A Framework for Modeling and Optimization of Prescient Instruction Prefetch*

Tor M. Aamodt [†‡] Pedro Marcuello [§] Paul Chow [‡] Antonio González [§]
Per Hammarlund [¶] Hong Wang [†] John P. Shen [†]

[†]Microprocessor Research, Intel Labs
Santa Clara, CA 95054, USA

[‡]Dept. of Electrical and Computer Engineering
University of Toronto, Canada

[§]Intel Barcelona Research Center
Universitat Politècnica de Catalunya, Spain

[¶]Desktop Products Group, Intel Corp.
Hillsboro, OR 97124, USA

ABSTRACT

This paper describes a framework for modeling macroscopic program behavior and applies it to optimizing prescient instruction prefetch—a novel technique that uses helper threads to improve single-threaded application performance by performing judicious and timely instruction prefetch. A helper thread is initiated when the main thread encounters a spawn point, and prefetches instructions starting at a distant target point. The target identifies a code region tending to incur I-cache misses that the main thread is likely to execute soon, even though intervening control flow may be unpredictable. The optimization of spawn-target pair selections is formulated by modeling program behavior as a Markov chain based on profile statistics. Execution paths are considered stochastic outcomes, and aspects of program behavior are summarized via path expression mappings. Mappings for computing reaching, and posteriori probability; path length mean, and variance; and expected path footprint are presented. These are used with Tarjan's fast path algorithm to efficiently estimate the benefit of spawn-target pair selections. Using this framework we propose a spawn-target pair selection algorithm for prescient instruction prefetch. This algorithm has been implemented, and evaluated for the Itanium® Processor Family architecture. A limit study finds 4.8% to 17% speedups on an in-order simultaneous multithreading processor with eight contexts, over nextline and streaming I-prefetch for a set of benchmarks with high I-cache miss rates. The framework in this paper is potentially applicable to other thread speculation techniques.

*E-mail addresses: {aamodt, pc}@eecg.toronto.edu, {pedro.marcuello,antoniox.gonzalez, per.hammarlund, hong.wang, john.shen}@intel.com. Tor Aamodt, and Paul Chow were partly supported by the Natural Sciences and Engineering Research Council of Canada.

SIGMETRICS'03, June 10–14, 2003, San Diego, California, USA.

Categories and Subject Descriptors

C.0 [**General**]: Modeling of computer architecture

General Terms

Algorithms, Measurement, Performance, Design, Theory

Keywords

Multithreading, Helper Threads, Analytical Modeling, Optimization, Path Expressions, Instruction Prefetch

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) [25, 8, 12] 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 [10] the performance of many single-threaded applications does not benefit from SMT. Recently, a number of proposals have been put forth to exploit SMT resources to improve the latency of single-threaded applications [7, 3, 28, 19, 16, 27, 1, 14, 6, 5, 4, 13]. 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. To our knowledge, there has been little if any published work that is 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.

Dubois and Song proposed assisted execution as a generic way to use multithreading resources to improve single threaded application performance [7]. 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 [3], and subsequently proposed hardware mechanisms for dynamically constructing and spawning subordinate microthreads to predict difficult-path branches [4]. Weiser [26] proposed *program slicing* to find that 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 [28]. They subsequently implemented hand-crafted speculative slices to precompute branch outcomes and data prefetch addresses [27]. Roth and Sohi [19] 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 [20]. Moshovos et al. proposed Slice Processors, a hardware mechanism for dynamically constructing and executing slice computations for generating data prefetches [16]. Annavaram proposed Dependence Graph Precomputation [1], which effectively uses the predicted stream of instructions to produce accurate dynamic slices. Luk proposed software controlled preexecution [14] as a mechanism to prefetch data by executing a future portion of the program. Collins et al. proposed speculative precomputation [6], and later dynamic speculative precomputation [5] 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 [13].

One task common to all these techniques is the selection of a trigger point where a helper thread should be spawned off for precomputation leading to a target branch or load. 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 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 independence between spawn and target common to such idioms, Marcuello and Gonzalez [15] 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.

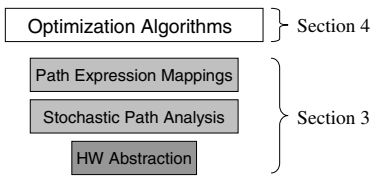

Figure 1: Modeling and Optimization Framework

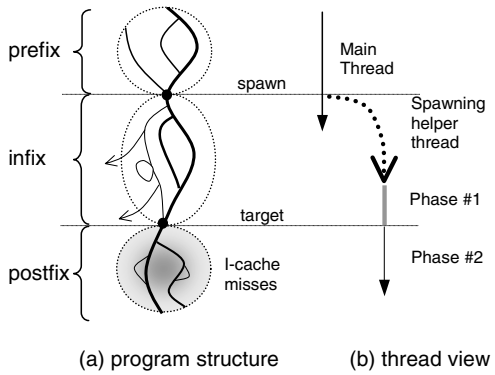

Figure 2: Prescient Instruction Prefetch. (a) Control-flow graph with spawn and target points highlighted. From profiling, the region labeled postfix is known to encounter heavy instruction cache misses. (b) Two phases of helper thread execution: Phase #1 live-in precomputation; Phase #2 postfix precomputation and instruction prefetch.

By using a spare thread context to speculatively execute a future, probabilistically control-independent, 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 [23]. Figure 1 depicts the overall framework for optimization. Using this framework, we propose and evaluate a simple 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. Finally, Section 6 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.

In general, the helper thread may need to precompute some live-in values before starting to execute the program code in the postfix region. Since effective instruction prefetch requires accurate resolution of branches in the postfix region, the precomputation consists of the backward slice of these branches. 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. 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 effectively 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 that proposed by Reinman et al. [18], in that it uses a global view of control-flow gained from 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.

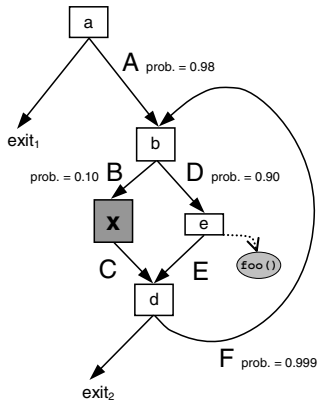

Figure 3: Control flow graph fragment with edge profile information. Block e calls subroutine `foo()`.

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

***Example:* Spawn-Target Selection Tradeoffs**

Figure 3 depicts a control-flow graph fragment. Nodes represent basic blocks, and edges represent potential control flow transfer between two 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 places 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 of its callees (if any) require more instruction storage than the cache capacity, 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 the type of challenges illustrated in this example, 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. 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[23] 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 6).

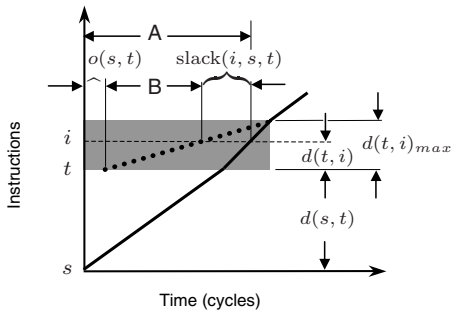

Figure 4: Illustration of Equations 1 and 2.

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 4):

$$\text{slack}(i,s,t) = \overbrace{(\text{CPI}_m(s,t)\cdot d(s,t) + \text{CPI}_m(t,i)\cdot d(t,i))}^{\text{A}} \underbrace{-\text{CPI}_h(t,i)\cdot d(t,i)}_{\text{B}} - o(s,t) \quad (1)$$

where $\text{CPI}_m(s,t)$ represents the average cycles per fetched instruction for the main thread when traversing the infix region between this particular instance of s and t; $\text{CPI}_m(t,i)$ represents the average cycles per fetched instruction for the main thread in the postfix region between t and i; $\text{CPI}_h(t,i)$ 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 not that of the helper thread, 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,

$$d(t,i)_{max} = \frac{\text{CPI}_m(s,t)\cdot d(s,t) - o(s,t)}{\text{CPI}_h(t,i) - \text{CPI}_m(t,i)} \quad (2)$$

Figure 4 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 point is where the dotted line intersects the solid line. The distance computed in Equation 2 corresponds to the height of the shaded box in Figure 4.

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. 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., $o(s,t) = 0$),

$$\begin{matrix}\#\text{ helper threads for} \\ \text{full coverage}\end{matrix} \geq \left\lceil \frac{\text{CPI}_h}{\text{CPI}_m} \right\rceil$$

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. An exploration of techniques to increase the efficiency of helper threads by decreasing CPI_h is beyond the scope of this paper.

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 [9]. 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 [2], 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 *necessity* of instruction prefetches can be assessed by analyzing the instruction footprint distribution along paths between the target and the spawn. The *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 *accuracy*, and *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 [22, 23].

Given a sequence of branch outcomes, the *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. 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 [15], 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%). Similarly, the *posteriori probability* 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 [22] is simply a regular expression summarizing all paths between two points in a graph. For example, in Figure 3 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:

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

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, "$\mathsf{A} \cup \mathsf{A}$" enumerates A twice so this path expression is ambiguous (for a more formal definition, see Tarjan [23, 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 $O(m\alpha(m, n))$ time on reducible flow graphs, where n is the number of nodes and m is the number of edges and α is a functional inverse of of Ackermann's function [23]. This means we get a path expression for every pair of basic blocks in a procedure in $O(nm\alpha(m, n))$ 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 1 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

Table 1: Path Expression Mappings

		Reaching Probability	Expected Path Length	Path Length Variance
concatenation	$[R_1 \cdot R_2]$	pq	$X + Y$	$v + w$
union	$[R_1 \cup R_2]$	$p + q$	$\frac{pX + qY}{p+q}$	$\frac{p(v+X^2)+q(w+Y^2)}{p+q} - \left(\frac{pX+qY}{p+q}\right)^2$
closure	$[R_1^*]$	$\frac{1}{1-p}$	$\frac{pX}{1-p}$	$\frac{p(v+X^2)}{1-p} + \left(\frac{pX}{1-p}\right)^2$

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 [22], and dataflow frequency analysis [17]. 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 1, 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, the assumption 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.

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 must start with edge A and end with edge B, however, 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:

$$\begin{aligned} \mathsf{P(a,x)} &= \mathsf{A} \cdot (((\mathsf{B} \cdot \mathsf{C}) \cup (\mathsf{D} \cdot \mathsf{E})) \cdot \mathsf{F})^* \cdot \mathsf{B} \\ [\mathsf{P(a,x)}] &= 0.98 \cdot \left(\frac{1.0}{1.0 - (0.1(\underline{0.0}) + 0.9(1.0)) \cdot (0.999)} \right) \cdot 0.10 \\ &\approxeq 0.97 \end{aligned}$$

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, 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 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 $< p, X, v >$ 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 1, 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 1 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.

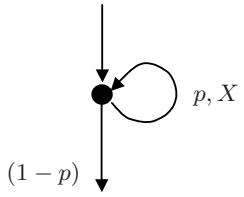

Figure 5: Analysis of the closure operator.

For a given path expression R, let $\sigma(R)$ represent the set of all paths enumerated by R. 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. 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:

$$\begin{aligned}[R_1 \cup R_2] \quad = \quad & E[R_1|\text{follow } p \in \sigma(R_1)] \cdot \Pr(\text{follow } p \in \sigma(R_1)) + \\ & E[R_2|\text{follow } p \in \sigma(R_2)] \cdot \Pr(\text{follow } p \in \sigma(R_2))\end{aligned}$$

To derive the formulation for closure we exploit the fact that the sum of a set of independent random variables is the sum of the expected values and focus on examining a loop with expected path length per iteration of X and backedge probability of p (see Figure 5). 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.

$$\begin{aligned}E[\,{}^{\text{closure}}_{\text{length}}\,] \quad = \quad & \sum E[\text{length}(p_i)] \cdot \Pr(p_i) \\ & p_i = \text{path formed by iterating } i \text{ times} \\ = \quad & 0 \cdot (1-p) + X \cdot p(1-p) + 2X \cdot p^2(1-p) + ...\end{aligned}$$

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:

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

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 detailed derivation is beyond the scope of this paper.

3.3.5 Expected Path Footprint

Assuming x and y are in the same procedure, 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:

$$F(x,y) = \frac{1}{RP(x,y)} \sum_v \text{size}(v) \cdot RP_\alpha(x, v|\neg y) \cdot RP_\beta(v, y) \quad (3)$$

where the summation runs over all blocks v on any path from x to y for which y only appears as an endpoint[1], 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 through y,

$$RP_\alpha(x, v|\neg y) = \begin{cases} RP(x,v) \text{ s.t. no path thru } y & \text{if } x \neq v, \\ 1 & \text{if } x = v \end{cases} \quad (4)$$

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). $RP_\beta(x, y)$ is defined as,

$$RP_\beta(x,y) = \begin{cases} RP(x,y) & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} \quad (5)$$

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 the sum of the fraction of times we traverse from x to y following path p, times the footprint of path p, over all paths from x to y such that y only appears as the endpoint of any particular path:

$$F(x,y) = \sum_{p \in \sigma(P_o(x,y))} \text{freq}(p|x,y) \cdot f(p)$$

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, 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 the latter equation 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:

$$F(x,y) = \frac{1}{RP(x,y)} \cdot \sum_{v \in B} \text{size}(v) \cdot \sum_{p \in P'_o} \Pr(p) \quad (6)$$

[1]This set can be found while evaluating the reaching probability.

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 $P_o^{'}$ 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 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 as the product of these factors:

$$\sum_{p \in P_o^{'}} \Pr(p) = RP_\alpha(x, v|\neg y) \cdot RP_\beta(v, y)$$

Substituting this result into Equation 6 yields Equation 3.

Note that $RP_\alpha(x, v|\neg y) \leq RP(x, v)$. Thus, we may conservatively estimate path footprints by using $RP_(x, v)$ rather than $RP_\alpha(x, v|\neg y)$. 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

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 is 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 due to 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. 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 6.

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

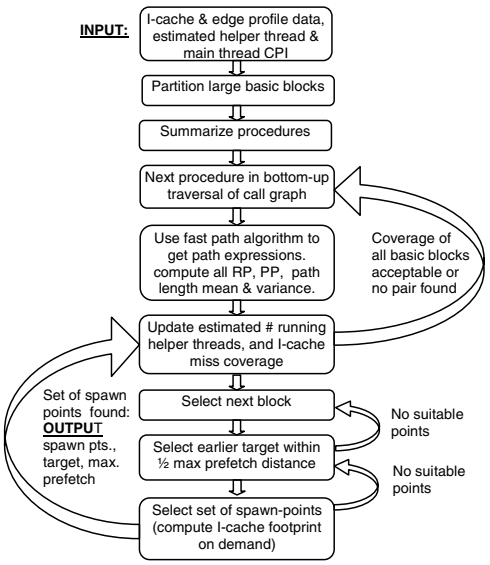

Figure 6: Spawn-Target Selection Algorithm

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 edge from the call instruction to the next instruction. 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. Spawn and target locations are constrained to the beginning of basic blocks to reduce complexity—without performing this step some instructions may not be prefetched as no admissible target would be close enough. 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 and variance, 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.

For each selected block v, potential targets are earlier blocks with high posteriori probability of being visited before v. Thus, the targets 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[2] 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 in turn, 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 threshold[3], by computing a heuristic value indicating their effectiveness as spawn points for the target. In particular, the merit of a given spawn-point s 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 footprints are less than the cache size, because this condition implies a greater likelihood the prefetched instructions are still in the I-cache (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. The third factor is the size of the postfix region the helper thread can prefetch by finding the maximum postfix length that always provides a minimum threshold probability[4] 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). 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 Spec2000 and one from Spec95 that incur significant instruction cache misses on the baseline processor model. The application binaries were built with the Intel Electron compiler, all, except for `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 2, which also quantifies the number of dynamic occurrences of these spawn target pairs, and the average distance between spawn and target (infix), and the 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 indicates helper threads do run slower than the main thread.

Table 2: Spawn-Target Characteristics

benchmark	# static	# dynamic	infix	postfix
145.fpppp	62	378528	622	162
177.mesa	34	210519	1186	255
186.crafty	166	560200	573	129
252.eon	152	407516	691	120
255.vortex	1348	438722	1032	142

The baseline architecture is a cycle-accurate research in-order SMT processor model for the Itanium® architecture [11] based upon SMTSIM [25] with microarchitectural parameters as summarized in Table 3. 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

[2]The latter was set experimentally to 100 Itanium® bundles. Each bundle contains two, or (more usually) three instructions.
[3]Experimentally set to 0.95
[4]Experimentally set to 0.5

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.

Table 3: Processor Resources

Threading	SMT processor with 2, 4, or 8 hardware threads.
Pipelining	In-order: 12-stage pipeline.
Fetch	2 bundles from 1, or 1 bundle from 2 threads prioritize main thread, helpers ICOUNT [24].
I-Prefetch	next line prefetch (triggered on miss) stream prefetcher (triggered by compiler hints) max. 4 outstanding prefetches per context
Branch Pred.	2k-entry gshare. 256-entry 4-way assoc. BTB. helper threads: oracle branch prediction
Issue	2 bundles from 1, or 1 bundle from 2 threads prioritize main thread, helpers: round-robin.
Function units	4 int., 2 fp., 3 br., and 2 data mem. ports
Register files per thread	128 int, 128 fp, 64 predicate, 8 br. 128 control registers.
Caches	L1 (separate I&D): 16KB (each). 4-way. 1-cyc. L2 (shared cache): 256KB. 4-way. 14-cycles L3 (shared cache): 3072KB. 12-way. 30-cycles. Fill buffer: 16 entries. caches have 64B lines. helper threads: infinite store buffer 1-cyc.
Memory	230-cycles. TLB miss penalty: 30 cyc.

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

To assess the accuracy of the modeling framework, 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. The `Evaluate()` subroutine accounts for roughly one quarter of all I-cache misses in `crafty`, and its control flow graph is non-trivial—containing 230 basic blocks, 345 edges, and several loops. Figure 7(a)-7(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 7(b), (c) and (d) are Itanium® bundles, while Figure 7(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. 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 7(a)-7(d) underscores the robustness of the path expression based methodology described in Section 3, and Table 1.

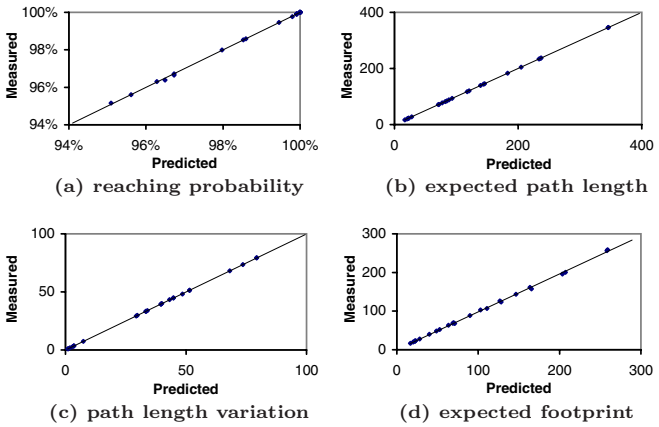

Figure 7: Monte Carlo Simulation vs. Framework.

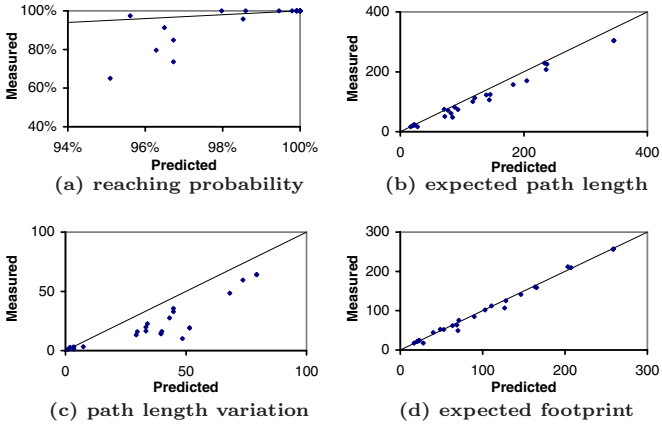

Figure 8: Execution Trace Statistics vs. Framework.

Figures 8(a)-8(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 8(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, 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 [21], and taking into account the impact of branch correlation (perhaps by extending the current approach to use higher order Mar-

kov chain models) these quantities may be predicted with greater accuracy, however such enhancements are beyond the scope of this paper.

5.3 Results

Figure 9 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.

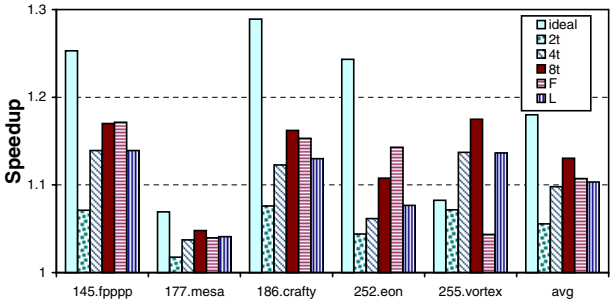

Figure 9: Limit study performance. "Perfect I$" speedup if all I-accesses hit. "nt" prescient instruction prefetch speedup for n thread contexts. "F" spawn-target pairs selected assuming four thread contexts and *fast* helpers (low CPI_h), performance modeled so that helpers do not block on an I-cache miss. "L" 4t and spawn-target pairs selected with *low* reaching probability threshold (0.75).

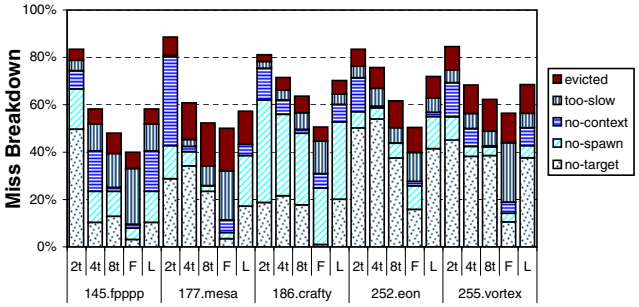

Figure 10: Breakdown of remaining I-cache misses relative to baseline.

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 10 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 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 a early enough location that was control flow correlated because we constrained spawn-target pairs to be within the same function body. Figure 10 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 9 and Figure 10, 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 finding ways to improve helper thread efficiency. 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 to include 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. 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. 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 prefetch, demonstrating the effectiveness of the framework, and prescient instruction prefetch. Important topics for future study are the impact of precomputation overhead and hardware optimizations for prescient instruction prefetch.

7. ACKNOWLEDGEMENTS

We would like to thank Murali Annavaram, Bob Colwell, Edward Grochowski, Steve (Shih-wei) Liao, James Psota, Ronny Ronen, Lesley Shannon, Perry Wang, Craig Zilles, and the anonymous referees for their valuable comments on this work.

8. REFERENCES

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *28th International Symposium on Computer Architecture*, pages 52–61, 2001.

[2] P. P. Chang, S. A. Mahlke, and W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software – Practice and Experience*, 21(12):1301–1321, 1991.

[3] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *26th International Symposium on Computer Architecture*, pages 186–195, 1999.

[4] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *29th International Symposium on Computer Architecture*, pages 307–317, 2002.

[5] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic Speculative Precomputation. In *34th International Symposium on Microarchitecture*, pages 306–317, 2001.

[6] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, pages 14–25, 2001.

[7] M. Dubois and Y. Song. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[8] J. Emer. Simultaneous multithreading: Multiplying alpha's performance. Microprocessor Forum, Oct. 1999.

[9] D. W. Hammerstrom and E. S. Davidson. Information Content of CPU Memory Referencing Behavior. In *4th International Symposium on Computer Architecture*, pages 184–192, 1977.

[10] G. Hinton and J. Shen. Intel's multi-threading technology. Microprocessor Forum, Oct. 2001.

[11] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–23, 2000.

[12] Intel Corporation. Special Issue on Intel Hyper-Threading Technology in Pentium® 4 Processors. Intel Technology Journal. Q1 2002.

[13] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 117–128, 2002.

[14] C.-K. Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *28th International Symposium on Computer Architecture*, pages 40–51, 2001.

[15] P. Marcuello and A. Gonzlez. Thread-Spawning Schemes for Speculative Multithreading. In *8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[16] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *15th International Conference on Supercomputing*, pages 321–334, 2001.

[17] G. Ramalingam. Data flow frequency analysis. In *SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 267–277, 1996.

[18] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *32nd International Symposium on Microarchitecture*, pages 16–27, 1999.

[19] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *7th International Symposium on High-Performance Computer Architecture*, pages 37–48, 2001.

[20] A. Roth and G. S. Sohi. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *35th International Symposium on Microarchitecture*, pages 430–441, 2002.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[22] R. E. Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, 1981.

[23] R. E. Tarjan. Fast Algorithms for Solving Path Problems. *Journal of the ACM*, 28(3):594–614, 1981.

[24] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, 1996.

[25] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, pages 392–403, 1995.

[26] M. Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, 1981.

[27] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture*, pages 2–13, 2001.

[28] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th International Symposium on Computer Architecture*, pages 172–181, 2000.