University of Pennsylvania, Department of Computer and Information Science Technical Report MS-CIS-02-23
available at http://www.cis.upenn.edu/~amir/pubs/tr/tselect-tr2002.pdf

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 conventional address-prediction driven prefetching is ineffective. 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 on p-threads and when they should be launched such that total execution time is minimized. P-thread selection 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 new data structure that compactly represents the space of all possible static p-threads.* ***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. Our framework finds the set of p-threads whose aggregate advantages sum to a maximum. The framework is simple and intuitively parameterized to model the salient microarchitecture features.*

We apply our framework to the task of choosing p-threads that cover L2 cache misses. Using detailed simulation, we study the effectiveness of our framework, and pre-execution in general, under different conditions. We measure the effect of constraining p-thread length, of adding localized optimization to p-threads, and of using various program samples as a statistical basis for the p-thread selection, and show that our framework responds to these changes in an intuitive way. In the microarchitecture dimension, we measure the effect of varying memory latency and processor width and observe that our framework adapts well to these changes. Each experiment includes a ***validation*** *component which checks that the formal model presented to our framework correctly represents actual execution.*

1 Introduction

Second-level cache misses constrain processor performance and will constrain it further 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 many prior instructions. Prefetching eliminates many misses. However, certain static *problem loads* defy address prediction and their misses elude prefetching.

Pre-execution is a recently proposed technique for dealing with problem loads[1]. 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*[2]—in a multithreaded processor. “Hoisting” is accomplished as the p-thread fetches and executes many fewer instructions than the main program thread and thus arrives at and initiates the cache miss first. The multithreaded execution model, in which p-threads are decoupled from the main program and

1. Pre-execution has also been proposed as a way of dealing with problem (i.e., frequently mis-predicted) branches. While we do not explicitly discuss branch pre-execution here, all of our methods do apply in that scenario.
2. These have been alternately called data-driven threads, p-threads and p-slices. The term *p-threads* is the “average” of the three.

one another, has many advantages. P-thread execution and cache miss initiation 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 thread continues fetching, executing and retiring instructions from the main program. With hardware multithreading becoming prevalent, pre-execution is gaining popularity [3, 8, 11, 14, 20].

The benefits and limitations of pre-execution have been well-documented. Here, we attack the problem of *p-thread selection*, 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 cache miss latency tolerance, p-thread resource consumption (important when p-threads share resources with the main thread), and prefetch coverage and accuracy. To date, p-thread selection has been approached both manually [20] and automatically [2, 3, 5, 7, 11] and with promising results. However, past approaches have been generally heuristic. We present a framework for attacking the problem in a formal, quantitative, and holistic fashion.

We focus on *static p-threads,* copies of which are launched repeatedly during program execution. The dynamic program intervals for which p-threads are chosen can be short, modeling on-the-fly p-thread generation, or a full run, modeling an off-line implementation. For each program sample, we select p-threads using what is effectively an analytical pre-execution limit study. First, we use an execution trace to enumerate all possible static p-threads. Then, we apply a simple model called *aggregate advantage* to calculate the performance benefit of each static p-thread aggregated over its dynamic invocations. Finally, we "solve" the selection problem by choosing the set of static p-threads that maximizes total performance benefit. Two novel components make this approach feasible. The first is *aggregate advantage*, which uses a few key abstractions to effectively model the microscopic interactions of a p-thread with the main thread using only a few intuitive high level parameters. The second is the *slice tree*, a data structure that compactly represents the space of all possible static p-threads and the relationships between them. The slice tree allows us to accurately assess miss coverage and to ensure that pre-execution work is not replicated. The framework also includes facilities for optimizing p-threads. Constructed from first principles, the framework is simple and, via a few intuitive parameters, applicable to a wide range of pre-execution implementations and processor configurations. In this work, we assume a simultaneous multithreading (SMT) [17] processor, where resources are shared among all threads. The framework, however, is easily adapted to other multithreaded models.

At first glance, the use of exhaustive analysis on dynamic execution traces seems impractical: the trace-driven approach meshes well with dynamic optimization while exhaustive search seems a better fit for off-line implementations. However, representative execution samples can be obtained for off-line analysis or reconstructed from profiles and the structure of the problem allows us to perform our exhaustive search using a simple iterative procedure that converges quickly. Independently, the framework has intrinsic value in that the p-threads it finds are optimal insofar as aggregate advantage accurately models pre-execution. The conditional optimality statement derives from the standard iterative techniques we use to solve the problem. To remove the condition, we use correlation and cross-validation methodologies to measure the fidelity of aggregate advantage. Our results show that, although simple, this formula is quite accurate under many conditions. While perhaps not always optimal in reality, the p-threads produced by our framework are often close to it. Thus, our framework provides a robust analytical foundation for future p-thread selection algorithms. In addition, it allows us to characterize p-threads and evaluate the performance potential of pre-execution under different processor and pre-execution configurations and conditions. In this paper, we do exactly that in the context of L2 misses. Our experiments confirm an intuitive result—maximum pre-execution effectiveness and the p-threads required to achieve it are a function of program structure and processor configuration. As we remove constraints, our framework naturally gravitates to this canonical set of p-threads.

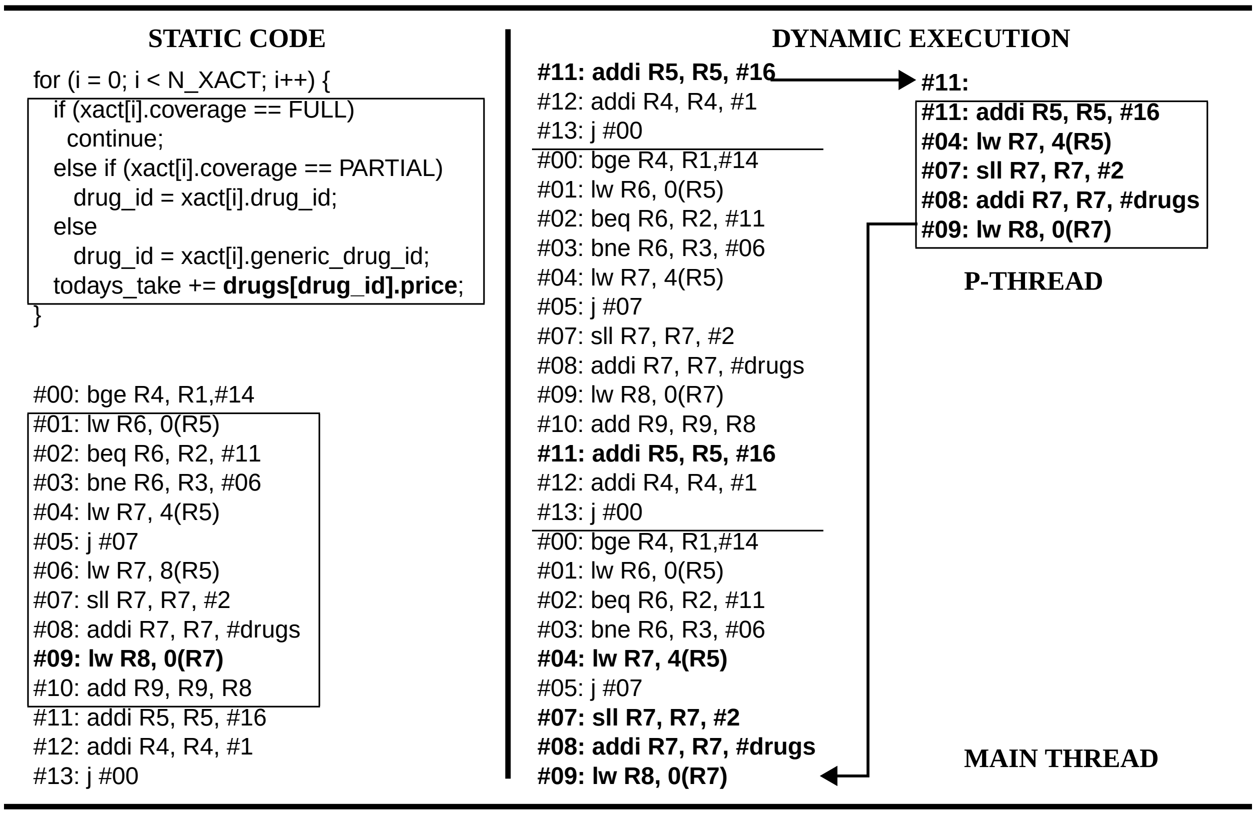
The next section provides background and describes the important aspects of the problem. Section 3 describes the framework in detail. The final three sections contain an experimental evaluation, related work, and our conclusions.

2 Background

We review pre-execution and introduce the p-thread selection problem using an example. Figure 1 shows a loop executed by a mythical pharmacy cash register. The loop iterates over the day's transactions and sums the appropriate prices for the purchased drugs. Load #09 (in bold), which accesses the price of the drug, is a static problem load. Its cache misses cannot be handled via conventional prefetching—their addresses do *not* form an arithmetic series—and must be attacked via pre-execution. The left side of the figure shows the static code. The right shows an p-thread-assisted dynamic execution—the main thread is on the left with loop iterations separated by horizontal lines, the p-thread is to the right. As a running example throughout the paper, we construct this p-thread from the ground up.

Abstract pre-execution model. Selecting proper p-threads requires an understanding of p-thread execution. A p-thread has two components: the *body* is a list of instructions that mimics 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. On the right side of the figure, the p-thread body is shown in a box with the trigger as a annotation on top. The p-thread and the main thread computation it mimics are in bold. Although not shown, a dynamic p-thread is launched by every main thread

FIGURE 1. Pre-Execution Example



```
for (i = 0; i < N_XACT; i++) {
  if (xact[i].coverage == FULL)
    continue;
  else if (xact[i].coverage == PARTIAL)
     drug_id = xact[i].drug_id;
  else
     drug_id = xact[i].generic_drug_id;
  todays_take += drugs[drug_id].price;
}

#00: bge R4, R1,#14
#01: lw R6, 0(R5)
#02: beq R6, R2, #11
#03: bne R6, R3, #06
#04: lw R7, 4(R5)
#05: j #07
#06: lw R7, 8(R5)
#07: sll R7, R7, #2
#08: addi R7, R7, #drugs
#09: lw R8, 0(R7)
#10: add R9, R9, R8
#11: addi R5, R5, #16
#12: addi R4, R4, #1
#13: j #00
```

```
#11: addi R5, R5, #16
#12: addi R4, R4, #1
#13: j #00
#00: bge R4, R1,#14
#01: lw R6, 0(R5)
#02: beq R6, R2, #11
#03: bne R6, R3, #06
#04: lw R7, 4(R5)
#05: j #07
#07: sll R7, R7, #2
#08: addi R7, R7, #drugs
#09: lw R8, 0(R7)
#10: add R9, R9, R8
#11: addi R5, R5, #16
#12: addi R4, R4, #1
#13: j #00
#00: bge R4, R1,#14
#01: lw R6, 0(R5)
#02: beq R6, R2, #11
#03: bne R6, R3, #06
#04: lw R7, 4(R5)
#05: j #07
#07: sll R7, R7, #2
#08: addi R7, R7, #drugs
#09: lw R8, 0(R7)
```

```
#11:
#11: addi R5, R5, #16
#04: lw R7, 4(R5)
#07: sll R7, R7, #2
#08: addi R7, R7, #drugs
#09: lw R8, 0(R7)
```

instance of instruction #11.

As shown in the figure, a p-thread's trigger and body are closely related: the body corresponds to the miss computation starting from the trigger. This relationship forms the basis of the *abstract pre-execution model*. 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 fewer instructions, i.e., only the cache miss computation as opposed to the full program. A different view of the trigger/body relationship allows us to *automate* p-thread selection: the body and trigger form a *dynamic backwards data-dependence slice* that starts at the problem instruction. For a given problem instruction, we *enumerate all possible static p-threads* by constructing successively longer backward slices. P-thread selection thus becomes a true selection process. From this enumerated set, we select the *static p-threads* whose dynamic instances will tolerate the most miss latency while incurring the least amount of overhead.

Our abstract model makes two assumptions about the p-thread sequencing model. First, p-threads are *control-less*—they are fixed sequences that are executed in their entirety. Second, p-thread launching is *not chained* [3]—only the main thread may launch p-threads (with chaining p-threads can launch other p-threads). These restrictions simplify the pre-execution implementation—allowing only the main thread to launch fixed size p-threads naturally ties p-thread progress to main thread, limiting redundant and runaway pre-execution and reducing early prefetching effects. They also allows us to analyze the benefit of a static p-thread as the aggregate benefit of its dynamic instances—we know exactly what each instance looks like and exactly how many of them there will be. At the same time, they do not excessively constrain the power of pre-execution. The primary use of control [20] and chaining [3] in pre-execution is to implement p-thread loops for increased lookahead and latency tolerance. In our model, where loops are needed, they are *simulated* by including multiple copies of the induction in a p-thread, an idiom called *induction unrolling* [2, 14]. Our example p-thread uses a copy of instruction #11 to effectively skip one loop iteration ahead.

Our framework uses the abstract pre-execution model—the parallel execution of an isolated computation and the dynamic main thread region which contains it—in its calculations. It ignores the "constant-overhead" mechanical details of pre-execution (e.g., the mechanism which initializes p-threads with seed values from the main thread) and incorporates the important aspects (e.g., how much sequencing bandwidth p-threads are allocated) as parameters.

Aspects of p-thread selection. Since p-threads are backwards slices of problem loads, the only thing we can vary in a p-thread is its length. While choosing a proper p-thread length may sound straightforward, it can be quite subtle. Obviously, a longer p-thread is launched earlier with respect to is target cache miss and will typically tolerate more latency. However, it also execute more instructions and consume more resources. In fact, if it executes too many instructions less latency will be tolerated. That is not all. A given static p-thread will launch a certain number of *useless dynamic instances*. There are two kind of useless p-threads: the first pre-executes loads that would have been cache hits anyway, the second pre-executes no main thread load at all (i.e., the main thread executes along a different path than the one the p-thread 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 dynamic misses*. A given instance of load #09 may be arrived at via two different computations: one containing #04 (drug_id=xact[i].drug_id), the other containing #06 (drug_id=xact[i].generic_drug_id). A longer p-thread that contains this portion of the computation will target only the subset of misses that exercise the corresponding instruction. To fully cover all #09 misses *potentially* requires two static p-threads. Our framework simultaneously examines all of these considerations and makes trade-offs between them *quantitatively*.

3 P-Thread Selection Framework

We now construct our framework, using first principles of pre-execution as guides. We first describe *aggregate advantage (ADV_{agg})*, 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 introduce the *slice tree* and show how it enables the simultaneous selection of multiple p-threads from multiple, partially overlapping computations. Finally, we describe two enhancements to the basic framework: p-thread merging and p-thread optimization.

3.1 Aggregate Advantage: Quantifying the Performance Impact of a Single Static P-Thread

Obtaining a backward data-dependence slice of a single load is straightforward, even in hardware [2, 11]. A slice comprising *N* instructions presents us with a choice of *N* possible p-threads that increase in length from 1 to *N* instructions. The basic p-thread selection problem is to choose the *slice suffix* that makes the best p-thread, allowing for the possibility that no sub-slice makes an acceptable p-thread.

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 considerations into a single numerical score, allowing them to be simultaneously optimized. The *advantage* (ADV) of a dynamic p-thread instance is the estimated number of cycles by which it accelerates program execution; the aggregate advantage of a static p-thread is the sum of the advantages of all of its dynamic instances. Advantage is the difference of two terms: *latency tolerance (LT)* is the number of cycles by which the p-thread accelerates its targeted cache miss, and *overhead (OH)* is the number of cycles by which the p-thread slows down the main thread by stealing resources from it. When computing ADV_{agg}, it is convenient to aggregate latency tolerance (LT_{agg}) and overhead (OH_{agg}) separately. *Every* dynamic p-thread exacts overhead on the main thread, but only dynamic p-threads that pre-execute actual dynamic cache miss computations achieve any latency tolerance. Useless p-threads have no latency tolerance associated with them because their associated main thread loads have no latency (they either hit in the cache or do not exist). If DC_{trig} is the dynamic count of triggers in the program (i.e., the number of times a p-thread is launched) and $DC_{pt\text{-}cm}$ is the number of times a given launched p-thread actually pre-executes a main thread miss, then:

$$ADV_{agg} = LT_{agg} - OH_{agg} \qquad \text{EQ 1.}$$

$$OH_{agg} = DC_{trig} * OH \qquad \text{EQ 2.}$$

$$LT_{agg} = DC_{pt\text{-}cm} * LT \qquad \text{EQ 3.}$$

where OH and LT are the overhead and latency tolerance for a single dynamic p-thread instance, respectively.

Overhead per dynamic p-thread (OH). Given our assumption of SMT execution, the number of sequencing cycles stolen from the main thread is the most direct way of measuring overhead. All other forms of contention are either subsumed by this measure (e.g., execution bandwidth), not easily estimated (e.g., bus bandwidth), or both (e.g., buffering resources). The number of cycles it takes to sequence a p-thread is the number of instructions in the p-thread

($SIZE_{pt}$) divided by the sequencing width of the processor (BW_{seq}). Since overhead is opportunity cost, we discount overhead for a given p-thread by the expected main thread sequencing utilization ($BW_{seq\text{-}mt}$ / BW_{seq}). For instance, if on an 8-wide processor the main thread fetches 4 instructions per cycle, then a p-thread is only penalized for half of its bandwidth consumption. One in two cycles it uses would not have been used by the main thread anyway.

$$OH = (SIZE_{pt} / BW_{seq}) * (BW_{seq\text{-}mt} / BW_{seq}) \qquad \text{EQ 4.}$$

Latency tolerance per useful dynamic p-thread (LT). P-thread latency tolerance (LT) is quantified as a difference in execution times of the p-thread and the main thread. Our execution time estimation metric is *sequencing-constrained dataflow-height (SCDH)*, a function that models both data-dependences and limited sequencing bandwidth. Starting from the trigger instruction (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 and the number of cycles it takes an *unassisted main thread* to do the same. The difference between these two estimates, $SCDH_{mt}$ - $SCDH_{pt}$, 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 on its behalf. Since it does not benefit the main thread to tolerate more latency than the latency of the miss, we bound LT by the original miss latency (L_{cm}).

$$LT = MIN(SCDH_{mt} - SCDH_{pt}, L_{cm}) \qquad \text{EQ 5.}$$

The recursive equations for SCDH are those for standard dataflow-height, except that the input height at a given instruction also models a *sequencing constraint (SC)*—the cycle at which the instruction is sequenced (fetched).

$$SCDH_{in} = MAX(SC, MAX_{dataflow\text{-}predecessors}(SCDH_{out})) \qquad \text{EQ 6.}$$

$$SCDH_{out} = SCDH_{in} + L_{exec} \qquad \text{EQ 7.}$$

To calculate SC for a given instruction, we divide the instruction's trigger distance ($DIST_{trig}$)—its distance in dynamic instructions from the trigger—by the available sequencing bandwidth ($BW_{seq\text{-}mt}$ for the main thread, $BW_{seq\text{-}pt}$ for the p-thread). $SCDH_{pt}$ is smaller than $SCDH_{mt}$ because of SC: the p-thread has fewer instructions to sequence through, so each p-thread instruction has a smaller $DIST_{trig}$ than its main thread counterpart. Now let us define the values used for $BW_{seq\text{-}mt}$ and $BW_{seq\text{-}pt}$. $BW_{seq\text{-}mt}$ is the rate at which the main thread *actually* sequences. To account for main thread speculative execution, we heuristically calculate $BW_{seq\text{-}mt}$ as the average of the unassisted main thread IPC and the sequencing width of the processor (BW_{seq}), weighted 2-to-1 in favor of the IPC. $BW_{seq\text{-}pt}$ is the rate at which a p-thread is *allowed* to sequence. We set $BW_{seq\text{-}pt}$ 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.

Working Example. To illustrate the working of ADV_{agg}, we select a p-thread for one particular dynamic computation of load #09 from our example, the one that contains instruction #04. We make the following assumptions. The loop executes 100 iterations. The first branch is taken 20 times such that only 80 iterations contain instances of #09. The second branch is taken 60 times, thus of the 80 iterations that contain instances of #09, 60 use the computation that includes #04, the remaining 20 use #06. Half of all #09 instances result in misses (there are 40 #09 misses). All operations have unit latency and cache miss latency is 8 cycles. Note, the highest possible ADV_{agg} score in this case is 320: 8 cycles of latency tolerance for each of the 40 #09 misses, with 0 overhead. This score is impossible to achieve if p-threads have non-zero cost. The processor is 4 wide, and the unassisted IPC of the loop is 1 ($BW_{seq\text{-}mt}$ is 2). Finally, our slicing mechanism examines 40 instructions and limits p-threads to fewer than 8 instructions, these constraints result in a slice with 6 instructions (plus the trigger) implying there are six p-thread candidates.

Figure 2 shows the ADV_{agg} calculation for each of the six candidate p-threads. The calculation for the winning p-thread is shaded. Each calculation is represented by two tables. The table on the left shows the SCDH calculations. In the p-thread, trigger distances ($DIST_{trig}$) are sequential and the sequencing constraint (SC) is obtained by dividing the

FIGURE 2. Working Example: using aggregate advantage a single static p-thread.

p-thread candidate #1			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
B	#08	addi R7, R7, #accts	0	0	0/1	0	0	0/1
A	#09	lw R8, 0(R7)	1	1	1/9	1	1	1/9

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
40	0	0	0
DC_{trig}	SIZE	OH	OH_{agg}
80	1	0.125	10
		ADV_{agg}	**-10**

p-thread candidate #2			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
C	#07	sll R7, R7, #2	0	0	0/1	0	0	0/1
B	#09	addi R7, R7, #accts	1	1	1/2	1	1	1/2
A	#09	lw R8, 0(R7)	2	1	2/10	2	2	2/10

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
40	0	0	0
DC_{trig}	SIZE	OH	OH_{agg}
80	2	0.25	20
		ADV_{agg}	**-20**

p-thread candidate #3			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
D	#04	lw R7, 4(R5)	0	0	0/1	0	0	0/1
C	#07	sll R7, R7, #2	3	2	2/3	1	1	1/2
B	#08	addi R7, R7, #accts	4	2	3/4	2	2	2/3
A	#09	lw R8, 0(R7)	5	3	4/12	3	3	3/11

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
30	1	1	30
DC_{trig}	SIZE	OH	OH_{agg}
60	3	0.375	23
		ADV_{agg}	**7**

p-thread candidate #4			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
E	#11	addi R5, R5, #16	0	0	0/1	0	0	0/1
D	#04	lw R7, 4(R5)	7	4	4/5	1	1	1/2
C	#07	sll R7, R7, #2	10	5	5/6	2	2	2/3
B	#08	addi R7, R7, #accts	11	6	6/7	3	3	3/4
A	#09	lw R8, 0(R7)	12	6	7/15	4	4	4/12

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
30	3	3	90
DC_{trig}	SIZE	OH	OH_{agg}
100	4	0.5	50
		ADV_{agg}	**40**

p-thread candidate #5			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
F	#11	addi R5, R5, #16	0	0	0/1	0	0	0/1
E	#11	addi R5, R5, #16	12	6	6/7	1	1	1/2
D	#04	lw R7, 4(R5)	19	10	10/11	2	2	2/3
C	#07	sll R7, R7, #2	22	11	11/12	3	3	3/4
B	#08	addi R7, R7, #accts	23	12	12/13	4	4	4/5
A	#09	lw R8, 0(R7)	24	12	13/21	5	5	5/13

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
30	8	8	240
DC_{trig}	SIZE	OH	OH_{agg}
100	5	0.625	63
		ADV_{agg}	**177**

p-thread candidate #6			main thread			p-thread		
			$DIST_{trig}$	SC	$SCDH_{in/out}$	$DIST_{trig}$	SC	$SCDH_{in/out}$
G	#11	addi R5, R5, #16	0	0	0/1	0	0	0/1
F	#11	addi R5, R5, #16	6	3	3/4	1	1	1/2
E	#11	addi R5, R5, #16	18	9	9/10	2	2	2/3
D	#04	lw R7, 4(R5)	25	13	13/14	3	3	3/4
C	#07	sll R7, R7, #2	28	14	14/15	4	4	4/5
B	#08	addi R7, R7, #accts	29	15	15/16	5	5	5/6
A	#09	lw R8, 0(R7)	30	15	16/24	6	6	6/14

$DC_{pt\text{-}cm}$	$SCDH_{diff}$	LT	LT_{agg}
30	10	8	240
DC_{trig}	SIZE	OH	OH_{agg}
100	6	0.75	75
		ADV_{agg}	**165**

trigger distance by 1 ($BW_{seq\text{-}pt}$). In the main thread, trigger distances are sparse and SC is obtained by dividing the $DIST_{trig}$ by 2 ($BW_{seq\text{-}mt}$). $SCDH_{pt}$ and $SCDH_{mt}$ are $SCDH_{out}$ of the load #09 instance. The table on the right shows the LT_{agg}, OH_{agg}, and ADV_{agg} calculations. In the LT_{agg} calculation, $SCDH_{diff}$ is $SCDH_{mt}$ - $SCDH_{pt}$, and latency tolerance (LT) is the minimum of $SCDH_{diff}$ and L_{cm} (8 in our example) per Equation 5. LT_{agg} is the product of $DC_{pt\text{-}cm}$ and LT per Equation 3. In the OH_{agg} calculation, SIZE is the number of instructions in the p-thread, OH is SIZE multiplied by 0.125 (using Equation 4 and plugging the value 4 for BW_{seq} and 2 for $BW_{seq\text{-}mt}$), and OH_{agg} is OH times DC_{trig} per Equation 2.

Neither of the first two candidates provides a fetch advantage (latency tolerance) over the main thread—starting at the trigger, the main thread and the p-thread fetch exactly the same instructions—while both incur overhead (which increases linearly with p-thread size). Pre-executing either will reduce performance. Notice, for both p-threads DC_{trig} is 80 while $DC_{pt\text{-}cm}$ is 40—each p-thread is executed 80 times, but only 40 executions correspond to misses.

Setting #04 (D) as the trigger for the third candidate imparts the p-thread with minimal sequencing advantage over the main thread—the p-thread gets to skip instructions #05 and #06—and 1 cycle of latency tolerance. However, in contrast with the first two candidates, #04 is executed only 60 times (#06 is executed the other 20 times) and the computation triggered by #04 results in only 30 misses (the other 10 #09 misses have instruction #06 in their computation). Unlike the first two, this p-thread has a positive advantage, tolerating 1 cycle of latency for each of 30 misses, and incurring 0.375 cycles of overhead for each of 60 p-threads launched.

The trigger for the fourth candidate is instruction #11 from the previous iteration. Since an instance of #11 occurs once per iteration, DC_{trig} is 100. $DC_{pt\text{-}cm}$ is still 30—the computation includes instruction #04 and correctly pre-executes only 30 misses. The changes in $DC_{pt\text{-}cm}$ and DC_{trig} observed for the last two candidates illustrate two general trends. First, within a given backwards slice, $DC_{pt\text{-}cm}$ monotonically decreases as p-thread length increases. This is an intuitive result: the longer the slice, the fewer dynamic computations it corresponds to. In contrast, DC_{trig} has no direct relationship to p-thread length. Trends aside, the fourth p-thread candidate is better than the third. Although the number of useless p-threads—computed by subtracting $DC_{pt\text{-}cm}$ from DC_{trig}—rises from 30 to 70 and the overhead of each p-thread increases, the additional 2 cycles of latency tolerance achieved for each miss produces a net gain.

Once the induction instruction, #11, is encountered in a slice, further p-thread growth generally comes from the addition of instances of this instruction. This pre-execution idiom is called *induction unrolling* [2, 14] and it generates most of the fetch advantage (lookahead) used by pre-execution to achieve latency tolerance. Each additional level of unrolling provides the latency tolerance of one full loop iteration for the price of one additional instruction. Induction unrolling falls naturally from dynamic backward slicing and is automatically performed to the level dictated by L_{cm}. The final two p-thread candidates are similar—the fifth uses a single level of induction unrolling, the sixth unrolls twice. The first unrolling provides the p-thread with an additional fetch advantage of 12 instructions over the main thread, which translates into 5 additional cycles of execution time advantage ($SCDH_{diff}$) for a total of 8. This is as much latency tolerance as we need. The score achieved for this p-thread is 177: full latency tolerance for 30 misses, at the cost of 63 overhead cycles. Predictably, the final candidate has worse projected performance. With full latency tolerance already achieved, adding another level of unrolling only serves to increase overhead.

3.2 Slice Tree: Selecting Multiple P-Threads Simultaneously

The pre-execution solution for a given program typically involves multiple p-threads. Even the simple problem we tackled in the previous section is not fully solved by a single p-thread. The p-thread we found covers only 30 of the 40 possible misses. In this section we show how to select a *set of p-threads* by solving multiple, partially overlapping sub-problems simultaneously. The specific question we answer is: “given the set of computations for *all* the misses of a given static load, what is the best set of p-threads for pre-executing as many of those misses as possible?”

One approach to any large problem is “divide-and-conquer”. In p-thread selection, the naive divide-and-conquer approach does not work because the underlying assumption that aggregate advantage adds—$ADV_{agg}(A+B) = ADV_{agg}(A) + ADV_{agg}(B)$—does not always hold. If two 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 has tolerated the latency of a miss, a second p-thread cannot tolerate it again. To ensure that p-threads across individual sub-problems do not overlap, we divide the p-thread selection problem for an entire program into sub-problems each of which handles the misses of a different static load. To solve each sub-problem, we use a new data structure—the *slice tree*—that naturally and precisely represents p-thread overlap.

Slice tree. The slice tree is a tree of static backward slices with the static load at the root. Each instruction node in a slice tree represents a static p-thread whose trigger is that instruction. The p-thread is constructed by walking from the node to the root. Figure 3 shows the slice tree representing both slices from our example. To save space, we represent linear tree regions as tables. The figure shows two partially overlapping slices targeting the misses of instruction #09 which is at the root of the tree (node A). The slice formed by the tables along the left path (nodes A-G) is the one we optimized in the previous section. The slice formed by the tables along the right path (nodes A-C,H-K) represents the “other” computation, the one that contains instruction #06 rather than instruction #04. The slices triggered by B and C are shared suffixes of the larger slices. When discussing p-threads in a slice tree, we talk about *parent-child relationships*. Given a parent p-thread, a child p-thread is constructed by extending the slice by one instruction. In the figure, C is a child p-thread of B, and D and H are children of p-thread C.

Each slice tree node is annotated with information that summarizes the behavior of dynamic instances of the corresponding static trigger *and* static p-thread. $DIST_{pl}$ is a concise representation of the average $DIST_{trig}$ in the main thread context: an instruction’s $DIST_{trig}$ with respect to any trigger is obtained by subtracting its $DIST_{pl}$ from the trigger’s, DC_{trig} and $DC_{pt\text{-}cm}$ were previously defined in Section 3.1. Note, DC_{trig} is a trigger property while $DC_{pt\text{-}cm}$ is a

FIGURE 3. Slice Tree

#	PC	inst	$DIST_{pl}$	$DC_{pt\text{-}cm}$	DC_{trig}
A	#09	lw R8, 0(R7)	0	40	80
B	#08	addi R7,R7,#drugs	1	40	80
C	#07	sll R7, R7, #2	2	40	80

#	PC	inst	$DIST_{pl}$	$DC_{pt\text{-}cm}$	DC_{trig}
D	#04	lw R7, 4(R5)	5	30	60
E	#11	addi R5, R5, #16	12	30	100
F	#11	addi R5, R5, #16	24	30	100
G	#11	addi R5, R5, #16	30	30	100

#	PC	inst	$DIST_{pl}$	$DC_{pt\text{-}cm}$	DC_{trig}
H	#06	lw R7, 4(R5)	3	10	20
I	#11	addi R5, R5, #16	12	10	100
J	#11	addi R5, R5, #16	24	10	100
K	#11	addi R5, R5, #16	30	10	100

p-thread property. For instance, DC_{trig} is the same for nodes E, F, G, I, J, and K as these are all instances of instruction #11. However, $DC_{pt\text{-}cm}$ is different for each one of these nodes as each corresponds to a different static p-thread.

The use of the slice tree to summarize information for a static p-thread across its dynamic executions means that temporal information is lost and that we cannot compute the interactions of p-threads with one another. Our framework calculates the aggregate effects for each static p-thread individually, implicitly assuming that dynamic p-threads do not interact, whether they be instances of the same static p-thread or not. This assumption is not egregious for L2 cache misses which are infrequent enough to make the likelihood of concurrent p-threads low. Either way, we willingly trade this inaccuracy for the computational leverage and statistical support provided by summary information.

P-thread overlap and addition of aggregate advantages. The degree of overlap between a p-thread and its parent is defined by $DC_{pt\text{-}cm}$. Consider the p-threads corresponding to instructions C, D, and H. P-thread C pre-executes all 40 misses, but cannot achieve latency tolerance for any of them. If we want a longer p-thread, one that can tolerate more latency, we are faced with a choice of two. P-thread D and its children (E-G) will pre-execute only 30 of the misses. P-thread H and its children (I-K) will target the other 10. That a parent p-thread's $DC_{pt\text{-}cm}$ is the sum of the $DC_{pt\text{-}cm}$ of its children is an invariant. Also note, a parent-child relationship is the only possible source of overlap between two p-threads. P-thread A can either be longer and more specialized than p-thread B or shorter and less specialized, not both at the same time.

Now that we understand the relationship between different p-threads in a slice, we can explain how to combine the aggregate advantages of two p-threads. If two p-threads are not a parent and child (either directly or indirectly) then their aggregate advantages simply add. If the two are a parent and child, then there is some component of aggregate latency tolerance that is counted in both, and this double counted component must be subtracted from the total. The number of misses attacked by both p-threads is given by the child's $DC_{pt\text{-}cm}$. The amount of latency that is "double-tolerated" for each one of these is LT of the parent. Because the parent thread tolerates less latency per miss, we typically associate advantage reduction with the parent p-thread.

$$ADV_{agg\text{-}reduced}(P) = ADV_{agg}(P) - LT(P) * DC_{pt\text{-}cm}(C) \qquad \text{EQ 8.}$$

where P is the parent and C is the child.

The solution of a composite p-thread selection problem (covering the misses of a single static load) is the set of p-threads whose aggregate advantages—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 using an iterative procedure rather than an exhaustive search. For each leaf (separate linear slice) in the slice 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 parent p-threads and reselect. The process terminates once the reductions performed in one iteration do not influence the p-threads selected in the next iteration.

Working example. Obtaining a complete solution for the slice tree in our example is trivial. Selecting the two p-threads separately we find that the best p-thread along the left hand side of the tree is p-thread F (found in the previous section) and that the best p-thread along the right side of the tree is p-thread J. Since these two p-threads do not overlap, no corrections must be made, and no further iterations are necessary.

3.3 Framework Extensions: Merging and Optimization

In addition to the basic facilities for solving the static p-thread selection problem, our framework contains two enhancements. First, we support merging of partially redundant p-threads to reduce overhead. Second, we support p-thread optimization or specialization. Both of these capabilities are automated.

Merging. The two p-threads chosen in the previous section do not overlap from a cache miss standpoint, each targets a disjoint set of misses. However, many of the instructions dynamically executed by these p-threads—e.g., every instance of instruction #11—are redundant. Rather than execute two separate p-threads, one with instructions #11, #04, #07, #08, #09 and one with instructions #11, #06, #07, #08, and #09, we create a single p-thread with instructions #11, #04, #06, #07, #08, and #09 that captures both computations. A merged p-thread achieves the same latency tolerance as separate instances of each of the original p-threads and incurs less overhead. Our merging algorithm merges p-threads with matching *data-flow prefixes.* A p-thread's data-flow prefix is its trigger instruction plus any contiguous chunk of data-flow sub-graph connected directly to any other instruction external to the p-thread (trigger included). Merging proceeds in dataflow order with register renaming and code duplication performed as needed to preserve the computational semantics of each of the original component p-threads. In our example, instructions #07, #08 and #09 cannot be merged in the final p-thread, they must be replicated: one copy must take the computation that contains instruction #04 to its completion and the other must complete the computation that contains #06.

Optimization. *Optimized p-threads* are p-threads that 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 calculations for SCDH_{pt} and SIZE_{pt} to use *any* sequence of instructions that is functionally equivalent to the actual sub-slice. P-thread optimization is both easier and more productive than full program optimization. First, since p-threads are control-less, traditional control-flow and iterative data-flow analyses are replaced by a simple linear scan. Second, only optimizations that are enabled by the highly specialized nature of the p-thread need be 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 the final candidate, the two instances of instruction #16 (addi R5, R5, #16) may be folded into a single instruction (addi R5, R5, #32). This optimization reduces both p-thread latency (the height of the dataflow graph is cut by one instruction) and overhead.

4 Experimental Evaluation

We evaluate our p-thread selection framework's capacity for selecting p-threads that target L2 misses. In section 4.2, we validate the framework's performance model by comparing predicted statistics against statistics measured from pre-execution simulations. In sections 4.3 and 4.4, we measure the framework's response to variations in several p-thread and machine parameters, respectively.

4.1 Methodology

Our experiments use a suite of tools built using the SimpleScalar Alpha AXP ISA and system call modules. A func-

	bzip2	**crafty**	**gap**	**gcc**	**mcf**	**parser**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
Instructions (M)	6000.00	2600.00	900.00	500.00	900.00	1300.00	1300.00	1700.00	300.00	1100.00
Loads (M)	1509.20	731.49	218.49	116.15	246.03	295.70	291.56	458.21	79.27	327.17
L2 misses (M)	21.05	0.93	1.79	0.93	70.48	5.15	9.64	1.17	0.41	7.69
IPC	**3.13**	**3.16**	**1.85**	**2.09**	**0.29**	**1.44**	**1.18**	**3.63**	**2.08**	**1.30**
Perfect L2 IPC	**5.41**	**3.56**	**2.79**	**2.71**	**1.86**	**2.51**	**2.31**	**4.78**	**2.70**	**2.19**

TABLE 1. Benchmark characterization

tional cache simulator generates program traces and constructs backward slices of all dynamic L2 misses and collects them into slice trees which are written out to files. The p-thread selection tool takes a slice tree file and parameters describing the processor (sequencing width, memory latency), unoptimized program performance (IPC), and p-thread construction constraints (maximum p-thread length, optimization level) and produces a list of static p-threads. This arrangement allows multiple p-thread sets for the same cache configuration but different pipeline, latency and p-thread optimization configurations to be generated quickly. Our default configuration has a maximum slicing scope of 1024 instructions, maximum p-thread length of 32 instructions, and full merging and p-thread optimization.

Performance results are obtained via a detailed timing simulator that models a parametrizable pipeline with register renaming, reservation stations, load speculation, and an event-driven memory hierarchy with realistic bandwidth contention. Our base configuration is an 8-wide dynamically scheduled processor, with a 14 stage pipeline, 80 reservation stations, and a maximum of 128 instructions in-flight. The front-end has a 32KB, 2-way set-associative instruction cache, 32-entry TLB, and 6K-entry hybrid branch predictor with 2K-entry BTB. The data-memory system includes a 16KB, 32B line, 2-way set associative, 2-cycle access, write-back data cache, a 256KB, 64B line, 4-way set-associative, 6-cycle access second-level cache, and a 32-entry TLB. Store-to-load forwarding via a 64-entry store queue also takes 2 cycles, with all memory accesses preceded by 1-cycle address generation. We model an infinite main memory with 70 cycle access latency, a 32B wide backside bus clocked at processor frequency, and a 32B memory bus clocked at one fourth processor frequency. 32 simultaneously outstanding misses are allowed.

The simulator models the run-time functions of pre-execution. A p-thread is launched when the main thread renames the corresponding trigger. The p-thread is allocated to one of three additional thread contexts or dropped if no context is available. P-thread instructions are injected into the execution core at register renaming in a bursty fashion, 8 instructions once every 8 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. A p-thread context is freed when all p-thread instructions have been renamed. Physical registers allocated to p-thread instructions are recycled in a circular fashion and our simulator models an additional 64 physical registers for p-thread use. The simulator does *not* model the p-thread selection/pre-execution interface assuming that p-threads are accessible in one cycle from an ideal p-thread cache that experiences no misses. Our (untested) assumption is that this interface has no first-order performance effects. Because our experiments target L2 misses, we disable the data cache fill path for p-thread loads—p-thread loads prefetch only into the L2. While prefetching into the first level cache improves performance, it perturbs our ability to validate the framework's performance model, an important aspect of this evaluation.

All simulation tools exploit sampling, cycling through off (fast-forwarding), warm-up (caches and branch predictor only) and on (full detail) phases at regular intervals. We have performed experiments (not shown) which confirm that, by both miss rates and IPCs, cyclic sampling is "equivalent" to unsampled execution. Our experiments use the SPEC2000 integer benchmarks compiled at "peak" optimization levels using the Digital Unix cc compiler. Perfor-

		bzip2	**crafty**	**gap**	**gcc**	**mcf**	**parser**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
Base	L2 misses (M)	21.05	0.93	1.79	0.93	70.48	5.15	9.64	1.17	0.41	7.69
	IPC	**3.13**	**3.16**	**1.85**	**2.09**	**0.29**	**1.44**	**1.18**	**3.63**	**2.06**	**1.30**
Pre-exec	P-threads launched (M)	102.19	6.86	5.41	6.74	26.23	25.88	13.75	5.03	4.62	32.25
	Instructions per p-thread	19.49	13.91	9.00	5.84	17.66	4.89	11.09	8.91	9.97	11.97
	Overhead IPC (execute)	2.97	3.15	1.85	2.09	0.29	1.44	1.18	3.63	2.05	1.29
	Overhead IPC (sequence)	2.97	3.16	1.85	2.09	0.29	1.44	1.18	3.63	2.07	1.29
	Misses covered (M)	13.40	0.61	0.81	0.40	8.36	1.46	4.77	0.43	0.34	5.19
	Misses fully covered (M)	4.63	0.31	0.59	0.30	6.54	0.87	4.30	0.13	0.18	3.96
	Latency tolerance IPC	3.54	3.13	1.93	2.22	0.30	1.54	1.38	3.82	2.27	1.63
	IPC	**3.32**	**3.12**	**1.93**	**2.21**	**0.30**	**1.53**	**1.39**	**3.81**	**2.25**	**1.62**
Predict	P-threads launched (M)	255.97	4.26	4.91	3.50	15.80	17.00	8.52	3.85	2.77	20.71
	Instructions per p-thread	21.81	13.39	9.22	6.03	19.29	4.97	12.03	9.47	9.94	12.57
	Overhead IPC	2.57	3.15	1.84	2.08	0.29	1.44	1.17	3.61	2.04	1.27
	Misses covered (M)	11.87	0.46	0.64	0.24	11.03	1.38	4.58	0.45	0.28	5.40
	Misses fully covered (M)	4.35	0.23	0.42	0.16	5.84	1.19	3.75	0.20	0.16	2.85
	Latency tolerance IPC	5.25	3.28	2.02	2.23	0.38	1.61	1.61	3.83	2.35	2.18
	IPC	**3.85**	**3.27**	**2.01**	**2.21**	**0.38**	**1.60**	**1.60**	**3.81**	**2.32**	**2.12**

TABLE 2. Basic results and performance model validation

mance numbers are reported using the training inputs sampled at 100M of every 1B instructions with 10M instruction warm-up phases. Unless otherwise noted, p-threads are selected using the same program sample on which they are subsequently measured. This arrangement allows our framework to produce performance predictions which may then be checked. A relevant benchmark characterization is shown in Table 1. Of the 16 benchmark/input combinations, we use only 10; *eon* (3 inputs), *gzip*, and *perlbmk* (2 inputs) exhibit negligible L2 miss rates.

4.2 Primary Performance Results

The middle section of Table 2 (*Pre-exec*, the shaded section) 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 both in part and in full. A miss is fully covered if the p-thread initiates the miss far enough in advance to overlap the full latency of the miss. If the latency is only partially overlapped, the miss is partially covered. *Overhead IPC* and *Latency tolerance IPC* are produced by simulations which model only p-thread cost or benefit and are used in the next section to validate the model. The results show that the p-threads selected by our framework generally improve performance. The p-threads cover between 10% (*mcf*) and 82% (*vpr.p*) of the L2 misses in the program—with full coverage in general achieved for about half of all misses covered—and result in performance improvements of up to 24% (*vpr.r*). One benchmark, *crafty,* experiences a 1% performance degradation due to the addition of p-threads. These results are good in absolute terms, but the point of using a quantitative framework is to obtain the *best possible* results. In other words, we want assurance that the reason only 10% of *mcf*'s L2 misses are covered is that the structure of the program is such that our pre-execution model *cannot* be used to cover the other 90%, not because the framework couldn't find them. We devote the next section—and parts of the following sections—to constructing experiments that increase our confidence in this regard.

4.3 Model Validation

One way to gain confidence in our framework is to compare the performance of p-threads it produces with that of all other sets of possible p-threads. This approach is infeasible. We take a different tack based on the following argument. Our framework uses standard optimization techniques to find good solutions for a given function. That we are sure of.

What we are not sure of is whether the function our framework optimizes, ADV_{agg}, accurately models reality such that solutions which are good in the model space are also good in the real world. Fortunately, the modeling fidelity of ADV_{agg} is easier to verify. In performing its selections, our framework implicitly makes diagnostic predictions of p-thread behavior. We check these predictions against simulated measurements. If they align, we can be confident that ADV_{agg} is the appropriate p-thread evaluation function, and that our framework produces solutions that are good in the real world. The bottom portion of Table 2 (*Predict*) lists the framework predictions that allow us to perform this validation. We validate overhead and latency tolerance individually to better identify model inaccuracies.

Overhead. Three diagnostics check our overhead model: number of p-threads launched, average number of instructions per dynamic p-thread, and performance of an overhead-only implementation. We address the last diagnostic first. Overhead performance degradation is measured in *two* ways. In the first (*execute*), p-threads execute as usual, but do not access the data cache (thus do not have the pre-execution effect). In the second (*sequence*), p-thread instructions consume sequencing cycles but are immediately discarded. The first simulation measures true p-thread overhead, the second measures overhead as modeled by our framework (i.e., the only cost of a p-thread instruction is the bandwidth used to sequence it). As we see in the table, these simulations often produce identical results indicating that our “overhead as sequencing bandwidth consumption” assumption is valid.

Estimates of performance loss due to overhead are generally accurate. Predictions of average p-thread length are self-fulfilling. Occasional p-thread launch count *over-estimation* (e.g., *bzip2*) is due to the finite number of p-thread contexts—a p-thread launch request is dropped if a thread context is not immediately available. This effect is especially prominent in programs that require many p-threads. Typically, however, p-thread launch counts are *under-estimated* due to the fact that our framework does not account for p-threads launched from wrong-path trigger instructions. We have run simulations in which p-threads are launched only from correct path triggers and observed nearly perfect correlation between predicted and simulated launch counts. Interestingly, wrong-path p-thread launches do not increase overhead. Since the majority of p-threads are short and are sequenced within a cycle or two of launch, wrong-path p-threads primarily contend with wrong-path instructions whose latency does not directly impact performance.

Latency Tolerance. Latency tolerance is also validated via three diagnostics: L2 misses covered (i.e., turned into either full or partial hits), L2 misses fully covered (i.e., turned into full hits) and performance of a latency-tolerance-only implementation. Miss coverage is measured by timestamping cache blocks with p-thread request, main thread request, and ready times. Fully and partially covered misses are detected by the appropriate relationships between timestamps and are tabulated at instruction retirement to avoid overinflating the counts with wrong-path data. Latency-tolerance impact is measured via an additional simulation in which p-threads are not charged for bandwidth.

Miss coverage, both full and partial, is more difficult to predict than overhead, as there are many factors which affect miss coverage that are not considered by ADV_{agg}. Miss coverage over-estimation (too few misses actually covered) is the result of p-thread issue delays caused by contention with the main thread and other p-threads. Full miss coverage overestimation (too few actual full miss coverages) implies *post-issue* delays for p-thread misses, with the primary source being contention in the memory bus. Full miss coverage under-estimation (too *many* actual full miss coverages—a good problem to have) indicates *main thread* delays, primarily due branch mis-predictions but also to contention with p-threads. Miss coverage underestimation (too many actual misses covered) implies the presence of unintentional L2 prefetches *within* a p-thread. A given static load may not have statistical character that merits the construction of p-threads for its own sake. However, such loads are sometimes embedded in p-threads targeted for other loads or consistently share cache lines with such embedded loads. Each of these factors acts to some degree in *every* benchmark, with the net effect determined by the dominant phenomena. For instance, full miss-coverage underestimation is dominant in benchmarks with high branch mis-prediction rates (e.g., *crafty*, *gcc* and *vpr.r*).

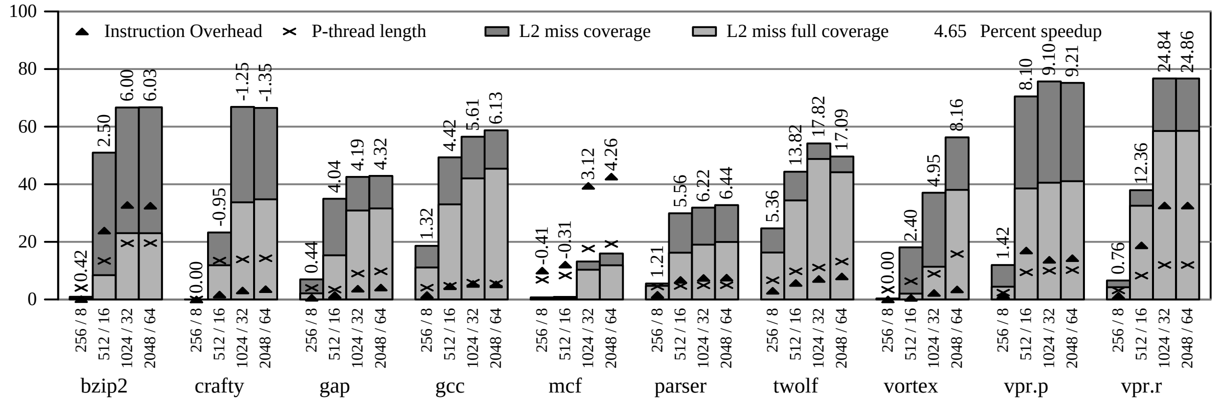
Unfortunately, the simulated metric that most poorly correlates with its predicted value is performance improvement. Even more unfortunately, it is generally *overestimated*. The primary cause is a single assumption built into our framework—that miss latency translates cycle for cycle into execution latency and, therefore, that miss latency tolerance translates cycle for cycle into performance improvement. Effectively, we assume that L2 misses are handled serially and are not overlapped with the execution of any instructions in the program. This is obviously not the case—in a dynamically scheduled processor, some degree of overlapping, either with other misses or other instructions in the program, is almost always possible—meaning that our framework is fooled into believing that there is more latency to tolerate than actually exists. An interesting piece of future work is to combine our framework with a critical path model [4] that can assign a “true” latency to each miss. Finally, while never completely true for L2 misses, the serialization assumption is even less true for L1 misses. Our experience shows that, while our framework easily finds p-threads that target L1 misses, its *predictions* in that scenario are less accurate.

We have presented a single validation experiment on a single design point in the processor and p-thread selection configuration space. We have performed similar experiments with other configurations—narrower processors, slower memories, and different thread selection parameters—and have obtained similar qualitative results. While our framework does not always predict end performance and diagnostics with perfect accuracy, in many cases it comes quite close. This suggests that ADV$_{agg}$, accurately captures pre-execution behavior under many conditions.

4.4 Sensitivity to P-Thread Selection Parameters

In this section and the next we measure our framework’s response—i.e., changes in the p-threads it produces—to variations in p-thread selection parameters (this section) and the underlying microarchitecture (Section 4.5). By measuring this response, we directly observe the performance potential of pre-execution under those same conditions. From this point forward, our results are presented graphically. The graphs all have a format similar to the one in Figure 4. Each bar in a group shows the results of one experiment using five diagnostics. Miss coverage (dark grey, top) and full miss coverage (light gray, bottom) are shown as stacked bars. Their units are in percentages of the number of L2 misses in the unoptimized program. Overhead is shown as a tick in each column and is computed as the number of

FIGURE 4. Combined impact of slicing scope and p-thread length.



p-thread instructions executed over the number of instructions retired by the main thread. Average dynamic p-thread length is shown as a cross mark. Finally, percent speedup over the base configuration is shown in text over each bar.

Slicing scope and p-thread length. Our first test measures the effects of *slicing scope*, the length of the dynamic trace that is examined to construct a p-thread, and *p-thread length*. Limiting these is a concession to the finite buffering of the p-thread constructor. Limiting *p-thread length* is also a concession to the implementation of the p-thread memory hierarchy. Figure 4 characterizes the performance of p-threads selected in four scope/length combinations. For instance, the left bar corresponds to a scope limit of 256 instructions and a maximum p-thread length of 8 instructions.

Two intuitive and comforting trends are evident. First, actual p-thread length, miss coverage, full miss coverage, and performance increase as p-thread selection constraints are relaxed. Second, they all saturate at some point and do not benefit from further relaxations. These trends imply that pre-execution performance potential is a strong property of program structure, that each combination of program and processor configuration has a *natural set of p-threads* and, more significantly, that *our framework gravitates to this natural set*. Quantitatively, the pre-execution performance of most programs effectively saturates at slicing windows of 512 instructions and with post-optimization lengths of 16 instructions, although several programs (e.g., *vortex*) benefit from further relaxations.

For brevity, we only present the combined effects of length and scope restrictions. The importance of each constraint individually varies from one benchmark to the next. Most programs are more sensitive to p-thread length constraints, unable to achieve any gain whatsoever with very short p-threads, even with large (2K instruction) slicing windows. This is an indication that miss computations in these programs are dense in the locus leading up to the miss—small computations are unable to obtain any sequencing advantage. Two programs which are more sensitive to scope restrictions are *parser* and *twolf*. This is signature of the complementary program structure, sparse computations which can achieve latency tolerance with small computations, but need large windows to “see” these computations.

P-Thread optimization and merging. One of the stated strengths of our framework is its support for p-thread optimization and merging. As Figure 5 shows, the addition of p-thread optimization and merging can have a profound performance impact on pre-execution, witness *vpr.r.*

Optimization reduces average p-thread length, often significantly (e.g., *crafty*, *parser*, *vortex*, and *vpr*). Less intuitively perhaps, it also often results in a significant increase in p-thread launches. With optimization reducing p-thread

FIGURE 5. Impact of p-thread optimization and merging.

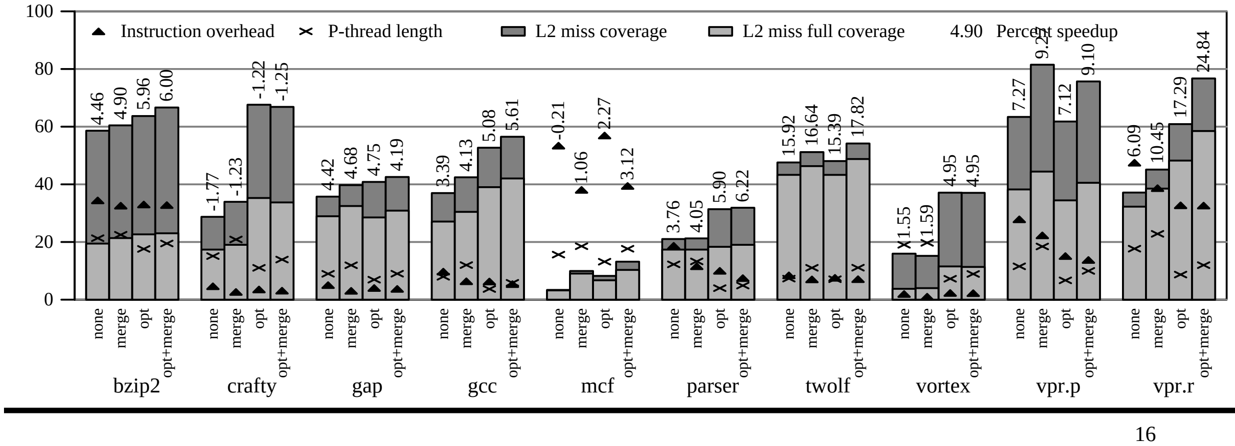
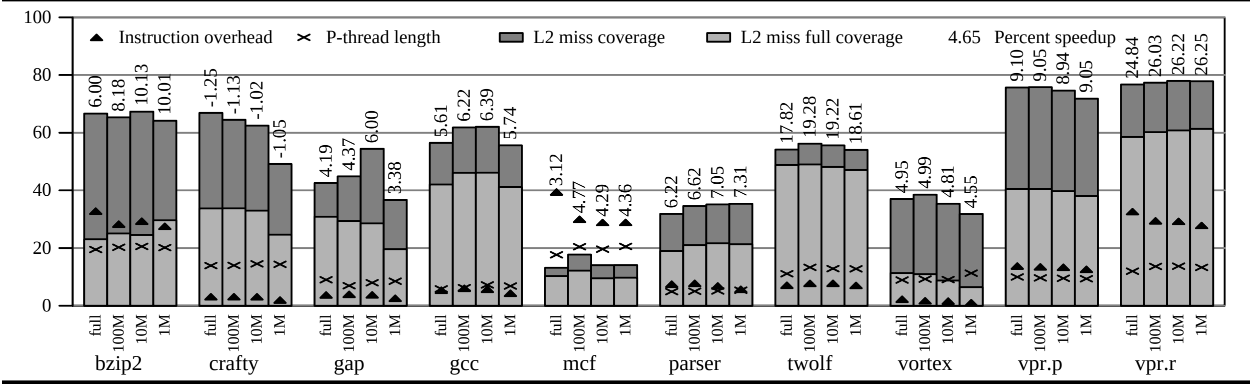


FIGURE 6. Impact of p-thread selection granularity.



length, increasing latency tolerance by compressing p-thread dataflow graphs and decreasing overhead, p-threads that were either unprofitable or illegal (i.e., too long) in their unoptimized forms become viable. An increase in the number of viable p-thread candidates results in increased miss coverage, more complete latency tolerance for covered misses, and improved performance. *Vortex* and *vpr.r* are prime examples of the power of this secondary effect, which is much stronger than the primary effect of overhead reduction for pre-existing p-threads. Of our three optimizations, we have found store-load pair elimination to be the most effective. Register-move elimination has almost no impact. Intuitively, our experience shows that optimization becomes increasingly effective as length constraints are tightened.

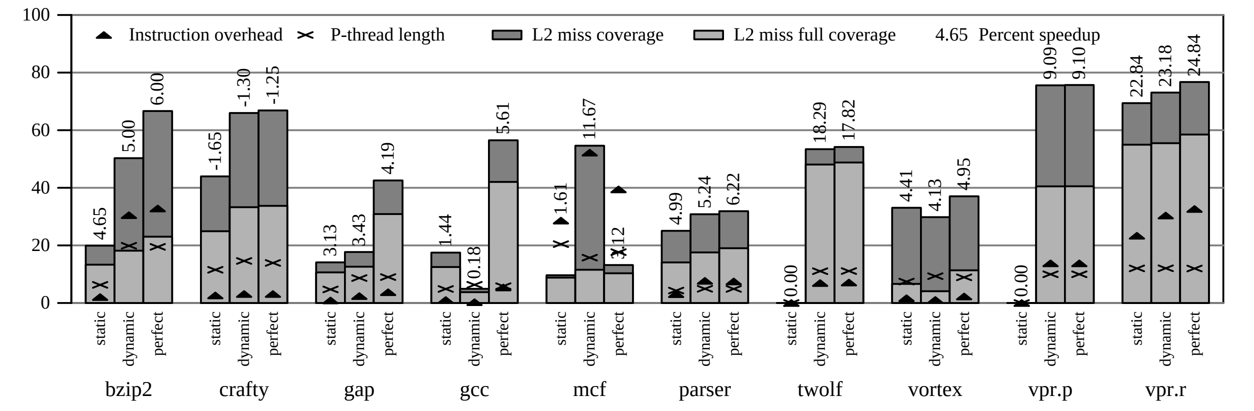
Merging primarily reduces overhead but does not increase the number of viable p-thread candidates—its performance effects are thus less pronounced. Merging increases p-thread length while decreasing the p-thread launch counts. Merging generally improves performance by reducing contention for p-thread contexts, although it can occasionally *increase* contention by creating long p-threads that occupy a single context for several 8-cycle sequencing periods.

P-thread selection granularity. Our default selection granularity is an entire (sampled) run of the program. Figure 6 compares this coarse grain approach with finer grain strategies in which p-threads are specialized for dynamic program regions of 100, 10, and 1 million instructions each. Intuition says that breaking the program into smaller chunks and optimizing each chunk separately will produce more highly specialized p-threads and higher performance. After all, at the limit of this process we would find a custom p-thread for every dynamic L2 miss!

The trends we expect to see are those we observe in *bzip2*, *parser*, *vpr.r* or the first three bars of *gap* and *gcc*—increased miss coverage, reduced overhead, and increased performance. Although these trends appear frequently, they are not consistent or even monotonic within a benchmark. Counter intuitive trends are most often seen at the finest (1 million instruction) granularity although they may appear sooner as they do in *vortex*. Finer selection granularities do not always mean increased miss coverage. If a p-thread is deemed profitable at a coarse-grain region, but not at all finer-grain sub-regions, coverage for any misses that do occur at unselected sub-regions will be lost. This phenomenon suggests a slight mis-calculation by our framework at fine granularities, specifically one that is overly-biased towards overhead. One reason for this may be that our overhead model is not quite accurate at very high or very low IPCs, which are typically seen only over small dynamic regions.

Overall, the consistency of results across grains suggests a certain amount of self-similarity in programs and builds confidence in our approach of using coarse-grain information (IPC) to model microscopic behavior.

FIGURE 7. Impact of p-thread selection input data-set.



Input Data Sets. Our evaluation to this point has shown that automated p-thread selection is possible given *perfect* information—a previous execution of the same program using the same input data. In reality, such information is never available. For our final experiment in p-thread selection parameter space, we vary the input data set used to construct p-threads to test the viability of performing automated p-thread selection in real world scenarios. We consider a *dynamic* scenario in which p-threads are selected on-line (ignoring for a moment that the exhaustive nature of our framework is not conducive to on-line implementation) using small profiling program phases. The dynamic scenario models p-thread selection as it would be implemented by a profile-driven, dynamically optimizing just-in-time compiler. We also consider a *static* scenario in which p-threads are selected using profiles from test inputs. The static scenario models p-thread selection implemented by a profile-driven static compiler. A simulated performance evaluation of p-threads selected under each scenario is shown in Figure 7.

The figure shows that good p-thread selection is theoretically possible under these two realistic implementation scenarios. P-threads selected in the dynamic scenario often approach the performance of those selected with perfect information. This is a testament to the fact that programs have a finite number of characteristic behaviors (determined by program structure) and further proof that our sampling methodology is sound. The less encouraging results from static scenario are the product of our choice of test inputs which use small data sets and incur significantly fewer L2 misses. In fact, the test data working sets for *twolf* and *vpr.p* fit into our L2 cache resulting in no p-threads being selected for those two benchmarks in the static scenario. However, for most other programs, static information is nearly as effective as dynamic information and even perfect information. This result reinforces our belief that p-threads and pre-execution performance potential are most strongly a function of program structure.

Occasionally, imperfect information yields better results than perfect information. Our framework makes several unrealistic assumptions—primarily that miss latency translates directly to program execution latency—which we can correct by essentially “lying” to the framework via adjusted parameters. In *mcf*, the framework’s interpretation of the characteristics of the dynamic sample better match the true characteristics of the perfect sample, than the framework’s interpretation of *those* characteristics. While manifestations of this problem are rare, they do suggest that future work may be needed to free the framework from these assumptions.

4.5 Sensitivity to Machine Parameters

An important aspect of our framework is its ability to accurately parametrize important processor features. In the interest of space, we cannot validate our framework's response to many parameters. We demonstrate our methodology using one important parameter, memory latency, which directly impacts latency tolerance requirements and hence p-thread structure. We have performed similar validation experiments on other parameters including processor width and L2 size, with similar results.

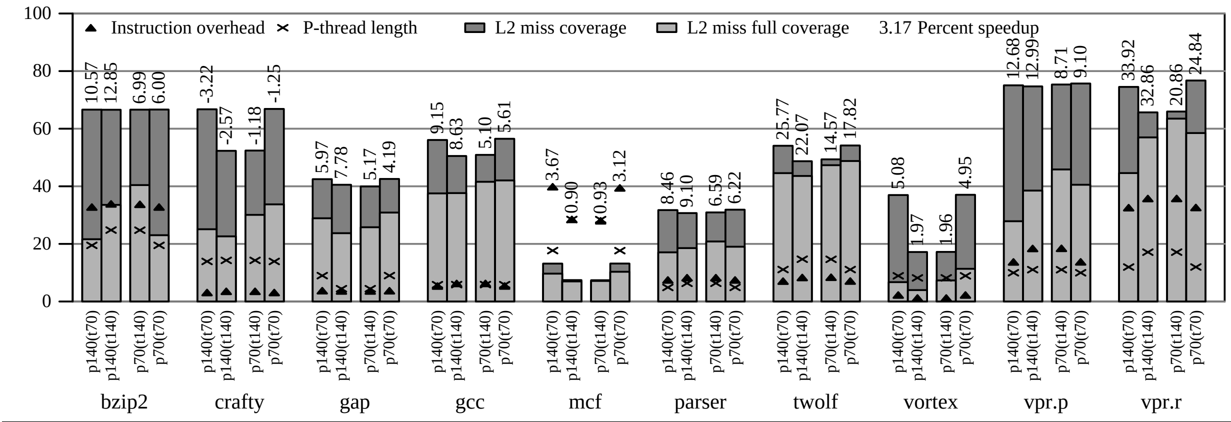
Intuition says that our framework models the underlying microarchitecture correctly if for a given configuration, the p-threads selected with that configuration in mind are better than p-threads selected for another configuration. We conduct a limited version of this study. For a given parameter *X*, let t_{X1} be the set of p-threads chosen using statistics obtained from a configuration where *X* is *X1* and $p_{X1}(t)$ be the performance of a processor in which the parameter has a value *X1* pre-executing the set of p-threads *t*. Within each study, we perform four experiments on two configurations, *X1* and *X2*—$p_{X1}(t_{X1})$, $p_{X2}(t_{X2})$, $p_{X1}(t_{X2})$, and $p_{X2}(t_{X1})$. We gain confidence in our framework's model of parameter *X* if $p_{X1}(t_{X1}) > p_{X1}(t_{X2})$ and $p_{X2}(t_{X2}) > p_{X2}(t_{X1})$. This cross-validation also allows us to test the framework's response to both under- and over- specification of parameter X.

Memory Latency. Results for memory latency are shown in Figure 8. We show four experiments split into two groups. Within each group, the simulated memory latency value is constant. Within each group, the right bar is the self-validation experiment and the left bar is the cross-validation experiment. In the left group, simulated memory latency is 140 cycles. In this group, the left bar corresponds to p-threads selected assuming 70 cycle memory latency (cross-validation) and the right bar to p-threads selected assuming 140 cycle latency (self-validation). In the right bar group, the roles are reversed. Here, we simulate a memory latency of 70 cycles, meaning that p-threads chosen assuming 70 cycle latency are used for self validation and p-threads chosen with a 140 cycle latency in mind are used for cross-validation. Due to our cross-validation methodology, the inner columns of a given bar group are similar to each other, as are the outer two columns.

Within each bar group, we are interested in two trends. First, we expect the self-validation experiments to outperform the corresponding cross-validation experiments. Second, we can intuitively gauge the framework's response to variations in a given parameter by comparing the self-validation experiments to one another.

Comparing $p_{70}(t_{70})$ with $p_{140}(t_{140})$ shows that our framework responds to memory latency variations in an intuitive way. A latency increase results in the selection of longer p-threads which cover fewer misses and fully cover fewer still. That it is the correct response is confirmed by cross-validation.

FIGURE 8. Response to variations in memory latency.



In $p_{70}(t_{140})$ (third bar in each group) we *over-specify* memory latency. We pretend that the processor has more latency to tolerate than it actually does, causing the framework produces more aggressive p-threads that more completely cover the latency that does exist. The framework does, in fact, produce longer p-threads and these fully cover more misses (the light gray bars are highest in this group). However, end performance is mixed. On most benchmarks, the expected effect is observed. As there is no more actual latency to tolerate, longer p-threads provide no additional benefit. At the same time, fewer total misses are covered as an increase in perceived latency makes previously viable p-threads unprofitable, with the net result being performance loss. However, on several benchmarks, the reverse happens. On these benchmarks, high memory bus contention—which our framework does not explicitly model—effectively increases memory latency and exploits the increased latency tolerance of the longer p-threads. By over-specifying latency, we are effectively helping the framework model bus contention.

$p_{140}(t_{70})$ is an *under-specification* experiment. By pretending that the underlying processor has less latency to tolerate than actually exists, we elicit the framework to produce less aggressive p-threads that can cover more total misses. Although sometimes unexpectedly successful (again, for the highlighted benchmarks) this strategy typically backfires. The framework produces more and shorter p-threads capable of tolerating less latency but doing so for more misses. The lost opportunity to tolerate longer latencies typically results in lower performance. However, in several cases, memory latency under-specification can produce *better* results. This is a concrete example of parameter adjustment overcoming the shortcomings of the framework. Recall, one of the framework's stated problems is that it incorrectly assumes that latencies are serial, i.e., that they translate into performance cycle for cycle. In *twolf* and *vortex*, by feeding the framework lower than actual latencies, we help it *simulate* the true conditions of naturally overlapped misses. Actual latency tolerance is not reduced, as there were never 140 cycles worth of latency per miss anyway. Reduced overhead and increased total miss coverage produce a net gain.

5 Related Work

The analysis of cache misses and techniques for avoiding and eliminating them has been has been an active subject of research for as long as caches have existed. Software, hardware, and cooperative techniques for prefetching regular and irregular accesses have been proposed and implemented [1, 6, 9, 12], including several that use finite state machines (FSMs) to mimic execution and generate prefetch addresses from loaded prefetch values [10, 13].

An early proposal to accelerate a single sequential program via prefetching using general-purpose threads was Assisted Execution [16]. Implementations of pre-execution in its current form include Speculative Data-Driven Multithreading (DDMT) [14], Speculative Pre-Computation [2, 3, 18], Speculative Slices [20], Software Controlled Pre-Execution [8], and Slice Processors [11]. Each implementation has its own special feature. Recent work proposes pre-execution on an in-memory processor creating a "push" prefetching model which cuts round-trip request/reply effects [15, 19].

Our framework complements this body of work. We parametrize the pre-execution run-time model. From a run-time perspective our results apply to all of these implementations, whether p-threads are executed on dedicated resources [3, 2, 11, 15, 19] or in a shared resource environment [8, 14, 20] and whether they are executed at the architectural level [8, 15, 19, 20] or the microarchitectural level [2, 3, 11, 14]. Our results are directly applicable to those implementations which also use static p-threads [8, 14, 15, 19, 20]. Short p-thread selection intervals (Section 4.2) can be used to for comparison with systems in which p-threads are generated continuously and on-the-fly [2, 3, 11]. Even so, applicability to dynamic p-thread selection systems is tenuous: such systems typically do not analyze p-thread behavior and do not explicitly target aggregate effects but rather continuously adapt and modify p-threads via feedback. For instance, one incarnation of Speculative Precomputation [2] initially selects a conservative p-thread and adds levels of induction unrolling if more latency tolerance is needed. In addition to implementation differences, there may be p-thread sequencing model differences. Our framework assumes control-less p-threads and no chaining, a model used by several proposed systems [2, 11, 14]. The use of chaining [3] or control flow [8, 15, 19, 20] reduces the applicability of our results, although extensions to handle these different p-thread sequencing models are possible.

More recently, several frameworks for compiler- or linker-based p-thread generation have appeared [5, 7, 8] one of which even employs inter-procedural analysis [5]. These frameworks form a strong implementation path for pre-execution. However, they have certain limitations which derive from their static nature. Primarily, they cannot analyze the latency tolerance of general dataflow graphs and must instead approximate these from high-level static constructs. Although the amount of main thread work available for overlapping can be approximated for simple loops, such analysis is difficult for general conditional and call constructs. By dealing with execution traces, our framework sees the dynamic instruction stream as a long straightline piece of code (in which loops are unrolled) and sidesteps this problem. We hope that some of the insight supplied by our framework can be combined with the practical aspects of these.

6 Conclusions

Memory latency is a significant component of total execution time for integer programs running on modern processors. With multithreading becoming prevalent, pre-execution—a recently proposed technique for effectively moving cache miss latency to other threads—is becoming popular. This paper presents a quantitative framework for reasoning about the performance potential of pre-execution and, as a useful side effect, for selecting static p-threads. Our framework contains two novel components. *Aggregate advantage* is a function that 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 external parameters allow it model most processor configurations.

We apply our framework to find static p-threads for covering L2 misses in the SPEC2000 integer benchmarks, and evaluate the performance of these p-threads using detailed timing simulation. In addition to measuring pre-execution performance under different p-thread selection and processor conditions, we evaluate the framework itself by checking its diagnostic predictions against simulated measurements and by verifying that it qualitatively responds to underlying parameter variations as an optimization framework would. We find that aggregate advantage models p-thread behavior accurately, and that it parametrizes the important aspects of the underlying processor—miss latency and processor width—accurately to a first order. The performance results themselves reveal several interesting facts about p-threads. P-thread effectiveness monotonically increases as selection constraints are relaxed but saturates at certain characteristic points. This behavior strongly suggests that pre-execution effectiveness and p-thread structure are properties of the program—a given program/processor pair is associated with a certain canonical set of static p-threads. Encouragingly, our framework gravitates to this set when left to its own devices.

There are several interesting directions for future work. Primarily, alignment of the framework's performance model and its assumptions with reality is a continuing process. This may involve adding a critical path modeling [4] component to the framework or enriching its vocabulary to allow it to quantitatively reason about naturally overlapped misses. Such an addition is important as it would allow us to better model p-threads for pre-executing L1 misses.

7 References

[1] T. Chen and J. Baer. "Effective Hardware Based Data Prefetching for High Performance Processors." *IEEE Transactions on Computers*, 44:609–623, May. 1995.

[2] J. Collins, D. Tullsen, H. Wang, and J. Shen. "Dynamic Speculative Precomputation." In *Proc. 34th International Symposium on Microrchitecture*, pages 306–317, Dec. 2001.

[3] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. "Speculative Pre-Computation: Long Range Prefetching of Delinquent Loads." In *Prc. 28th International Symposium on Computer Architecture*, pages 14–25, Jul. 2001.

[4] B. Fields, S. Rubin, and R. Bodik. "Focusing Processor Policies via Critical Path Prediction." In *Proc. 27th Annual International Symposium on Computer Architecture*, pages 74–85, Jul. 2001.

[5] D. Kim and D. Yeung. "Design and Evaluation of Compiler Algorithms for Pre-Execution." In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)*, Oct. 2002.

[6] A. Lai, C. Fide, and B. Falsafi. "Dead-block prediction and dead-block correlating prefetchers." In *Prc. 28th International Symposium on Computer Architecture*, pages 144–154, Jul. 2001.

[7] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. "Post-Pass Binary Adaptation for Software-Based Speculative Pre-Computation." In *Proc. ACM 2002 Conference on Programming Language Design and Implementation*, Jun. 2002.

[8] C.-K. Luk. "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors." In *Prc. 28th International Symposium on Computer Architecture*, pages 40–51, Jul. 2001.

[9] C.-K. Luk and T. Mowry. "Compiler Based Prefetching for Recursive Data Structures." In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[10] S. Mehrotra and W. Harrison. "Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Program." In *Proc. 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[11] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. "Slice Processors: An Implementation of Operation-Prediction." In *Proc. 2001 International Conference on Supercomputing*, Jun. 2001.

[12] T. Mowry, M. Lam, and A. Gupta. "Design and evaluation of a compiler algorithm for prefetching." In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[13] A. Roth, A. Moshovos, and G. Sohi. "Dependence Based Prefetching for Linked Data Structures." In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

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

[15] Y. Solihin, J. Lee, and J. Torrellas. "Using a User Level Memory Thread for Correlation Prefetching." In *Proc. 29th International Symposium on Computer Architecture*, pages 171–182, May 2002.

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

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

[18] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, and J. Shen. "Memory Latency Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation." In *Proc. 8th International Syposium on High-Performance Computer Architecture*, Jan. 2002.

[19] C.-L. Yang and A. Lebeck. "Push vs. Pull." In *Proc. 2000 International Conference on Supercomputing*, May 2000.

[20] C. Zilles and G. Sohi. "Execution Based Prediction Using Speculative Slices." In *Prc. 28th International Symposium on Computer Architecture*, pages 2–13, Jul. 2001.