Optimization of Data Prefetch Helper Threads with Path-Expression Based Statistical Modeling

Tor M. Aamodt
Dept. of Electrical and Computer Engineering
University of British Columbia
2332 Main Mall, Vancouver,
British Columbia, V6T 1Z4, CANADA
aamodt@ece.ubc.ca

Paul Chow
Dept. of Electrical and Computer Engineering
University of Toronto
10 King's College Road,
Toronto, Ontario, M5S 3G4, CANADA
pc@eecg.utoronto.ca

ABSTRACT

This paper investigates helper threads that improve performance by prefetching data on behalf of an application's main thread. The focus is data prefetch helper threads that lack branch instructions and which generate prefetches for one dynamic instance of a delinquent load instruction per spawned helper thread. This form of helper thread, sometimes called a *simple p-thread*, has been studied previously by Roth et al. [29, 26] who proposed a framework for optimizing their impact. A key step in that framework is predicting the performance impact of a helper thread. In this paper we propose and evaluate a novel performance prediction technique that achieves comparable results yet requires less detailed information about dynamic program behavior. This technique extends a path expression based statistical modeling framework [2] by incorporating information about branch correlation (which we show is important) and by considering data flow information in a statistical manner. Significantly, the profile information we use is similar to that provided within current optimizing compilers. This paper also provides the first comprehensive assessment of the sources of modeling error relevant to predicting the performance impact of simple p-threads.

Categories and Subject Descriptors

C.0 [**General**]: Modeling of computer architecture; D.3.4 [**Programming Languages**]: Processors—*Optimization*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Measurement, Performance, Design, Theory

Keywords

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

1. INTRODUCTION

A significant performance bottleneck in current microprocessors is the latency of memory accesses that miss in the cache hierarchy. While hardware based prefetch mechanisms can reduce the frequency and latency of cache misses, existing hardware-only prefetch mechanisms are not effective for all memory access patterns. Recently, significant interest has centered on the use of so-called helper threads for prefetching data [12, 28, 23, 37, 11, 29, 20, 18, 17, 36, 26]. In this paper we study techniques for predicting the benefit of compiler generated helper threads [20, 18]. We quantify the sources of modeling error in prior schemes and we propose a technique to overcome one of the more significant sources of modeling errors.

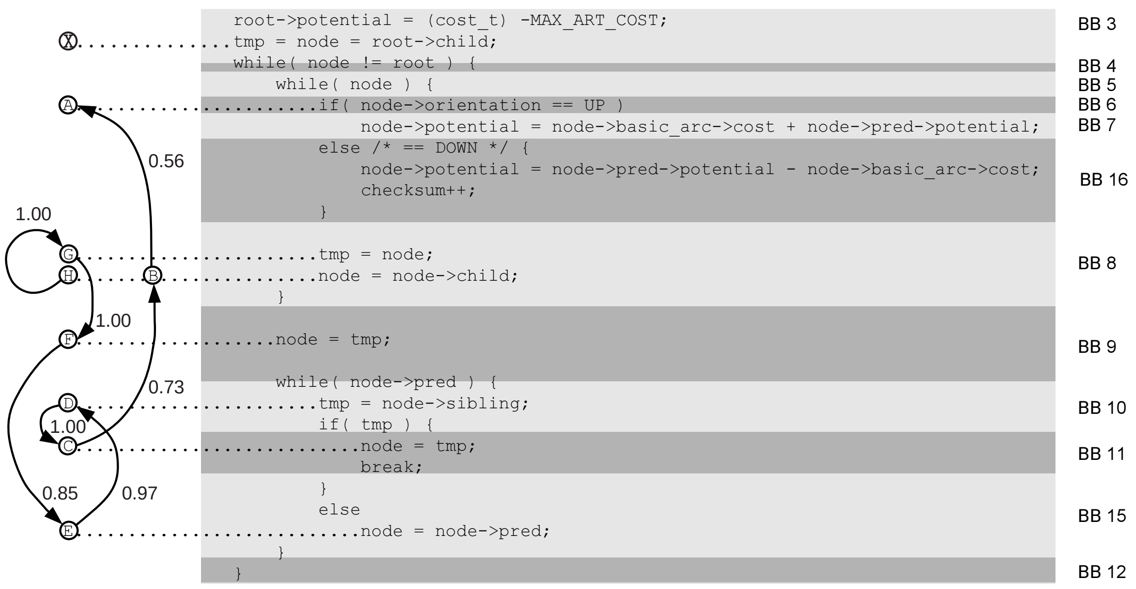
In contrast to earlier execution trace based helper thread performance prediction techniques our approach is compatible with traditional optimizing compiler implementation frameworks. In particular, we show how to achieve the benefits of the aggregate advantage framework [29, 26] within a modern optimizing compiler framework [1]. The key to the aggregate advantage framework is a structure called the *slice tree*. The slice tree is a directed graph with vertices representing instructions and edges linking instructions in a simple prefetch thread (p-thread). A natural way to construct the slice tree is by running the program and extracting the backward slice of each performance degrading event (i.e., L2 data cache misses). The main drawback of this approach is that it requires a detailed microarchitecture simulator. In this paper we describe how to build the slice tree using information that is more accessible from within an optimizing compiler framework.

The rest of this paper is organized as follows. In Section 2, Roth and Sohi's [29] optimization technique is summarized. Section 3 summarizes the findings of our detailed investigation into sources of modeling error in p-thread optimization. In Section 4, two extensions of our earlier path expression modeling technique [2] are proposed, which can be used instead of execution trace data. Section 5 presents our experimental evaluation of the proposed techniques. Section 6 summarizes related work, and Section 7 concludes.

2. OPTIMIZATION OF P-THREADS

In this section we briefly summarize the features of the aggregate advantage framework [29, 26] made accessible to modern optimizing compiler frameworks by the path expression based techniques proposed in this paper.

Figure 1 illustrates a slice for a delinquent load in the



```
root->potential = (cost_t) -MAX_ART_COST;
tmp = node = root->child;
while( node != root ) {
    while( node ) {
        if( node->orientation == UP )
            node->potential = node->basic_arc->cost + node->pred->potential;
        else /* == DOWN */ {
            node->potential = node->pred->potential - node->basic_arc->cost;
            checksum++;
        }

        tmp = node;
        node = node->child;
    }
    node = tmp;

    while( node->pred ) {
        tmp = node->sibling;
        if( tmp ) {
            node = tmp;
            break;
        }
        else
            node = node->pred;
    }
}
```

Figure 1: Example slice for a delinquent load in 181.mcf. Delinquent load is dereference `node->orientation` in basic block labeled BB6, on line marked A. Corresponding helper thread for this slice would be spawned when the main thread reaches H. Edges in graph (on left side of figure) are labeled with the relative frequency the dependency occurred given that the successor was executed (this is the *posteriori reaching definition probability* defined in Section 4.3). This occurs when control flow follows the *implied control flow* (defined in Section 4). See corresponding slice tree in Figure 2.

benchmark `mcf` from the SPEC 2000 integer benchmark suite [33]. The code is from the function `refresh_potential` and several lines directly contributing to the address computation are highlighted on the left (our p-threads include instructions from both the *value* and *address slice* [38]). The corresponding slice tree is shown in Figure 2. Each node in the tree represents an instruction in the program binary, and is labeled with a letter corresponding to the line from the source code. The values next to each circle in Figure 2 represent the aggregate advantage measured in millions of cycles. The best slice to use as a p-thread—determined by measuring the actual speedup obtained—is emphasized in bold in the slice tree starting from the node with a double edge labeled B (variously referred to as the *trigger instruction* [28, 11], *lead instruction* [23], or *spawn point* [9]). Note that in this case, the best p-thread to use is *not* the one with the highest aggregate advantage value [26]. The aggregate advantage for each node is computed using the following equations, which essentially transform average values measured during simulation into the estimated benefit the corresponding helper thread would provide [29]:

$$\begin{aligned}
\mathrm{ADV}_{agg}(p) &= \mathrm{LT}_{agg}(p) - \mathrm{OH}_{agg}(p) && (1)\\
\mathrm{LT}_{agg}(p) &= \mathrm{DC}_{pt-cm}(p) \cdot \mathrm{LT}(p) && (2)\\
\mathrm{LT}(p) &= min(\mathrm{SCDH}_{mt}(p) - \mathrm{SCDH}_{pt}(p), \mathrm{L}_{cm}) && (3)
\end{aligned}$$

In the above equations $\mathrm{ADV}_{agg}(p)$ is the aggregate advantage of the helper thread p, $\mathrm{LT}_{agg}(p)$ is the aggregate latency tolerance of the helper thread, and $\mathrm{OH}_{agg}(p)$ is the aggregate overhead. The *aggregate latency tolerance* is the number of cycles of stalled execution that are removed by the helper thread. The *aggregate overhead* is the number of additional cycles incurred to support p-thread execution assuming (hypothetically) the data prefetched by the helper thread were "ignored" by the main thread so that the main thread had to access these values from wherever they were in the memory hierarchy before the helper thread attempted to prefetch them. As latency tolerance is only provided by those helper thread instances that prefetch data not already in the cache that the main thread subsequently uses, the aggregate latency tolerance can be expressed as a product of $\mathrm{DC}_{pt-cm}(p)$, the dynamic count of helper threads that pre-execute *actual misses*, and $\mathrm{LT}(p)$, the average per-helper-thread-instance latency tolerance of helper thread p. In turn, the latter can be expressed as the minimum of the average difference between the schedule constrained data flow heights of the execution sequence of the main thread and the helper thread ($\mathrm{SCDH}_{mt}(p)$ and $\mathrm{SCDH}_{pt}(p)$ respectively) and the latency of a cache miss (L_{cm}). For the above example we measured the following values for node B (measured over a segment of 100M simulated instructions):

$$\begin{aligned}
\mathrm{DC}_{pt-cm}(p) &= 480753\\
\mathrm{SCDH}_{mt}(p) &= 670\\
\mathrm{SCDH}_{pt}(p) &= 111\\
\mathrm{L}_{cm}(p) &= 300
\end{aligned}$$

Substituting these values into Equations 1 to 3 yields an aggregate (latency) advantage of:

$$\begin{aligned}
\mathrm{ADV}_{agg}(p) &= 480753 * min(670 - 111, 300)\\
&= 144.2 \times 10^6 \ \ \text{[cycles]}
\end{aligned}$$

However, the actual execution time saved when this helper thread was used was 96.3M cycles (which translates into a speedup of 12.6%). As observed by Petric and Roth [26],

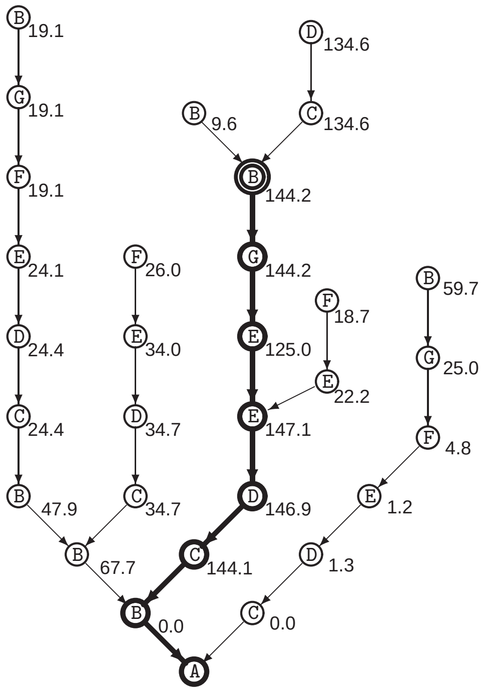


Figure 2: Slice tree for delinquent load in Figure 1. Values beside nodes in slice tree are aggregate advantage in millions of cycles. The best helper thread, as determined by the actual speedup obtained, begins at the node B with the double edge and follows the highlighted path to the delinquent load instruction at the root node labeled A.

part of this inaccuracy on the side of optimism results from ignoring whether the prefetches reduce the latency of misses on the critical path. In this paper we explore other important sources of prediction inaccuracy. Note that the largest aggregate advantage is for E—implying this is the best helper thread, but the p-thread starting at this node has an actual speedup of only 1.4%. Thus, accurate modeling of the impact of helper threads is important to making good optimization choices.

3. SOURCES OF MODELING ERROR

Before describing our techniques for predicting the benefit of helper threads, we summarize the results of our investigation into the sources of modeling error for earlier frameworks (detailed quantitative data are presented later). In this paper we focus on the fundamental question of how to predict the benefit of potential helper threads given a specific set of profile information assuming the same inputs will be seen again. Prior work has established that the benefit of p-threads selected using the aggregate advantage framework with one set of program inputs is often robust across program inputs [29]. In practice, if this were not the case, one could combine the results of profile runs using different input sets [8]. Alternatively, one might adaptively tune or even select among different helper threads based upon program phase—for example, prior work has explored selectively throttling helper threads on a real (not simulated) hardware system using performance counters [17]. We believe that since (as we show) there are significant sources of modeling error when using so-called "perfect information" (profile information for the same input data used to evaluate performance), a important challenge is accurately interpreting whatever profile information is available. In the following section we provide a framework for making more accurate modeling available within the context of an optimizing compiler and extend it to address one of the key sources of modeling error.

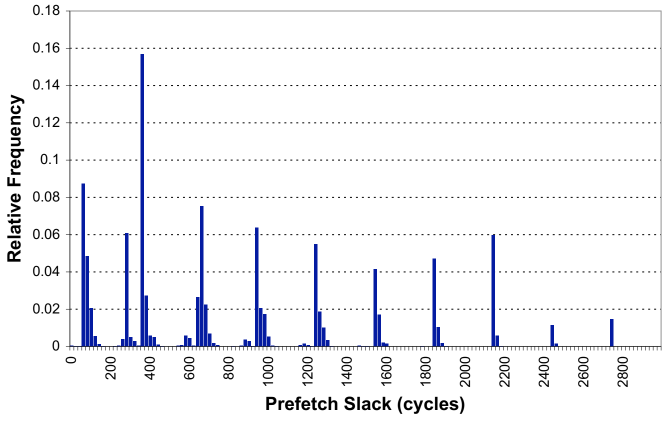


Figure 3: P-thread prefetch slack distribution for 175.vpr (idealized execution model: 4B cache lines, perfect branch prediction)

Figure 3 shows one significant source of modeling error—the distribution in the latency tolerance of p-threads. Note that the average value of this distribution is the quantity $\mathrm{SCDH}_{mt}(p) - \mathrm{SCDH}_{pt}(p)$ from Equation 3. The distribution in Figure 3 is multimodal and indicates a significant fraction (18.3%) of the prefetches have slack that happens to be less than the latency of memory (300 cycles in our simulations). The isolated and well defined peaks in this distribution may indicate that the main thread iterates around a loop a variable number of times before reaching the target load. Their are two problems resulting from approximating this latency tolerance distribution with an average value. One, which affects 175.vpr, is that using the average value will significantly over-estimate the benefit of this helper thread. On the other hand, for the Olden benchmark treeadd [7] we discovered that while the latency tolerance distribution had an average near zero, several individual helper thread instances were actually able to offer significant prefetch benefit in the context of the multithreaded microarchitectures we simulated. In this case very beneficial helper threads were entirely overlooked.

The results of our detailed investigation into all sources of modeling discrepancies concluded that the main sources of error are:

Prefetch Slack Distribution: Due to the nonlinearity in the transfer function between latency tolerance and execution time savings, measuring only average quantities can lead to significant inaccuracies.

Prefetch Slack Sensitivity: Out-of-order superscalar processors hide the latency of L1 data cache misses off the critical path [32]. Thus each cycle earlier a target delinquent load is prefetched may *not* save a cycle of execution time [26]. We found the impact is equivalent to a constant multiplicative factor applied to the minimum of the prefetch slack and the memory latency.

Branch Outcome Correlation: The path expression framework described in [2] assumes that branch outcomes are uncorrelated. This approximation may lead the modeling framework to underestimate the frequency of execution paths from the spawn point to the target delinquent load causing the benefit of the associated helper threads to be underestimated.

Spatial Locality / Loss of Temporal Locality: Typical memory systems use caches that store several words of memory in a single block (or cache line). The slice of a delinquent load may contain other loads. During application profiling these other loads may appear to have a high hit rate, but when these loads are executed in a helper thread they may trigger cache misses and consequently the helper thread would have less latency tolerance than predicted.

Resource Constraints: Estimating the number of helper threads running concurrently was found to be important.

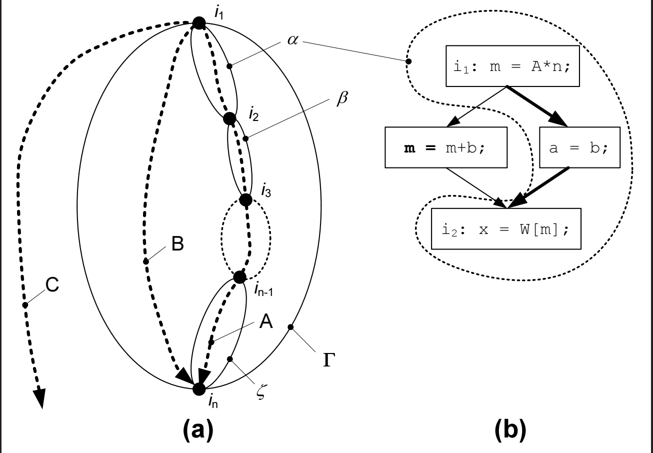
Equivalent Linked Data Traversals: In at least one case (181.mcf) it was observed that significant prefetch benefit was obtained even for those dynamic p-thread instances where the main thread did not follow the control flow assumed when creating the helper thread. The reason for this was found to be that the application would begin by traversing a different path through the linked data structure associated with the target load of the helper thread, it would later return back to the same node visited when the helper thread was spawned, and then followed the control flow assumed by the helper thread. Again significant optimization potential can be overlooked in this case.

Correlation of Cache Misses to Control Flow Paths: The assumption (used later in Equation 6) that the probability of a cache miss is independent of the control flow leading up to the load instruction does not always hold in practice. For example, the probability of a cache miss occurring when the main thread follows the implied control flow (defined in Section 4) of the p-thread starting at the node B with the double edge in Figure 2 is 0.766 versus an average L2 cache miss rate 0.432 for the target load. Similar observations have been made by others [24, 21].

In the following section we introduce two models that provide the optimization benefits of the aggregate advantage framework within the context of an optimizing compiler. We then revisit the sources of inaccurate prediction in Section 5.

4. PATH EXPRESSION MODELING

The values used in the aggregate advantage optimization framework are statistical quantities. In this section we describe an algorithm for computing these quantities from data that is easy to obtain within the framework of present day feedback directed optimizing compiler frameworks. The algorithm we describe uses path expressions [34, 35, 2]. A path expression is a regular expression summarizing all paths between two vertices in a directed graph. From a delinquent load we incrementally build up a p-thread as follows: An instruction is included in the p-thread if it is a data flow predecessor of an instruction already in the p-thread and it is the "most likely" instruction to generate a live-in value to the existing p-thread (a variable is a "live-in" to a p-thread if it is used in that p-thread before being defined). We quantify "most likely" using a novel form of *data flow frequency analysis* [27].



Consider whether the control flow the main thread follows after spawning a p-thread "matches" the control flow assumed when constructing the slice defining the p-thread. The set of paths the main thread can follow from spawn to target for which this is true is the *implied control flow*.

More formally, label the instructions in the slice from i_1 to i_n in order from the spawn to the target instruction. Then, the implied control flow of a p-thread is the subset of paths *in the main thread* from i_1 to i_n that pass through instructions i_k (where $1 < k < n$) in order, such that along a given path no variable which is live between any two consecutive instructions i_k and i_{k+1} *in the p-thread* is redefined by an instruction between i_k and i_{k+1} *in the main thread*. We say a path in the main thread between two consecutive slice instructions is *admissible* if it avoids definitions of variables live at that point in the p-thread. For example, in part (b) in the figure above, the definition of `m` on the left hand side of the branch kills the definition of `m` at i_1 before it reaches i_2 along the execution path that takes the left hand side of the branch, but the path along the right hand side is admissible.

In part (a) Γ represents the set of *all* paths from i_1 to i_n. The implied control flow from i_1 to i_n is the subset of Γ composed of the concatenation of the path expression α summarizing admissible control flow between instruction i_1 and i_2 with the path expression β summarizing admissible control flow between instruction i_2 and i_3, etc... up to the path expression ζ summarizing admissible control flow between instruction i_{n-1} and i_n. After spawning, the main thread either follows the implied control flow of the p-thread (e.g., 'A'), a path that likely results in a different load address (e.g., 'B') or a path that avoids the target load altogether (e.g., 'C').

Figure 4: Implied Control Flow

4.1 Expected Benefit

To make optimization choices, we compute the *expected benefit* of using a helper thread. Given a sequence of instructions $i_1, i_2, ...i_n$ making up a p-thread p, we compute the expected benefit of the p-thread as follows:

$$\begin{aligned}\mathrm{E}[\mathrm{benefit}(p)] \quad &= \quad \mathrm{f}_{sp} \cdot \mathrm{P}[\mathrm{miss}] \cdot \Big(\prod_{k=1}^{n-1} RDP(k|p)\Big) \cdot \\ &\qquad min\big((\mathrm{E}[\mathrm{lat}_{mt}] - \mathrm{E}[\mathrm{lat}_{ht}]), \mathrm{lat}_{mem}\big) \quad (4)\end{aligned}$$

The quantity $RDP(k|p)$ is the *reaching definition probability* from instruction i_k to i_{k+1} for the helper thread p. This is the probability that program execution goes from instruction i_k to instruction i_{k+1} without defining any variables that are "live" [25] between instruction i_k and i_{k+1} in the helper thread.

Building upon an earlier path expression based approach to statistical control flow modeling [2], we compute $RDP(k|p)$ by first finding the path expression from i_k to i_{k+1}. The path expression is transformed into a probability by recursively replacing the concatenation and union operators from the regular expression with the product and sum of the path probabilities, respectively. The closure operator for a path with probability p is replaced with the quantity $1/(1-p)$. The edges of the control flow graph are replaced by their respective transition probabilities as determined by profiling the branch outcomes of the application (e.g., "taken" or "not taken" for conditional branches). To obtain the desired reaching definition probability we carry out the above recursive evaluation after first assigning zero probabilities to edges leaving basic blocks that contain instructions which re-define (i.e., "kill" [25]) live variables within the p-thread (as well as to the edges leaving i_{k+1} as required by the procedure for computing statistical quantities outlined in [2]). The set of paths that remains forms what we call the *implied control flow* of the helper thread between instructions i_k and i_{k+1}.

Thus, the implied control flow of a simple p-thread is the set of paths the application main thread may follow from the spawn point to the target delinquent load, which satisfy the following conditions: The execution path encounters each instruction in the program slice used to define the p-thread in the same order as the p-thread, and no variable live between two instructions in the p-thread is redefined by any other instruction encountered between the corresponding two slice instructions in the main thread. Note that if the main thread follows the implied control flow of the p-thread after spawning that p-thread, it is guaranteed to compute the same address for the delinquent load as the helper thread (following the implied control flow is a sufficient condition, but not always a necessary condition for computing an address to *accurately* prefetch the data referenced by a future load instruction). See Figure 4.

The quantity $\mathrm{E}[\mathrm{lat}_{mt}]$ in Equation 4 is the expected latency of the main thread from the spawn to the delinquent load and is evaluated using the path expression method for computing expected path length [2], modified to weight basic blocks by the average number of cycles they take to execute and modifying edge probabilities as done when computing the reaching definition probability. As we are only interested in the latency of the main thread when it follows implied control flow of the helper thread, the quantity $\mathrm{E}[\mathrm{lat}_{mt}]$ can be decomposed as follows:

$$\mathrm{E}[\mathrm{lat}_{mt}] = \sum_{k=1}^{n-1} latency(k|p) \tag{5}$$

where the quantity $latency(k|p)$ is the expected number of cycles for execution of the main thread to go from instruction i_k to i_{k+1} along the implied control flow of p-thread p. One way to obtain the information required to compute this quantity in practice may be to profile average *cycles per instruction* at the individual instruction or basic block level using hardware performance monitors using a utility such as Intel's VTune performance analyzer [16].

The quantity $\mathrm{E}[\mathrm{lat}_{ht}]$ in Equation 4 is the number of cycles it takes the helper thread to execute after spawning. To compute this quantity the execution latencies of each instruction in the helper thread are simply added. This ignores the potential for ILP in the helper thread, but requires less analysis. The assumption of little ILP for the helper threads turns out to be a valid one for the simple p-threads we examined in this study as the dependency graphs are often a single chain of dependent instructions.

The quantity f_{sp} in Equation 4 is the frequency the spawn instruction is executed (determined via basic block profiling), the quantity $\mathrm{P}[\mathrm{miss}]$ is the relative frequency that the delinquent load misses in the cache, and finally, the quantity lat_{mem} is the latency of main memory.

4.2 Connection to Aggregate Advantage

The connection between Equation 4 for the expected benefit of spawning a p-thread and Equation 1 for computing the aggregate advantage of a p-thread is as follows: $\mathrm{DC}_{pt-cm}(p)$ is the number of instances of the slice p that lead to an occurrence of the delinquent load that suffers a cache miss. This is approximately equal to the total number of slices leading to the delinquent load multiplied by the cache miss rate of the delinquent load:

$$\mathrm{DC}_{pt-cm}(p) \approx \mathrm{DC}_{pt} \cdot \mathrm{P}[\mathrm{miss}] \tag{6}$$

The approximation in Equation 6 is due to the assumption that the probability of encountering a data cache miss is independent of the control flow path leading up to the target delinquent load instruction, and that, for a particular instance of the target delinquent load to have its data prefetched, the main thread must follow the implied control flow of the p-thread spawned to prefetch the data.

The quantity DC_{pt} on the right hand side in Equation 6 is equal to the frequency the spawn instruction is executed by the main thread (f_{sp}), multiplied by the relative frequency that execution subsequently follows the implied control flow of the p-thread. The relative frequency that execution follows the implied path is the ratio of the frequency that the main thread execution follows the implied path ($\mathrm{f}_{implied\ cf}$) over the frequency with which the spawn point is reached (f_{sp}). The ratio of $\mathrm{f}_{implied\ cf}$ over f_{sp} can be estimated using the path expression framework so that DC_{pt} can be expressed as:

$$\begin{aligned} \mathrm{DC}_{pt} &= \mathrm{f}_{sp} \cdot \frac{\mathrm{f}_{implied\ cf}}{\mathrm{f}_{sp}} \\ &= \mathrm{f}_{sp} \cdot \Big(\prod_{k=1}^{n-1} RDP(k|p)\Big) \end{aligned}$$

To estimate the quantity LT_{agg} we use the value $\mathrm{E}[\mathrm{lat}_{mt}] - \mathrm{E}[\mathrm{lat}_{pt}]$ described above.

4.3 Predecessor Selection and Pruning

The form of static program slicing required by our algorithm takes as input an instruction (perhaps with additional context information) and returns the set of dataflow predecessors (for simple p-threads we do not include control dependencies in the slice). For simplicity, rather than building the entire slice tree, let us examine how to generate a single p-thread equivalent to one path in the slice tree. For concreteness we assume a greedy algorithm is used to select the path.

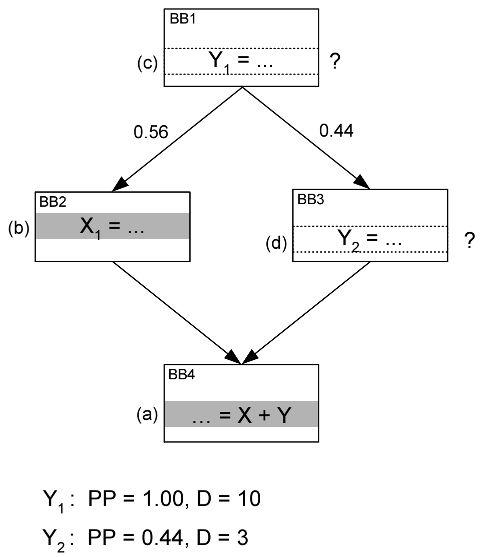


Figure 5: Example illustrating need for predecessor pruning during p-thread slicing.

There are two challenges to using static program slicing to build a p-thread in this manner. The first is ensuring the p-thread being created is actually in the slice tree, and the second is selecting the best instruction to add to the p-thread. Without further analysis it is not clear which instructions from the set of dataflow predecessors found via static slicing can be added to the existing p-thread and result in a valid p-thread. For example, in Figure 5, two instructions define register Y, however since X_1 has already been selected to be in the p-thread, adding Y_2 would lead to an incorrect p-thread. The underlying reason is that Y_2 does not lie on a path leading to X_1 without intersecting another instruction already selected to be in the p-thread (in this case instruction "a" in BB4).

To ensure the p-thread being created is a correct p-thread (which implies it has a node in the slice tree), we filter the set of dataflow predecessors before a new instruction is added to the p-thread from the slice. We do this by pruning away predecessors inconsistent with the implied control flow of the current p-thread.

From among the remaining data flow predecessors we use the following greedy heuristic to select the (estimated) best new instruction to include in the slice: For each of the data flow predecessors, we compute the *posteriori reaching definition probability* (PRDP) excluding any execution paths that include other data flow predecessors currently under consideration. The PRDP of a dataflow predecessor to a particular instruction is the reaching probability in the reversed graph excluding edges leaving basic blocks that contain an instruction that kills the definition generated by the predecessor. For each live-in variable to the p-thread, we select the predecessor with the largest PRDP.

4.4 Example

To illustrate the algorithm let us consider Figure 1 again. To aid understanding, Figure 6 illustrates the control flow graph for the function `refresh_potential()` with the implied control flow of the desired helper thread traced out with bold arrows. The implied control flow of the desired p-thread starts in basic block 8, follows the backward branch to basic block 6, goes along either path back to basic block 8, then falls out of the loop to basic block 9, continues to basic block 10, then to basic block 15, and back to block 10, then goes around the outer loop via blocks 11, 12, 13 and 5 to finally reach the delinquent load at 6.

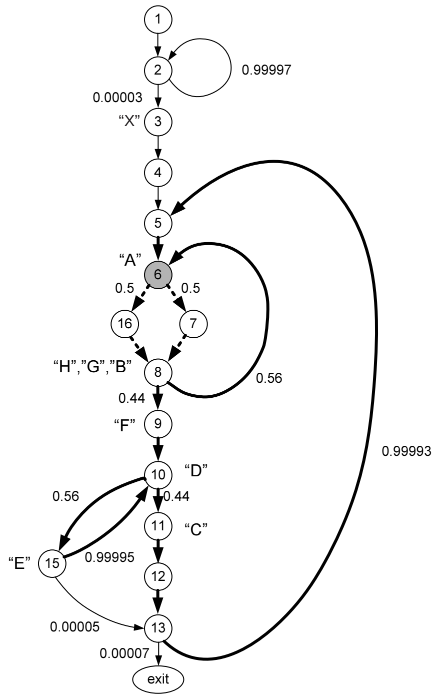


Figure 6: Control flow graph for code in Figure 1. Circles represent basic blocks. Numbers inside circles are the basic block number from the right hand side of Figure 1. Edges are labeled with the branch transition probability.

Our algorithm starts by initializing the p-thread to include only the delinquent load marked A (in Figure 1 and Figure 6). Static slicing indicates that instructions B, C, E, F and X are dataflow predecessors of A. The posteriori reaching definition probabilities for these instructions are then evaluated resulting in values of 0.560, 0.440, 0.000, 0.000, and 0.000 respectively. These values are computed by evaluating the posteriori probability [2] excluding paths through instructions that define "node" in Figure 1). Since the value for B is largest, we include B in the p-thread.

To determine when to stop adding instructions to the p-thread, we examine the expected benefit of the p-thread as each additional instruction is added. Table 1 shows the value of the reaching definition probability (RDP) for the implied control flow of the entire p-thread as nodes are added (column a), the spawn frequency (column b), the cache miss frequency (column c), the expected slack (column d), and the expected benefit (column e). Also included are values from the aggregate advantage calculation, namely DC_{pt-cm} (column f), the difference $\Delta\mathrm{SCDH} = \mathrm{SCDH}_{mt-pt}$ in the schedule constrained data flow heights of the main thread and the helper thread (column g), and the aggregate advantage computed using the aggregate advantage technique (column h). Finally, the cycles actually saved if the p-thread is used

Sp. Inst.	(a) $\prod$ RDP	(b) f_{sp}	(c) P[miss]	(d) E[slack]	(e) E[benefit] (cycles)	(f) DC_{pt-cm}	(g) ΔSCDH	(h) ADV_{agg} (cycles)	(i) Actual (cycles)
B	0.56	2.55M	0.43	0	0	842682	0	0	0.5M
C	0.56	1.12M	0.43	254	68M	567304	254	144.1M	5.4M
D	0.313	2.55M	0.43	260	**159M**	567304	260	146.9M	17.6M
E	0.138	1.43M	0.43	260	89M	565767	260	**147.1M**	12.0M
F	0.077	1.12M	0.43	260	39M	480753	260	125.0M	10.4M
G	0.077	2.55M	0.43	363	45M	480753	363	144.2M	87.5M
H	0.077	2.55M	0.43	559	25M	480753	559	144.2M	**96.3M**

Table 1: Comparison of static slicing with statistical control flow analysis using our path expression optimization metric in column (e) versus Roth and Sohi's aggregate advantage metric in column (h) and actual speedup in column (i).

are in column (i). To isolate it as a source of modeling error, the estimate prefetch slack in column (d) is simply measured using SCDH_{mt-pt} (rather than Equation 5).

We see that the path expression model predicts the peak benefit to occur for the p-thread starting at node D, whereas the peak aggregate advantage value is for node E. Furthermore, the best speedup is actually obtained for the p-thread starting at node H. Examining columns (a) through (h) it can be seen that the discrepancy in predictions between the path expression technique and the aggregate advantage technique is largely due to the value RDP computed using the path expressions. For example, for the p-thread starting at node H, the low value of the predicted benefit is most heavily influenced by the low value of RDP. It turns out that the value of RDP significantly underestimates the actual value. For example, the probability of following the implied control flow from H to A is actually roughly 0.5 rather than 0.077.

The main contributor to the suboptimal optimization decision made using the expected benefit computed using our path expression technique is the computation of RDP (employing Equation 5 increases error further [1]). In the following section we show how to dramatically improve the accuracy of the RDP prediction. This improvement to the path expression based approach enables it to select helper threads similar to those that would be selected using simulator based execution tracing and the aggregate advantage framework but using information more easily collected during application profiling. We discuss the reason that aggregate advantage does not select the helper thread corresponding to H in Section 5.3.

4.5 Branch Correlation

In this section we show that by computing control flow statistics over a second-order control flow graph, the accuracy of our predictions can be improved significantly. We construct a higher-order Markov chain model of control flow as follows: Each state consists of the combination of a basic block and the recent control flow history leading up to that basic block. The "first-order" Markov chain from the last section thus consists of states encoding no branch outcome history leading up to the current basic block, while a second order Markov chain consists of states which represent both the last basic block visited and the current basic block. Even higher-order models could be constructed by including additional branch history in each state. We call the graph representing the nodes and state transitions of a second-order Markov chain a *second-order control flow graph*.

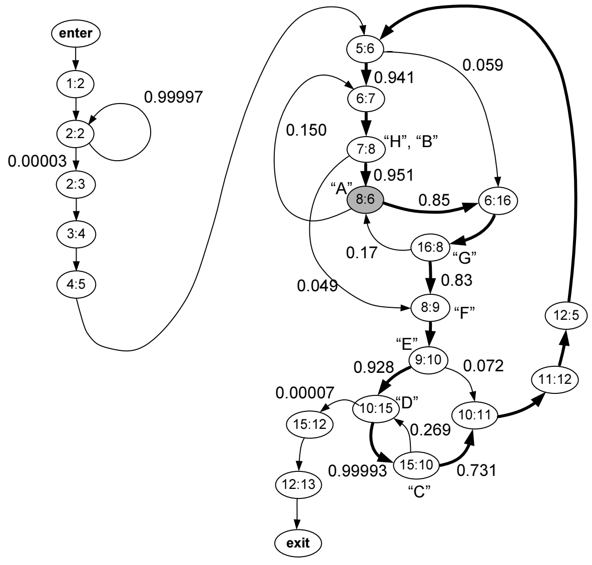


Figure 7: Second-order control flow graph for code in Figure 1.

Figure 7 illustrates the second-order control flow graph for `refresh_potential()`. Each node in this graph represents a branch outcome in the function `refresh_potential()` and is labeled using the convention n:m, where n and m are the basic block numbers used in Figures 1 and 6. Here n represents the previous basic block the program executed, while m represents the basic block where the program is currently executing. The bold lines indicate the implied control flow of the p-thread we wish to select. Note that, even though the original source code's control flow is reducible, this graph is non-reducible [25]. This does not impact our approach since Tarjan provides an algorithm [35] that can be used to extract path expressions even for non-reducible graphs.

Table 2 provides data similar to that in Table 1 but for the enhanced version of our path expression optimization algorithm. Examining Table 2 in comparison to Table 1 we observe that while (a) the same slice was chosen by both of our schemes, and (b) neither our schemes nor aggregate picked the best p-thread, that (c) the expected benefit computed using the second-order control flow graph is now *significantly* closer to the (less accessible from a compiler writer's perspective) aggregate advantage value for longer slices. For example, node H is predicted to have a benefit of 141M cycles using the second-order control-flow graph, versus 25M for the first-order control flow graph (in Table 1) and 144.2M for the aggregate advantage. We argue such improvements in prediction accuracy are necessary to obtain better optimization results.

5. EXPERIMENTAL EVALUATION

In this section we evaluate our path expression data prefetch helper thread optimization algorithm. After describing our experimental methodology, we show that the proposed optimization algorithm compares favorably to an enhanced version of the aggregate advantage methodology at predicting the speedup obtained when employing a given p-thread to prefetch for a given (static) problem load instruction. Then we delve deeper into the sources of modeling error that exist in both approaches and measure sensitivities to various aspects of the modeling framework and the simulated microarchitecture.

5.1 Methodology

To evaluate the various approaches to simple p-thread optimization we selected five benchmarks from the SPEC 2000 integer benchmark suite [33] and two benchmarks from the Olden benchmark suite [7] that exhibited large performance degradation due to data cache misses. Simulation data presented in this section was collected using the benchmark reference inputs over a 100 million instruction segment after fast-forwarding to a representative location of program execution found using the SimPoint toolkit [31]. Data cache, traditional edge profiling, and second-order control flow profile information was obtained by running the same input and program segment as used to evaluate performance (as noted in Section 3 prior research has shown that p-threads selected using aggregate advantage are relatively insensitive to changes in program inputs [29], and furthermore in this work we focus on how best to interpret profile information assuming it accurately represents likely program behavior). We extended version 3.0d of the SimpleScalar microprocessor simulator [6] to model an SMT processor with the microarchitecture parameters given in Table 3.

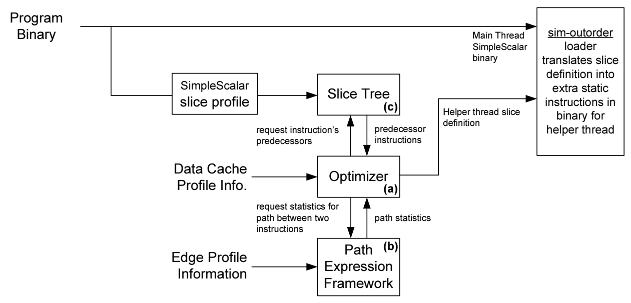

Program Binary
Main Thread SimpleScalar binary
sim-outorder loader translates slice definition into extra static instructions in binary for helper thread
SimpleScalar slice profile
Slice Tree (c)
Helper thread slice definition
request instruction's predecessors
predecessor instructions
Data Cache Profile Info.
Optimizer (a)
request statistics for path between two instructions
path statistics
Edge Profile Information
Path Expression Framework (b)

Figure 8: Helper thread generation infrastructure

Two automated software frameworks for augmenting applications with helper threads were implemented and extensively evaluated. Both leverage the path expression based techniques described in Section 4. In this paper we focus on the infrastructure illustrated in Figure 8. Due to practical considerations this infrastructure employs information from the slice tree data structure rather than performing true static program slicing. We have experimented extensively [1] with developing compiler based infrastructure using Grammatech's CodeSurfer [14] program slicer and the SUIF compiler intermediate representation [15]. A practical issue we encountered with the latter approach was correlating information between these two frameworks' different intermediate representations. The main source of inaccuracy in the infrastructure we employ in this paper is the use of perfect memory disambiguation—the dynamic data addresses accessed by load and store instructions are used during the creation of the slice tree, which means the slice tree contains more accurate dependence information than is available using static slicing. Note that our goal is to accurately predict the benefit of a given slice if used as a p-thread, which is orthogonal to the effect of pointer analysis on determining which slices to analyze. Prior work [20] has established that compilers can statically construct p-threads capable of achieving a speedup. As in Section 4.4 we use the main thread and expected helper thread latency information captured during slice tree generation (using the microprocessor simulator) rather than Equation 5.

5.2 Target Load Selection

With a framework that produces accurate predictions of performance improvement for helper threads it is possible to make good tradeoffs in the selection of which loads can most benefit from helper thread based prefetching. Since the aim of this work is to develop such a framework and in the process investigate the sources of inaccuracy in its predictions we fix the selection of target loads. Table 4 lists up to the top five load instructions for each benchmark ranked in order of their estimated impact on performance. The potential speedup varies from 7% for vortex to 148% for treeadd indicating that these applications may benefit significantly from the use of data prefetch helper threads.

#	Bench.	Target Load	Frequency	Speedup	L1 miss rate	L2 miss rate*	Sensitivity
1	art	0x404c80	1067500	1.396	50%	50%	1.01
2	art	0x404c98	1067500	1.357	100%	100%	0.25
3	art	0x4045e8	567943	1.030	98%	100%	0.07
4	art	0x404358	284532	1.002	62%	98%	0.02
5	art	0x404c88	1067500	1.000	100%	100%	0.00
6	bh	0x403000	322628	2.161	100%	88%	1.01
7	bzip2	0x40bad0	410549	1.285	67%	65%	0.85
8	bzip2	0x40aca0	174995	1.064	50%	45%	1.07
9	bzip2	0x40abc0	174995	1.060	48%	43%	1.07
10	bzip2	0x40ab50	174995	1.056	47%	42%	1.07
11	bzip2	0x40ac30	174995	1.056	46%	42%	1.07
12	mcf	0x4009f8	2546268	1.294	57%	75%	0.66
13	mcf	0x400a18	1269615	1.000	100%	96%	0.00
14	mcf	0x400a50	1276653	1.000	99%	99%	0.00
15	treeadd	0x400508	1048575	2.483	50%	50%	1.10
16	vortex	0x4880e8	28965	1.072	121%	82%	1.00
17	vortex	0x4673d8	114655	1.061	23%	97%	0.94
18	vortex	0x466668	263188	1.025	10%	45%	0.82
19	vortex	0x45dc00	15027	1.012	63%	51%	1.01
20	vortex	0x461958	424163	1.011	6%	33%	0.55
21	vpr	0x41e5e8	394677	1.092	21%	81%	1.04
22	vpr	0x41dc38	152291	1.054	65%	77%	0.59
23	vpr	0x41e6b0	384331	1.035	63%	78%	0.18
24	vpr	0x41f400	377369	1.028	27%	72%	0.34
25	vpr	0x41f570	1905810	1.020	14%	71%	0.08

Table 4: Target Load Selection ("Sensitivity" is defined in Section 5.3.1)

5.3 Framework Accuracy

To assess the accuracy of the data prefetch helper thread optimization framework, helper threads were generated for the selected target loads using the infrastructure illustrated

Sp. Inst.	(a) $\prod$ RDP	(b) f_{sp}	(c) P[miss]	(d) E[slack]	(e) E[benefit] (cycles)	(f) DC_{pt-cm}	(g) ΔSCDH	(h) ADV_{agg} (cycles)	(i) Actual (cycles)
B	0.951	2.55M	0.43	0	0	842682	0	0	0.5M
C	0.897	1.12M	0.43	254	72.5M	567304	254	144.1M	5.4M
D	0.897	2.55M	0.43	260	**169M**	567304	260	146.9M	17.6M
E	0.897	1.43M	0.43	260	94.3M	565767	260	**147.1M**	12.0M
F	0.833	1.12M	0.43	260	68.9M	480753	260	125.0M	10.4M
F	0.691	2.55M	0.43	363	150M	480753	363	144.2M	87.5M
H	0.651	2.55M	0.43	559	141M	480753	559	144.2M	**96.3M**

Table 2: Comparison of static slicing with second-order statistical control flow analysis using path expressions versus aggregate advantage.

Table 3: Processor Configurations

Threading	SMT with 4 thread contexts	Func. Units	4 iALU, 2 LD/ST, 4 fpALU 1 iMUL/DIV, 1 fpMUL/DIV
Superscalar Pipeline	4-way issue/decode/commit 256 entry RUU, 64 entry LSQ	Reg. Files (per thread)	32 GPR, 32 FPR
Fetch	8 inst./thread (max 2 threads/cyc) priority main thread	Caches	L1I 32KB, 2-way, 32B blks, 1-cyc L1D 16KB, 4-way, 32B blks, 2-cyc L2 1 MB, 4-way, 64B blk, 14-cyc
Branch Prediction	max 2 taken branch/cyc/thread hybrid predictor with (a) 2048-entry bimodal (b) 256-entry global predictor (c) 1024-entry selector 64-entry return address stack	Memory	300-cycle

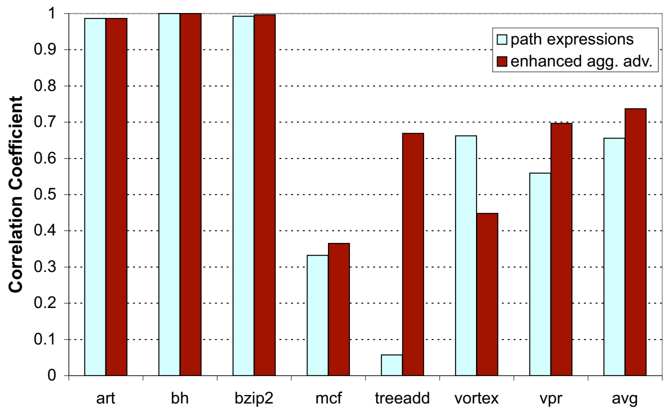


Figure 9: Comparing accuracy of path expression and aggregate advantage frameworks

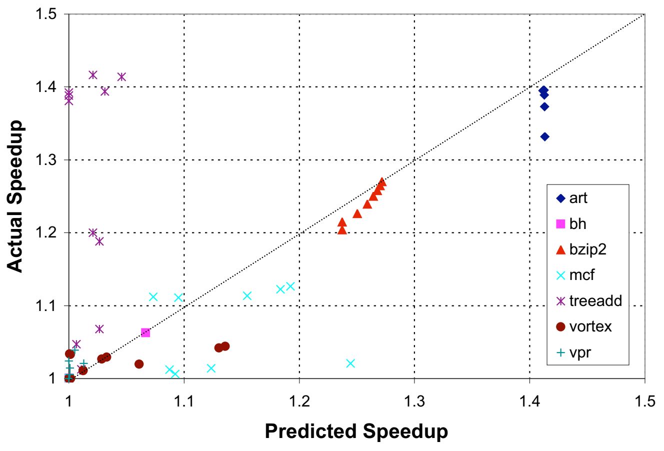


Figure 10: Actual versus predicted speedup for path expression based modeling.

in Figure 8. The actual benefit of the selected helper threads was measured using detailed performance simulation and compared with the speedup predicted by our path expression framework (including branch correlation information). To quantify the prediction accuracy of our framework, we compute the correlation coefficient of predicted speedups and actual speedups. Figure 9 shows that on average the compiler friendly path expression based technique has correlation coefficient of 0.65, which is fairly close to the value 0.74 obtained using an enhanced version of the execution trace based aggregate advantage. The enhanced aggregate advantage technique we employ in this paper measures the slack of each instance of a slice and compares it against the nonlinearity of the prefetch slack sensitivity function (see Figure 12 described in Section 5.3.1) rather than first averaging the values. In contrast the path expression based technique uses the average and standard deviation of the estimated helper thread latency measured during slice tree extraction and applies a Gaussian approximation to obtain the resulting predicted speedup. To better understand why the correlation coefficient is less than one, Figure 10 shows the actual and predicted speedup values for our path expression based approach. The amount of correlation varies significantly by benchmark. While the correlation for bzip2 is very good, for treeadd the actual speedup was found to be *greater* than predicted.

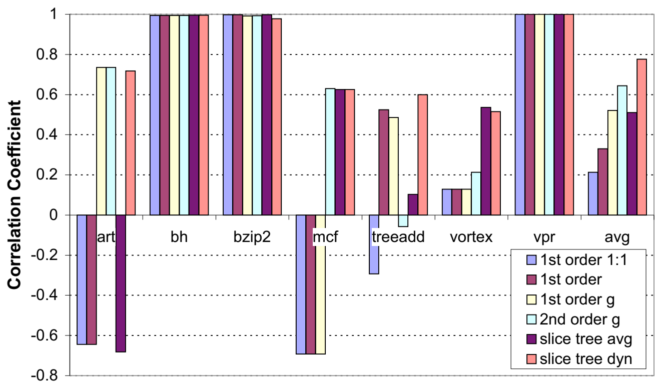


Figure 11: Correlation of predicted and actual speedups (idealized execution model: 4B data cache lines, perfect branch prediction)

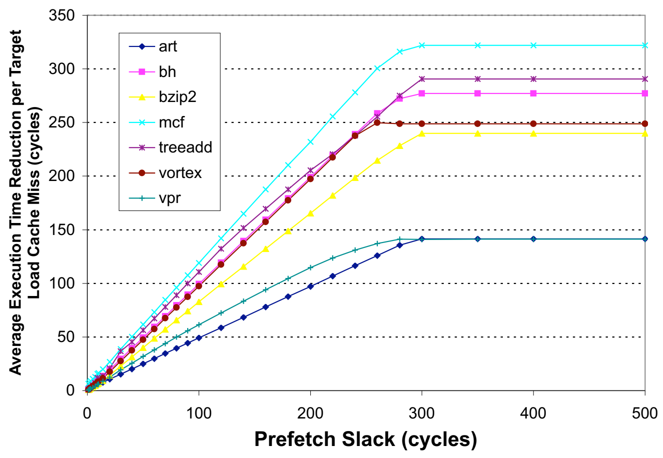


Figure 12: Sensitivity of execution time to prefetch slack

Figure 11 plots the correlation coefficient of the predicted and actual speedups for several of the p-thread modeling techniques already described, both with and without the use of second-order control flow information. In particular, the bars labeled *1st order 1:1* are for the path expression modeling prediction with prefetch sensitivity assumed to be one cycle improvement for each cycle of prefetch slack (for cache misses), up to the latency of main memory; *1st order* includes the impact of using the prefetch sensitivity measured earlier; *1st order-g* also includes the impact of prefetch slack distribution assuming a Gaussian latency distribution; *2nd order-g* uses the second order control flow path expression model, prefetch slack sensitivity and the Gaussian latency distribution model; *slice tree avg* represents a version of aggregate advantage formulation that includes the effects of prefetch slack sensitivity [26]; and *slice tree dyn* goes one step further and accounts for the effects of prefetch slack distribution (this is the same data presented earlier as the enhanced aggregate advantage technique).

The data in Figure 11 highlights the importance of using second-order control flow profile information, prefetch slack sensitivity and latency distribution information.

Below we examine the sources of modeling errors in more detail. Additional results are contained in Chapter 5 of the associated dissertation [1].

5.3.1 Prefetch Slack Sensitivity

Figure 12 plots the relationship between the prefetch slack for target loads that miss in the L1 cache, and the resulting reduction in execution time. As observed by Petric and Roth [26] the reduction is typically less than one cycle of execution time saved per cycle of prefetch slack. We define the *prefetch slack sensitivity* as the slope of a best fit line to the speedup/prefetch-slack function in Figure 12 from zero prefetch slack to the main memory latency of 300 cycles. The magnitude of the prefetch slack sensitivity was measured and this sensitivity was incorporated in the predictions made by both the enhanced aggregate advantage prediction technique and our path expression based prediction technique.

5.3.2 Prefetch Slack Distribution

Figure 3 plots a histogram of the prefetch latencies of the best p-thread found for the selected target load in 175.vpr. There is a wide distribution of latency tolerances (or prefetch slack) in the prefetches generated by this static p-thread. It is possible to predict the impact such variation has on performance by computing the *path length variance* [2]. We evaluated this approach using a simple Gaussian approximation and found that it reduces prediction error by 13%. However, it should be clear that a Gaussian is a poor approximation to the form of multimodal distribution in Figure 3 and hence further research to more accurately account for this effect would be beneficial.

5.3.3 Branch Outcome Correlation

Figure 11 shows that both 181.mcf and 255.vortex benefit from the application of second order control flow information to capture the impact of correlated branch outcomes (see data labeled *2nd order-g*).

5.3.4 Spatial Locality

We explored the impact of using 4-byte blocks to eliminate the effects of spatial locality on p-thread performance impact, which is not modeled by our proposed technique. The benchmarks that show the most significant change are treeadd and 181.mcf. The behavior for 181.mcf was investigated in detail and it was found that the profile data for 32B cache lines did not accurately predict the prefetch slack of the helper threads. The reason was that a load in the helper thread that almost always hit when executed as part of the main thread, was found to miss when executed by the helper thread in advance of another memory access that was not in the slice. When the cache line size is instead set to 4B (the size of a pointer), this effect is eliminated and the predictions for helper thread latency are closer to the actual observed latency.

5.3.5 Resource Constraints

Figure 13 illustrates the impact of making additional thread contexts available to run the selected helper threads for 256.bzip2. As additional thread contexts are made available the resulting speedup is closer to that predicted using the proposed modeling technique.

5.3.6 Equivalent Linked Data Structure Traversals

Figure 14 plots the predicted and actual speedups for 181.mcf assuming a memory hierarchy that stores data in

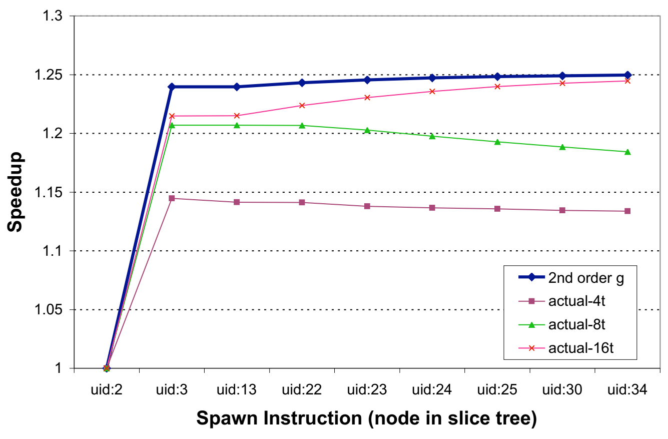


Figure 13: Increasing thread resources for 256.bzip2 reduces prediction error (idealized execution model: 4B cache lines, perfect branch prediction).

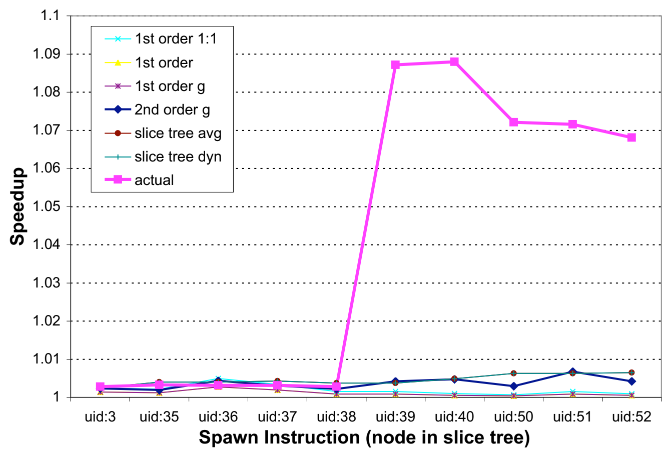


Figure 14: Predicted and actual speedups for 181.mcf with 4B cache lines and perfect branch prediction: Disparity due to unmodeled equivalent linked data structure traversals.

4-byte blocks. The large actual speedup encountered for the longer p-threads was found to be a result of p-threads where the main thread did not follow the implied control flow assumed during p-thread creation. Further analysis revealed this was because the main thread would return to the same location in the data structure and *then* access the data which has been "incorrectly" prefetched by the p-thread.

6. RELATED WORK

In this section we summarize related work not mentioned elsewhere in this paper. We believe the second-order control flow graph introduced in this paper is novel. The notion of profiling paths through a program dates to Fisher's work on trace scheduling [13]. Ball and Larus proposed efficient path profiling [4]. Larus proposed whole program paths [19]. Data flow frequency analysis using "two-edge profiling" was proposed by Mehofer and Scholz [22], who subsequently proposed data flow frequency analysis using whole program paths [30]. We note that [22] uses the first-order control flow graph and the quantity $p(u, v, w)$ which is the two edge probability. In the context of predicting helper thread performance impact, we believe our work is more general since it uses the second-order control flow graph and path expressions for analysis. Path expressions allow one to predict the expected latency of the main thread to help compute the expected benefit of a helper thread. As noted earlier, the path expression framework also enables one to compute the variance in this latency [2]. It is not clear how to compute these quantities using the approaches in [22, 30]. Our predecessor selection and pruning algorithm in Section 4.3 could be viewed as a form of *dynamic program slicing* [3] using path expressions. Prior work has looked at program optimizations based upon analysis of branch correlation. For example, Bodik [5] proposed statically analyzing branch correlation to remove redundant branches via code restructuring.

7. SUMMARY

This paper proposes and evaluates a technique for optimizing simple p-threads using control flow profile information and building on the aggregate advantage framework of Roth and Sohi [29] and the path expression modeling framework described in [2]. This technique combines the effectiveness of the aggregate advantage optimization framework with the efficiency and compiler friendly implementation advantages of a path expression based framework. We evaluated extending the control flow modeling approach in [2] by incorporating branch outcome correlation information. This improves the accuracy of the speedup predictions, as does more accurately modeling of the distribution of prefetch latencies. Further research on compile time mechanisms to predict prefetch slack sensitivity and incorporating more realistic memory disambiguation (perhaps including probabilistic pointer analysis techniques [10]) is needed to evaluate the practicality of the proposed approach in the setting of an optimizing compiler. Our investigation into the sources of modeling errors indicates that modeling dataflow events such as equivalent linked data structure traversals and the impact of spatial locality on helper thread execution may yield significant improvements in prediction accuracy.

8. ACKNOWLEDGEMENTS

This work supported partly supported by the Natural Sciences and Engineering Research Council of Canada. We thank the anonymous reviewers along with Todd Mowry, Andreas Moshovos, and Greg Steffan for their valuable comments on this work.

9. REFERENCES

[1] T. M. Aamodt. *Modeling and Optimization of Speculative Threads*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 2006.

[2] T. M. Aamodt, P. Marcuello, P. Chow, A. González, P. Hammarlund, H. Wang, and J. P. Shen. A Framework for Modeling and Optimization of Prescient Instruction Prefetch. In *ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 13–24, 2003.

[3] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Conf. on Programming Language Design and Implementation*, pages 246–256, 1990.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Int'l Symp. on Microarchitecture*, pages 46–57, 1996.

[5] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Conf. on Programming Language Design and Implementation*, pages 146–158, 1997.

[6] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com, 1997.

[7] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, Department of Computer Science, June 1996.

[8] 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.

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

[10] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Interprocedural probabilistic pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15(10):893–907, 2004.

[11] 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 Int'l Symp. on Computer Architecture*, pages 14–25, 2001.

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

[13] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[14] GrammaTech. Codesurfer. http://www.grammatech.com, 2007.

[15] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[16] Intel Corporation. Intel®VTune^{TM} Performance Analyzer. http://www.intel.com, 2007.

[17] D. Kim, S.-W. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *2nd Intl. Symp. on Code Generation and Optimization (CGO 2004)*, pages 27–38, 2004.

[18] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *ASPLOS-X*, pages 159–170, 2002.

[19] J. R. Larus. Whole program paths. In *Conf. on Programming Language Design and Implementation*, pages 259–269, 1999.

[20] 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 *Conf. on Programming Language Design and Implementation*, pages 117–128, 2002.

[21] C.-K. Luk. *Optimizing the Cache Performance of Non-numeric Applications*. PhD thesis, Department of Computer Science, University of Toronto, January 2000.

[22] E. Mehofer and B. Scholz. Probabilistic data flow system with two-edge profiling. In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 65–72, 2000.

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

[24] T. C. Mowry and C.-K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *30th Int'l Symp. on Microarchitecture*, pages 314–320, 1997.

[25] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[26] V. Petric and A. Roth. Energy-effectiveness of pre-execution and energy-aware p-thread selection. In *32nd Int'l Symp. on Computer Architecture*, pages 322–333, 2005.

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

[28] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *7th Int'l Symp. on High-Performance Computer Architecture*, pages 37–48, 2001.

[29] A. Roth and G. S. Sohi. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *Int'l Symp. on Microarchitecture*, pages 430–441, 2002.

[30] B. Scholz and E. Mehofer. Dataflow frequency analysis based on whole program paths. In *The 11th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 95–103, 2002.

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[32] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Int'l Symp. on Microarchitecture*, pages 148–159, 1998.

[33] Standard Performance Evaluation Corporation. SPEC 2000 CPU benchmarks. http://www.spec.org/.

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

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

[36] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading on an experimental itanium 2 processor-based platform. In *ASPLOS*, pages 144–155, 2004.

[37] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Int'l Symp. on Computer Architecture*, pages 2–13, 2001.

[38] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th Int'l Symp. on Computer Architecture*, pages 172–181, 2000.