A Quantitative Framework for Automated Pre-Execution Thread Selection

Amir Roth
Department of Computer and Information Science
University of Pennsylvania
amir@cis.upenn.edu

Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin–Madison
sohi@cs.wisc.edu

Abstract

Pre-execution attacks cache misses for which address-prediction driven prefetching fails. In pre-execution, copies of cache miss computations are isolated from the main program and launched as separate threads called p-threads whenever the processor anticipates an upcoming miss. P-thread selection is the task of deciding what computations should execute as p-threads and when they should be launched such that total execution time is minimized. It is central to the success of pre-execution.

We introduce a framework for automated static p-thread selection, a static p-thread being one whose dynamic instances are repeatedly launched during the course of program execution. Our approach is to formalize the problem quantitatively and then apply standard techniques to solve it analytically. The framework has two novel components. The slice tree is a data structure that compactly represents a set of static p-threads and the relationships among them. Aggregate advantage is a formula that uses raw program statistics and computation structure to assign each candidate static p-thread a numeric score based on estimated latency tolerance and overhead aggregated over its expected dynamic executions.

We use the framework to select p-threads that cover L2 misses and study its effectiveness under different conditions via detailed simulation. We measure the effect of constraining p-thread length, locally optimizing p-threads, using different program samples as a statistical basis for selection, and varying several machine parameters. Our framework responds to these changes in an intuitive way. We also validate that aggregate advantage correctly models actual pre-execution.

1 Introduction

Second-level cache misses constrain processor performance, a problem that will worsen as memory latencies relatively increase. Driven by address prediction, non-binding prefetching hides memory latency by speculatively hoisting the cache miss portion of a load, overlapping it with prior instructions. Prefetching eliminates many misses, but certain static *problem loads* defy address prediction and their misses elude prefetching.

Pre-execution is a way to deal with problem loads. Pre-execution sidesteps address prediction and generates prefetch addresses by executing a *copy of the load computation* in parallel with the main program as a separate thread called a *p-thread* (these have also been called p-slices or data-driven threads) in a multithreaded processor. Hoisting is accomplished as the p-thread fetches and executes fewer instructions than the main program thread and thus initiates the cache miss first. The multithreaded execution model, which decouples p-threads from the main program and from one another, has many advantages. P-thread cache miss initiations are accelerated because p-threads are isolated from stalls and squashes that occur in the main thread. Overlapping is enhanced because while a cache miss stalls the p-thread, the main program thread continues fetching, executing and retiring instructions. With multithreaded processors becoming prevalent, pre-execution is gaining popularity [5, 10, 11, 15, 19].

The benefits and limitations of pre-execution have been well documented. Here, we attack the problem of *p-thread selection* [12], the task of deciding which p-threads to pre-execute and when to pre-execute them. P-thread selection is a crucial component of pre-execution. It is also a complex task that must balance many inter-related, often antagonistic concerns including miss latency tolerance, p-thread resource consumption (important when p-threads share resources with the main thread), and prefetch coverage and accuracy. Existing p-thread selection methods—both manual [19] and automatic [4, 5, 8, 9, 11]—are successful, but largely heuristic. We present a framework for attacking the problem in a formal, quantitative, and holistic way. The framework produces *static p-threads*, copies of which are launched repeatedly during program execution. The intervals for which p-threads are chosen can be short, modeling on-the-fly generation, or a full program run, modeling off-line selection.

Our framework selects p-threads by effectively conducting an analytical pre-execution limit study. This approach is possible because we consider only p-threads which can be directly derived from the original program or optimized versions thereof. This restriction (which is not severe) allows us to use an execution trace to *enumerate all possible p-threads of this form and select the best among them*, rather than generate p-threads from scratch. The *slice tree* is a data structure that compactly represents a set of static p-threads and the relationships between them. It allows us to accurately assess miss coverage, ensuring that pre-execution work is not replicated and aids in decomposing the problem into orthogonal sub-problems. We apply a simple model called *aggregate advantage* to calculate the performance benefit of each candidate static p-thread aggregated over its dynamic executions. Aggregate advantage uses a few key abstractions to model the interactions of a p-thread with the main thread. Finally, we "solve" the selection problem by choosing the set of p-threads that maximizes total performance benefit. The framework also includes facilities for optimizing and merging p-threads. Constructed from first principles, the framework is simple and, via a few intuitive parameters, applicable to a range of pre-execution implementations and processor configurations. For instance, in this work we

assume a simultaneous multithreading (SMT) substrate, but the framework can model processors that execute p-threads on dedicated resources.

Because it considers all possible static p-threads and uses standard optimization techniques to maximize expected performance, our framework has value beyond its p-thread selection capabilities. As a formal model of pre-execution, it can be used to *study pre-execution performance potential* under different configurations. The results of these studies would be rough—the framework makes several assumptions to achieve computational leverage and its performance model, *aggregate advantage*, is simplistic—but may provide useful insight and intuition. The framework also forms an analytical foundation for future p-thread selection algorithms.

We evaluate our framework by using it to select p-threads targeting L2 misses for the SPEC2000 benchmarks. We validate the framework's performance model by comparing predicted performance results and diagnostics to those observed during pre-execution simulation. We measure pre-execution sensitivity to variations in p-thread construction and machine parameters.

The next section describes the selection problem. Section 3 details the framework. The final three sections contain an evaluation, related work, and our conclusions.

2 Background

We review pre-execution and introduce the p-thread selection problem using an example. The loop at the top of Figure 1 iterates over a list of pharmacy transactions and sums the appropriate drug prices. Load #09 (rx[rxid].price), is a problem load. We attack its misses—whose addresses do *not* form an arithmetic series—via pre-execution. The bottom of the figure shows a p-thread-assisted execution—main thread on the left with loop iterations separated by horizontal lines and p-thread to the right. In Section 3, we show how our framework constructs this p-thread.

Abstract pre-execution model. A p-thread has two components: the *body* is a list of instructions that constitutes a cache miss computation, the *trigger* is a PC of an instruction in the main thread. A *static p-thread* is a trigger/body pair. A *dynamic p-thread* is an instance of a p-thread body launched when the main thread executes an instance of the corresponding trigger. In the figure, the p-thread body is shown in a box with the trigger as an annotation on top. A dynamic p-thread instance is launched by every main thread instance of #11.

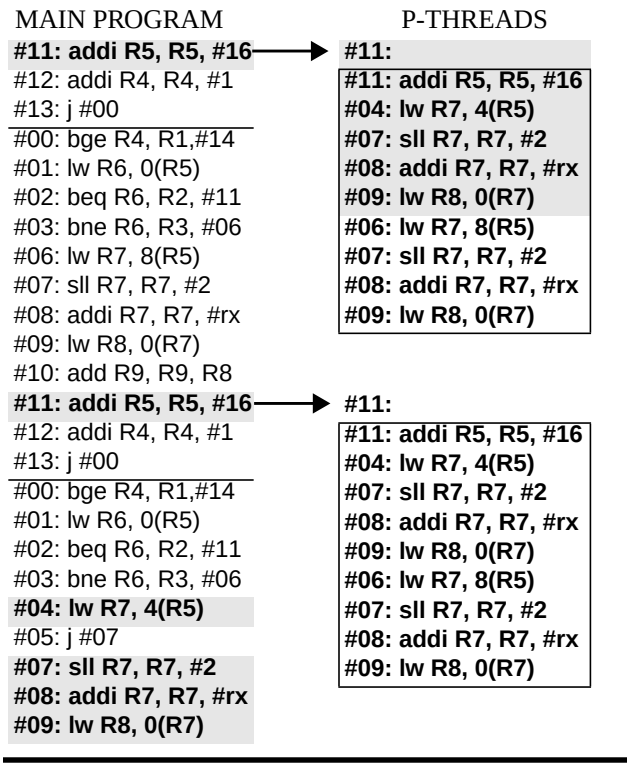
A *linear p-thread* corresponds to one dynamic computation. Our example p-thread merges two linear p-threads: [#11,#04,#07,#08,#09], a slice that includes instruction #04 (rxid=xact[i].rxid), and [#11,#06,#07,#08,#09], which includes instruction #06 (rxid=xact[i].gen_rxid). As it is not known at launch time which computation a given iteration will execute, our framework hedges and executes both. This is simpler and often faster than having the p-thread figure out which to execute. Our framework deals primarily with linear p-threads which it extracts from dynamic instruction traces. Merging is performed as a post-processing step. Our example will focus on the first linear p-thread. This p-thread and the main thread computation it corresponds to are shaded.

A p-thread's target cache miss, trigger, and body are related by the dynamic execution of the original program. A given p-thread's body is the computation of the target miss starting from the trigger. This relationship forms the basis of the *abstract pre-execution model*. Starting at the trigger, the p-thread and main thread execute in parallel, with the p-thread arriving at the cache miss first by virtue of fetching and executing only the load computation as opposed to the full program. Our framework uses this abstract model in its calculations. For a given trigger/miss pair, it estimates how much faster the body executes stand-alone (i.e., as a p-thread) than when embedded in the full program.

The miss/trigger/body relationship also lets us turn the problem from p-thread construction into *p-thread selection*. The body and trigger form a dynamic backwards data-dependence slice that starts at the miss. We can *enumerate all possible linear p-threads* that are derived from

FIGURE 1. Pre-execution running example

```
for (i = 0; i < N_XACT; i++) {     // 100 iterations
  if (xact[i].cover==FULL)
    continue;                      // 20 times
  else if (xact[i].cover==PART)
    rxid = xact[i].rxid;           // 60 times (#04)
  else
    rxid = xact[i].gen_rxid;       // 20 times (#06)
  receipts += rx[rxid].price;      // 80 times/40 misses (#09)
}
```



the program by constructing successively longer backward slices. The trace in Figure 1 yields five candidates (written as trigger:[body]): #08:[#09], #07:[#08,#09], #04:[#07,#08,#09], #11:[#04,#07,#08,#09], and #11:[#11,#04,#07,#08,#09].

P-thread sequencing. Our p-threads are *control-less* and *unchained*—they are fixed instruction sequences, executed in their entirety, and launched only by the main thread. These restrictions allow us to analyze a static p-thread as the aggregate of its dynamic instances—we know exactly what each instance looks like and how many of them there are. Control-less-ness also simplifies dynamic backward slicing. Disallowing control and chaining does not constrain the power of pre-execution. These features are primarily used to implement p-thread loops for increased latency tolerance. We simulate loops by including multiple induction copies in a p-thread, an idiom called *induction unrolling* [4, 15]. Our example p-thread uses one level of unrolling, it uses a copy of instruction #11 to skip one loop iteration ahead.

Aspects of p-thread selection. Since it is a backward slice, the only thing we can vary in a p-thread is its length. Choosing a proper p-thread length can be subtle. Obviously, a longer p-thread is launched earlier with respect to its target miss and will typically tolerate more latency. It also executes more instructions and consumes more resources. However, that is not all. A given static p-thread will launch a certain number of *useless dynamic instances*. An instance is useless because the load it pre-executes either: 1) hits in the cache anyway, or 2) never arrives, i.e., the main thread executes along a different path than the one the p-thread implicitly assumes. Our example p-thread is launched once per loop iteration by instruction #11 while not every loop iteration contains an instance of load #09. Increasing p-thread length often increases the incidence of useless p-threads of the second kind. Another phenomenon is that longer p-threads, while tolerating more latency per miss, *cover fewer misses*. In our example, a given instance of load #09 may be computed using either instruction #04 or #06. A p-thread that contains one of these two instructions will cover only the corresponding subset of misses. Covering all #09 misses requires two linear p-threads. Our framework simultaneously examines all of these considerations and makes trade-offs between them quantitatively.

3 P-Thread Selection Framework

We construct the framework from first principles. First, we introduce the *slice tree*, a data structure for representing static p-threads and the relationships among them. Next, we describe *aggregate advantage*, a formula that quantifies the performance impact of static p-thread candidates and show how it is used to select the best p-thread from within a single computation. We then show how the slice tree enables the selection of multiple linear p-threads from multiple, partially overlapping computations. Finally, we discuss merging and optimization.

3.1 Structure, Basic Assumptions, and Raw Data

Finding good p-threads is computationally intensive. The control-less nature of the p-threads allows us to explicitly analyze a single dynamic p-thread execution and estimate the aggregate effects of all executions of the static p-thread using multiples of expected launch and miss coverage counts. Our framework analyzes statistical data of this nature. In this work, we collect data from program traces, but static estimation may also be used.

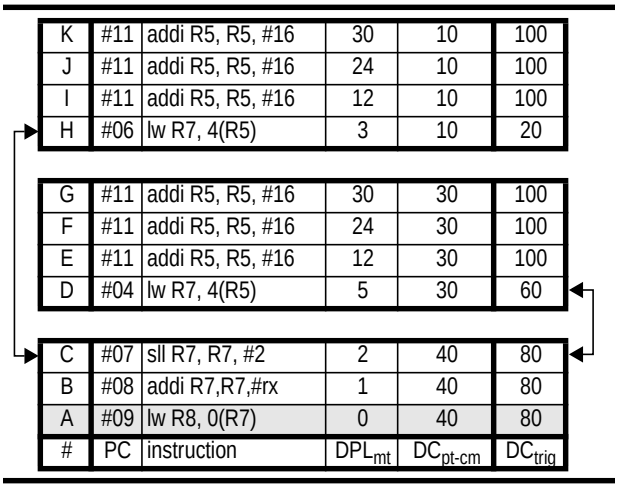
Slice tree. The data we collect is compactly represented in a data structure called a *slice tree*, a tree of static backward slices with the static problem load at the root. Each tree node represents an instruction and a static p-thread whose trigger is that instruction. The p-thread is constructed by walking from the node to the root. The tree arrangement is used to represent *static p-thread overlap*—p-threads overlap if their dynamic executions target overlapping sets of dynamic misses. We will deal with the ramifications of p-thread overlap in section 3.3.

Figure 2 shows the slice tree representing both linear slices from our example. Instruction #09 (node A) is the root. The p-thread from our example is represented by node F: it's trigger is #11 and its body is constructed by walking up from F to the root [#11,#04,#07,#08,#09]. The slice formed by the nodes A–C and H–K is the alternate computation of #09, the one that contains #06. For pedagogical reasons, we include an extra level of induction unrolling in each linear slice (nodes G and K).

Raw statistics. Each tree node (p-thread) is associated with three statistics. DC_{trig} counts the dynamic instances of the trigger in the trace; it is an estimate of the number of times the p-thread would be launched. $DC_{pt\text{-}cm}$ counts the number of times this list of instructions appears in the trace as the *backwards slice of a miss*. $DC_{pt\text{-}cm}$ estimates the number of misses a p-thread would cover. Note, DC_{trig} is a trigger property while $DC_{pt\text{-}cm}$ is a p-thread property. For instance, nodes E and I have the same DC_{trig} values (#11 triggers both) but different $DC_{pt\text{-}cm}$ values (they correspond to different p-threads).

To gain computational leverage, the slice tree explic-

FIGURE 2. Slice tree



itly stores only p-thread instructions. The parallel work in the main thread is represented by a single count, DPL_{mt}, which is the average number of dynamic main thread instructions that exist between the instruction in question and the target load. Our calculations actually count main thread work from the trigger ($DTRIG_{mt}$): an instruction's $DTRIG_{mt}$ with respect to any trigger is obtained by subtracting its DPL_{mt} from the trigger's. For example, assuming F is the trigger, B's $DTRIG_{mt}$ is 23 (24–1). The main thread sequences 23 instructions before it can execute B. B's $DTRIG_{pt}$ is 4, the p-thread sequences only 4 instructions before it can execute B.

Our approach has two limitations. First, by representing only slice instructions we do not consider the instructions outside the slice as triggers. Non-slice triggers *may* have lower DC_{trig} values, and require fewer useless launches to cover the same number of misses. More seriously, by not retaining temporal information we implicitly assume that a single dynamic p-thread instance is active at a time. This assumption is not egregious for L2 cache misses. We willingly trade these inaccuracies for the computational leverage provided by summary information.

Divide and conquer. We create a separate slice tree for the computations of each static problem load. Since p-threads for different static loads do not overlap–they do not target the same misses–we treat each tree as a subproblem and solve it separately. A post-pass merges the linear p-threads that result from the solutions of all trees.

3.2 Estimating the Benefit of a P-Thread Candidate

Backward data-dependence slicing is straightforward, even in hardware [4, 11]. An *N*-instruction slice presents a choice of *N* linear p-threads. P-thread selection amounts to choosing the backwards sub-slice that makes the best p-thread. P-thread goodness is measured by a function, aggregate advantage (ADV_{agg}), whose definition is summarized in Table 1.

Aggregate Advantage (ADV_{agg}). P-thread selection balances four considerations: *latency tolerance*, *overhead*, *miss coverage* and *useless p-thread frequency*. Longer p-threads tolerate more latency per miss, but incur more overhead, generally cover fewer misses, and generally result in more useless p-thread instances. *Aggregate advantage (ADV_{agg})* combines these into a single numerical score, allowing them to be simultaneously optimized. The aggregate advantage of a static p-thread is the estimated number of cycles by which its dynamic instances collectively accelerate program execution. Shown in equation 1 of Table 1, aggregate advantage is the difference of two terms. *Aggregate latency tolerance (LT_{agg})* is the number of cycles by which a p-thread's dynamic instances accelerate cache misses. *Aggregate overhead (OH_{agg})* is the number of cycles by which a p-thread's instances slow down the main thread by stealing resources from it. Note the multipliers for LT_{agg} and OH_{agg}. Every launched p-thread instance (DC_{trig}) exacts overhead, but only instances that pre-execute actual misses ($DC_{pt\text{-}cm}$) achieve any latency tolerance. Useless instances tolerate no latency because their corresponding main thread loads have none.

Overhead per dynamic p-thread (OH). On an SMT processor, the number of sequencing cycles stolen from the main thread is the most direct measure of p-thread overhead. Other forms of contention are either subsumed by this measure (e.g., execution slots), not easily estimated (e.g., bus bandwidth), or both (e.g., buffer occupancy). The number of cycles it takes to sequence a p-thread instance is its length (SIZE) divided by the sequencing width of the processor ($BWSEQ_{proc}$). Since overhead is opportunity cost, we discount it by the expected main thread sequencing utilization ($BWSEQ_{mt}/BWSEQ_{proc}$). For instance, if the main thread utilizes only half of the available sequencing bandwidth, then a p-thread is only penalized for half of its bandwidth consumption. Half the cycles would not have been used by the main thread.

Latency tolerance per useful dynamic p-thread (LT). P-thread latency tolerance is estimated as a difference in

TABLE 1. Static p-thread selection framework summary

Equation or Definition	Description
EQ1. $ADV_{agg}(p) = LT_{agg}(p) - OH_{agg}(p)$	**Aggregate advantage**
EQ2. $OH_{agg}(p) = DC_{trig}(p) * OH(p)$	**Aggregate overhead:** overhead is incurred for every dynamic p-thread instance (DC_{trig})
EQ3. $LT_{agg}(p) = DC_{pt\text{-}cm}(p) * LT(p)$	**Aggregate latency tolerance:** latency is tolerated only for instances that pre-execute actual misses ($DC_{pt\text{-}cm}$)
EQ4. $OH(p) = (SIZE(p)/BWSEQ_{proc}) * (BWSEQ_{mt}/BWSEQ_{proc})$	**Per dynamic instance overhead:** overhead is discounted by main thread sequencing utilization ($BWSEQ_{mt}/BWSEQ_{proc}$)
EQ5. $LT(p) = MIN(SCDH_{mt}(p) - SCDH_{pt}(p), L_{cm})$	**Per useful dynamic instance latency tolerance:** latency tolerance is bounded by cache miss latency (L_{cm})
EQ6. $SCDH_t(pi) = MAX(SC_t(pi), SCDH_t(DFP(pi))) + L_{op}(pi)$	**Completion time for i^{th} p-thread instruction:** DFP(pi) are pi's dataflow predecessors, $L_{op}(pi)$ is its operation latency
EQ7. $SC_t(pi) = DTRIG_t(pi) / BWSEQ_t$	**Sequencing constraint on i^{th} p-thread instruction:** instruction's distance from trigger divided by sequencing bandwidth
EQ8. $BWSEQ_{mt}$, $BWSEQ_{pt}$, $BWSEQ_{proc}$, L_{cm}	**Constant parameters**
EQ9. $ADV_{agg\text{-}red}(p) = ADV_{agg}(p) - LT(p) * DC_{pt\text{-}cm}(CHILD(p))$	**Reduced aggregate advantage**

miss computation execution times. Starting from the trigger (when the main thread and p-thread begin executing in parallel), we calculate the number of cycles it would take the p-thread to execute the cache miss ($SCDH_{pt}$) and the number of cycles it takes an *unassisted main thread* to do the same ($SCDH_{mt}$). The difference between these estimates is the number of cycles by which the p-thread hoists the miss with respect to the main thread and thus the amount of latency it tolerates. Since there is no benefit to tolerating more latency than the latency of the miss, we bound LT by the original miss latency (L_{cm}).

Our execution time estimation function is *sequencing-constrained dataflow-height (SCDH)*. It is the standard recursive dataflow-height equation with an additional *sequencing constraint (SC)* which models the cycle at which the instruction is sequenced (fetched). SCDH can be used to model both standalone p-thread execution ($SCDH_{pt}$) and embedded main thread execution ($SCDH_{mt}$) using proper definitions of the sequencing constraint. To calculate SC for a given instruction, we divide *DTRIG*—its distance in dynamic instructions from the trigger—by the sequencing bandwidth. $SCDH_{pt}$ is smaller than $SCDH_{mt}$ because of SC: the p-thread sequences fewer instructions, so each instruction's $DTRIG_{pt}$ is smaller than its $DTRIG_{mt}$.

Constant parameters. Our calculations parameterize the underlying machine using four constants. $BWSEQ_{proc}$ is the sequencing bandwidth of the processor and is straightforward to define. $BWSEQ_{pt}$ is the rate at which a p-thread is *allowed* to sequence. We set $BWSEQ_{pt}$ to 1 because p-threads are single computations that execute serially and there is no sense allocating a p-thread more sequencing bandwidth than it will use. $BWSEQ_{mt}$ is the rate at which the main thread *actually* sequences. This constant is tricky to set. Not only is it difficult to measure—we are really interested in the main thread's sequencing rate *up to* the miss, not including the miss—it also varies (often wildly) over the course of the program such that a constant rate may not accurately represent any single program region. L_{cm} is the miss latency. Like $BWSEQ_{mt}$, our framework treats L_{cm} as a constant, even though a conventional processor naturally overlaps misses—with program instructions or even other misses—to varying degrees.

Working Example. To illustrate the use of ADV_{agg}, we select a p-thread from the linear computation of load #09 that contains instruction #04 (nodes A–G in Figure 2). As shown in Figure 1, the loop executes 100 iterations, 80 iterations execute instances of #09, of which 40 are misses. Of the 80 #09 computations, 60 contain instruction #04 and 20 use #06. All operations have unit latency, and miss latency is 8 cycles. Note, the highest possible ADV_{agg} score in this case is 320: 8 cycles of latency tolerance for all 40 misses, with 0 overhead. This score is impossible to achieve if p-threads have non-zero cost. The processor is 4 wide, $BWSEQ_{mt}$ is 2, and $BWSEQ_{pt}$ is 1.

Figure 3 shows the ADV_{agg} calculations for five of the six candidates (B–G). The winning p-thread, F, is shaded. Each calculation is shown as a table. The top portion computes $SCDH_{mt}$ and $SCDH_{pt}$. The left term under max() is the sequencing constraint. Notice, $DTRIG_{mt}$ values are sparse, while $DTRIG_{pt}$ values are sequential. Each instruction has a single dataflow-predecessor; the right value under max is SCDH of the previous instruction. The bottom row calculates ADV_{agg}, plugging 8 for L_{cm}, 2 for $BWSEQ_{mt}$, and 4 for $BWSEQ_{proc}$.

Candidate C (B is similar so we do not show it) provides no sequencing advantage over the main thread. Starting at the trigger, both main thread and p-thread execute exactly the same instructions. However, it does incur overhead. Pre-executing this candidate will *increase* execution time by 20 cycles. Notice, DC_{trig} is 80 while $DC_{pt\text{-}cm}$ is 40; only 40 of the 80 executions will cover misses.

FIGURE 3. ADV_{agg} calculation working example

	candidate C	$SCDH_{mt}$	$SCDH_{pt}$
#07	sll R7, R7, #2	max(0/2, 0)+1 = 1	max(0/1, 0)+1 = 1
#08	addi R7, R7, #rx	max(1/2,1)+1 = 2	max(1/1, 1)+1 = 2
#09	lw R8, 0(R7)	max(2/2, 2)+8 = 10	max(2/1, 2)+8 = 10
[40 * min(10–10, 8)]– [80 * (2/4) * (2/4)] = 0–20 = **–20**			

	candidate D	$SCDH_{mt}$	$SCDH_{pt}$
#04	lw R7, 4(R5)	max(0/2, 0)+1 = 1	max(0/1, 0)+1 = 1
#07	sll R7, R7, #2	max(3/2, 1)+1 = 3	max(1/1, 1)+1 = 2
#08	addi R7, R7, #rx	max(4/2, 3)+1 = 4	max(2/1, 2)+1 = 3
#09	lw R8, 0(R7)	max(5/2, 4)+1 = 12	max(3/1, 3)+8 = 11
[30 * min(12–11, 8)]– [60 * (3/4) * (2/4)] = 30–23 = **7**			

	candidate E	$SCDH_{mt}$	$SCDH_{pt}$
#11	addi R5, R5, #16	max(0/2, 0)+1 = 1	max(0/1, 0)+1 = 1
#04	lw R7, 4(R5)	max(7/2, 1)+1 = 5	max(1/1, 1)+1 = 2
#07	sll R7, R7, #2	max(10/2, 5)+1 = 6	max(2/1, 2)+1 = 3
#08	addi R7, R7, #rx	max(11/2, 6)+1 = 7	max(3/1, 3)+1 = 4
#09	lw R8, 0(R7)	max(12/2, 7)+8 =15	max(4/1, 4)+8 = 12
[30 * min(15–12,8)]– [100 * (4/4) * (2/4)] = 90–50 = **40**			

	candidate F	$SCDH_{mt}$	$SCDH_{pt}$
#11	addi R5, R5, #16	max(0/2, 0)+1 = 1	max(0/1, 0)+1 = 1
#11	addi R5, R5, #16	max(12/2, 1)+1 = 7	max(1/1, 1)+1 = 2
#04	lw R7, 4(R5)	max(19/2, 7)+1 = 11	max(2/1, 2)+1 = 3
#07	sll R7, R7, #2	max(22/2, 11)+1 = 12	max(3/1, 3)+1 = 4
#08	addi R7, R7, #rx	max(23/2, 12)+1 = 13	max(4/1, 4)+1 = 5
#09	lw R8, 0(R7)	max(24/2, 13)+8 = 21	max(5/1, 5)+8 = 13
[30*min(21–13,8)]– [100*(5/4)*(2/4)] = 240–63 = **177**			

	candidate G	$SCDH_{mt}$	$SCDH_{pt}$
#11	addi R5, R5, #16	max(0/2, 0)+1 = 1	max(0/1, 0)+1 = 1
#11	addi R5, R5, #16	max(6/2, 1)+1 = 4	max(1/1, 1)+1 = 2
#11	addi R5, R5, #16	max(18/2, 4)+1 = 10	max(2/1, 2)+1 = 3
#04	lw R7, 4(R5)	max(25/2, 10)+1 = 14	max(3/1, 3)+1 = 4
#07	sll R7, R7, #2	max(28/2, 14)+1 = 15	max(4/1, 4)+1 = 5
#08	addi R7, R7, #rx	max(29/2, 15)+1 = 16	max(5/1, 5)+1 = 6
#09	lw R8, 0(R7)	max(30/2, 16)+8 = 24	max(6/1, 6)+8 = 14
[30*min(24–14, 8)]–[100*(6/4)*(2/4)]= 240–75 = **165**			

Candidate D provides minimal sequencing advantage—the p-thread skips instructions #05 and #06—and 1 cycle of latency tolerance. Notice, #04 is executed only 60 times (#06 is executed the other 20 times) and the computation triggered by #04 covers only 30 misses. The p-thread incurs overhead for each of 60 p-threads launched for a positive ADV_{agg} of 7 cycles.

Since instances of #11 occur once per iteration, candidate E's DC_{trig} is 100. $DC_{pt\text{-}cm}$ is still 30—the computation includes instruction #04 and correctly pre-executes only 30 misses. Note, $DC_{pt\text{-}cm}$ monotonically decreases as p-thread length increases. A longer slice corresponds to fewer dynamic computations. In contrast, DC_{trig} has no relationship to p-thread length. Trends aside, candidate E is better than candidate D. Although the number of useless p-threads (DC_{trig}–$DC_{pt\text{-}cm}$) grows from 30 to 70 and per p-thread overhead increases, the additional 2 cycles of latency tolerance per miss produces a net gain.

The final two candidates are similar: F uses a single level of induction unrolling, G unrolls twice. The first unrolling provides candidate F with an additional sequencing advantage of 12 instructions over the main thread, which translates into 5 additional cycles of execution time advantage for a total of 8. This is as much latency tolerance as we need. The score for this candidate is 177: full latency tolerance for 30 misses, at the cost of 63 overhead cycles. With full latency tolerance already achieved, the added unrolling of candidate G only increases overhead.

3.3 Selecting Multiple P-Threads Simultaneously

P-thread F, which we found in the previous section, covers only 30 of the 40 #09 misses. We now show how to select a *set of p-threads* by optimizing multiple, partially overlapping linear slices simultaneously.

P-thread overlap. As hinted in Section 3.1, the possibility of overlap makes a naive approach inappropriate for selecting multiple p-threads that cover the misses of a single static load. If two static p-threads *overlap*—if at least one dynamic miss is pre-executed by both of them—then their aggregate latency tolerances do not add. Once one p-thread tolerates the latency of a miss, a second p-thread cannot tolerate it again.

P-threads in a slice tree obey parent-child relationships. Shorter, less specialized p-threads which cover more misses are parents. Longer, more specialized p-threads which cover fewer misses but more latency per miss are children. In our Figure 2 example, C is the child of B, D of C, E and H of D, and so on. The parent-child relationship is transitive, but not total: e.g., E and I are not parent and child. The parent-child relationship quantifies p-thread overlap. A child covers some subset of the misses covered by its parent. The size of this subset is given by its $DC_{pt\text{-}cm}$. In fact, one slice tree invariant is that a parent's $DC_{pt\text{-}cm}$ is the sum of the $DC_{pt\text{-}cm}$ of its children.

Addition of aggregate advantages. Equation 9 in Table 1 defines the addition of latency tolerances (and hence advantages). If two p-threads are not a parent and child (either directly or indirectly) then their advantages simply add. If they are a parent-child pair, then the number of misses covered by both p-threads is $DC_{pt\text{-}cm}$(child) and the amount of latency that is "doubly-tolerated" for each of these is LT(parent). The latency tolerance correction—LT(parent) * $DC_{pt\text{-}cm}$(child)—is associated with the parent p-thread.

Iterative algorithm. The set of p-threads covering the misses of a single load is the one whose ADV_{agg}—where latency tolerance reductions due to overlap have been accounted for—sum to a maximum. Because p-threads within a slice tree obey certain relationships to one another, we can find this set iteratively rather than via exhaustive search. For each separate linear slice in the tree, we select a p-thread as in the previous section. If any of the independently selected p-threads overlap, we reduce the advantages of the parents and reselect. The process terminates once reductions performed in one iteration do not change the p-threads selected in the next.

Working example. Obtaining a complete solution for our example slice tree is trivial. Selecting p-threads for the two linear slices separately, we choose F and J. As these two p-threads do not overlap, no corrections must be made, and no further iterations are necessary.

Overlap explains why multiple p-threads from the same linear slice are typically not chosen. In Figure 3, candidates D, E, F and G all have positive ADV_{agg}. Yet the framework—which maximizes ADV_{agg} *sum*—selects only F. Why? Assume both E and F are initially selected. F's LT_{agg} is 240 cycles (8 cycles for 30 misses); E's is 90 cycles (3 cycles for 30 misses). Since F is E's child, its presence reduces E's LT_{agg} by the shared latency tolerance, LT(E) * $DC_{pt\text{-}cm}$(F). Since E and F cover the same set of misses, F makes E completely superfluous!

3.4 Extensions: P-thread Merging and Optimization

Our framework includes two automated enhancements to basic p-thread selection: merging of partially redundant p-threads and p-thread optimization.

Merging. The two linear p-threads chosen in the previous section target disjoint sets of misses. However, instances of instruction #11 dynamically executed by these p-threads are redundant. To eliminate this redundancy, our framework merges the two p-threads. A merged p-thread achieves the same latency tolerance as separate instances of each of the original linear p-threads and incurs less overhead. Our framework merges p-threads with matching triggers. Merging proceeds in dataflow order, with register reassignment and code duplication performed as needed to preserve the semantics of the original p-threads. In our example, instructions #07, #08 and #09 are replicated in the final p-thread—one copy completes the computation that contains instruction #04, the other completes the computation that contains instruction #06.

Optimization. *Optimized p-threads* are not exact copies of dynamic computations from the program, but rather specialized versions of them. We fit p-thread optimization into our framework by allowing the $SCDH_{pt}$ and SIZE cal-

culations to use *any* instruction sequence that is functionally equivalent to the program derived sub-slice. P-thread optimization is both easier and more productive than its full program counterpart. Since our p-threads are control-less, traditional control-flow and iterative data-flow analyses are replaced by linear scans. Also, only optimizations that are enabled by the specialized nature of the p-thread are considered. Register allocation was already performed by the compiler that generated the initial program and scheduling is unnecessary since a p-thread is a single computation. We have found that *store-load pair elimination* and *constant folding* capture most p-thread optimization opportunities. Figure 2 contains one optimization opportunity: in candidate G, the two instances of instruction #16 (addi R5, R5, #16) may be folded into a single instruction (addi R5, R5, #32), reducing both p-thread latency (the data-flow height is cut by one instruction) and overhead.

4 Experimental Evaluation

We evaluate our framework by using it to select p-threads that target L2 misses. In section 4.4, we validate its performance model by comparing predicted statistics against statistics measured from pre-execution simulations. In sections 4.5 and 4.6, we measure its response to variations in several p-thread and machine parameters.

4.1 Methodology

We experiment with a suite of tools built using the SimpleScalar Alpha ISA and syscall modules. A cache simulator collects backward slices of L2 misses into slice trees which are written to files. A p-thread selection tool takes a slice tree file and parameters describing the processor (e.g., sequencing width) and p-thread constraints (e.g., p-thread length) and produces a list of p-threads. Default selection settings are maximum slicing scope and p-thread length of 1024 and 32 instructions, respectively, and full merging/optimization.

Performance results are obtained via detailed timing simulation. Our base configuration is a 6-wide dynamically scheduled processor, with a 6K-entry hybrid branch predictor, 14 stage pipeline, 80 reservation stations, and a maximum of 128 instructions or 64 memory operations in-flight. We model 16KB, 32B line, 2-way set-associative primary caches, a 256KB, 64B line, 4-way set-associative, 6-cycle access L2, and an infinite, 100-cycle access main memory. A 32B wide memory bus is clocked at one fourth processor frequency. 32 simultaneously outstanding misses are allowed. Because we target L2 misses, we disable the data cache fill path for p-thread loads—these prefetch only into the L2. Data cache prefetching improves performance but diminishes our ability to validate the framework.

Our pre-execution run-time implementation resembles data-driven multithreading (DDMT) without register integration [12, 15]. P-thread contexts are lightweight, consisting only of a map table. When a trigger is renamed, a p-thread is allocated to one of three additional contexts, or dropped if no context is available. The context is initialized with a copy of the main thread's map table allowing the p-thread to read pre-launch main thread register values. The context is freed when all p-thread instructions have been *renamed*. P-threads are injected into the execution core at register renaming in bursts, 6 instructions once every 6 cycles per active p-thread. P-thread instructions are allocated physical registers and reservation stations and contend with main thread instructions for these resources and for scheduling slots. They are not allocated ROB entries, are not retired, and do not modify architected state. Since we focus on the performance of the p-threads not of the selection algorithm, we do not model the p-thread selection/pre-execution interface; p-threads are accessible in one cycle from an ideal cache. We assume that this interface has only secondary performance effects.

We use the SPEC2000 integer benchmarks compiled at peak optimization levels using the Digital Unix cc compiler. We use 9 of the 12 benchmarks; *eon*, *gzip*, and *perlbmk* have negligible L2 miss rates. Performance numbers are reported using the training inputs cyclically sampled at 100M of every 1B instructions with 10M instruction warm-up phases. P-threads are selected using the same program sample on which they are subsequently measured, allowing us to check the framework's performance predictions. The top section of Table 2 characterizes the benchmarks. We show both baseline IPC and IPC with a perfect L2. IRPC measures instructions renamed per cycle. Since we sequence p-threads at register renaming, we use this value as our estimate of $BWSEQ_{mt}$.

4.2 P-thread Selection Performance

Although selection algorithm performance is *not* the focus of this paper, a few words about the cost of our implementation are in order. Profiling takes time proportional to the length of the trace plus the number of L2 misses scaled by the size of the slicing window. With a maximum of 32 instructions per p-thread, a typical benchmark's slice trees have about 200,000 nodes (p-thread candidates). Advantage calculations take time proportional to p-thread length, due to the SCDH definition. Once initial advantages are computed, selection time is proportional to the number of candidates; the median number of iterations per slice tree is 1. Merging and optimization take time proportional to the static size of the linear p-threads.

4.3 Absolute Performance Results

The *Pre-exec Actual* (shaded) portion of Table 2 shows the performance of the p-threads selected by our framework. In addition to IPC, we list the number of p-threads launched, the average number of instructions per p-thread and L2 misses covered both partially and in full. *Overhead IPC* and *Latency tolerance IPC* are produced by simulations which isolate p-thread cost and benefit, respectively; these are used to validate the model. Our framework selects p-threads that generally improve performance. The p-threads cover between 19% (*mcf*) and 84% (*vpr*) of the L2 misses—full coverage is achieved for about half of the covered misses—and result in performance improvements of up to 25% (*vpr*). One benchmark, *crafty*, experiences a 1% performance degradation.

4.4 Model Validation

In the scope of this work, absolute results are less important than demonstrations of the framework's computational power and modeling fidelity. Our framework uses standard techniques to find optimal solutions for a given function. What we need is confidence that its assumptions and the function it optimizes, ADV_{agg}, accurately model reality such that good solutions in the model space are also good in reality. Put differently, we want to know that only 19% of *mcf*'s misses are covered because the p-threads don't exist, not because the framework can't find them.

Our framework's calculations are explicit predictions of p-thread behavior. As one form of validation, we check these against simulated measurements. We check overhead and latency tolerance separately to help pinpoint model inaccuracies. Predicted diagnostics are shown in the bottom of portion of Table 2 (*Pre-exec Predict*).

Overhead. *P-thread length* predictions are self-fulfilling. *P-thread launch counts* are occasionally over-estimated (e.g., *bzip2*) due to the finite number of p-thread contexts. Most often, they are under-estimated because the framework does not model wrong-path triggering.

Overhead performance degradation is measured in two ways. In *execute*, p-thread instructions execute, but L2 misses are not satisfied (these p-threads do not have the pre-execution effect). In *sequence*, p-thread instructions consume sequencing cycles but are immediately discarded. The first simulation measures true overhead, the second measures overhead as modeled by our framework—the only cost of a p-thread instruction is the bandwidth consumed to sequence it. The result proximity of these two simulations to baseline IPC may suggest that p-thread overhead is an insignificant factor in p-thread selection. As we show in Section 4.5, this is not the case. Overhead is low precisely because our framework explicitly minimizes it.

Latency Tolerance. To measure *miss coverage*, we time-stamp cache blocks with p-thread request, main thread request, and ready times. At retirement, p-thread covered would-have-been misses are identified, and classified as *full* or *partial*, by time-stamp relationships. Miss coverage is difficult to predict, as it is affected by several factors that ADV_{agg} does not account for. For partial or general miss coverage, *over-estimation* (too few misses actually covered) indicates p-thread issue delays caused by contention with the main thread and other p-threads. *Under-estimation* (too many misses actually covered—a good problem to have) implies the presence of unintentional prefetches within p-threads (a particular load may not merit p-thread for its own sake, but may be embedded in a p-thread targeted for another). For full miss coverage in particular, over-estimation implies post-issue p-thread delays due to memory bus contention. Under-estimation indicates main thread delays, primarily due to branch mispredictions. Each effect is present to some degree in every benchmark.

We isolate *performance improvement due to latency-tolerance* via a simulation in which p-thread instructions are not charged for bandwidth they consume. Performance improvement is the most difficult metric to predict. It is almost universally over-estimated because of our framework's assumption that miss latency translates cycle-for-cycle into execution latency and, therefore, that miss latency tolerance translates directly into speedup. As we mentioned earlier, this is certainly not the case. A dynamically scheduled processor often partially overlaps L2 misses with program instructions and even with L1 and

TABLE 2. Basic results and performance model validation

		bzip2	crafty	gap	gcc	mcf	parser	twolf	vortex	vpr
Base	Sampled instructions (M)	6000.00	2600.00	900.00	500.00	900.00	1300.00	1300.00	1700.00	1100.00
	Loads (M)	1509.20	731.49	218.49	116.15	246.03	295.70	291.56	458.21	327.17
	L2 misses (M)	20.15	0.91	1.91	0.72	63.52	4.59	9.07	1.17	6.88
	IRPC ($BWSEQ_{mt}$)	*2.75*	*4.00*	*2.11*	*2.92*	*0.38*	*1.82*	*1.60*	*2.94*	*1.57*
	IPC	**2.43**	**2.73**	**1.56**	**1.72**	**0.22**	**1.21**	**0.93**	**2.74**	**1.08**
	Perfect L2 IPC	**4.16**	**3.06**	**2.52**	**2.32**	**1.71**	**2.33**	**2.14**	**3.54**	**2.07**
Pre-exec Actual	P-threads launched (M)	171.88	8.51	16.95	9.60	88.96	53.72	61.98	16.58	69.94
	P-thread length	15.47	15.02	6.73	5.51	8.43	5.59	8.31	12.00	9.56
	Overhead IPC (execute)	2.20	2.71	1.55	1.71	0.22	1.20	0.93	2.72	1.07
	Overhead IPC (sequence)	2.23	2.72	1.56	1.72	0.22	1.20	0.93	2.73	1.08
	Misses covered (M)	14.11	0.58	1.36	0.39	12.53	1.95	6.68	0.87	5.75
	Misses fully covered (M)	3.61	0.24	0.65	0.24	3.83	1.01	4.12	0.26	3.08
	Latency tolerance IPC	2.70	2.72	1.70	1.82	0.23	1.30	1.17	2.94	1.37
	IPC	**2.47**	**2.70**	**1.68**	**1.81**	**0.23**	**1.29**	**1.14**	**2.91**	**1.35**
Pre-exec Predict	P-threads launched (M)	338.77	3.05	9.23	3.26	38.85	22.39	25.10	7.81	31.62
	P-thread length	18.51	14.28	7.40	5.64	8.91	5.74	9.50	11.75	9.92
	Overhead IPC	2.03	2.72	1.55	1.71	0.22	1.21	0.93	2.71	1.07
	Misses covered (M)	15.60	0.50	1.15	0.29	18.14	1.87	7.18	0.74	6.37
	Misses fully covered (M)	8.19	0.01	0.57	0.12	16.17	1.37	4.68	0.22	4.81
	Latency tolerance IPC	5.85	2.84	1.79	1.84	0.36	1.42	1.51	2.96	2.36
	IPC	**3.99**	**2.82**	**1.78**	**1.83**	**0.36**	**1.41**	**1.49**	**2.92**	**2.30**

other L2 misses. Simply, the framework often believes that there is more latency to tolerate than actually exists.

Summary. Experiments for other configurations (e.g., narrower processor, slower memory) show similar results. Our framework's predictions are not perfect, but are close in many cases. This suggests that ADV_{agg} is a reasonable model for L2 miss pre-execution under many conditions.

4.5 Contribution of Framework Components

Another way to demonstrate the merit of our framework is to isolate the effects of its components. Our framework explicitly accounts for three factors unaccounted for by previous schemes: 1) required latency tolerance, 2) overhead, and 3) p-thread overlap. Figure 4 shows four experiments, starting with greedy p-thread selection and successively incorporating these considerations. We show five diagnostics for each experiment. Full and partial miss coverages, as percentages of baseline L2 misses, are stacked bars. P-thread overhead, computed as p-thread instructions sequenced over the main thread instructions retired, is an uptick. Average dynamic p-thread length is a cross. Finally, percent speedup is shown as rotated text.

In *greedy* selection (first bar from the left), p-threads are chosen for maximum miss coverage and prefetch distance without regard for actual latency tolerance requirements. Here the framework selects many long p-threads. In most programs, this is a losing strategy. A greedy approach does yield high numbers of fully covered misses, but the long p-threads produce many useless instances, and much of the bandwidth consumed by others would be better spent on short p-threads that increase total miss coverage. The high overhead sometimes produces sharp slowdowns (*crafty*, *vortex*). Surprisingly, the greedy approach occasionally outperforms the framework (*bzip2*, *mcf*). This occurs when, due to bus contention, effective memory latency is higher than the constant 100 cycles (L_{cm}) we target. The high overhead of the greedy approach is mitigated by the finite number of thread contexts.

In the second experiment, *+lt*, we account for latency tolerance requirements—we select p-threads that are sufficiently long to cover the targeted latency and no longer—but ignore overhead. In line with our reasoning from the previous paragraph, this strategy outperforms the greedy approach, increasing coverage while reducing overhead, except for those benchmarks in which effective memory latency exceeds 100 cycles.

In experiment *+oh* we consider p-thread overhead but not p-thread overlap. We simulate this strategy by running our framework for a single iteration over each slice tree, effectively ignoring overlap corrections. From our results in Sections 4.3 and 4.4, it may appear that pre-execution overhead is insignificant. However, this is misleading: overhead is low only because our framework explicitly minimizes it. Accounting for overhead *a priori* produces higher performance p-threads in all but one benchmark (*bzip2*). This effect is especially prominent in benchmarks which are swamped by the overhead of greedy p-threads (*crafty*, *vortex*).

Experiment *+ovlp* adds p-thread overlap considerations and corresponds to our complete framework. Redundancy due to overlap is a form of overhead and has similar impact. In all benchmarks but *gap*, overhead decreases while performance increases.

4.6 Sensitivity to P-Thread Selection Parameters

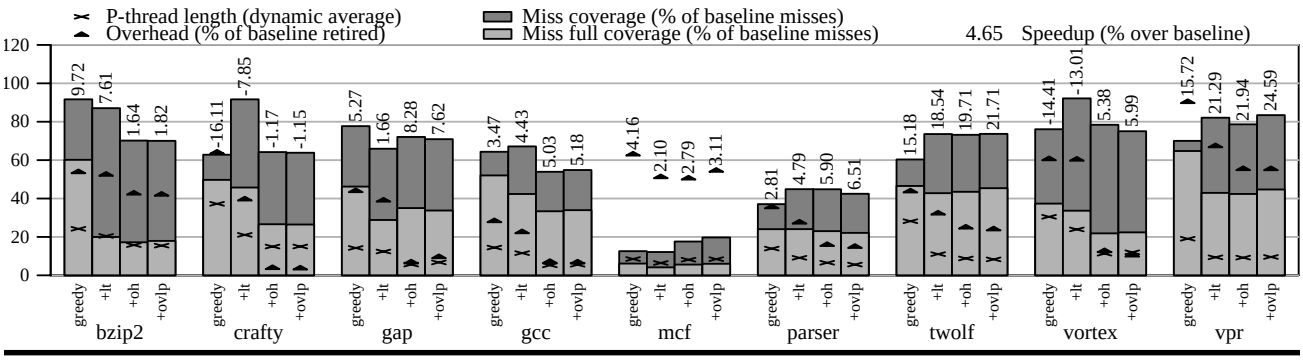
We measure our framework's sensitivity—i.e., changes in the p-threads it selects—to variations in p-thread selection parameters. Figure 5 shows three sensitivity experiments for each of three benchmarks, chosen because they represent a wide range of baseline IPC's: *gcc* (1.72), *twolf* (0.93), and *vortex* (2.74).

Slicing scope and p-thread length. Slicing scope (the length of the dynamic trace examined to construct linear p-threads) and p-thread length are physical constraints on p-thread construction and the implementation of the p-thread memory hierarchy. In the left graph of Figure 5, we begin with a scope length combination of 256 and 8, respectively, and successively relax each constraint up to our default settings of 1024 and 32.

As p-thread selection constraints are relaxed, actual p-thread length, miss coverage, full miss coverage, and performance all increase. However, they do not increase indefinitely; at some point, they saturate and do not benefit from further relaxation. Quantitatively, the saturation point of most programs lies at slicing scopes of 512 instructions and p-thread lengths of 16 instructions. *Gcc*'s performance dips slightly as constraints are relaxed. This anomaly is due to the finite number of p-thread contexts, which our framework does not model.

Parameter sensitivity varies from one program to the

FIGURE 4. Contribution of framework components via comparison with simple approaches



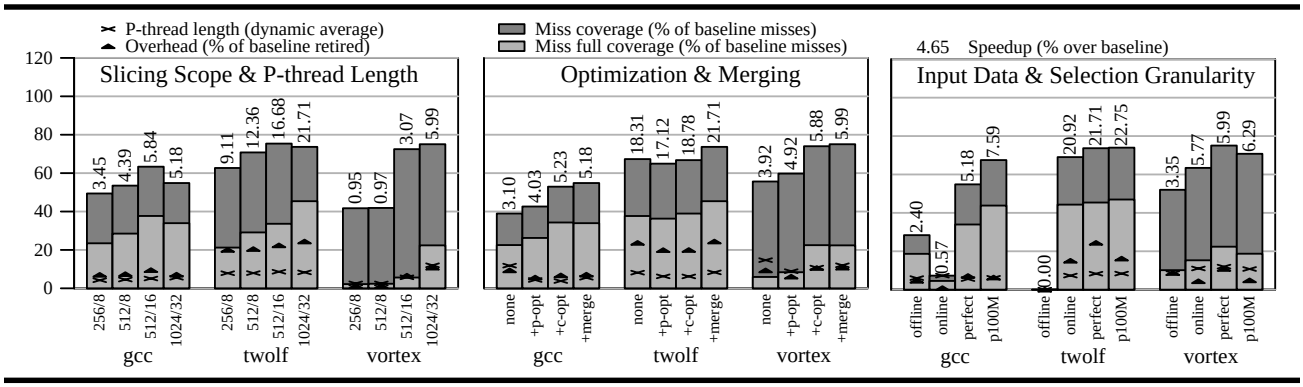


FIGURE 5. Sensitivity to p-thread selection parameters

next. Most programs are more sensitive to p-thread length constraints, unable to achieve any gain with short p-threads, even at large scopes. Miss computations in these programs are dense in the locus leading up to the miss—small computations are unable to obtain any sequencing advantage. *Twolf* is especially sensitive to scope restrictions. It has the complementary program structure: sparse computations which can achieve latency tolerance with small computations, but need large slicing scopes to "see" these computations.

P-Thread optimization and merging. The second graph in Figure 5 isolates the two effects of p-thread optimization and the effect of merging.

The first configuration employs neither optimization nor merging. It produces long, high-overhead p-threads, lower miss coverage—due to optimization's second effect described below—and 2–3% lower performance gains than the full-powered framework.

Experiment *+p-opt* (post-optimization) examines the first effect of optimization. Here we optimize a set of p-threads initially chosen without optimization in mind. Optimization reduces p-thread length and overhead. *Twolf*'s performance suffers slightly under post-optimization due to an interaction with the finite number of thread contexts; this is rare behavior. Experiment *+c-opt* (choose-optimization) isolates optimization's second effect. Here we account for optimized length when choosing p-threads to begin with, enabling us to choose p-threads that were either unprofitable or illegal (i.e., too long) in their unoptimized forms. As shown in *vortex* and *gcc* this "secondary" effect is often stronger than the primary effect of overhead reduction for existing p-threads. Experience also shows that optimization is increasingly effective as length constraints are tightened.

Experiment *+merge* adds merging to the selection process and corresponds to our full framework. Merging reduces overhead but does not increase the number of viable candidates—its performance effects are less pronounced than those of optimization.

Input data set. We have shown that good p-thread selection is possible given *perfect* information: a trace of program execution on the same input data. We now test the viability of p-thread selection in real world scenarios.

To model a profile-driven static compiler implementation of p-thread selection, we select p-threads *offline* using profile data from different (smaller) inputs. P-threads selected offline often approach the performance of p-threads selected with perfect information, supporting our intuition that p-threads and pre-execution performance potential are a function of program structure. Offline selection does have a new failure mode, however. *Twolf*'s test input data set fits in the L2 cache and offline selection generates no p-threads.

To model a dynamically optimizing virtual machine [1], we select p-threads *online* using a short profiling/selection phase before each pre-execution sample. Predictably, online selection generally outperforms offline selection and approaches the performance of "perfect" p-threads. *Gcc*'s shortfall is due to rapidly changing phases.

The online result actually incorporates two effects: p-thread selection using a different sample from within the same program and *per-sample p-thread specialization*. To isolate the latter, we use perfect information to create specialized p-threads for each 100M instruction sample. Intuition says that this approach will produce higher performance. After all, at the limit of this process we find a custom p-thread for every dynamic L2 miss! Experiment *p100M* confirms this intuition.

4.7 Sensitivity to Machine Parameters

An important aspect of the framework is its ability to parameterize major processor features. We use *cross-validation* to measure this fidelity. Intuition says that feature X is modeled correctly if for configurations X1 and X2, p-threads chosen for configuration X1 perform best on configuration X1 *and* those chosen for X2 perform best on X2. If p-threads chosen for X1 perform best on both configurations, then the framework has a bias towards X1.

Figure 6 shows cross-validation studies for three parameters. A single study includes four experiments, pX1cX1—p-threads chosen for configuration X1 executed on configuration X1—pX2cX1, pX1cX2, and pX2cX2. Experiments are grouped by simulated configuration (cX); the left bar in a group is the "cross" experiment. While "cross" experiments validate the model, comparing "self" experiments—pX1cX1 with pX2cX2—provides insights into the impact of parameter variations on pre-execution.

Memory Latency. Memory latency impacts p-thread

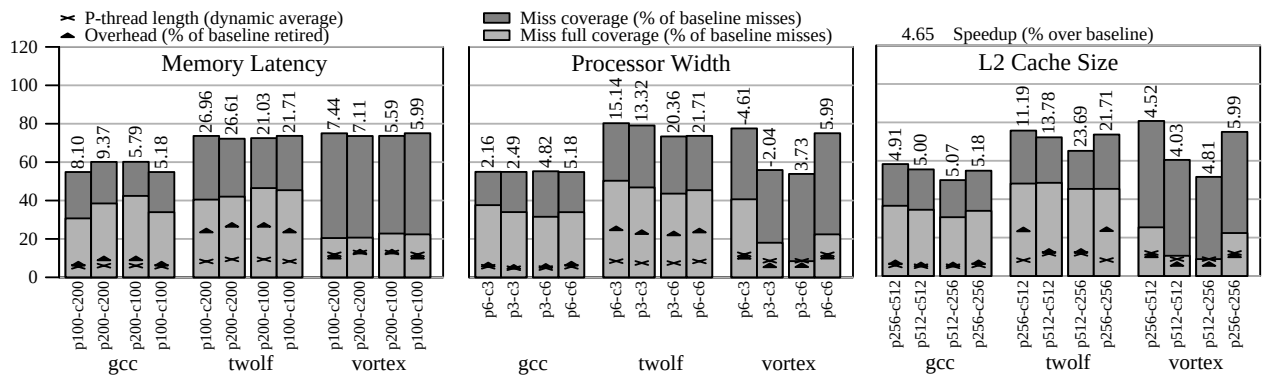


FIGURE 6. Sensitivity to machine parameters

structure directly by changing latency tolerance requirements (L_{cm}). In Figure 6, we measure the effect of latency variations using latencies of 100 and 200 cycles.

Comparing *p100-c100* with *p200-c200* shows that memory latency changes produce intuitive p-thread changes. Increased latency demands longer p-threads which exact more overhead. Predictably, miss coverage remains the same but fewer misses are fully covered. As memory latency accounts for a larger fraction of total execution time, relative performance improvement due to pre-execution increases.

Cross-validation shows that our framework does not model memory latency very accurately; one may obtain better performance by specifying both higher-than-actual (*gcc*) and lower-than-actual (*twolf, vortex*) latency values. We have seen this problem before. Our framework uses a constant value for memory latency (L_{cm}), which is set to the uncontested L2-memory round trip time. The *effective* memory latency is often higher or lower than this value, due to bus contention or natural overlapping, respectively, and typically varies from the misses of one static load to the next and across the dynamic misses of a single static load. By specifying higher or lower latencies, we may be helping the framework simulate true conditions.

Processor width. Processor width ($BWSEQ_{proc}$) effects p-thread selection by controlling the sequencing rate of the main thread ($BWSEQ_{mt}$) and the overhead discount ($BWSEQ_{mt}/BWSEQ_{proc}$). We validate our modeling of this parameter using a 3-wide processor in addition to our default 6-wide machine.

Comparing *p3-c3* with *p6-c6*, we see that our framework responds to processor width changes predictably. A wider processor demands longer p-threads; it allows the main thread to sequence faster—a p-thread must skip more instructions to obtain the same absolute sequencing advantage—and is more overhead tolerant.

From this small sample, the framework appears to model processor width correctly. Lying to it about processor width generally produces poorer results. One exception is *twolf*, where our framework seems to be biased towards width over-specification. *Twolf* has poor branch prediction accuracy. $BWSEQ_{mt}$ overestimates the cost of overhead because nearly half of the instructions sequenced by the main thread are wrong path instructions.

L2 cache size. L2 cache size affects p-thread selection by changing the number of misses. The number of misses can change the effective computation latency when misses are serialized (i.e., pointer-chasing). We select and cross-select p-threads for a 512KB L2.

Larger caches produce fewer misses and require p-threads that produce fewer useless instances; such p-threads tend to be shorter. They also typically require fewer static p-threads as some portions of the working set fit in the L2. By exposing less total memory latency, they reduce the relative impact of pre-execution.

Modeling fidelity results for L2 cache size are mixed: the model appears to be correct in *gcc*, biased towards larger caches in *twolf*, and biased towards smaller caches in *vortex*. Because number of cache misses does not directly enter into the framework as a parameter, the effect of cache size manifests only via interactions. We have already seen that *twolf* is more overhead tolerant than its $BWSEQ_{mt}$ value indicates. It would better tolerate the higher number of p-threads and the longer p-threads it requires due to heavy use of pointer data structures. In contrast, *Vortex*'s branches are predicted with 99% accuracy; it is less overhead tolerant than $BWSEQ_{mt}$ indicates.

5 Related Work

Techniques for hoisting cache misses have been studied for as long as caches have existed. Several have used data-driven instruction execution as a means for generating prefetch addresses [13]. An early proposal to accelerate a single sequential program via decoupled thread-based prefetching was Assisted Execution [17]. Implementations of pre-execution in its current form include Speculative Data-Driven Multithreading (DDMT) [15], Speculative Pre-Computation [4, 5], Speculative Slices [19], Software Controlled Pre-Execution [10], Slice Processors [11], and "push" prefetching by pre-execution in in-memory processors [16, 18].

Our framework complements this work. By parameterizing pre-execution's run-time model, we can apply our results to all of these implementations, whether p-threads execute on dedicated resources [5, 4, 11, 16, 18] or in a shared resource environment [10, 15, 19] and whether

their execution is architecturally visible [10, 16, 18, 19] or not [4, 5, 11, 15]. Our results directly apply to implementations that use static p-threads [10, 15, 16, 18, 19]. Applicability to dynamic p-thread selection systems is more tenuous [4, 5, 11]. Such systems do not explicitly target aggregate effects but rather continuously modify p-threads via feedback (e.g., Speculative Precomputation [4] adds levels of induction unrolling if more latency tolerance is needed). Our framework assumes control-less p-threads and no chaining, a model used by several systems [4, 11, 15]. Chaining [5] and control flow [10, 16, 18, 19] complicate selection, but framework extensions to handle them may be possible.

Compiler and linker based p-thread generators [8, 9, 10] form an implementation path for pre-execution. However, they must reconstruct the dynamic effects p-threads from static program constructs, a non-trivial exercise for anything except for simple loops. By dealing with traces, our framework sees the dynamic instruction stream as straightline code (in which loops are unrolled) and sidesteps this problem. We hope to combine the analysis of our framework with the practical aspects of these systems.

Pre-execution has also been used to target difficult branches [2, 3, 6, 14, 15, 19]. We have applied the methods we present here to branch pre-execution [12] by setting desired latency tolerance to the pipeline-induced branch resolution latency. The framework is oblivious to the implementation and performance of the branch outcome communication mechanism.

6 Conclusions

Memory latency is a significant component of total execution time for integer programs. With multithreading becoming prevalent, pre-execution—a recently proposed technique for effectively moving cache miss latency to other threads—is becoming popular. We present a quantitative framework for selecting static p-threads and reasoning about the performance potential of pre-execution. Our framework contains two novel components. *Aggregate advantage* combines the important p-thread selection criteria—latency tolerance per miss, overhead, and ratio of p-threads launched to misses covered—into a single numerical value, allowing these often antagonistic considerations to be simultaneously optimized. The *slice tree* is a data structure that naturally represents the set of all possible candidate p-threads and the overlap relationships between them, allowing non-redundant solutions comprising multiple p-threads to be found. The framework is built from first principles. A few parameters allow it to model most processor configurations.

We use our framework to find static p-threads covering L2 misses in the SPEC2000 benchmarks. We measure pre-execution performance under different p-thread selection and processor conditions and evaluate the framework itself by checking its predictions against simulated measurements and by observing that it qualitatively responds to parameter variations as an optimization framework would.

An important direction for future work is the tuning of the framework's performance model, especially its estimates of effective memory latency. One possibility is to augment the framework with a critical path [7] pre-processor that can assign a "true cost" to each miss or an average cost to each static problem load. Alternatively, we could enrich the framework with temporal information to allow it to reason about miss overlapping internally.

Acknowledgments

This work was supported by NSF grants CCR-9900584 and EIA-0071924 and an Intel Graduate Fellowship. We thank Yale Patt and the referees for their valuable comments.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. "Dyanmo: A transparent dynamic optimization system." *PLDI-2000*, Jun. 2000.

[2] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. "Simultaneous Subordinate Microthreading (SSMT)." *ISCA-26*, May 1999.

[3] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. "Difficult Path Branch Prediction using Subordinate Microthreads." *ISCA-29*, May 2002.

[4] J. Collins, D. Tullsen, H. Wang, and J. Shen. "Dynamic Speculative Precomputation." *MICRO-34*, Dec. 2001.

[5] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. "Speculative Pre-Computation: Long Range Prefetching of Delinquent Loads." *ISCA-28*, Jul. 2001.

[6] A. Farcy, O. Temam, R. Espasa, and T. Juan. "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes." *MICRO-31*, Dec. 1998.

[7] B. Fields, S. Rubin, and R. Bodik. "Focusing Processor Policies via Critical Path Prediction." *ISCA-27*, Jul. 2001.

[8] D. Kim and D. Yeung. "Design and Evaluation of Compiler Algorithms for Pre-Execution." *ASPLOS-10*, Oct. 2002.

[9] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. "Post-Pass Binary Adaptation for Software-Based Speculative Pre-Computation." *PLDI-2002*, Jun. 2002.

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

[11] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. "Slice Processors: An Implementation of Operation-Prediction." *ICS-15*, Jun. 2001.

[12] A. Roth. *Pre-Execution via Speculative Data Driven Multithreading*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Aug. 2001.

[13] A. Roth, A. Moshovos, and G. Sohi. "Dependence Based Prefetching for Linked Data Structures." *ASPLOS-8*, Oct. 1998.

[14] A. Roth, A. Moshovos, and G. Sohi. "Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation." *ICS-13*, Jun. 1999.

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

[16] Y. Solihin, J. Lee, and J. Torrellas. "Using a User Level Memory Thread for Correlation Prefetching." *ISCA-29*, May 2002.

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

[18] C.-L. Yang and A. Lebeck. "Push vs. Pull." *ICS-14*, May 2000.

[19] C. Zilles and G. Sohi. "Execution Based Prediction Using Speculative Slices." *ISCA-28*, Jul. 2001.