Prescient Instruction Prefetch

Tor Aamodt[1,2], Pedro Marcuello[3], Paul Chow[2], Per Hammarlund[4], Hong Wang[1]

[1]Microprocessor Research, Intel Labs
[2]Department of Electrical and Computer Engineering, University of Toronto
[3]Departament d´Arquitectura de Computadors, UPC, Barcelona
Intel Barcelona Research Center, Intel Labs
[4]Desktop Products Group, Intel Corp.

ABSTRACT

This paper introduces prescient instruction prefetch, a 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. The execution of the helper thread prefetches instructions starting at a distant target point that identifies a code region that the main thread is likely to execute soon and tends to incur I-cache misses. This paper formulates the identification of appropriate spawn and target points for the associated helper threads as an optimization problem by modeling runtime program behavior as a Markov chain with statistics derived from profile data. This formulation enables the accurate estimation of important statistical quantities related to helper thread execution via a simple and efficient computational strategy based upon Tarjan's fast path expression algorithm. Using this formulation we propose a spawn-target pair selection algorithm. This algorithm has been implemented for the Itanium Processor Family (IPF) architecture. As an initial limit study we present simulation results indicating that helper threads given perfectly predicted register and memory live-in values at the target can achieve speedups in the range of 6% to 63% on an in-order SMT machine with four hardware thread contexts for a select set of benchmarks that have high I-cache miss rates. With increasing levels of realism, we find speedups ranging from 0.5% to 54% are feasible when taking into account the overhead of live-in precomputation. The approach presented in this paper is potentially applicable to other thread speculation techniques.

Keywords: instruction prefetch, path expression algorithm, analytical modeling, optimization, multithreading.

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. This paper focuses on workloads that incur significant I-cache misses. We investigate the use of spare simultaneous multithreading (SMT) [20] thread resources for prefetching instructions. SMT has been shown to be an effective way to boost throughput performance with limited impact on processor die area [9]. However, many single-threaded applications do not benefit from SMT. Recently, a number of proposals have been put forth to exploit SMT resources and helper threads to improve the latency of single-threaded applications. In particular, several studies have investigated using helper threads in the form of program-slice based precomputation for reducing the latency of delinquent loads, and branch mispredictions. However, to our knowledge, there has been little if any published work that is focused specifically on improving I-cache performance by exploiting helper threads.

Song and Dubois proposed *assisted execution* as a generic way to use multithreading resources to improve single-threaded application performance [18]. Chappel 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 using SSMT to provide a very large local pattern history based branch predictor [3] and later proposed hardware mechanisms for dynamically constructing and spawning subordinate microthreads to predict difficult-path branches [4]. Zilles and Sohi analyzed the dynamic backward slices of performance degrading instructions, and suggested the feasibility of using program slices to pre-execute them [22]. They subsequently implemented hand crafted speculative slices to precompute branch outcomes and data prefetch addresses [1]. Roth and Sohi [17] proposed using data driven multithreading (DDMT) to dynamically prioritize sequences of operations leading to branches that mispredict or loads that miss. Moshovos et al. proposed Slice Processors, a hardware mechanism for dynamically constructing and executing slice computations for generating data prefetches [1]. Annavaram et al. proposed Dependence Graph Precomputation [1]. Luk proposed software controlled preexecution [12]. Collins et al. proposed speculative precomputation [7], and later dynamic speculative precomputation [6] as techniques to leverage spare SMT resources for generating long range data prefetches. They showed the importance of chaining to achieve effective data prefetching. Liao et al. extended this work by implementing a post-pass compilation tool to augment a program with automatically generated precomputation threads for data prefetching [11].

One task common to these techniques is the selection of a trigger point where a helper thread should be spawned off for precomputation preceding a *target* branch or load. As a matter of convenience, throughout this paper, we call this trigger the *spawn* point[1]. For instruction prefetch, the target identifies not just a single instruction, but rather the beginning of an entire region of code. A similar notion can be found in speculative multithreading. For instance, Marcuello and Gonzalez recently proposed using a notion of reaching probability to define *control-quasi-independent points* [13], which essentially serve as spawn-target pairs.

This paper makes three key contributions: One, we propose using spare SMT thread contexts to perform *prescient instruction prefetch* for single-threaded applications. Two, we present a formal mathematical framework for effective selection of spawn-target pairs that generalizes, in a rigorous way, the probabilistic framework introduced by Marcuello and Gonzalez [13]. Three, it proposes and quantitatively evaluates a specific optimization algorithm for spawn-target pair selection within this framework.

By using a spare thread context to speculatively execute a probabilistically control-independent future region of the same program but far ahead of the main thread, prescient instruction

[1] The spawn point is associated with a point in the execution of the main thread. In the case of chaining triggers [7][6][11], the association with the main thread is implicit via a sequence of helper threads. In this paper we focus on spawn points for instruction prefetch that are embedded directly in the main thread.

prefetch can fetch most instructions that the main thread is likely to encounter soon, while accurately executing those instructions pertaining to the control flow.

In this paper we present an algorithm for combining control-flow edge profile information with instruction cache miss profile information to judiciously select spawn-target pairs for instruction prefetch. The basic idea behind the algorithm is to estimate the average amount of useful prefetching generated when triggering a helper thread based upon the profile information. We evaluate the potential of the overall technique via detailed performance simulation assuming varying levels of realism starting with a model in which live-in register and memory values at the target are assumed to be predicted perfectly at no cost as soon as the helper thread spawns. We then investigate the feasibility of the technique under the more realistic assumption of using program slicing to precompute the subset of live-ins required to correctly execute the branches following the target.

The rest of this paper is organized as follows: In Section 2 we introduce the prescient instruction prefetch paradigm. Section 3 describes the mathematical framework we use to characterize program behavior, formulates key statistical quantities related to helper thread execution, and describes our novel use of path expressions to compute these statistical quantities. In Section 4 we present an algorithm that uses the analytical framework to select the spawn-target pairs that define instruction prefetch helper threads. Section 5 discusses our simulation results, and Section 6 concludes.

2. Prescient Instruction Prefetch

Prescient instruction prefetch involves the use of helper threads to perform instruction prefetch to reduce I-cache misses for single-threaded applications. Here the term prescient carries two connotations: One, that the helpers are initiated in a timely and judicious manner based upon a global analysis of program behavior, and two, that the instructions prefetched are useful as the helper thread follows the same path through the program that the main thread will follow when it reaches the code the helper thread prefetches.

Profile information is used to identify code regions that incur significant I-cache misses; we then further identify *target* points as instructions closely preceding the basic blocks that incur the I-cache misses. For each target point identified in the single-threaded application, a set of corresponding *spawn* points are identified that can serve as trigger points for initiating the execution of a helper thread for prefetching instructions after the target point. 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 1 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.

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. As shown in Figure 1(b), helper thread execution consists of two phases: The first phase, *target live-in precomputation*, reproduces the effect of the code

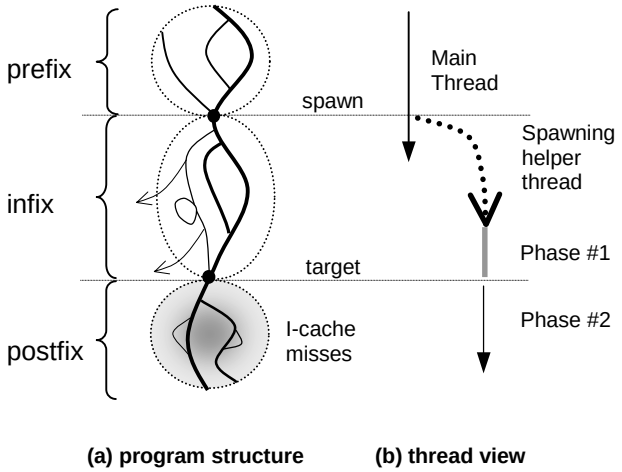

Figure 1: Precient Instruction Prefetching. (a) Control-flow graph with *spawn* and *target* points highlighted. From profiling, the region labeled postfix is known to encounter heavy instruction cache misses. (b) Two phases of helper thread execution: Phase #1 *target* live-in precomputation; Phase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

skipped over in the infix that relates to the resolution of branches in the postfix. 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 distinguishes itself from traditional branch predictor based instruction prefetch mechanisms, such as that proposed by Reinman et al. [16], in that it uses a global view of control-flow to aid in anticipating potential performance degrading regions of code from a long-range. The key challenges are: One, identification of an appropriate set of distant yet strongly control-flow correlated pairs of spawn-target points; and two, accurate resolution of branches after the target.

We tackle these challenges in the following subsections by introducing a rigorous Markov model based formulation of global control-flow and instruction memory behavior, and showing how to analyze this model to optimize the selection of spawn-target pairs for helper threads. Before going into detail we consider a brief example.

Example: Spawn-Target Pair Selection Tradeoffs

Consider the control-flow graph fragment in Figure 2. In this figure squares represent basic blocks, and edges represent potential control flow due to branches. The shaded block labeled `X` is known from cache profiling to suffer many instruction cache misses. Each edge is labeled with the probability the main program will take the given path. The two 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 the probability of exiting the loop each iteration (i.e., from block `d`) is much smaller than the probability of transitioning to `X`. Even though each choice of spawn is roughly as "good" as any other in this sense, as we show next, not all spawn points are necessary as effective as each other.

For instance, consider the impact of the size of subroutine `foo()` on spawn-target pair selection. If `foo()` together with blocks `b`, `d`, and `e` fit inside the instruction cache, then initiating a prefetch of block `X` starting at block `a` is good because the loop will likely iterate several times before the branch at the end of `b` transitions to `X`. On the other hand, if `foo()`together with its callees

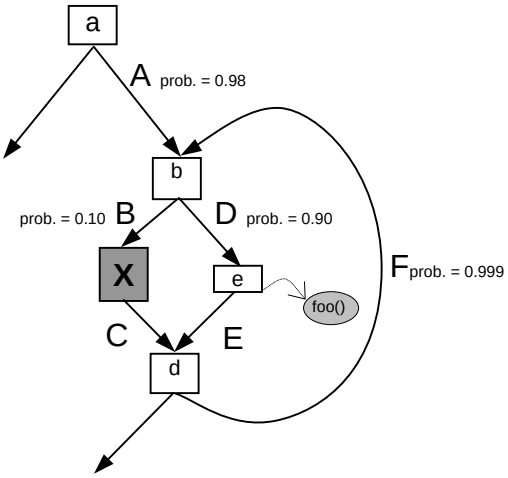

Figure 2: Control flow graph fragment with edge profile information. Block `e` calls subroutine `foo()`. Each edge is labeled with a transition probability (prob.) unless the value is exactly one.

require more space than the cache capacity, there is no point in spawning a prefetch thread directly targeting X from *any* block because the instructions it will fetch will almost certainly be evicted from the cache by a call to `foo()` before the main thread reaches X again due to the low probability of branch b transitioning to X. A better solution in this case would be to target block b because evaluating the branch at the end of block b will determine whether to prefetch X. A good set of spawn points for b in this case might be the beginning of blocks a and d, particularly if these blocks contain a significant amount of code unrelated to the branch at the end of block b. If we can only choose one spawn locations among a and d due to resource limitations, the more profitable choice is d because a would only cover misses in the event that control goes directly from a to b to X, while d would cover the cache misses in all other cases. In Section 3 we formulate a mathematical framework for quantifying the tradeoffs highlighted in this example.

3. Analytical Framework

Effective spawn-target pairs should meet the following necessary conditions: One, the helper thread must run ahead, but not so far ahead as to evict instructions soon to be used by the main thread. Two, the spawn point and the target point should be highly correlated during the main execution, in other words when the main thread reaches the spawn point, the target is highly likely to be seen. Three, the helper thread must follow the same path the main thread will execute when it reaches the target.

In this section a set of key statistical quantities characterizing these necessary conditions, and thus essential to effective spawn-target pair selection, are introduced. Furthermore, a general yet efficient technique for computing these techniques is also described. The resulting analytical framework serves as a foundation for the algorithm to be described in Section 4.

3.1. Prefetch Slack

Analytically, we model the *prefetch slack* of a particular instance of instruction *i*, targeted by a helper thread spawned at *s* with target *t*, using the following expression:

$$\text{slack}(i,s,t) = \underbrace{\left(\text{CPI}_{\text{m}}(s,t)\cdot d(s,t) + \text{CPI}_{\text{m}}(t,i)\cdot d(t,i)\right)}_{\text{A}} - \underbrace{\text{CPI}_{\text{h}}(t,i)\cdot d(t,i)}_{\text{B}} - o(s,t) \quad (1)$$

here *d(x,y)* denotes the *distance* between instructions *x* and *y*, measured in number of instructions executed; *o(s,t)* represents the

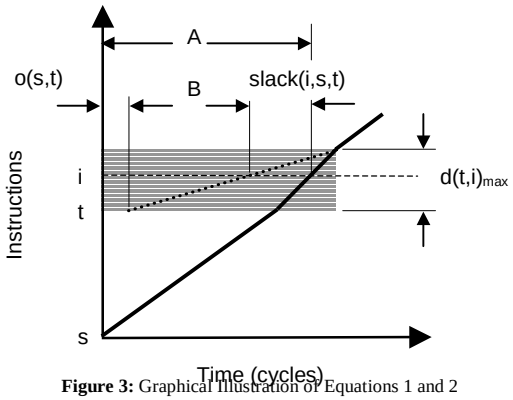

Figure 3: Graphical Illustration of Equations 1 and 2

overhead in cycles incurred by the helper thread of spawning at *s* and performing precomputation for live-ins at *t*; $\text{CPI}_{\text{m}}(s,t)$ represents the average cycles per fetched instruction of the main thread in going between the particular instance of *s* and *t*; $\text{CPI}_{\text{m}}(t,i)$ represents the average cycles per fetched instruction of the main thread in going between the particular instance of *t* and *i* in postfix; and $\text{CPI}_{\text{h}}(t,i)$ represents the average number of cycles per fetched instruction for the helper thread in going between the particular instance of *t* and *i* in postfix.

A given instance of a helper thread can reduce the fetch latency of the target instruction in the main thread if it has *positive* prefetch slack. Effective prefetching will increase the average IPC of the main thread but not necessarily that of the helper thread so that while $\text{CPI}_{\text{h}}(t,i)$ stays relatively constant, $\text{CPI}_{\text{m}}(t,i)$ decreases leading to an upper bound on how far ahead a helper thread the performs useful prefetching can run before the main thread will catch up to it[2]. In particular, the main thread will catch the helper thread if slack(*i*,*s*,*t*) is zero. From Equation 1 this is equivalent to:

$$d(t,i)_{\max} = \frac{CPI_m(s,t)}{CPI_h(t,i) - CPI_m(t,i)} \cdot \left(d(s,t) - o(s,t)\right) \quad (2)$$

Figure 3 portrays a graphical representation of these concepts by plotting instructions executed versus time along some portion of an execution trace. The solid line indicates the progress of the main thread, 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 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.

Note that the infix slice, corresponding to the first of the two phases of helper thread execution (as shown in Figure 1), depends strongly upon the distance between spawn and target. This is because increasing the amount of program executed between spawn and target increases the number of operations that can potentially affect the outcome of branches in the postfix. To model this effect requires dataflow analysis. For practical purposes, in this paper we assume this overhead increases proportional to spawn-target distance.

3.2. A Statistical Model of Program Execution

In this section we describe a statistical model of program execution. This model captures the effects of salient events such as control flow

[2] On an SMT processor, as studied in this paper, helper thread IPC may also be diminished by resource contention with the main thread. On a CMP processor, where helper threads run on different cores from the main thread, such constraints can be relaxed.

transitions and instruction cache accesses. It does not accurately model the effects of branch mispredictions and data cache misses, which are instead approximated using an average CPI assuming instructions hit in the cache. This model is the basis for the spawn-target selection algorithm.

Two points in a control-flow graph are strongly *control-flow correlated* if the execution of one implies the high probability of the execution of the other. Control-flow correlation consists of two factors: the probability of reaching the target from the spawn given that the current state is the spawn; and the posteriori probability that the spawn precedes the target given that the current state is the target. A high reaching probability increases the likelihood that the speculative execution of the helper thread will be helpful. A high posteriori probability means the associated spawn point usually precedes a given target whereas a low posteriori probability means the target has alternative predecessors that do not include the spawn point.

3.2.1. Modeling Control Flow

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 [8]. A Markov chain is defined by a set of states and a set of 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 readily gleaned from the traditional edge profile [2] that measures the probability that one block flows to another assuming independent branch outcomes.

For *inter*-procedural control flow we model the effect of procedure calls with a restriction on transitions into and out of procedures such that a callee must return to its caller. In particular, we model a transition from the current procedure's Markov model to a subroutine's when the basic block representing the current state ends in a procedure call. When the state associated with the exit block of a callee is entered, control returns to its caller.

This model ignores correlation between branch outcomes, however it can be generalized to exploit higher order branch correlation by defining states to be path segments. Mehofer and Scholz have applied such an approach to probabilistic dataflow analysis by using two-edge profile information [14].

3.2.2. Modeling Instruction Cache Accesses

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 the program entirely determines the contents of this cache independent of the program's static code layout. 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.

3.3. Computing Statistics using Path Expressions

In this section we describe important statistical quantities related to spawn-target point selection. These include the reaching probability, posteriori probability, expected path length, and expected path footprint. We show how to map the evaluation of these quantities onto the classic *path expression problem* [19], which can be efficiently solved using Tarjan's fast path algorithm [20].

3.3.1. Path Expressions

A path expression is simply a regular expression summarizing all paths between two points in a graph. For instance, in Figure 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 is summarized by the path expression *P(a,X)*:

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

Here "$\cup$", "$\cdot$", and "*" denote the regular expression operators union, concatenation, and closure, respectively, and parenthesis 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 zero or more complete iterations around a loop. For reasons that will become more apparent in Section 3.3.2 it is important that the path expressions we use are *unambiguous* in the sense that no path can be enumerated two ways. For example, "$A \cup A$" enumerates *A* twice so this path expression is ambiguous.

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, and the algorithm we describe has overall time complexity of $O(p \cdot n \cdot m \cdot \log n)$, where *p* is the number of procedures in the program and *n* and *m* are the number of basic blocks and control-flow edges, respectively, in the largest function.

3.3.2. Reaching Probability

The *reaching probability, RP(x,y)*, between two 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 [13], 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%). The *intra-procedural* reaching probability can be determined as follows. Ignoring procedure calls, label all transitions in the Markov model with their respective transition probability. We evaluate RP(*x*,*y*), for $x \neq y$, by first setting the probability of edges leaving *y* to zero so that paths through *y* are effectively not included, then "evaluating" the path expression which summarizes all paths from *x* to *y*. Given path expressions R_1 and R_2, with probabilities *p* and *q*, respectively, the regular expression operators are evaluated recursively following the operator precedence: parenthesis, closure, concatenation, and then union. For each operator we apply the "mapping" defined in Figure 4, where the path expression on the left side evaluates to the value computed by the formula on the right hand side.

concatenation	$[R_1 \cdot R_2]$	$= pq$
union	$[R_1 \cup R_2]$	$= p + q$
closure	$[R_1^{*}]$	$= \frac{1}{1 - p}$

Figure 4: Reaching probability mapping

The mapping in Figure 4 is not arbitrary and we note that it also arises in other applications such as the solution of systems of linear equations [19]. The validity of using this mapping for computing 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 union of two path expressions works out to 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 each outcome along the path in the same way that the probability of tossing a coin twice in a row and getting two heads is one-half squared. 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 it implies this is the same as adding the probabilities of each individual path enumerated by the combined path expression.

The mapping for closure arises when considering loops. A detailed explanation is beyond the scope of this paper, however we note that the number computed by the closure mapping is always between one and infinity. This number, which is the sum of the probabilities of taking 0,1,2,… iterations around the loop does not represent a probability because the events added are not independent. This quantity is actually the expected number of times the associated loop will iterate before exiting given it is entered at least once. Note that the path expression based analysis is valid as long as the profile data represents a program run to completion so that in particular there are no loops with loop probability of one.

Example: Reaching Probability Calculation.

Let us determine the reaching probability from a to X in Figure 2: Each path from a to X 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:

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

$$[P(a, X)] = 0.98 \cdot \left(\frac{1}{1.0 - (0.1(\underline{0.0}) + 0.90(1.0)) \cdot (0.999)}\right) \cdot 0.10$$

$$\cong 0.97$$

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 not implicitly enumerated.

This example illustrates an important point: The probability of reaching a block can only be determined by examining global behavior. The probability of taking the direct path 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 , is 0.97, which much higher than 0.098. An alternative perspective on this result is that the probability of exiting the loop per iteration, 0.001, is 100 times lower than the probability of reaching *X* each iteration, 0.1 which means that there are many chances to reach X and it is unlikely for the loop to execute to completion without making at least one transition to X.

3.3.3. Posteriori Probability

Another important quantity in our analysis framework is the probability of having previously visited state *X* since the last occurrence of *Y* (if any), given the current state is *Y*, which we call the *posteriori probability* of *X* given *Y*. For a specific target, choosing a spawn point with low posterior probability is inefficient because it will only trigger prefetch threads for a small fraction of the occurrences of the target.

The intra-procedural posteriori probability from *x* to *y*, $x \neq y$, is found by applying the mapping in Figure 4 on the Markov chain obtained by reversing control flow edges and labeling them with the frequency a predecessor precedes the successor rather than the frequency with which the successor follows the predecessor, after setting the probability of edges from *x* to successors of *x*, as well as the edges from predecessors of *y* to *y* to zero (in both cases, referring to the new edge orientation).

3.3.4. Expected Path Length

Given a sequence of branch outcomes, the *path length* is the number of instructions executed along the associated path through the program. We are interested in estimating the *expected path length* given that execution is currently in block *x* and the program subsequently reaches block *y*. To compute this, with each edge we associate a tuple. The first element represents the probability of branching from the predecessor to the successor, and the second element represents the length of the predecessor basic block. Similarly, for path expressions R_1 and R_2 we associate tuples $\langle p,X \rangle$ and $\langle q,Y \rangle$ where the first element represents the sum of the probabilities over all the associated paths and the second element represents the expected number of instructions executed. The rules for combining these tuples to evaluate path expressions are listed in Figure 5.

concatenation	$[R_1 \cdot R_2]$	$= \langle pq, X + Y \rangle$
union	$[R_1 \cup R_2]$	$= \langle p + q, \frac{pX + qY}{p + q} \rangle$
closure	$[R_1^{*}]$	$= \langle \frac{1}{1 - p}, \frac{pX}{1 - p} \rangle$

Figure 5: Expected path length mapping

This mapping results from analyzing the expected value of the path length given the probabilities of following any particular paths determined by the edge profile data. For brevity the derivation is omitted.

3.3.5. Expected Path Footprint

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. 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:

$$F(x, y) = \frac{1}{RP(x, y)} \cdot \sum_{v} \text{size}(v) \cdot RP_{\alpha}(x, v, y) \cdot RP_{\beta}(v, y) \qquad (3)$$

where the summation runs over all blocks *v* on any path from *x* to *y* for which *y* only appears as an endpoint (this set can be determined while evaluating the reaching probability from *x* to *y*), *size(v)* is the number of instructions in basic block *v*, and $RP_{\alpha}(x,v,y)$ is defined as,

$$RP_{\alpha}(x, v, y) = \begin{cases} \text{reaching probability } x \text{ to } v, \text{ no path through } y & , x \neq v \text{ and } y \neq v \\ RP(x, y) & , x \neq v \text{ and } y = v \\ 1 & , x = v \end{cases}$$

and $RP_{\beta}(x,y)$ is defined as,

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

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 weight the size of each block by the probability it is reached at least once.

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 and *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*. The latter operation can be performed efficiently while evaluating the reaching probability.

4. Spawn-Pair Selection Algorithm

In this section we describe the spawn-target selection algorithm. We have implemented this algorithm for the Itanium® architecture and collected performance data is presented in Section 5 supporting its effectiveness. A high-level block diagram of the algorithm is shown in Figure 6 and described throughout the rest of this section.

The inputs to the algorithm are edge and instruction cache miss profiles (first box in Figure 6) and the output is a set of mutually independent spawn-target pairs. The edge profile data is supplied in the form of the program's procedure control flow graph annotated with basic-block and branch frequencies, and an instruction cache profile indicating the frequency of instruction cache misses in each block. The control flow graphs include information about the procedure calls and returns such as the target of procedure calls and the frequency a given subroutine is called from a given call site. For our purposes we consider a call site to be a basic block boundary, and augment the graph with a symbolic edge from the call instruction to the next instruction. This symbolic edge has edge probability equal to one and path length initially undefined until all subroutines called by the enclosing function have been summarized.

The algorithm begins by splitting control flow graph nodes that represent basic blocks with more instructions than the maximum number of postfix instructions a helper is permitted to execute into a sequence of nodes, each representing only a portion of the basic block. The motivation for this is that spawn and target locations are constrained to the beginning of basic blocks to reduce complexity and without performing this step some instructions could not be prefetched because no potential target point would be close enough that a helper thread could start there and reach the instruction. Next, the algorithm computes procedure summary information. In particular, the expected path length from entry to exit, and reaching probabilities from the entry to each block are computed. The latter are 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 frequency of cache misses they generate. We only target basic blocks that account for the top 90% of all instruction cache misses. In each procedure visited, the reaching probability, expected path length, posteriori probability, and expected posteriori path length are computed between each pair of basic blocks according to the method described in Section 3.3.

We maintain an estimated number of cache misses occurring in each block initialized to the value determined via instruction cache profiling. For each block within the procedure, in descending order of absolute cache miss frequency, we select a target and a set of independent spawn points for that target then update the estimated remaining cache misses in all blocks. The target is an earlier block with high posteriori probability of having been seen before the selected block. The selection process for target and spawn are coupled in the following way: A set of potential targets with posteriori expected distance (computed using the mapping in Figure 5 on the reverse graph) less than the maximum prefetch distance is generated and ranked in descending order by distance from the selected block. By selecting the target to be an earlier block in this

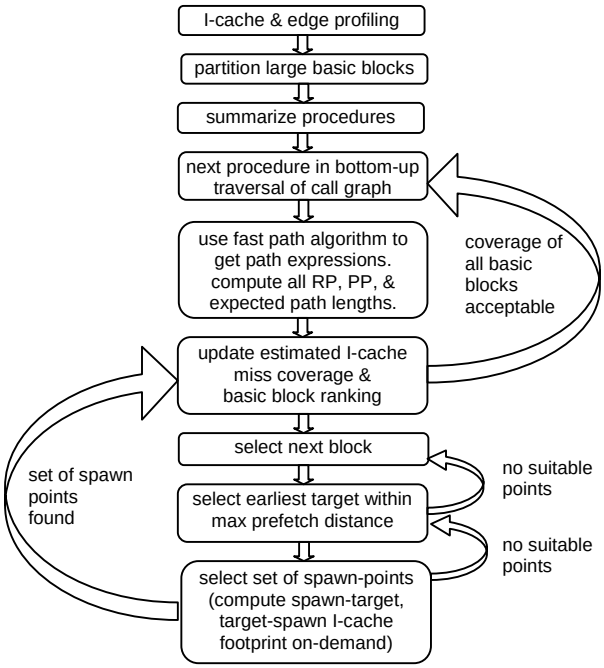

Figure 6: Spawn-Target Selection Algorithm

manner we are simply trying to avoid selecting spawn-target pairs with overlapping postfix regions. For each potential target in turn, a set of independent spawn points is found using the process described in the next paragraph.

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 0.95, by computing a heuristic indicating their effectiveness as spawn points for that target. In particular, the merit of a given spawn-point *s* is computed as the product of several factors. The first factor is the posteriori probability that the spawn precedes the target. Together with the restriction on reaching probability they ensure the spawn and target are highly control flow correlated. The second factor uses the expected path footprint to penalize spawn points with small average target-to-spawn footprints because this condition imply a greater likelihood that the target will is still in the cache since it was last visited. Similarly, this term penalizes spawn points with expected spawn-to-target path footprint larger than the instruction cache capacity. The next factor takes into account the expected slack of prefetches issued by the helper thread by using the expected path length from spawn to target. As slack increases, this term becomes better point that most instructions run in the helper thread would be prefetched before the main thread reaches them.

To keep track of the effect of previously selected spawn target pairs we keep a running estimate of the remaining expected cache misses for each block. We update the expected remaining cache misses for a given block reachable by a helper thread by estimating the amount of miss coverage that block receives using the posteriori probability and a factor quantifying normalized slack to account for the fact that not all prefetches need be timely to contribute to performance.

5. Simulation Results

This section presents a performance evaluation of the prescient instruction prefetch approach based upon cycle accurate simulation. We examine the performance impact of prescient instruction prefetch threads defined by the spawn-target selection algorithm described in

Section 3 under varying assumptions about the spawn overhead and microarchitecture.

5.1. Hardware Model

Our baseline is a research in-order SMT processor model for the Itanium® architecture [10]. The microarchitectural details are summarized in Table 1. By default we assume the processor is equipped with a hardware instruction prefetch mechanism that supports the Itanium® instruction prefetch hints (`br.many`).

We evaluate the potential of prescient instruction assuming varying levels of realism starting with a model in which live-in register and memory values at the target are assumed to be predicted perfectly at no cost as soon as the helper thread spawns. We then investigate the feasibility of the technique under the more realistic assumption of using program slicing to precompute the subset of live-ins required to correctly execute the branches following the target.
We assume a hardware mechanism capable of perfect memory disambiguation [5] and supporting store-to-load forwarding. Furthermore we assume stores executed by helper threads do not commit to memory.

In all cases we assume that stores in the helper thread do no commit to the memory hierarchy but forward their values to subsequent loads in the helper thread assuming perfect memory disambiguation.

5.1.1. Perfect Live-in Prediction (PLP)

Ideally, the helper thread would begin executing at the target as soon as the spawn is encountered by the main thread and would initially see the same architectural state that the main thread will be in when it reaches the target. We model this case by triggering a helper thread when the main thread commits a spawn-point. If no free thread contexts are available the spawn-point is ignored. The helper thread begins fetching from the target the following cycle and runs ahead the maximum prefetch distance before exiting. If the main thread catches up with it before that point, the helper thread stops fetching instructions. Once the helper thread instructions drain from the pipeline the thread context is available to run other helper threads.

Threading	SMT processor with 4 hardware thread contexts.
Pipelining	In-order: 12-stage pipeline.
Fetch per cycle	2 bundles from 1 thread or 1 bundle each from 2 threads
Inst. Prefetch	instruction stream prefetcher triggered by compiler hints maximum 4 outstanding prefetches per thread context
Branch predict.	2k-entry GSHARE. 256-entry 4-way associative BTB.
Issue per cycle	2 bundles from 1 thread or 1 bundle each from 2 threads main thread has priority, helper threads round robin.
Function units	4 int. units, 2 FP units, 3 branch units, 2 memory port
Register files per thread	128 integer registers, 128 FP registers, 64 predicate registers, 8 branch registers, 128 control registers.
Cache structure	L1 (separate I & D): 16KB each. 4-way. 2-cycle latency. L2 (shared cache): 256KB. 4-way. 14-cycle latency. L3 (shared cache): 3072KB. 12-way. 30-cycle latency. Fill buffer: 16 entries. All caches have 64-byte lines.
Memory	230-cycle latency. TLB Miss Penalty: 30 cycles.

Table 1: Processor Resources

5.1.2. Variable Slice (VS)

Next we model the effect of having to precompute the subset of live-ins required to properly resolve the control flow in the postfix region. As a bounding box analysis, we assume the processor has access to the dynamic value and address backward slices [22] needed to accurately resolve the control flow of the postfix region for each dynamic spawn-target pair instance. In the present setting the value slice refers to those operations that compute values directly used to determine branch outcomes, while the address slice are those operations required to compute load and store addresses to facility dataflow via store-to-load dependencies in the value slice [22]. Because the helper thread will execute the code following the target, the live-in precomputation should not include those slice instructions whose original version was from the postfix region.

While we model the value and address slice, for this paper we do not model the control slice. In addition the slices we model could also be made smaller by applying slice optimizations that have been shown to be highly effective in prior studies [17][22][6][4][11].

We model two configurations: In the first we assume single-cycle access to slice instructions. In the second configuration the slices experience normal cache hierarchy latencies and contend with regular application code for space in the instruction cache. In the later case we do not model the potential effect of hardware instruction prefetching on the slice instructions themselves.

benchmark	static # pairs	dyn. #pairs	avg. dist.	#slc inst.
145.fpppp	50	18311	528	68
177.mesa	28	14750	766	321
186.crafty	119	31949	472	57
252.eon	73	19168	732	175
255.vortex	1436	21371	930	51

Table 2: Spawn-target characteristics: static #pairs = number of spawn target pairs selected by our algorithm, dyn. #pairs = the number of spawn-target instances seen during simulation, avg. dist. = average spawn-target distance during simulation, #slc inst. = average number of (infix) slice instructions per spawn-target pair.

5.2. Methodology

We selected 4 benchmarks from Spec2000, and 1 from Spec95 that incur significant instruction cache misses on the baseline processor model. We profiled the branch frequencies and profile behavior by running the programs to completion. Our spawn-target generation algorithm selected between 28 and 1436 spawn-target pairs per benchmark as shown in Table 2, which also quantifies several characteristics of the slices executed for the VS configurations.

To evaluate the performance we collect data for 5 million instructions starting after warming up cache hierarchy while fast-forwarding past the first billion instructions (100 million for 145.fpppp) assuming no prescient instruction prefetch. In our initial investigations we examined data for up to 100 million instructions, but found very little variation in performance over this range.

We examine prescient instruction prefetch assuming a maximum postfix length of 300 instructions.

5.3. Results

Figure 7 illustrates the speedup of the various prescient instruction prefetch configurations against the baseline model running without hardware instruction stream prefetching enabled.

For reference purposes the first bar from the left in Figure 7 shows the speedup if all instruction accesses are assumed to hit in the first level cache and shows speedups ranging from 1.18 to 2.5 with a geometric average of 1.58. This indicates the performance of the benchmarks we study does indeed suffer significantly from instruction cache misses. The next bar represents the speedup of the hardware instruction stream prefetch mechanism and shows speedups ranging from 1.05 to 1.53 with geometric average speedup of 1.21. The next two bars measure performance for the idealized PLP model. In the first case the hardware stream prefetch mechanism is disabled and we see that average performance improvement of 1.27, which is better than that of the hardware mechanism alone. When we combine the two mechanisms we obtain an overall speedup of 1.38 on average.

The remaining bars show the speedup in the case that the minimum dynamic backward value and address slice are executed prior to the helper thread beginning to fetch instructions from the postfix region. Assuming slice operations require no latency to fetch and do not displace instructions from the main thread we obtain

speedups varying from 0.5% for 177.mesa up to 54% for 145.fpppp with an average speedup of 15% without the hardware instruction prefetch mechanism, and between 3% and 87% when combined with the hardware instruction prefetch mechanism. In the later case, the average additional benefit beyond that of the hardware stream prefetch mechanism is 5.7%.

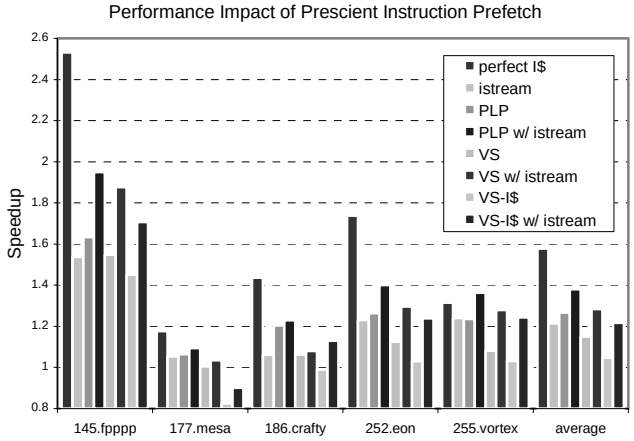

Figure 7: Performance Impact of Prescient Instruction Prefetch after selecting spawn pairs with the algorithm described in Section 4. From left to right the bars represent: instructions always hit in first level cache (perfect I$), hardware instruction prefetch mechanism (istream), perfect live-in prediction with (PLP) and without hardware prefetch (PLP w/ istream), and variable slices assuming: no hardware prefetch and single cycle access (VS), hardware prefetch and single cycle access (VS w/istream), no hardware prefetch an slice instructions contend for memory (VS-I$) and hardware prefetch with instructions contending for memory (VS-I$ w/ istream).

If slice operations contend for space in the memory hierarchy we still obtain an average speedup of 4.7% for the case without the hardware stream prefetch hardware, however in this case both 177.mesa and 186.crafty suffered a net decrease in performance. If we combine the stream prefetch hardware the overall speedup is still higher, but when compared against the hardware prefetch mechanism alone the benefit is on average quite negligible, and in some cases the mechanism reduces performance. The reason for the lower speedups when executing slice instructions is two-fold: On the one hand for benchmarks like 177.mesa the number of slice instructions required is on average half of the dynamic instruction stream up to the target which means that little if any instruction prefetching gets done. On the other hand these slices must be stored somewhere regardless of whether the work they do ends up being useful hence they cause a rise in capacity misses for the main thread. For instance for 177.mesa the number of capacity misses for the main thread doubles when modeling storage of slices in the standard memory hierarchy.

6. Conclusion

In this paper we propose *prescient instruction prefetch* a technique for speeding up single-threaded applications suffering from heavy instruction cache misses. We describe an analytical framework and propose, implement and evaluate an optimization algorithm for selecting prefetch helper thread spawn-target pairs. This study shows potential speedups of up to 63% assuming live-in generation is free. More realistic models highlight the need to include analysis to make tradeoffs between the spawn-target distance and the size of the required precomputation slice. In particular, the analysis methodology developed in this paper is complementary to and would benefit from integration with a compiler based slice analysis such as that studied by Liao et al. [11]. The quantitative evaluation based on an Itanium® implementation of the algorithm demonstrates its effectiveness and areas of further improvement through further algorithm tuning as well as microarchitectural optimization.

7. Acknowledgements

We would like to thank Antonio Gonzalez, Murali Annavaram, James Psota, Edward Grochowski, Shih-wei Liao, Perry Wang, and John Shen for their valuable comments on this work. We appreciate the helpful suggestions from the referees.

8. REFERENCES

[1] M. Annavaram, J. Patel, E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *ISCA 28*, July 2001.

[2] P. Chang, S. Mahlke, W. Hwu, Using Profile Information to Assist Classic Code Optimizations, Software Practice and Experience, vol. 21, no.12, pp. 1301-1321, 1991.

[3] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *ISCA 26*, May 1999.

[4] R. Chappell, F. Tseng, A. Yoaz, Y. Patt, Difficult-Path Branch Prediction Using Subordinate Microthreads, *In ISCA 29*, June 2002.

[5] G. Chrysos, J. Emer, Memory Dependence Prediction Using Store Sets. *ISCA 25*, 1998.

[6] J. Collins, D. Tullsen, H. Wang, J. Shen, Dynamic Speculative Precomputation. In *Micro 34*, December 2001.

[7] J. Collins, H. Wang, D. Tullsen, C, Hughes, Y. Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *ISCA 28*, July 2001.

[8] D. Hammerstrom and E. Davidson, Information content of CPU memory referencing behavior, *ISCA 4*, 1977.

[9] G. Hinton and J. Shen. Intel's multi-threading technology. *Microprocessor Forum*, October 2001.

[10] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, R. Zahir, Introducing the IA-64 Architecture. *IEEE Micro*, Sept-Oct 2000.

[11] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, J. Shen, Post-Pass Binary Adaptation for Software-Based Speculative Precomputation, In PLDI, June 2002.

[12] C. K. Luk, Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors, In *ISCA 28*, June 2001.

[13] Pedro Marcuello and Antonio Gonzalez, *Thread-Spawning Schemes for Speculative Multithreading, HPCA 8, Jan 2002.*

[14] E. Mehofer and B. Scholz, Probabilistic Data Flow Systems with Two-Edge Profiling, In *Dynamo*, 2000.

[15] A. Moshovos, D. Pnevmatikatos, A. Baniasadi. Slice procesors: an implementation of operation-based prediction. In *ICS*, June 2001.

[16] G. Reinman, T. Austin, B. Calder, A Scalable Front-End Architecture for Fast Instruction Delivery, In *ISCA 26*, May 1999.

[17] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *HPCA-7*, Jan., 2001.

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

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

[20] R.E. Tarjan, Fast algorithms for solving Path problems. Journal of the ACM, 3(28):591-642, July 1981.

[21] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA 22*, June 1995.

[22] C. Zilles and G. Sohi. Understanding the backward slices of performance degrading instructions. In *ISCA 27*, May 2000.

[23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA 28*, July 2001.