Energy-Effectiveness of Pre-Execution and Energy-Aware P-Thread Selection

Vlad Petric, Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{vladp,amir}@cis.upenn.edu

Abstract

Pre-execution removes the microarchitectural latency of "problem" loads from a program's critical path by redundantly executing copies of their computations in parallel with the main program. There have been several proposed pre-execution systems, a quantitative framework (PTHSEL) for analytical pre-execution thread (p-thread) selection, and even a research prototype. To date, however, the energy aspects of pre-execution have not been studied.

Cycle-level performance and energy simulations on SPEC2000 integer benchmarks that suffer from L2 misses show that energy-blind pre-execution naturally has a linear latency/energy trade-off, improving performance by 13.8% while increasing energy consumption by 11.9%.

To improve this trade-off, we propose two extensions to PTHSEL. First, we replace the flat cycle-for-cycle load cost model with a model based on a critical-path estimation. This extension increases p-thread efficiency in an energy-independent way. Second, we add a parameterized energy model to PTHSEL (forming PTHSEL$_{+E}$) that allows it to actively select p-threads that reduce energy rather than (or in combination with) execution latency.

Experiments show that PTHSEL$_{+E}$ manipulates pre-execution's latency/energy more effectively. Latency targeted selection benefits from the improved load cost model: its performance improvements grow to an average of 16.4% while energy costs drop to 8.7%. ED targeted selection produces p-threads that improve performance by only 12.9%, but ED by 8.8%. Targeting p-thread selection for energy reduction, results in "energy-free" pre-execution, with average speedup of 5.4%, and a small decrease in total energy consumption (0.7%).

1 Introduction

L2 cache misses are a major obstacle to ILP. Address-prediction driven prefetching eliminates many misses, but a small number of static loads—"problem" loads—defy address prediction and generate disproportionate numbers of misses. *Pre-execution* generates timely and accurate prefetch addresses for problem loads using execution rather than address prediction. Pre-execution isolates problem load computations, then redundantly executes copies of them in parallel with the main program on the spare hardware contexts of a multithreaded processor. The chosen computations are called *p-threads*. Pre-execution has been used to target both branch mispredictions and cache misses, but has been most successful in targeting L2 misses. Load latency reduction can be "communicated" from a p-thread to the main program via simple cache prefetching. L2 misses are also difficult to overlap with other main thread work. The latter fact both makes pre-execution more valuable, and simplifies the automated analysis and selection of p-threads. In previous work we presented a framework for analytically mining p-threads from program profiles using constrained optimization [19]. The framework—which in this paper we call PTHSEL—enumerates all static p-threads of a given length or less, uses formulae to estimate the performance advantage of each p-thread, and chooses the set that maximizes this advantage. It has proven to produce good L2-miss targeting p-threads. Recent years have seen several proposed implementations of pre-execution [3, 4, 6, 7, 15, 17, 18, 21, 24, 25] and even some physical research prototypes [23]. To this date and to our knowledge, no one has examined pre-execution's energy aspects.

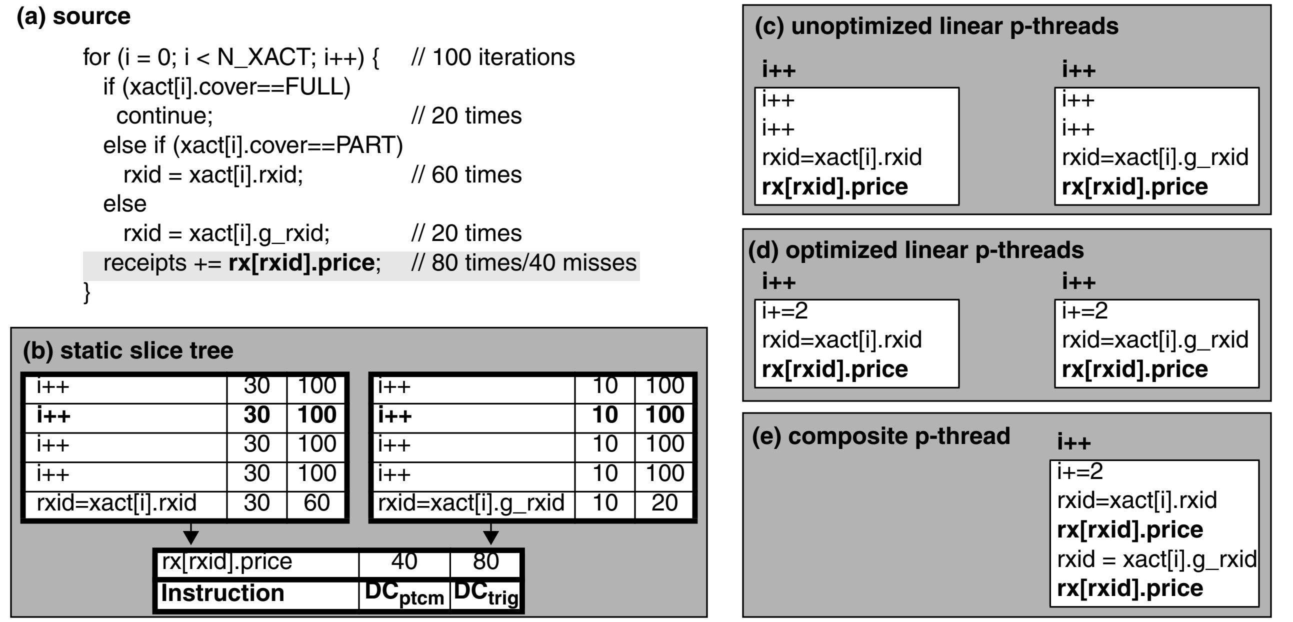
Like many performance techniques, pre-execution uses energy—in the form of redundant execution—to reduce execution latency. Unlike some other techniques, however, it is driven by a formal framework that explicitly controls the amount of redundancy. In this paper, we ask two questions. First, what is the energy/latency trade-off of pre-execution as driven by an energy-unaware tool like PTHSEL? Second, can this trade-off be manipulated? In other words, can we add energy-cognition to PTHSEL (forming PTHSEL$_{+E}$) and allow it to select p-threads that minimize energy consumption?

To answer the first question, we analyze the performance and energy consumption of pre-execution using p-threads as selected by the original, energy-unaware PTHSEL. We find that latency-oriented p-threads (L-p-threads) have a quasi-linear latency/energy trade-off, producing average speedups of 13.8%, while increasing energy consumption by 11.9%. As an answer to the second question, we enhance PTHSEL in two ways.

First, we replace PTHSEL's linear load-latency-reduction to global-execution-time-reduction function with a parametric function based on an approximation of the critical path [9]**.** A combination of two critical path estimates also models the interaction costs [8] of contemporaneous L2 misses. PTHSEL's original ignorance of interaction costs left it vulnerable to selecting ineffectual p-threads. While not directly energy-aware, this extension allows PTHSEL to make more judicious p-thread choices, primarily by recognizing that some problem loads have a rather small effect on global execution time. The new miss cost model is responsible for producing performance-positive p-threads for *mcf*, and raises the average performance gain from 13.8% up to 16.4%. Its main benefit, however, is a significant improvement in the accuracy of latency gain estimation. This added precision allows us to build on top of the existing, latency-oriented, PTHSEL equations.

Second, we enhance PTHSEL with a parametric energy model that explicitly accounts for the energy costs

0-7695-2270-X/05/$20.00 (C) 2005 IEEE



(a) source

```
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
  else
    rxid = xact[i].g_rxid;          // 20 times
  receipts += rx[rxid].price;       // 80 times/40 misses
}
```

(b) static slice tree

i++	30	100
i++	**30**	**100**
i++	30	100
i++	30	100
rxid=xact[i].rxid	30	60

i++	10	100
i++	**10**	**100**
i++	10	100
i++	10	100
rxid=xact[i].g_rxid	10	20

rx[rxid].price	40	80
Instruction	**DC_{ptcm}**	**DC_{trig}**

(c) unoptimized linear p-threads

```
i++                      i++
i++                      i++
i++                      i++
rxid=xact[i].rxid        rxid=xact[i].g_rxid
rx[rxid].price           rx[rxid].price
```

(d) optimized linear p-threads

```
i++                      i++
i+=2                     i+=2
rxid=xact[i].rxid        rxid=xact[i].g_rxid
rx[rxid].price           rx[rxid].price
```

(e) composite p-thread

```
i++
i+=2
rxid=xact[i].rxid
rx[rxid].price
rxid = xact[i].g_rxid
rx[rxid].price
```

FIGURE 1. Pre-execution and p-thread selection example. (a) static code, (b) static slice tree, (c) unoptimized linear p-threads, (d) optimized linear p-threads, (e) merged composite p-thread.

(and potential benefits) of p-threads. The energy-aware framework, which we call PTHSEL$_{+E}$, also contains formulae for composing the existing latency criterion and the new energy one. Via a composition parameter, PTHSEL$_{+E}$ can be retargeted to select p-threads that optimize execution time, energy consumption, or any combination of the two like ED [10] or ED^2[16]. Energy oriented p-threads achieve only a 5.4% performance improvement, but actually reduce energy by 0.7%. ED-oriented p-threads effectively balance the two concerns, improving performance by 12.9%, at an energy increase of 3%, yielding the greatest ED improvement, 8.8%.

One parameter that impacts pre-execution's latency/energy trade-off is the idle energy factor: the fraction of maximum per-cycle energy the processor consumes when idle and which can only be saved using drastic, long/term energy reduction techniques like power gating (i.e., "deep sleep"). A zero idle factor makes latency-oriented pre-execution quite unattractive from an energy standpoint, but retargeting p-threads for ED restores a linear latency/energy trade-off. When the idle factor is high, pre-execution can be used as an energy reduction tool.

The rest of the paper is organized as follows. Section 2 reviews pre-execution and the original p-thread selection framework, PTHSEL. Section 3 presents our extensions for retargeting PTHSEL to optimize for energy consumption or combinations of latency tolerance and energy consumption like ED. Section 4 evaluates these extensions using cycle-level performance and energy simulation.

2 Pre-Execution Primer

We begin by reviewing pre-execution and the original, energy-unaware p-thread selection framework (PTHSEL) [19]. Figure 1a shows a loop that executes 100 iterations. The bold statement is a "problem" load. It misses 50% of the time and its address stream is not easily predictable by conventional predictors.

Pre-execution generates prefetch addresses for problem loads by redundantly executing their computations. An isolated problem load computation is called a *static p-thread* and consists of a *trigger* and a *body*. The trigger is a PC in the original program or, alternatively, a special jump instruction that has been planted into the main program. The body is a list of instructions that comprises the computation. When the main thread encounters a trigger, it spawns a copy of the corresponding body and executes it on a spare hardware context. Starting at the trigger, the main thread and p-thread execute in parallel. However, while the main thread fetches and executes all instructions, the p-thread fetches and executes only the load's computation. As a result, it fetches and issues the load first. By the time the main thread reaches problem load, the corresponding block is hopefully already cached.

Figure 1e shows the final p-thread chosen for the problem load. This p-thread, which is triggered by the induction statement **i++**, demonstrates many of the subtleties and complexities of p-thread selection. For now, notice that it tolerates the load's latency by effectively skipping two loop iterations ahead via **i+=2**, then pre-executes both possible load computations, once with **rxid=xacts[i].rxid** and once with **rxid=xacts[i].g_rxid**.

2.1 One Implementation: DDMT

There have been several proposed implementations of pre-execution [3, 4, 6, 7, 15, 17, 18, 21, 24, 25]. S*peculative Data-Driven Multithreading (DDMT)* [18] is representative of most, but has two unique features. First, DDMT executes p-threads in *lightweight* mode [5, 18]. P-thread instructions (p-instructions) are allocated reservation stations and physical registers, but not ROB or LSQ entries; they do not impact architectural state because they simply do not retire. More significantly, DDMT restricts p-threads to be *control-less* (fixed instruction sequences that execute in their entirety) and *unchained* (only the main thread spawns p-threads). Our example illustrates both restrictions and their work-arounds. Within a given iteration, the problem load is control-dependent on the branch **if (xacts[i].cover==FULL)**. Rather than including this branch in the p-thread, PTHSEL "assumes" that it is taken. A similar "choice" includes both **rxid=xacts[i].rxid** and **rxid=xacts[i].g_rxid** in the p-thread and excludes the branch that selects between them. P-thread chaining is used to jump multiple loop iterations ahead. In DDMT, chaining is replaced by *induction unrolling*, the inclusion of multiple induction statements in a p-thread. Our example p-thread contains two-levels of induction unrolling (**i+=2**). Control-less-ness and non-chaining preclude runaway p-threads, and simplify p-thread analysis: for each static p-thread, we know how many dynamic instances there will be and how many instructions each will contain.

2.2 PTHSEL: Automated P-Thread Selection

P-threads are (optimized) backward slices of problem loads. The difficulty in selecting p-threads is not in constructing the slices, it's in knowing when to stop, i.e., determining how large a p-thread should be or, equivalently, how far ahead of the target load it should be spawned. Tolerating longer latencies requires "hoisting" the trigger further upstream from the target load. However, hoisting increases p-thread overhead by forcing it to include more instructions. It also reduces the number of misses covered while potentially increasing the number of useless p-threads. The second effect comes from the increased number of branches between the trigger and target load which increases the probability that the main thread will deviate from the path implicitly assumed by the p-thread. The considerations described above are all associated with choosing the "sweet-spot" of a *linear p-thread*: a p-thread consisting of the computation of a single load. In general, the misses of a problem load will be computed by multiple instruction sequences along multiple paths. How does one choose a set of static p-threads that covers as much latency for as many misses as possible while minimizing overhead? And how does one choose p-threads for multiple problem loads? PTHSEL answers these questions quantitatively.

Overview: dynamic slicing, linear p-threads, static slice trees, and merging. PTHSEL uses a divide-and-conquer approach. First, it extracts linear p-thread candidates from dynamic program traces via backward data-dependence slicing. Candidates are grouped by static problem load and organized into slice trees. The root of the tree is the problem load itself. Each candidate is represented by a node—its trigger is the node itself, while the body is the path from the node to the root. Figure 1b shows the slice tree for the problem load in the example. A fork in the tree indicates a control-decision which effects the problem load's data slice. Here, the fork is due to the branch **if (xacts[i].cover==PART)**. A search procedure examines each tree independently and selects a subset of the p-threads. A post-pass merges linear p-threads (potentially targeting different static loads) with common triggers, under the assumption that merging improves p-thread characteristics (i.e., it lowers overhead and doesn't impact latency tolerance). From the slice tree in Figure 1b, the procedure selects the two linear p-threads (1c), which are optimized (1d) and merged (1e).

Aggregate Latency Advantage ($LADV_{agg}$). PTHSEL's centerpiece is a function that estimates the benefit of each static p-thread as the number of cycles by which its dynamic instances reduce execution time. Previously called aggregate advantage (ADV_{agg}), we now call this function *aggregate latency advantage* ($LADV_{agg}$) to distinguish it from energy metrics. Control-less-ness and non-chaining allow the cumulative effects of a p-thread's dynamic instances to be calculated by simple multiplications. Shown in Equations L1–L3 of Table 1, $LADV_{agg}$ is the difference of two components. *Aggregate latency reduction* ($LRED_{agg}$) is the latency tolerance of one dynamic instance multiplied by the number of covered misses (DC_{ptcm}). *Aggregate latency overhead* (LOH_{agg}) is the overhead of one dynamic instance multiplied by the number of instances spawned (DC_{trig}). DC_{trig} and DC_{ptcm} are mined from the traces and annotate the slice tree (1b).

The final components of $LADV_{agg}$ are tolerated

TABLE 1. PTHSEL Latency Model

Equation or Definition	Description
L1. $LADV_{agg}(p) = LRED_{agg}(p) - LOH_{agg}(p)$	**Aggregate latency advantage**
L2. $LOH_{agg}(p) = DC_{trig}(p) * LOH(p)$	**Aggregate latency overhead:** overhead is incurred for every dynamic p-thread instance (DC_{trig})
L3. $LRED_{agg}(p) = DC_{pt\text{-}cm}(p) * LRED(p)$	**Aggregate latency tolerance:** latency is tolerated only for instances that pre-execute actual misses ($DC_{pt\text{-}cm}$)
L4. $LOH(p) = (SIZE(p)/BWSEQ_{proc}) * (BWSEQ_{mt}/BWSEQ_{proc})$	**Per dynamic instance overhead:** overhead is discounted by main thread sequencing utilization ($BWSEQ_{mt}/BWSEQ_{proc}$)
L5. $BWSEQ_{proc}$, L_{cm}	**External per-microarchitecture parameters:** processor width, and memory latency, supplied by manufacturer or reverse engineered.
L6. $BWSEQ_{mt}$	**External per-program parameters**: unoptimized IPC
L7. $LADV_{agg}$ –= $LRED(p) * DC_{pt\text{-}cm}(CHILD(p))$	**Reduced aggregate advantage**

latency (LRED) and overhead (LOH) per dynamic p-thread instance. P-thread overhead is modeled as fetch bandwidth consumption, discounted by main thread consumption of the same (bandwidth is considered free if the main thread is not using it). The latency tolerance calculation is more involved and we do not show it for space reasons. It is described in detail complete with working examples in the original paper [19].

Correcting for overlaps. Although PTHSEL maximizes LADV_{agg} sum, it does not select all p-threads with positive LADV_{agg}. Two static p-threads *overlap* if they target an overlapping set of dynamic misses. In our example, a p-thread that has two unrolled instances of **i++** overlaps with one that has three. PTHSEL recognizes overlaps as parent-child tree relationships and discounts the LADV_{agg} of each selected p-thread by the shared latency tolerance it has with any selected child p-threads (L7). If discounting turns LADV_{agg} negative, the p-thread is de-selected.

3 Energy-Effectiveness of Pre-Execution

A performance technique like pre-execution is *energy-effective* if it reduces latency at a greater rate than that with which it increases energy consumption. Equivalently, an energy technique is energy-effective if it reduces energy consumption at a greater rate than that with which it increases execution time. Two metrics for measuring energy effectiveness are *energy-delay (ED)* [10] and the *energy-delay*^{2} *(ED*^{2}*)* [16]; energy-effective techniques have relative-to-baseline ED and/or ED^2 of less than 1.

A performance technique need not strictly trade latency for energy; it can reduce *both* given a vehicle for translating execution time reductions into energy savings. Processors consume some energy even when "idle". We use the term *idle energy* to lump together energy consumption due to leakage, imperfect/absent clock gating, and the clock-gating control circuitry itself. By reducing execution time, a performance technique saves idle energy. These savings can be "virtual" if the processor reduces the idle energy consumption per task by executing more tasks in a given amount of time. They can also be real if the processor supports aggressive energy conservation techniques like power-gating and dramatic frequency/voltage scaling which can be employed when the processor is truly idle. One example of such a technique is the Mobile Pentium 4's sleep modes, which employ voltage scaling as well as clock gating, to reduce static power consumption by as much as 40% and total power consumption by 9.5% [11]. We conservatively choose 5%—similar to the energy savings corresponding to a transition from "Stop-Grant" to "Deep Sleep"—to be our idle energy factor. This number is likely to increase as leakage becomes more prominent.

3.1 Experimental Setup

We begin by describing our experimental tools.

Timing Simulator. Our simulator is built using the SimpleScalar Alpha AXP machine definition and system call interface [2]. It models a dynamically scheduled multithreaded processor with aggressive control speculation, MIPS-R10000 style register renaming, and a two-level on-chip memory hierarchy. The default configuration is a 6-way superscalar processor, with a 15 stage pipeline, 128-entry ROB, 80 reservation stations and 384 physical registers. 384 registers suffice to hold the architectural state of four threads and the in-flight values of 128 instructions. The processor is actually equipped with 8 thread contexts, exploiting the fact that p-threads have no architectural state. Our experiments show that even with 8 thread contexts, pre-execution only requires about 20 additional registers to hold in-flight p-thread values. However, we stress that such a processor would not be a viable, general-purpose, multithreaded machine. We use an 8K-entry hybrid branch predictor, with a 2K-entry BTB. The on-chip memory hierarchy includes a 32KB, 2-way set-associative, 1-cycle access instruction cache, 16KB, 2-way set-associative, 2-cycle access data cache, a 256KB, 4-way set-associative, 12-cycle access L2, and 64-entry instruction and data TLBs. The L2 and memory buses are 16-bytes wide; the memory bus is clocked at 1/4 processor frequency. We model an infinite main memory with a 200 cycle latency. The processor can issue 2 loads and 1 store every cycle and handle up to 16 outstanding misses.

Energy Model. We enhance the simulator with a modified version of Wattch [1]. Wattch models energy for arrays (TLBs, branch predictor tables, ROB, reservation stations, and rename tables), important combinational circuits (ALUs), buses, and the clock and uses CACTI 3.0 [20] to model cache energy. We model a clock-gating style in which all structures draw some fixed fraction of their maximum per-cycle energy even when unused with the remainder consumed proportionally based on port usage. The per-structure energy consumption of our processor is: branch predictor/BTB (4.4%), i-cache/TLB (18.1%), window/ROB/result-bus (13.6%), register file (14.2%), ALUs (5.5%), d-cache/TLB/LSQ (8.6%), L2 cache (13.6%), and clock (22%). This breakdown corresponds to an unrealistic cycle in which every port of every structure is accessed. We assume a 5% idle energy factor [11]. Our other technology parameters are 100nm process, a 3GHz core frequency, and a 1.2V supply voltage.

Benchmarks. We perform our experiments on those SPEC2000 integer benchmarks that suffer from L2 misses. Most proposed implementations of pre-execution, and DDMT in particular, use statically generated p-threads. For a program/processor combination the will not benefit from pre-execution, the executable is not augmented with p-thread code. The remaining benchmarks are not affected by pre-execution one way or the other. We compile the programs for the Alpha EV6 architecture using the Digital Unix C-compiler with optimizations `-O4 -fast`. The simulator extracts all nops at no (simulated) cost. We simulate the programs to completion on their *train* inputs, using 2% sampling—with 2% cache/branch predictor warm-up—with 10M instructions per sample.

P-Thread Selection. We select p-threads using raw statistics mined from the same runs as those p-threads will subsequently optimize. This "ideal profiling" allows us to perform validation experiments by comparing predicted values with simulated ones. As previously shown [19], PTHSEL is sensitive to algorithm configuration and certain microarchitectural parameters, but is robust across

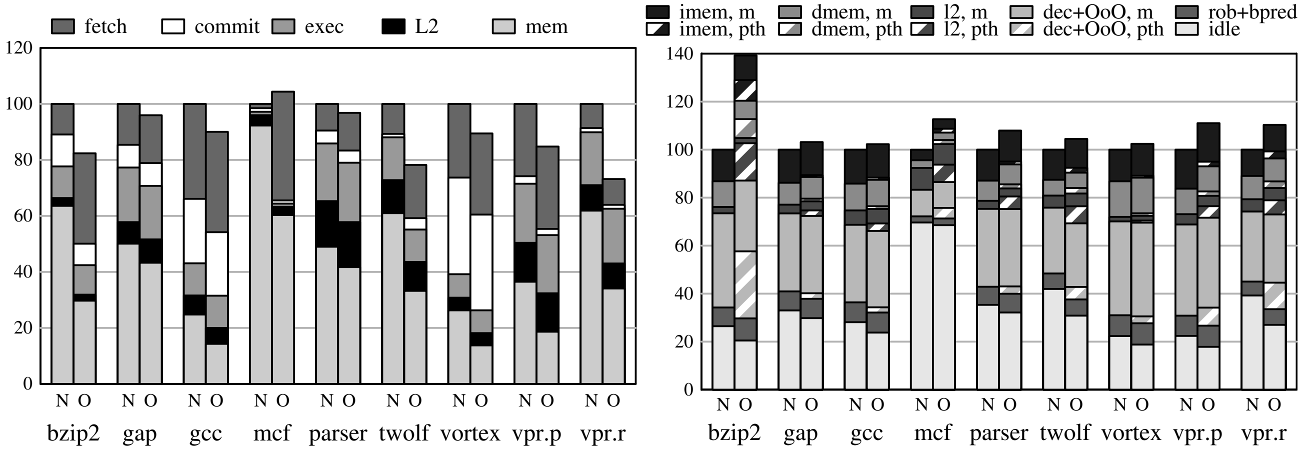


FIGURE 2. Latency and energy analysis of PTHSEL driven pre-execution. Left graph shows latency (critical path) breakdowns, right shows energy. Unoptimized execution (N) and PTHSEL driven pre-execution (O).

program inputs. We re-establish this result in an energy context in Section 5.1. Our default selection settings are a 2048-instruction slicing window and 64 instructions per linear p-thread.

3.2 Results and Discussion

Figure 2 shows relative execution time and energy breakdowns for un-optimized executions (N) and executions augmented with latency-oriented PTHSEL p-threads (O). For these programs, PTHSEL produces p-threads that yield average reductions in execution time of 13.8%, but does so at an energy cost of 11.9%. The execution time and energy breakdowns help illuminate.

Latency analysis. The execution time breakdown is constructed using a critical-path methodology [9]. The model breaks the program's critical path into five categories which are shown as a bar stack. From the bottom, the categories are: memory latency, L2 latency, execution latency, commit bandwidth, and fetch bandwidth/latency. The performance effects of a finite-instruction window and branch mispredictions are also part of the fetch bar. Although misses to memory are localized to a relatively small number of problem loads and account for only 1% of the total number of critical path edges, their accumulated latencies range from 25% (*gcc*) to 92% (*mcf*) of the total running time. Pre-execution manages to reduce, sometimes considerably, the L2 miss component of the execution time, but it does so at the expense of main thread fetch (*bzip2*, *mcf*, *twolf*). A very high number of p-threads and p-instructions (e.g. a 48% increase in instruction count for *bzip2* and as high as 310% for *mcf*) effectively starve the main thread. Part of the problem is failure on the part of PTHSEL to adequately model the bandwidth overheads of pre-execution. PTHSEL uses a divide-and-conquer approach that can only model the aggregate bandwidth overheads of the dynamic instances of individual static p-threads in isolation. PTHSEL can and will select many p-threads each of which will exact reasonable overhead in isolation, but which in union will choke the processor.

Counter-intuitively, the deeper problem is not that the p-threads are "bad" in aggregate, but that many of them are "bad" individually. PTHSEL's assumption that latency tolerance translates directly into latency gains is too aggressive, and can lead to the generation of many p-threads that in reality produce net performance losses (e.g., *mcf*). This inaccuracy in PTHSEL's miss-cost model is the main motivation for our first extension. By using a more realistic model, we effectively prune the "bad" p-threads and indirectly reduce fetch overhead.

Energy analysis. PTHSEL's built-in assumption that L2 misses are not naturally overlapped has two negative effects energy-wise. First, it leads PTHSEL to erroneously believe that all loads require latency tolerance equal to the full memory latency and to produce p-threads that are unnecessarily long and energy hungry. Second, it leads PTHSEL to erroneously believe that load latency tolerance translates cycle-for-cycle with performance improvement and to produce p-threads that don't actually reduce execution latency, but of course do consume energy. PTHSEL contains this built-in assumption because it provides computational traction. In PTHSEL's latency-only world, overhead is sub-linear fetch contention, so simplifications that result in excessive overhead are tolerated. From an energy standpoint, p-thread overhead is (at least) linear and simplifications that ignore it can have undesired results.

While it may be possible to adjust the assumption, a more straightforward way of counter-acting it is to account for energy overhead explicitly. The energy breakdown shown in the right graph of Figure 2 suggests that this should be feasible. The breakdown distinguishes the following components: fetch (instruction cache and TLB), structures that are accessed by p-loads (data cache, TLB, load/store queue), the L2 cache, structures that are accessed by all p-instructions (decoder, map table, instruction window, adder/ALU, register file, and result bus), structures that are not directly affected by p-instructions (branch predictor and re-order buffer) and, finally, idle energy. Main thread accesses are solid, p-thread accesses are striped. Our experiments show that with the exception of the L2 cache, the (dynamic) energy overheads of pre-execution are linear with respect to p-instruction count. At the same time, the idle energy benefit of pre-execution is linear with respect to execution time. These suggest that predicting the energy effects of p-threads should be easy, provided that a few energy constants are known. This observation motivates our second extension.

4 Energy-Aware P-Thread Selection

PTHSEL models pre-execution overhead as sequencing bandwidth contention between p-threads and the main program. On programs with low main thread bandwidth utilization, it considers p-threads to be almost "free", and thus is likely to choose p-threads that significantly increase the number of executed instructions. If energy is a concern, this strategy may backfire. We present two extensions to PTHSEL that improve latency energy trade-off.

4.1 Criticality-Based Load Cost

PTHSEL operates under the assumption that one cycle of load latency reduction translates into one cycle of execution time reduction. One way to manipulate PTHSEL without forcing it to explicitly model energy internally, is by supplying it with a latency-reduction to execution-time-reduction function for each static problem load.

From a critical path [9, 22] perspective, reducing the latency for a single dynamic miss is beneficial until a secondary critical path that does not include the miss is formed. Therefore, for a single instance of a problem load, the latency-reduction to execution-time reduction function is the identity (i.e., one cycle for one cycle) from zero to the point where the instruction loses criticality, and flat afterwards. Different instances of the load will have different "saturation" points, and the curve obtained by averaging over all instances will be smooth.

We supply PTHSEL with this function by extending the tool that generates the slice tree which PTHSEL subsequently analyzes. The original tool is a simple trace analyzer. We augment it with a simple critical-path model [9] that considers edges due to dataflow, branch mispredictions, in-order fetch and retirement, and a finite sized ROB. Representing and computing a dense function is expensive so the analyzer computes execution time reductions for only a few load latency reduction points: 25%, 50%, 75% and 100%; PTHSEL linearly interpolates between these.

Accounting for interactions. The interaction cost [8] of two contemporaneous L2 misses is positive—the gain of tolerating both exceeds the sum of tolerating each one individually. This cost stems from the inability of the ROB to effectively overlap such misses with non-miss work; if only one miss is covered, the other wedges the ROB.

PTHSEL is a divide-and-conquer framework that deals with one problem load at a time. Without critical path information, PTHSEL over-estimates pre-execution's capabilities in this scenario, believing that hiding each miss individually results in linear execution time reductions and that tolerating both results in additive reductions. The overall result is a consistent over-estimation of performance gains [19]. Ironically, the critical path function swings the pendulum too far the other way. Since each load individually has low criticality, PTHSEL will now believe that neither is worth a p-thread. Accounting for interactions accurately requires operations on power-sets of overlapping problem loads which is computationally infeasible. Instead, we average the pessimistic critical path function described above with a more optimistic one which is calculated assuming all contemporaneous misses are resolved. An example should help here.

Consider two L2 misses that complete in the same cycle; memory latency is 100 cycles. PTHSEL assigns each load 100 cycles of latency savings for a total of 200. That's too high. Since neither load by itself is critical, the pessimistic critical path function assigns each load 0 cycles of latency savings. That's too low. The optimistic critical path function is similar to PTHSEL—it considers all other misses to be resolved but does account for secondary critical paths—and assigns each load 90 cycles of savings (say). By averaging the pessimistic and optimistic estimates, we assign each load 45 cycles of latency savings, for a total of 90. This estimate permits PTHSEL to target both loads independently albeit less aggressively than it probably should, but maintain a reasonable estimate for global latency reduction.

4.2 Explicit Energy Model

Our final extension adds an explicit parameterized energy model to PTHSEL to allow it to select p-threads that target reductions in energy or any combination of energy and latency. The augmented framework is called PTHSEL_{+E} and uses the same basic algorithms. The extensions are a set of new formulae for p-thread energy benefit and overhead that culminate in a new p-thread evaluation function, $EADV_{agg}$, which evaluates p-threads based on their aggregate energy benefit. The formulae, shown in Table 2, are built in layers. EADV_{agg} builds on LADV_{agg}, the latency advantage computed by PTHSEL. Together LADV_{agg} and EADV_{agg} can be combined to form evaluation functions that maximize advantages of composite quantities like ED or ED^2.

In the subsequent discussion, all quantities are in units of energy (J). When multiplied, it is always by unit-less quantities like accesses or cycles (which we correlate, but not equate, with time). Energy constants are given per cycle or per access, e.g., $\text{E}_{\text{idle/c}}$ is the processor's idle energy per cycle and $\text{E}_{\text{L2/a}}$ is the energy of an L2 access.

Aggregate energy advantage (EADV_{agg}). In form, the EADV_{agg} calculation (equations E1–E3) mirrors LADV_{agg}, it is the difference between aggregate energy reduction (ERED_{agg}) and overhead (EOH_{agg}). A p-thread's energy reduction (equation E2) is proportional to its latency reduction (LADV_{agg}); the constant of proportionality is the per-cycle idle energy consumption, $\text{E}_{\text{idle/c}}$. This model fits with PTHSEL's assumption that memory latencies are not naturally overlapped with much useful computation, i.e., that the processor is idle during an L2 miss.

Similar to latency overhead (LOH_{agg}), energy overhead (equation E3) is proportional to the estimated number of p-thread spawns. Equations E4–E7 show the per dynamic p-thread energy overhead model which we divide into three components: (i) fetch, (ii) execution, and (iii) L2 access. Fetch energy is calculated for the p-thread as a whole. P-threads are sequenced from the instruction cache

Equation or Definition	Description
E1. $EADV_{agg}(p) = ERED_{agg}(p) – EOH_{agg}(p)$	**Aggregate energy advantage**
E2. $ERED_{agg}(p) = LADV_{agg}(p) * E_{idle/c}$	**Aggregate energy reduction:** is proportional to the p-thread’s aggregate latency advantage ($LADV_{agg}$).
E3. $EOH_{agg}(p) = DC_{trig}(p) * EOH(p)$	**Aggregate energy overhead:** overhead is incurred for every dynamic p-thread instance (DC_{trig})
E4. $EOH(p) = E_f(p) + E_x(p) + E_{L2}(p)$	**Per dynamic p-thread energy overhead: a combination of fetch, execution, and L2 access energy**
E5. $E_f(p) = ceil(SIZE(p)/BWSEQ_{proc}) * E_{f/a}$	**Fetch energy overhead: instruction cache access**
E6. $E_x(p) = SIZE(p)*E_{xall/a} + ALU(p)*E_{xalu/a} + LOAD(p)*E_{xload/a}$	**Execution energy overhead:** separates loads from ALU insns
E7. $E_{L2}(p) = LOAD(p) * MISSRATE_{L1}(p) * E_{L2/a}$	**L2 energy overhead:** proportional to the number of loads in the p-thread times a “global” p-thread data cache miss rate
E8. $E_{f/a}$, $E_{xall/a}$, $E_{xalu/a}$, $E_{xload/a}$, $E_{L2/a}$, $E_{idle/c}$	**External microarchitecture energy parameters:** structure access energy constants supplied by processor manufacturer or reverse engineered
C1. $CADV_{agg}(p) = [L_0^W*E_0^{1-W}]$ $– [(L_0–LADV_{agg}(p))^W*(E_0–EADV_{agg}(p))^{1-W}]$	**Aggregate composite advantage**
C2. W	**Composition weight parameter:** 0 for latency, 1 for energy, 0.5 for ED, 0.67 for ED^2
C2. E_0, L_0 or E_0/L_0	**External application parameters:** unoptimized latency and energy, or their ratio
C3. $CADV_{agg}(a+b) = [L_0^W*E_0^{1-W}]$ $– [(L_0–LADV_{agg}(a+b))^W*(E_0–EADV_{agg}(a+b))^{1-W}]$	**Adding composite advantages:** $LADV_{agg}$ and $EADV_{agg}$ can be added directly

TABLE 2. $PTHSEL_{+E}$ energy and latency-energy composition models.

in processor width ($BWSEQ_{proc}$) sized blocks at a frequency that achieves an overall bandwidth of 1 instruction per cycle. This policy simplifies p-thread scheduling and reduces instruction cache accesses. Its effect is that a p-thread consumes fetch energy proportional to $ceil(SIZE(p)/BWSEQ_{proc})$ (equation E5).

From an execution standpoint, p-instructions are split into two categories: loads and ALU instructions (p-threads contain neither stores nor branches). P-instructions execute in a lightweight mode, they are not allocated ROB or LSQ entries and are not retired. P-thread loads, however, do access the main thread LSQ to allow them to pick up values from pre-trigger stores. We assume that each p-instruction consumes renaming, instruction window, register read (or bypass), register write, and result bus energy. The model amalgamates these structures together and assumes that a single access to all of them consumes $E_{xall/a}$ energy. P-thread ALU instructions also consume ALU energy, $E_{xalu/a}$ per access. P-thread loads also consume address generation, and data cache/TLB/LSQ energy; $E_{xload/a}$ for a single access to all structures (equation E6).

The most difficult energy component to model is the L2. In the best case, a p-thread misses in the data cache and accesses the L2 once, on the target problem load. Note, in the default DDMT implementation, this L2 access is not “energy-free” as it does not replace an access that would have been performed by the main program. When targeting L2 misses, DDMT prefetches only into the L2, bypassing the L1. However, even if L1 prefetching is performed, some number of pre-executed target loads will be useless (needlessly consuming L2 energy) and these, in fact, may actually pollute the L1 and result in even more L2 accesses. The more difficult modeling problem involves embedded non-problem p-thread loads which may miss as well. We assume that embedded p-thread load misses in the L1 at the same rate as the corresponding main program load (equation E7).

The constants $E_{f/a}$, $E_{xall/a}$, $E_{xalu/a}$, $E_{xload/a}$, $E_{L2/a}$, and $E_{idle/c}$ are supplied to $PTHSEL_{+E}$ as external parameters. They can be published by the hardware vendor or reverse engineered using hardware counters [12]. For our configuration, they correspond to 9%, 4.9%, 0.8%, 3.8%, 13.6% and 5% of maximum per-cycle energy consumption.

Notice, there is no energy cost associated with p-thread spawning. DDMT forks p-threads micro-architecturally by taking a checkpoint of the physical register map table and assigning it to a free thread context. Shadow map tables are constantly updated to support branch speculation. Taking a checkpoint involves disabling updates on a shadow copy, i.e., flipping a bit. Assigning a checkpoint to a free sequencer involves writing a 3 or 4 bit value—processors support between 8 and 16 checkpoints—into a table.

Composite metrics. $PTHSEL_{+E}$’s latency and energy models can be combined to model composite quantities like ED or ED^2 and to produce p-threads that target these composites. Equations C1–C4 in Table 2 demonstrate the required extensions. The parameter W is the (exponential) weight given to latency in the composite calculation; when W is 1 $PTSHEL_{+E}$ models and optimizes latency, a W of 0 optimizes energy, to model ED and ED^2, W is set to 0.5 and 0.67, respectively.

The modeling of composites presents two previously unseen challenges. Latency and energy are “pure” quantities and their reduction can be maximized in absolute terms without regard to relative improvement. Maximizing the reduction of a composite that allows gains in one to be traded for losses in the other requires knowledge of relative latency and energy improvements, which implies knowledge of the unoptimized program's absolute latency and energy consumption. L_0 and E_0, respectively, are provided to $PTHSEL_{+E}$ as external per-application parameters. E_0 is absolute (not per-cycle) and includes idle energy. In practice, the ratio E_0/L_0 may be easier to measure or estimate than either absolute quantity individually. In that case, $PTHSEL_{+E}$ uses some reasonably large number for L_0 and that number multiplied by E_0/L_0 for E_0. A related complication is that composite advantages cannot be added. For instance, maximizing $CADV_{agg}$ of a set of p-threads is *not* tantamount to finding a set of p-threads whose $CADV_{agg}$ were maximized independently. In the same way that accurately estimating $CADV_{agg}$ requires knowledge of global quantities, it also requires knowledge of the global effects of other p-threads. We acknowledge this limitation but do not correct it, as doing so would forfeit significant computational traction.

5 Experimental Evaluation

Section 3.2 presented an energy characterization of O-p-threads, p-threads selected by the latency-only PTHSEL. We now compare these with p-threads selected by $PTHSEL_{+E}$. We also measure the response of $PTHSEL_{+E}$ to changes in several microarchitecture parameters.

5.1 Re-Targeting P-Threads with $PTHSEL_{+E}$

In this section, we measure the latency, energy, and ED characteristics of latency-oriented L-p-threads and judge our extensions by repeating the measurements for energy- and ED-oriented p-threads, respectively. In the process, we verify that re-targeting is "robust", i.e., that L-p-threads result in better latency reduction than energy- and ED-oriented ones (E-p-threads and P-p-threads), etc.

The top graph in Figure 3 shows latency (bar), energy (triangle), and ED (cross) improvements of L-, E-, and P-p-threads. Here, improvement means "reduction", a cross on the positive side of the Y axis means that ED is reduced. The second graph presents a p-thread/pre-execution characterization. We show L2 misses covered both partially (dark bars) and in full (light bars) as a fraction of the number of baseline L2 misses, the number of p-instructions executed (diamond) as a fraction of main thread instructions committed, the ratio of misses covered to p-threads spawned which we call the usefulness (cross), and average p-thread length (text). Latency and energy breakdowns are shown in the bottom two graphs.

PTHSEL latency-oriented O-p-threads. We have already discussed the shortcomings of PTHSEL and its latency oriented p-threads in the abstract. Figure 3 illustrates these shortcoming quantitatively. By failing to account for overlapping in any way, PTHSEL greedily chooses p-threads under the mistaken assumption that their lookahead will translate to performance. It consistently generates the longest p-threads, inducing the highest instruction overhead. Its ultra-aggressive p-threads do cover the most misses and do achieve full coverage for the most misses, but the profligate use of p-threads results in sub-optimal performance improvements and consistently high energy overheads. In *mcf*, overhead swamps the benefit of prefetching and yields slowdowns.

Latency-oriented L-p-threads. L-p-threads (selected by $PTHSEL_{+E}$) differ from O-p-threads (selected by PTHSEL) in their use of a criticality-based miss-cost model. By selectively throttling p-thread selection in situations where latency reduction may not translate into performance improvement, overhead is reduced and performance is actually increased. L-p-threads achieve the best latency reduction, improving performance by an average (GMean) of 16.4%, and achieve speedup for *mcf*. That the overlapping consideration is well-placed can be seen in the fact that L-p-threads cover only slightly fewer misses than O-p-threads, but reduce overhead more appreciably while doing so, e.g., *mcf*, *vpr.route*. Despite the corrected view of overhead, the weak LOH_{agg} cost model still produces p-threads that consistently increase energy consumption. While the energy cost of a dynamic p-thread is proportional to its length, its latency cost as modeled by LOH_{agg} is only the sub-linear *interference* between the p-thread and the main thread. To achieve high miss coverages, L-p-threads execute large numbers of instructions—*bzip2*'s L-p-threads increase executed instructions by 44%—and many useless instances. On average, they *increase* energy consumption by 8.7%, but increases as high as 29% (*bzip2*) are observed. However, this is still a super-linear latency/energy trade-off, or ED improvement, of 6.6%.

Energy-oriented E-p-threads. By adding a per p-instruction linear term to the cost model, $EADV_{agg}$ only selects p-threads that pay for their own energy consumption. While this reduces miss coverage (sometimes drastically), the remaining p-threads are much better behaved from an energy standpoint, executing far fewer p-instructions and spawning considerably fewer useless instances.

Unlike L-p-threads, E-p-threads either reduce (or at least don't increase) the energy consumption of most benchmarks. The very small increase in *vpr.place* (0.37%) is due to optimistic $LADV_{agg}$ estimates which translate into high $ERED_{agg}$ predictions. As expected, E-p-threads consistently achieve the highest energy reductions (0.7% on average) but also the lowest performance improvements (5.4% on average). ED improves by 5.8%.

ED-oriented P-p-threads. P-p-threads balance energy and latency concerns. The strong energy overhead filter keeps energy consumption reasonable (3% increase on average, but with higher variance than E-p-threads). At the same time, the low latency overhead filter aggressively pursues miss coverage and yields a 12.9% performance improvement. This balance produces the best ED reduction, an average of 8.8%. Miss coverage, instruction overhead and p-thread usefulness track our intuition—they all fall between the values corresponding to the L- and E-oriented p-threads.

ED^2 -oriented P2-p-threads. Voltage/frequency scaling is a circuit-level technique that provides an energy/latency trade-off characterized by constant ED^2 [16]. As shown in section 4.2, $PTHSEL_{+E}$ can be easily configured

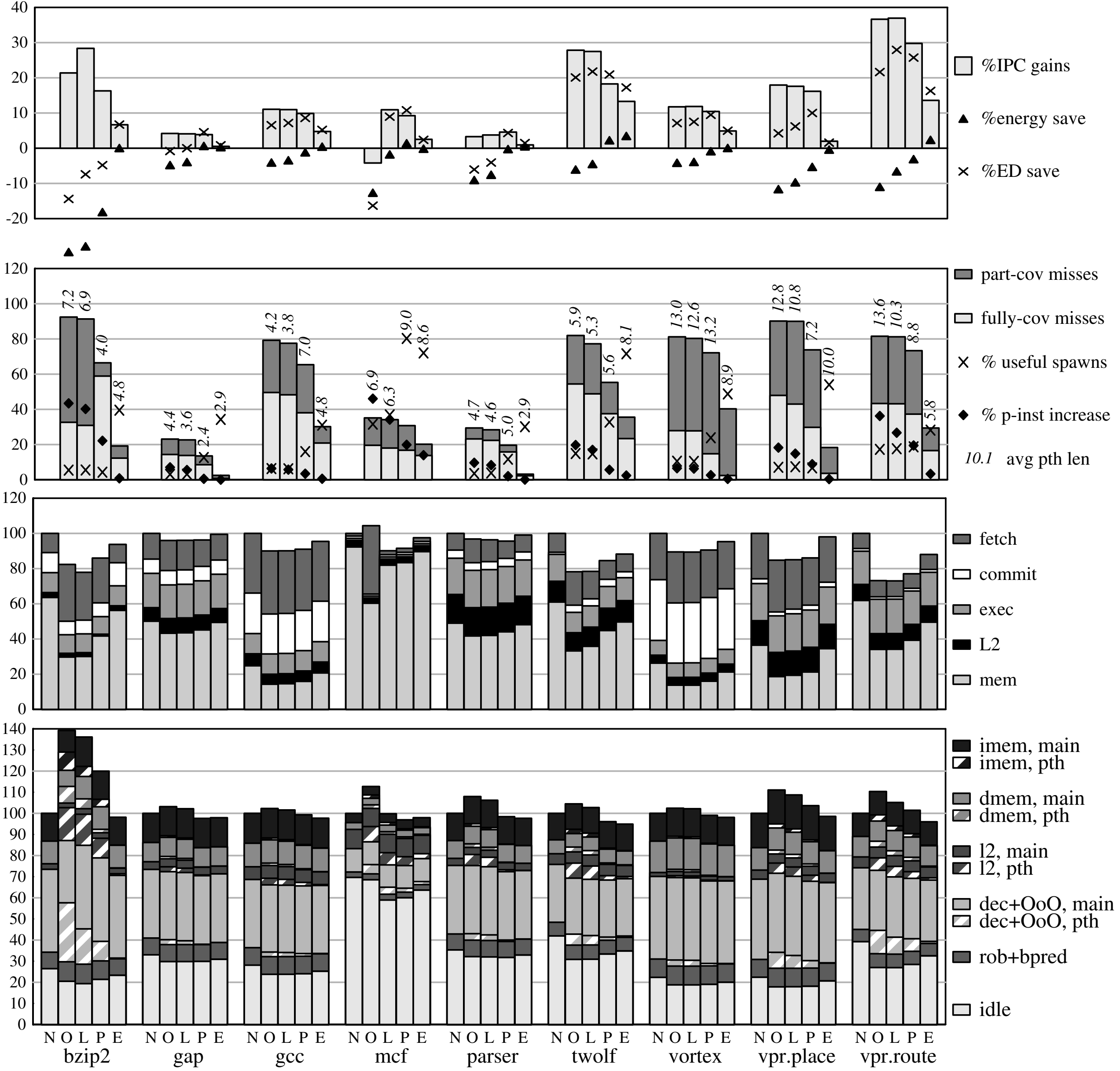


FIGURE 3. Analysis of PTHSEL$_{+E}$ driven pre-execution. P-threads targeting latency (L), energy (E), and ED (P), and PTHSEL latency p-threads (O). The top graph summarizes latency, energy, and ED improvements. The next shows pre-execution diagnostics. The bottom two graphs show execution time and energy breakdowns. Breakdowns of unoptimized execution (N) are shown for reference

to select p-threads that target reductions in ED^2 (P2-p-threads). To reduce clutter, however, our graphs do not explicitly show either relative ED^2 or P2-p-threads.

One reason we can safely make this omission is that that P2-p-threads are very similar, both structurally and behaviorally, to L-p-threads. This is intuitive. The latency benefit of L-p-threads is much larger than their energy cost. Even weighing energy and latency equally (by targeting ED) effectively emphasizes latency. ED^2 puts even more emphasis on latency reduction.

L-p-threads already provide a super-linear energy/latency trade-off and improve ED^2 by an average of 19% with all benchmarks “in the black.” Re-targeting for ED^2 only increases this gain by 1%, to 20%. By comparison criticality-based modeling of L2-miss cost yields an improvement of 6%.

Summary. Within PTHSEL$_{+E}$, the latency and energy targets are “robust”, each optimizes its respective metric: L-p-threads achieve the best performance and E-p-threads produce the best energy reduction. “Metric robustness” can be viewed as a form of model validation, when a quantity is optimized in model space the same quantity improves relatively in the real world. ED is a less robust target, achieving the best ED improvement overall but in only two thirds of the benchmarks individually. PTHSEL$_{+E}$ uses additive approximations based on independence assumptions, whereas ED is inherently a non-additive measure. However, in those cases in which ED is

	Validation Expression	gcc	parser	vortex	vpr.place
Latency Prediction	$(L_{BASE} - L_{PE}) / LADV_{agg}$	0.93	0.64	0.72	0.92
Energy Prediction	$(E_{BASE} - E_{PE}) / EADV_{agg}$	0.67	0.69	0.84	1.15
ED Prediction	$(P_{BASE} - P_{PE}) / PADV_{agg}$	1.21	0.76	0.66	0.69

TABLE 3. PTHSEL$_{+E}$ Model Validation

"properly aligned", the ordering of each target with respect to each metric is intuitive: P-p-threads reduce more latency than E-p-threads but less than L-p-threads, and reduce energy more effectively than L-p-threads but less so than E-p-threads. On average, retargeting changes metric trade-offs by 2–5%. However, on a per case basis (e.g., *bzip2*, *twolf*, *vpr*), its impact can be much larger.

5.2 Model Validation

As previously shown [19], absolute predictive accuracy in a p-thread selection framework is not necessary. From a practical standpoint, relative accuracy—the ability to adjust p-threads correctly in response to underlying parameter changes—is (much) more important. Nevertheless, in Table 3 we check PTHSEL$_{+E}$'s predictions for latency, energy, and ED reductions with actual reductions from simulated runs. Since we measure *actual* / *predicted* ratios, numbers close to 1 indicate that the estimate is accurate. Numbers under 1 are *over*-estimations. To save space, we show results for 4 benchmarks and L-p-threads. E- and P-p-threads have similar behavior.

PTHSEL systemically over-estimated LADV$_{agg}$ by as much as 60% [19]. PTHSEL$_{+E}$'s criticality based miss-cost model reduces these to 36% (e.g., an expected improvement of 30% may yield an actual improvement of 20%, not no improvement at all). Some over-estimation remains because the "non-overlapping L2 miss" assumption is inherently unchanged. LADV$_{agg}$ over-estimation translates into optimistic ERED$_{agg}$ predictions but here the non-overlapping assumption acts as a correction. Energy overhead estimations err in both directions (underestimations are due to wrong path spawning) but are within 33% relative. Since these are L-p-threads, many increase energy consumption. PADV$_{agg}$ (our name for CADV$_{agg}$ that targets ED) multiplies the inaccuracies of EADV$_{agg}$ with those of LADV$_{agg}$. Again, the improved latency model limits prediction errors to 36%.

5.3 Robustness to Profiling Data

Previous work showed that p-thread selection is stable with respect to profiling input [19]. Our primary study (Section 5.1) uses p-threads selected from ideal profiles. Figure 4 repeats that study with realistic profiling, selecting p-threads from profiles of different input (*ref*) runs.

For most benchmarks, performance, energy and ED gains with realistic p-threads are near those obtained with ideal p-threads. Generally speaking, realistic profiling degrades gain by at most 20% relative (e.g., from 16% to 13%). However, there are a few exceptions. In *bzip2*, the ref input is less memory-critical than the train input. This difference manifests when PTHSEL+E selects p-threads aggressively. L-p-threads lose 60% of their gain relative to ideal profiling. P- p-threads are affected to a lesser degree and E-p-threads are unaffected. *Twolf*'s problem is a sampling mismatch. Less aggressive sampling and the use of multiple input set profiles should alleviate both problems.

Realistic profiling also degrades metric robustness for ED. This is not surprising. PADV$_{agg}$ makes assumptions about the absolute values of E_0 and L_0. These assumptions may not hold closely in a realistic profiling scenario.

5.4 Sensitivity Analysis

We measure the response of each of PTHSEL$_{+E}$ targets—latency, energy, and ED—to changes in microarchitecture parameters. Due to space constraints, we show only three benchmarks for each experiment: two that are representative and one that is "interesting", i.e., particularly sensitive, insensitive, or counter-intuitively sensitive to the change.

Idle Energy Factor. The fraction of maximum energy structures consume when idle greatly impacts pre-execution's latency/energy trade-off. A low factor reduces overall energy consumption, but also diminishes the energy saving effects of latency reduction. A high factor increases energy consumption, but amplifies the latency-to-energy reduction lever. The top chart in figure 5 characterizes L-, E- and P-p-threads for three idle energy factors: 0% (a pro-

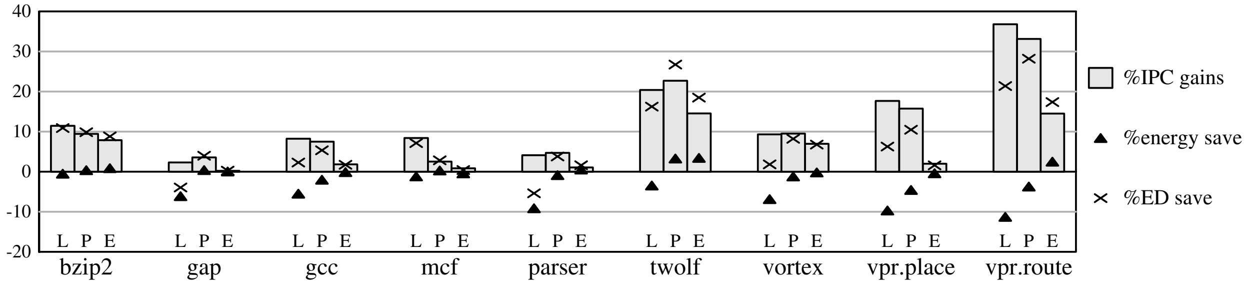


FIGURE 4. PTHSEL$_{+E}$ with realistic profiling/test inputs

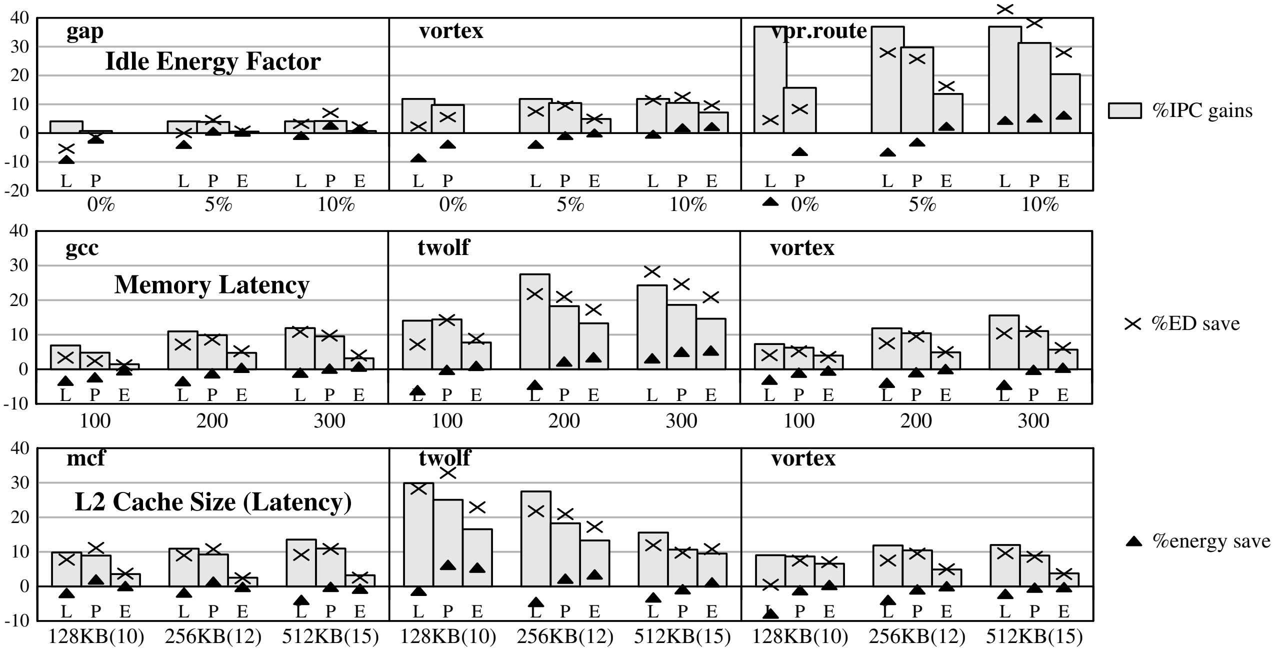


FIGURE 5. Sensitivity of PTHSEL$_{+E}$ with respect to microarchitectural parameters

cessor with no “deeper sleep”), our default 5%, and 10%.

Without a latency-to-energy reduction lever, the 0% idle-energy scenario strongly opposes pre-execution from an energy standpoint. In fact, there are no E-p-threads here since when $E_{idle/c}$ is 0, all $EADV_{agg}$ values are negative. PTHSEL$_{+E}$ can still select P-p-threads using pre-execution’s naturally high latency/energy trade-off. At 0% idle energy, L-p-threads have a strongly sub-linear latency/energy trade-off, and produce dramatic energy increases (*vpr.route*’s L-p-threads increase energy consumption by 23%). Surprisingly, retargeting PTHSEL$_{+E}$ to ED—by setting the W parameter to 0.5—restores the linear trade-off. This suggests that pre-execution can be a sensible energy-conscious choice even in this scenario, which is strongly biased against it. All p-threads—even purely latency-oriented L-p-threads—reduce ED^2 in this scenario, by an average of 8.5% (not shown). In contrast, an idle energy factor of 10% improves the energy and ED characteristics of all p-threads. L-p-threads achieve performance improvement at a lower energy cost, while both E- and P-p-threads have a longer energy reduction lever. In this scenario, pre-execution—targeted towards energy or ED—may actively be used to reduce energy. For instance, in this case E-p-threads reduce *vpr.route* energy by 7%.

Memory Latency. Longer memory latencies imply more latency per load for p-threads to tolerate. Tolerating more latency requires earlier spawns, which result in longer, higher-overhead p-threads with lower coverage. Increased memory latencies result in more idle energy consumption, but also make PTHSEL$_{+E}$’s assumption of lack of natural overlapping less aggressive. Figure 4 shows L-, E- and P-p-threads executing for three memory latencies: 100, our default 200, and 300 cycles.

Intuitively, pre-execution’s performance gains increase with growing memory latencies, but at a slower rate than those latencies themselves. The increased gains drag energy and ED gains upwards. The drag is especially strong on E- and P-p-threads. Surprisingly, longer memory latencies don’t increase p-thread length substantially and thus are more energy efficient relatively than their short-latency counterparts. Longer tolerated latencies require more induction unrolling, a fixed cost and extremely energy efficient idiom for arithmetic inductions.

L2 Cache Size/Latency. Larger L2s generate fewer misses and consume more energy per access. Intuitively, pre-execution will be more effective in smaller L2s.

The bottom graph of figure 5 shows L-, E- and P-p-threads for three L2 cache size (and corresponding hit latencies): 128KB (10 cycles), our baseline 256KB (12), and 512KB (15). The somewhat surprising result is that a smaller L2 does not necessarily imply a monotonic improvement in relative performance/energy gains. In most benchmarks (e.g., *twolf, vortex*), the dominant effect of smaller caches is increased overall latency tolerance and energy reduction. In *mcf*, the smaller L2 produces more misses, but fetch overheads of the additional p-threads overwhelm latency tolerance, reversing the trend.

6 Related Work

Recent years have seen many proposed [3, 4, 6, 7, 15, 17, 18, 21, 24, 25] and even some prototype [23] implementations of pre-execution. These have targeted both loads and branches and have used both software and hardware to generate and spawn p-threads. They have all studied pre-execution in the performance context. Here, we explore pre-execution’s latency/energy trade-offs and its energy and ED reduction properties. Our findings apply most directly to data-driven multithreading (DDMT) [18], the system with which we experimented, but can be used as rough approximations for other systems.

This work extends a previously proposed analytical framework for p-thread selection [19], augmenting it with evaluation functions that compare and select p-threads based on energy and ED characteristics. Other systems that automatically generate p-threads using a compiler [13] or binary rewriter [14] may be extended with energy awareness as well.

7 Conclusions and Future Work

Like many other techniques, pre-execution uses redundancy (translation: energy) to reduce latency. Unlike many other techniques, however, pre-execution is associated with a quantitative framework that allows its latency reduction properties to be analyzed and maximized via the selection of good p-threads. This work explores the latency/energy trade-offs of pre-execution and presents two extensions to the original framework (PTHSEL) that enable the selection of p-threads that target energy or ED reduction rather than latency reduction. Our new framework is called PTHSEL$_{+E}$.

Experiments with a modified SimpleScalar/Wattch simulator and the SPEC2000 integer benchmarks show that with PTHSEL$_{+E}$, latency-oriented pre-execution has a super-linear latency/energy trade-off. It reduces execution time by an average of 16.4% for those benchmarks that suffer from a large number of L2 misses while increasing energy consumption by 8.7%, an ED improvement of 6.5%. The latency/energy lever is due to increasing sparsity of data dependences proceeding backwards from problem loads. Latency—and via a "deep sleep" mode, energy—can often be substantially reduced at the cost of a single p-instruction. Retargeting PTHSEL$_{+E}$ to produce p-threads that reduce energy achieves a performance improvement of only 5.3%, but reduces total energy consumption. ED-targeted p-threads strike a good balance, achieving a latency reduction of 12.9% at a modest energy increase, for an ED improvement of 8.8%.

Pre-execution's latency/energy lever depends on the idle energy factor, the fraction of peak per-cycle energy the processor consumes when idle for short stretches and which can only be saved using drastic, longer-timescale measures. A high factor makes marginal p-instruction energy cheap, latency reduction translate to energy reduction more effectively, and pre-execution more attractive. Our experiments use a conservative factor of 5%. At 10%, pre-execution can be used as an energy reduction tool.

Although we do not explicitly evaluate branch pre-execution [5, 18, 25], it is potentially an effective energy-reduction technique. Our extensions should apply to branch p-threads with one modification. When targeting L2 misses, we assume that the processor would have been idle during saved cycles, such that energy is saved at a rate of $E_{idle/c}$. When targeting branches, we should assume that the processor would have been typically busy during saved cycles and that energy is saved at a rate of $E_{total/c}$.

Acknowledgements

We thank the reviewers for their feedback. This work was supported by NSF CAREER Award CCF-0238203.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Framework for Architectural Level Power Analysis and Optimizations." In *ISCA-27*, Jun. 2000.

[2] D. Burger and T. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

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

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

[5] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. "Microarchitectural Support for Pre-Computation Microthreads." In *MICRO-35*, Nov. 2002.

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

[7] 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 *ISCA-28*, Jul. 2001.

[8] B. Fields, R. Bodik, M. Hill, and C. Newburn. "Using Interaction Costs for Microarchitectural Bottleneck Analysis." In *MICRO-36*, Dec. 2003.

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

[10] R. Gonzalez and M. Horowitz. "Energy Dissipation in General Purpose Microprocessors." *IEEE Journal of Solid-State Circuits*, 31(9), Sep. 1996.

[11] Intel Corporation. *Mobile Intel Pentium 4 M-Processor Datasheet*, Jun. 2003. http://www.intel.com/design/mobile/datashts/250686.htm.

[12] R. Joseph and M. Martonosi. "Run-Time Power Estimation in High Performance Microprocessors." In *ISLPED-01*, Aug. 2001.

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

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

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

[16] A. Martin, M. Nystroem, and P. Penzes. "ET2: A Metric for Time and Energy Efficiency of Computation." Technical Report CSTR:2001.007, CalTech, 2001.

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

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

[19] A. Roth and G. Sohi. "A Quantitative Framework for Pre-Execution Thread Selection." In *MICRO-35*, Nov. 2002.

[20] P. Shivakumar and N. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model." Technical report, COMPAQ Western Research Laboratory, 2001.

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

[22] S. Srinivasan and A. Lebeck. "Load Latency Tolerance in Dynamically Scheduled Processors." In *MICRO-31*, Nov. 1998.

[23] P. Wang, J. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. Yunus, T. Sych, and J. Shen. "Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Machine." In *ASPLOS-XI*, Oct. 2004.

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

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