presented in this paper is probably best viewed as a toolkit for creating accurate analytical models of behavior taking place on the timescale required to execute 100’s to 1000’s of instructions [Eeckhout et al. 2003].

Sarkar [Sarkar 1989] presented a program analysis approach to predict program execution time and variance using profile information. That work differs from ours in that they analyze execution time between restricted pairs of basic blocks (entry and exit points of procedures) and their algorithm is limited to analyzing reducible flow graphs. We compute statistics between arbitrary basic blocks within a procedure. Moreover, path expressions can be found efficiently for irreducible flow graphs [Tarjan 1981b], and indeed the higher order control flow graphs analyzed in Section 5.2.2 to account for branch correlation typically were non-reducible even through the underlying program control flow was reducible. Path expression analysis has been applied in optimizing compilers for data flow analysis [Tjiang and Hennessy 1992].

There has been much work related to helper threads. Dubois and Song proposed assisted execution as a generic way to use multithreading resources to improve single threaded application performance [Dubois and Song 1998]. Chappell et al. proposed simultaneous subordinate multithreading (SSMT), a general framework for leveraging otherwise spare execution resources to benefit a single-threaded application. They first evaluated the use of SSMT as a mechanism to provide a very large local pattern history based branch predictor [Chappell et al. 1999], and subsequently proposed hardware mechanisms for dynamically constructing and spawning subordinate microthreads to predict difficult-path branches [Chappell et al. 2002].

Weiser [Weiser 1981] proposed program slicing to find the subset of a program impacting the execution of a particular statement to aid program understanding. Combining these themes, Zilles and Sohi proposed analyzing the dynamic backward slice of performance degrading instructions so as to pre-execute them [Zilles and Sohi 2000]. They subsequently implemented hand-crafted speculative slices to pre-compute branch outcomes and data prefetch addresses [Zilles and Sohi 2001]. Roth and Sohi [Roth and Sohi 2001] proposed using data driven multithreading (DDMT)
to dynamically prioritize sequences of operations leading to branches that mispredict or loads that miss. Concurrent with our work, they propose a framework for optimizing the selection of slices for loads that miss in the second-level cache based upon an analysis of program traces [Roth and Sohi 2002]. Moshovos et al. proposed slice processors, a hardware mechanism for dynamically constructing and executing slice computations for generating data prefetches [Moshovos et al. 2001]. Annavaram proposed dependence graph precomputation [Annavaram et al. 2001], which effectively uses the predicted stream of instructions to produce accurate dynamic slices. Luk proposed software controlled preexecution [Luk 2001] as a mechanism to prefetch data by executing a future portion of the program. Collins et al. proposed speculative precomputation [Collins et al. 2001], and later dynamic speculative precomputation [Collins et al. 2001] as techniques to leverage spare SMT resources for generating long range data prefetches by executing slices that compute effective addresses for loads that miss often. They showed that chaining the precomputation of a load that repeatedly misses can help tolerate long memory latencies. Liao et al. extended the work of Collins et al. by implementing a post-pass compilation tool to augment a program with automatically generated precomputation threads for data prefetching [Liao et al. 2002]. Kim et al. demonstrated that helper threads can improve performance on real hardware [Kim et al. 2004]. Wang et al. demonstrated the use of helper threads using switch on event multithreading, also on a real hardware platform [Wang et al. 2004]. More recently, Zhang et al. have applied a dynamic optimization framework to enable dynamic code optimizations that customize data prefetch helper threads to variations in program phase [Zhang et al. 2007].

Recently, Quiñones et al., explore optimizations for precomputation slices used to compute live in values for speculative multithreading [Quiñones et al. 2005]. The path expression framework presented in this work is applicable to speculative multithreading [Sohi et al. 1995; Steffan and Mowry 1998; Marcuello et al. 1998; Akkary and Driscoll 1998]. Incorporating branch correlation information as described in Section 5.2.2 and [Aamodt 2006; Aamodt and Chow 2007] may improve the accuracy of the performance predictions used to select spawn-target pairs for thread speculation, but detailed evaluation of this is beyond the scope of this paper.

7. CONCLUSION

In this paper we propose prescient instruction prefetch, a technique for speeding up single-threaded applications suffering from heavy instruction cache misses by leveraging spare thread contexts to run helper threads to prefetch future instructions. In addition, we introduce an analytical framework and describe, implement and evaluate an optimization algorithm for selecting prefetch helper thread spawn-target pairs. This study shows potential speedups of 4.8% to 17% over an Itanium® processor model with nextline and I-stream pre-fetch, demonstrating the effectiveness of the framework, and prescient instruction prefetch. The modeling framework has been applied to data prefetch helper threads [Aamodt and Chow 2007], and can potentially be applied to speculative multithreading.
A. PATH LENGTH VARIANCE OPERATORS

The union and closure operators for the path length variance are derived below. The commonality in the approach to their derivation is to express the expected value in terms of the conditional expectation over a set of disjoint paths.

A.1 Union Operator

Let $Z$ represent a random variable whose value is the length of a path selected at random from the set of paths between two specific, disjoint vertices $s$ and $t$ in a positively weighted directed graph. Furthermore assume that the corresponding set of paths can be partitioned into two non-empty sets $x$ and $y$ and let $X$ and $Y$ represent corresponding random variables whose value is the length of the paths between $s$ and $t$ in the sets $x$ and $y$. Then the expected path length variance of $Z$ is given by:

$$\text{VAR}(Z) = \mathbb{E}[(Z - \mathbb{E}[Z])^2]$$

$$= \mathbb{E}[Z^2] - \mathbb{E}[Z]^2$$

$$= \mathbb{E}[Z^2 | \text{select path in } X] \cdot \text{Pr}[\text{select path in } X] + \mathbb{E}[Z^2 | \text{select path in } Y] \cdot \text{Pr}[\text{select path in } Y] - \mathbb{E}[Z]^2$$

$$= \mathbb{E}[X^2] \cdot \left(\frac{p}{p+q}\right) + \mathbb{E}[Y^2] \cdot \left(\frac{q}{p+q}\right) - \left(\frac{\mathbb{E}[X] \cdot p + \mathbb{E}[Y] \cdot q}{p+q}\right)^2$$

$$\text{VAR}(Z) = \frac{p \cdot (v_X + m_X^2) + q \cdot (v_Y + m_Y^2)}{p+q} - \left(\frac{p \cdot m_X + q \cdot m_Y}{p+q}\right)^2$$

Where $p$ and $q$ represent the probabilities of starting at $s$ and following a path in, respectively, $X$ or $Y$ to reach $t$; $m_X$ and $m_Y$ represent the mean path length starting from $s$ and going to $t$ through a path in respectively $X$ or $Y$; $v_X$ and $v_Y$ represent the path length variance starting from $s$ and going to $t$ through a path in respectively $X$ or $Y$. The individual steps of the derivation are elaborated below:

**Eqn. 7 to 8** $\mathbb{E}[Z]$ is a constant. The expected value of a constant multiplied by a random variable is the expected value of the random variable multiplied by the constant—i.e., $\mathbb{E}[Z \cdot \mathbb{E}[Z]] = \mathbb{E}[Z] \cdot \mathbb{E}[Z] = \mathbb{E}[Z]^2$.

**Eqn. 8 to 9** It is well known that $\mathbb{E}[A] = \mathbb{E}[\mathbb{E}[A|B]]$—see for instance Leon-Garcia [Leon-Garcia 1994, eqn 4.37 on pp. 215]. To transform Equation 8 into Equation 9 we consider $A := Z^2$, and define $B$ to be the bernoulli random variable with outcome ‘$b_X$’ representing the event that a path in $X$ was selected, and outcome ‘$b_Y$’ representing the event that a path in $Y$ was selected. Since $B$ is a discrete random variable we have $\mathbb{E}[\mathbb{E}[A|B]] = \sum_{b_i} \mathbb{E}[A|b_i] \cdot \text{Pr}(b_i)$ [Leon-Garcia 1994, pp. 216].

**Eqn. 9 to 10** To evaluate $\text{Pr}(b_i)$ we note that $\text{Pr}(b_X) = \text{Pr}(X|Z)$ because we are only interested in the event that a path from $s$ to $t$ is selected from $X$ if it also lies in $Z$. Then we have $\text{Pr}(X|Z) = \frac{\text{Pr}(X \cap Z)}{\text{Pr}(Z)}$ =
\[
\frac{\Pr(X)}{\Pr(Z)} = \frac{\Pr(X)}{\Pr(X) + \Pr(Y)} = \frac{p}{p + q}. \quad \text{A similar result holds for } \Pr(bY).
\]

Eqn. 10 to 11 Rearrange and substitute for \(E[X], E[Y], E[X^2], \) and \(E[Y^2].\)

A.2 Closure Operator

Let \(Z\) represent a random variable whose value is the length of a path selected at random from the set of paths starting and ending at vertex \(x\) in a positively weighted directed graph. Then the expected path length variance of \(Z\) is given by:

\[
\text{VAR}(Z) = E\left( (Z - E[Z])^2 \right) \quad (12)
\]

\[
= \sum_{n=0}^{\infty} E\left( (Z_N - E[Z])^2 \middle| N = n \right) \cdot \Pr[N = n] \quad (13)
\]

\[
= \sum_{n=0}^{\infty} E\left( Z_N^2 - 2Z_N E[Z] + E[Z]^2 \middle| N = n \right) \cdot \Pr[N = n] \quad (14)
\]

\[
= \sum_{n=0}^{\infty} \left( E[Z^2 \middle| N = n] - 2E[Z_N \middle| N = n]E[Z] + E[Z]^2 \right) \cdot \Pr[N = n] \quad (15)
\]

\[
= \sum_{n=0}^{\infty} \left( nE[X^2] + n(n+1)E[X]^2 - 2nE[X]E[Z] + E[Z]^2 \right) \cdot \Pr[N = n] \quad (16)
\]

\[
= \sum_{n=0}^{\infty} \left( n \sigma^2 + n(n-1)\mu^2 - \frac{2n\mu^2 p}{1-p} + \left( \frac{\mu p}{1-p} \right)^2 \right) \cdot (1-p)p^n \quad (17)
\]

\[
= \sigma(1-p) \sum_{n=0}^{\infty} np^n + (1-p)\mu^2 \sum_{n=0}^{\infty} n^2 p^n - (1-p)\mu^2 \sum_{n=0}^{\infty} np^n - 2\mu^2 \sum_{n=0}^{\infty} np^n + \frac{\mu^2 p}{1-p} \sum_{n=0}^{\infty} p^n \quad (18)
\]

\[
= \sigma(1-p) \frac{p}{(1-p)^2} + \mu^2 (1-p)p + \frac{p+1}{(1-p)^3} - \mu^2 (1-p) \frac{p}{(1-p)^2} - 2\mu^2 p \frac{p}{(1-p)^2} + \frac{\mu^2 p^2}{1-p} \quad (19)
\]

\[
= \sigma(1-p) \frac{p}{(1-p)^2} + \mu^2 (1-p)p + \frac{p+1}{(1-p)^3} - \mu^2 (1-p) \frac{p}{(1-p)^2} - 2\mu^2 p \frac{p}{(1-p)^2} + \mu^2 \frac{p^2}{1-p} \quad (20)
\]

\[
= \sigma p \frac{1}{1-p} + \mu^2 p + \frac{p+1}{(1-p)^2} - \mu^2 p \frac{1}{1-p} - 2\mu^2 \frac{p^2}{(1-p)^2} + \mu^2 \frac{p^2}{1-p} \quad (21)
\]

\[
= \sigma p \frac{1}{1-p} + \mu^2 p \frac{1}{(1-p)^2} - \mu^2 p \frac{1}{1-p} \quad (22)
\]
\[
\frac{\sigma p}{1-p} + \frac{\mu^2 p - \mu^2 p(1-p)}{(1-p)^2} = \frac{(v + m^2)p}{1-p} + \left( \frac{pm}{1-p} \right)^2
\] (23)

Where \( p \) is the probability of repeating another iteration after entering the loop, \( m \) is the expected path length of a single iteration of the loop body, and \( v \) is the path length variance of a single iteration of the loop body. The individual steps of the derivation are elaborated below:

**Eqn. 12 to 13** Apply the expansion \( \mathbb{E}[A] = \mathbb{E}[\mathbb{E}[A|B]] \), with \( A := (Z - \mathbb{E}[Z])^2 \) conditioned on \( B = N \), the number of iterations around the loop. The notation \( Z_N \) is used to indicate that the value of \( Z \) is conditioned on the number of iterations.

**Eqn. 13 to 14** Expand the squared expression.

**Eqn. 14 to 15** Apply \( \mathbb{E}[c \cdot X] = c \cdot \mathbb{E}[X] \), where \( c \) is a constant.

**Eqn. 15 to 16** Let \( Z_N = \sum_{i=0}^{N} X_i \) where \( X_i \) is the path length of the \( i \)-th iteration of the loop.

**Eqn. 16 to 17** Expand the summations and reorganize assuming the \( X_i \) are independent random variables so that \( \mathbb{E}[X_i \cdot X_j] = \mathbb{E}[X_i]\mathbb{E}[X_j] \) for \( i \neq j \) (recall the covariance of two independent random variables \( X \) and \( Y \) is zero, and that \( \text{COV}(X,Y) := \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \); see [Leon-Garcia 1994, pp.233-234]).

**Eqn. 17 to 18** Substitute \( \sigma \) for \( \mathbb{E}[X^2] \) and \( \mu \) for \( \mathbb{E}[X] \). Note that \( \mathbb{E}[Z] = \frac{p\mu}{1-p} \) as shown in Section 3.3.4.

**Eqn. 18 to 19** Expand the product of terms and move constants outside of the summation.

**Eqn. 19 to 20** Apply the following simplifications:

\[
\sum_{n=0}^{\infty} p^n = \frac{1}{1-p}; \quad \sum_{n=0}^{\infty} np^n = \frac{p}{(1-p)^2}; \quad \sum_{n=0}^{\infty} n^2 p^n = \frac{p(p+1)}{(1-p)^3}
\]

The first is the well known geometric series summation. The others can be derived by differentiation.

**Eqn. 20 to 21** Cancel identical factors from numerator and denominator.

**Eqn. 21 to 22** Expand and cancel terms.

**Eqn. 22 to 23** Cross multiply and combine the second and third terms.

**Eqn. 23 to 24** Simplify the second term and replace \( \mu := \mathbb{E}[X] \) by letting \( m := \mathbb{E}[X] \), and replace for \( \sigma := \mathbb{E}[X^2] \) by using the relation \( \text{VAR}(X) := \mathbb{E}[X^2] - \mathbb{E}[X]^2 \) and letting \( v := \text{VAR}(X) \).

REFERENCES


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.