

Learning-Based SMT Processor Resource Distribution via Hill-Climbing

Seungryul Choi
Department of Computer Science
University of Maryland
choi@cs.umd.edu

Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland
yeung@eng.umd.edu

Abstract

The key to high performance in Simultaneous Multithreaded (SMT) processors lies in optimizing the distribution of shared resources to active threads. Existing resource distribution techniques optimize performance only indirectly. They infer potential performance bottlenecks by observing indicators, like instruction occupancy or cache miss counts, and take actions to try to alleviate them. While the corrective actions are designed to improve performance, their actual performance impact is not known since end performance is never monitored. Consequently, potential performance gains are lost whenever the corrective actions do not effectively address the actual bottlenecks occurring in the pipeline.

We propose a different approach to SMT resource distribution that optimizes end performance directly. Our approach observes the impact that resource distribution decisions have on performance at runtime, and feeds this information back to the resource distribution mechanisms to improve future decisions. By evaluating many different resource distributions, our approach tries to learn the best distribution over time. Because we perform learning on-line, learning time is crucial. We develop a hill-climbing algorithm that efficiently learns the best distribution of resources by following the performance gradient within the resource distribution space.

This paper conducts an in-depth investigation of learning-based SMT resource distribution. First, we compare existing resource distribution techniques to an ideal learning-based technique that performs learning off-line. This limit study shows learning-based techniques can provide up to 19.2% gain over ICOUNT, 18.0% gain over FLUSH, and 7.6% gain over DCRA across 21 multithreaded workloads. Then, we present an on-line learning algorithm based on hill-climbing. Our evaluation shows hill-climbing provides a 12.4% gain over ICOUNT, 11.3% gain over FLUSH, and 2.4% gain over DCRA across a larger set of 42 multiprogrammed workloads.

1. Introduction

Simultaneous Multithreading (SMT) is an important architectural technique, as evidenced by the widespread attention it has

This research was supported in part by NSF CAREER Award #CCR-0000988, by DARPA AFRL grant #F30602-01-C-0171, and by DARPA grant #NBCH1050022. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

received from academia [2, 4, 20, 19, 14], and by industry's willingness to incorporate it into commercial processors [7, 6]. Given the continued importance of chip-level multithreading, research that improves SMT performance without increasing its power consumption will remain highly relevant in future systems.

The key to high performance in SMT processors lies in optimizing the distribution of resources to simultaneously executing threads. Several resource distribution techniques have been studied in the past [2, 4, 20, 19, 14]. One shortcoming of these previous techniques is they optimize performance only *indirectly*. As illustrated in Figure 1a, existing techniques make resource distribution decisions based on hardware monitors of per-thread resource usage (e.g., instruction occupancy or cache miss counts); the hardware monitors do not reflect actual performance. From this resource usage information, the resource distribution mechanisms infer potential performance bottlenecks and take actions to try to alleviate them (e.g., stop fetching a thread that has consumed too many resources, or flush a thread that has suffered a cache miss). While these actions are designed to improve performance, their actual performance impact is not known since the resource distribution mechanisms never directly monitor end performance.

Because resource distribution mechanisms optimize performance only indirectly, opportunities for performance gains may be missed for two reasons. First, resource distribution mechanisms are designed to target a small set of important performance bottlenecks; however, SMT processors exhibit a myriad of behaviors that are highly sensitive to workload mix. Existing resource distribution mechanisms cannot possibly anticipate all bottlenecks for all workloads, missing performance opportunities in some cases. Second, resource distribution mechanisms are designed to improve performance in general, but they are not designed to be optimal for any specific case. Hence, even for the anticipated performance bottlenecks, further performance gains might still be possible.

We propose a different approach to SMT resource distribution that optimizes end performance *directly*. Our approach observes the impact that resource distribution decisions have on performance at runtime and feeds this information back to the resource distribution mechanisms to improve future decisions, as illustrated in Figure 1b. By successively applying and evaluating different resource distributions, our approach tries to *learn* the best distribution over time. Learning is performed continuously to adapt whenever the workload's resource needs change. Because our approach learns based on actual performance, the resource distribution decisions it makes are customized to the performance bottlenecks of the workload, reducing missed performance opportunities. Moreover, whenever learning for a particular workload behavior succeeds, our approach finds the best resource distribution for that behavior. Our approach can also optimize for a specific performance goal (e.g., throughput, speedup, or fairness) by simply using the

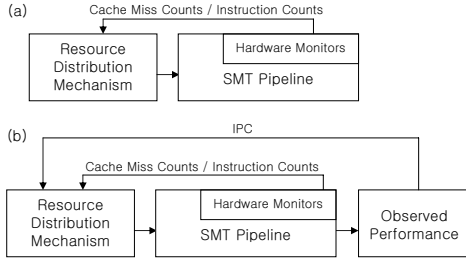


Figure 1. (a) Existing resource distribution techniques optimize performance indirectly by making decisions based on hardware monitors only. (b) Learning-based resource distribution examines actual performance to learn the best resource distribution.

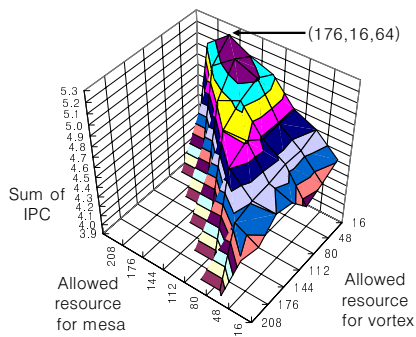


Figure 2. IPC of mesa, vortex, and fma3d during a 32K-cycle time interval as the fraction of resources distributed to each thread is varied. The X- and Y-axes show the resource distribution for mesa and vortex (fma3d receives the remaining resources). The arrow indicates the resource distribution with peak performance.

appropriate performance metric for feedback.

Since we perform learning on-line, learning time is crucial to the success of our approach. A key observation enabling fast learning is that performance does not change randomly as a function of resource distribution; instead, the performance sensitivity is often “hill-shaped.” For example, Figure 2 shows the performance of three applications—mesa, vortex, and fma3d—running simultaneously on an SMT processor during a time interval of 32K cycles. The graph plots IPC as the fraction of resources distributed to individual threads is varied. In the figure, performance follows a well-defined hill shape, with a clear performance peak. From our experience, many workloads exhibit such hill-shaped behavior. We exploit this behavior by using a *hill-climbing* algorithm to learn the best resource distribution. Because learning is guided by the slope of the hill, our hill-climbing algorithm reaches the best resource distribution after sampling only a small portion of the resource distribution space, thus leading to low learning times.

This paper investigates SMT resource distribution techniques that use hill-climbing to learn the best resource share. Specifically, we apply learning to dynamically distribute key SMT processor hardware structures across simultaneously executing threads. Our study begins by comparing an ideal off-line learning algorithm against existing techniques to quantify the best performance

improvements that learning-based techniques can achieve. Our limit study reveals an ideal learning-based technique outperforms ICOUNT [20] by 19.2%, FLUSH [19] by 18.0%, and DCRA [2] by 7.6% on 21 multiprogrammed workloads, demonstrating there is significant room for learning to improve performance. Next, we present the hill-climbing algorithm for performing learning on-line. Our evaluation reveals hill-climbing provides an 12.4%, 11.3%, and 2.4% performance boost over ICOUNT, FLUSH, and DCRA, respectively, on a comprehensive set of 42 multiprogrammed workloads. Finally, we extend hill-climbing to perform learning based on program phases, and report our preliminary experience with this phase-based technique.

The rest of this paper is organized as follows. Section 2 discusses related work, and Section 3 studies the ideal learning-based technique. Next, Section 4 presents and evaluates our hill-climbing resource distribution technique, and Section 5 describes its extension. Finally, Section 6 concludes the paper.

2. Related Work

Prior research has tried to boost SMT processor performance by improving the distribution of hardware resources to threads. One important approach is to optimize the selection of threads to fetch every cycle. ICOUNT [20] and FPG [10] are examples of such *SMT fetch policies*. These techniques monitor indicators of resource usage, such as resource occupancy (ICOUNT) or branch prediction accuracy (FPG). Every cycle, the threads using their resources most efficiently (*e.g.*, with low occupancy or few branch miss-predicts) are given fetch priority. By favoring fast threads, ICOUNT and FPG increase overall throughput.

Unfortunately, fetch policies do not effectively handle long-latency operations, especially cache-missing loads. Once a thread suffers a long-latency cache-missing load, continuing to fetch the thread clogs the pipeline with stalled instructions, preventing other threads that would otherwise gainfully use the resources from receiving them. Fetch policies like ICOUNT reduce, but do not stop, the fetch of stalled threads, so they cannot prevent resource clog. Several techniques address resource clog by explicitly limiting resource distribution to threads with long-latency memory operations. The first approach is to fetch-lock stalled threads. Techniques in this category differ in how they detect the stall condition. STALL [19] triggers fetch-lock when a load remains outstanding beyond some threshold number of cycles; DG [4] triggers fetch-lock when the number of cache-missing loads exceeds some threshold; and PDG [4] uses a cache-miss predictor to trigger fetch-lock.

One problem with fetch-locking is resource clog can still occur because the stall condition is detected either too late or unreliably. Instead of anticipating resource clog and fetch-locking, a second approach is to allow resource clog to occur but immediately recover by flushing the stalled instructions. This is the approach taken by FLUSH [19]. FLUSH is effective in preventing resource clog; however, flushing is wasteful in terms of fetch bandwidth and power consumption. Hybrid approaches (*e.g.*, STALL-FLUSH [19]) minimize the number of flushed instructions by first employing fetch-lock, and resorting to flushing only when resources are exhausted.

A third approach is to partition the processor resources. The simplest is static partitioning [5, 13, 14], but these techniques cannot adapt to changing workload behavior. In contrast, DCRA [2] partitions dynamically based on memory performance. Threads with frequent L1 cache misses are given large partitions, allowing them to exploit parallelism beyond stalled memory opera-

tions. Threads that cache-miss infrequently are guaranteed some resource share since stalled threads are not allowed beyond their partitions. Hence, DCRA prevents resource clog by containing stalled threads. Moreover, DCRA computes partitions based on the threads’ anticipated resource needs, increasing distribution to the threads that can use resources most efficiently.

Compared to previous techniques, learning-based SMT resource distribution is most similar to DCRA. Like DCRA, our approach also uses dynamic partitioning to address resource clog and improve resource usage efficiency. However, a key distinction is learning-based SMT resource distribution makes partitioning decisions based on performance feedback, thus optimizing end performance. In contrast, DCRA and other previous techniques perform resource distribution based on hardware monitors like instruction and cache miss counts. Hence, they optimize performance only indirectly, potentially missing opportunities for performance gains as discussed in Section 1. Exploiting performance feedback also permits optimization to a user-definable performance goal—like throughput, per-thread speedup, or fairness—by simply changing the performance metric used to drive learning. Previous techniques cannot tailor their optimizations to a specific performance goal. Because it takes time for our learning algorithm to process performance feedback, we update partitioning decisions periodically. Thus, our technique lies somewhere in between DCRA (update every cycle) and static partitioning (fixed) in terms of its responsiveness to dynamic runtime behavior.

Finally, our approach borrows from program phase analysis [16, 17]. Like these techniques, our approach breaks program execution into sequences of fixed-size epochs to facilitate performance analysis and feedback for runtime optimization. In particular, Dynamic Back-end Assignment (DBA) [9] uses epoch-based feedback to drive partitioning of clustered multithreaded processors. Like DBA, we also perform partitioning based on performance feedback; however, we control partitioning at a much finer granularity (per resource entry instead of per cluster), and we design and evaluate a detailed algorithm for performing partitioning in an on-line fashion.

3. Limits of Learning-Based SMT Resource Distributions

We begin our investigation with a limit study. To facilitate the study, this section develops an ideal learning algorithm that determines a pseudo-optimal resource distribution off-line via exhaustive search. Our off-line learning algorithm incurs zero overhead for computing the resource distributions.

3.1. Off-Line Exhaustive Learning

All of the SMT resource distribution techniques studied in this paper perform learning based on *phases*, an approach borrowed from existing phase detection and prediction techniques [16, 17]. We divide SMT execution into a linear sequence of *epochs* or fixed-size time intervals. For each epoch, the resource distribution mechanism specifies a partitioning of select shared processor resources across the simultaneous threads. During epoch execution, threads are allowed to consume up to (but no more than) the allotted resources within their partition. Hence, partitioning guarantees every thread receives some fraction of each shared resource.

Normally, the resource distribution mechanism decides the partitioning for each epoch based on performance feedback acquired via processor statistics counters from previously executed epochs.

In contrast, our off-line exhaustive learning algorithm decides the partitioning based on performance feedback from the *currently executing epoch*. At the beginning of each epoch, we execute the epoch once for every possible partitioning of the shared resources. Amongst these exhaustive trials, we select the trial with the highest measured performance, and advance the machine state accordingly. The execution time of the best trial is charged to execution time while the cost of sampling all other trials are ignored, and then the process is repeated for subsequent epochs. Although such off-line learning is impractical for real machines, its evaluation via simulation yields insights into the performance of learning-based SMT resource distribution.

Unfortunately, simulating off-line exhaustive learning is computationally expensive because of the exhaustive trials. Due to excessive simulation times, we are able to study off-line learning for SMT processors with 2 hardware contexts only. However, the insights derived from our study carry over to larger SMT machines. Later in Section 4.4, we will evaluate learning-based techniques on SMT processors with more hardware contexts. In the next two sections, we address several design issues pertaining to epochs and hardware resource partitioning that impact the performance of off-line exhaustive learning (as well as hill-climbing).

3.1.1. Epochs

Epoch Size. Epoch size, measured in processor cycles, is an important parameter for any phase-based technique because it affects adaptivity. If epoch size is too large, then learning may not adapt quickly enough to changes in the workload’s resource demands. If epoch size is too small, then inter-epoch behavior may become too dynamic, making learning difficult. We ran several experiments to measure the sensitivity of performance to epoch size assuming our hill-climbing algorithm, which we will present in Section 4. Based on these experiments, we found a 64K-cycle epoch size consistently yields good performance. Smaller epochs are likely to provide higher performance for off-line exhaustive learning since dynamic inter-epoch behavior is not a concern when learning off-line. However, for our limit study, we are interested in characterizing the limits of *on-line* learning algorithms, so we use a 64K-cycle epoch size for all of the experiments in this section.

Epoch Performance. Learning-based SMT resource distribution uses performance feedback to make resource partitioning decisions that optimize end performance. An important question is what performance metric should we choose to drive the learning algorithms? In the past, three performance metrics have been used to characterize SMT performance: average IPC, average weighted IPC [18], and harmonic mean of weighted IPC [11]. These metrics are defined below, where IPC_i is the IPC of the i th thread in the SMT machine, $SingleIPC_i$ is the IPC of the stand-alone execution of the i th thread, and T is the number of threads.

$$\text{Avg_IPC} = \frac{\sum IPC_i}{T} \quad (1)$$

$$\text{Avg_Weighted_IPC} = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{T} \quad (2)$$

$$\text{Harmonic_Mean_of_IPC} = \frac{T}{\sum \frac{SingleIPC_i}{IPC_i}} \quad (3)$$

Each metric reflects a different performance goal. Average IPC quantifies throughput improvement; average weighted IPC quantifies execution time reduction; and harmonic mean of weighted IPC quantifies both performance improvement and fairness. For evaluating off-line exhaustive learning, we will use the average

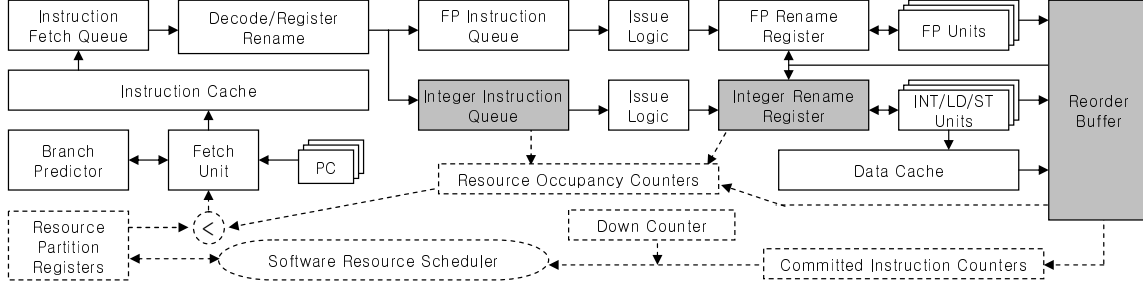


Figure 3. Block-level diagram of our SMT processor model. Shaded boxes indicate shared hardware structures that are partitioned by learning-based resource distribution. Dotted boxes indicate additional hardware needed for our hill-climbing algorithm, presented in Section 4.

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	32-IFQ, 80-Int IQ, 80-FP IQ, 256-LSQ
Rename reg / ROB	256-Int, 256-FP / 512 entry
Functional unit	6-Int Add, 3-Int Mul/Div, 4-Mem Port 3-FP Add, 3-FP Mul/Div
Branch Predictor Parameters	
Branch predictor	Hybrid 8192-entry gshare/2048-entry Bimod
Meta table/BTB/RAS	8192 / 2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
DL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
UL2 config	1Mbyte, 64byte block, 4 way, 20 cycle lat
Mem config	300 cycle first chunk, 6 cycle inter chunk

Table 1. SMT simulator settings.

weighted IPC metric. Later, when we evaluate hill-climbing, we will use all three performance metrics.

3.1.2. Hardware Resource Partitioning

This paper applies learning-based resource distribution to a detailed SMT processor model. Figure 3 illustrates the processor we assume. Like other techniques that explicitly control resource distribution (e.g., DCRA), we dynamically partition several shared hardware resources in the SMT pipeline that significantly impact performance. Specifically, we target the integer issue queue (IQ), integer rename registers, and reorder buffer (ROB),¹ which are shaded gray in Figure 3. In addition to controlling these structures, distributing fetch bandwidth is also crucial to SMT performance. Unfortunately, it is infeasible to partition fetch using learning-based resource distribution due to the high frequency in which partitioning decisions must be made. Hence, we rely on the ICOUNT fetch policy [20] to distribute fetch bandwidth across threads.

One problem with applying learning to all of the shaded structures in Figure 3 is the resource distribution space becomes intractably large. Given S shared structures, E_i entries for structure i , and T threads, the number of unique ways to distribute the resources is $\prod_{i=1}^S E_i^{(T-1)}$. Off-line exhaustive learning must try all of these unique cases for every epoch. To reduce the search space, we observe that a thread’s usage of different hardware resources is *not* independent; instead, the number of entries of each resource

¹Most SMT processors implement private ROB’s to simplify per-thread commit. We assume a shared ROB to be consistent with DCRA [2]. Our approach would still work for private ROB’s—we would ignore the ROB’s, and partition the remaining shared resources only.

type a thread occupies is often related. (For example, a thread can never use more rename registers than the number of ROB entries it holds). Hence, many cases do not need to be explored. We exploit this observation in two ways. First, we assume the number of integer IQ entries, integer rename registers, and ROB entries occupied by a thread are *in proportion to one another*. Rather than partition every resource independently, our learning algorithm partitions a single resource only, and then applies the same partition proportionally to all other resources. Second, we do not explicitly partition the floating point IQ and rename registers. By partitioning the integer IQ, integer rename register, and ROB, we indirectly control how many floating point resources each thread consumes, making learning for these resources less critical.

These simplifications reduce the number of unique resource distributions to $E_{max}^{(T-1)}$, where $E_{max} = \max_{i=1}^S (E_i)$, making off-line exhaustive learning significantly more tractable. However, the resource distribution space is still very large, especially for large T . Hence, we constrain our study of off-line exhaustive learning to SMT machines with 2 hardware contexts.

3.2. Experimental Methodology

We conduct a limit study of learning-based SMT resource distribution using our off-line exhaustive learning algorithm. Our experiments are performed on a detailed event-driven simulator of an SMT processor that models the pipeline illustrated in Figure 3. The simulator is derived from sim-ssmt [12], an extension of the out-of-order processor model in SimpleScalar [1], and has been used previously to study SMT techniques [3, 8]. For our evaluation, we model an 8-way issue SMT processor with 2 hardware contexts and a 512-entry reorder buffer. The processor and memory system settings for our simulations are listed in Table 1.

We extended sim-ssmt to support dynamic partitioning of the integer IQ, integer rename registers, and ROB. We keep a per-thread count of the entries occupied in each resource, and allow a thread to fetch instructions as long as it hasn’t exceeded its partition limit in any resource. If any resources become exhausted, the corresponding thread is fetch-locked until it releases some of its entries in the exhausted partition(s). In addition to resource partitioning, we also use ICOUNT to select the threads from which to fetch every cycle.

To implement off-line exhaustive learning from Section 3.1, our simulator checkpoints every processor memory structure (register file, pipeline registers, branch predictors, caches, etc.) as well as main memory at the beginning of each epoch. Then, we run a simulation for every partitioning of the 256 integer rename register across 2 threads (the integer IQ and ROB partitions are set in

App	Skip	Rsc	Freq	Type		App	Skip	Rsc	Freq	Type		App	Skip	Rsc	Freq	Type	
bzip2	1.1B	72	No	Int	ILP	perlbnk	1.7B	59	No	Int	ILP	eon	0.1B	82	No	Int	ILP
vortex	0.1B	102	High	Int	ILP	gzip	0.2B	83	High	Int	ILP	parser	1.0B	90	High	Int	ILP
gap	0.2B	208	No	Int	ILP	crafty	0.5B	125	High	Int	ILP	gcc	2.1B	112	High	Int	ILP
apsi	2.3B	127	No	FP	ILP	fma3d	1.9B	72	No	FP	ILP	wupwise	3.4B	161	No	FP	ILP
mesa	0.5B	110	No	FP	ILP	quake	0.4B	100	No	FP	MEM	vpr	0.3B	180	High	Int	MEM
mcf	2.1B	97	Low	Int	MEM	twolf	2.0B	184	High	Int	MEM	art	0.2B	176	No	FP	MEM
lucas	0.8B	64	No	FP	MEM	ammp	2.6B	173	High	FP	MEM	swim	0.4B	213	No	FP	MEM
applu	0.8B	112	No	FP	MEM												

Table 2. SPEC CPU2000 benchmarks used to create our multiprogrammed workloads.

proportion to the integer rename register partition). To save simulation time, we only try every other possible partitioning, reducing the number of exhaustive trials to 127 per epoch. Each simulation starts from the checkpoint and lasts for 64K cycles, the epoch size. After the exhaustive trials complete, we run one final simulation using the best partitioning to advance to the next epoch. The best partitioning is chosen using the weighted IPC metric, Equation (2) from Section 3.1.1. In addition to off-line exhaustive learning, our simulator also models the ICOUNT, FLUSH, and DCRA policies to facilitate a comparison against existing techniques.

Our experiments are driven by 42 multiprogrammed workloads created from 22 SPEC CPU2000 benchmarks. Table 2 lists our benchmarks. We use the pre-compiled alpha binaries from Chris Weaver² which are built with the highest level of compiler optimization. All of our benchmarks use the reference inputs. From the benchmarks, we created multiprogrammed workloads by following the methodology in [2, 19]. We first categorized the SPEC benchmarks into either high-ILP or memory-intensive programs, labeled “ILP” and “MEM,” respectively, in Table 2. Then, we created 3 groups of 2-thread and 3 groups of 4-thread workloads. Table 3 lists our multiprogrammed workloads. The ILP2 and ILP4 workloads group high-ILP benchmarks; the MEM2 and MEM4 workloads group memory-intensive benchmarks; and the MIX2 and MIX4 workloads group both high-ILP and memory-intensive benchmarks. Since we simulate only 2 hardware contexts for off-line exhaustive learning, we use the 2-thread workloads from Table 3. Later in Section 4.4, when we evaluate hill-climbing, we will also use the 4-thread workloads as well.

We selected simulation regions for our multithreaded workloads in the following way. First, we used SimPoint [15] to analyze the first 16 billion instructions (or the entire execution) of each benchmark, and picked the earliest representative region reported by SimPoint. In our SMT simulations, we fast-forward each benchmark in the multithreaded workload to its representative region. Table 2 reports the number of skipped instructions in each benchmark during fast forwarding. Finally, we turn on detailed multithreaded simulation, and simulate for 100 million “on-line” instructions (*i.e.*, not including the “off-line” exhaustive trials needed to find the best partitionings). Due to the cost of simulating the exhaustive trials, we are unable to simulate more instructions for our limit study; however, the regions we simulate are representative thanks to the SimPoint analysis. When evaluating on-line learning in Section 4.4, we will use larger simulation regions of 1 billion instructions. (The “Rsc” and “Freq” columns in Tables 2 and 3 will be discussed in Section 4.4.2).

3.3. Off-Line Learning Results

Figure 4 compares off-line exhaustive learning (labeled “OFF-LINE”) against ICOUNT, FLUSH, and DCRA. The figure plots weighted IPC versus different resource distribution techniques applied to the 2-thread multiprogrammed workloads. Comparing

²These SPEC CPU2000 alpha binaries are available at the SimpleScalar website.

App	Rsc	App	Rsc
ILP			
apsi eon	209	apsi eon fma3d gcc	392
fma3d gcc	184	apsi eon gzip vortex	393
gzip vortex	184	fma3d gcc gzip vortex	368
gzip bzip2	155	gzip bzip2 eon gcc	349
wupwise gcc	273	mesa gzip fma3d bzip2	337
fma3d mesa	182	crafty fma3d apsi vortex	425
apsi gcc	239	apsi gap wupwise perlbnk	555
MIX			
applu vortex	214	ammp applu apsi eon	493
art gzip	259	art mcf fma3d gcc	457
wupwise twolf	345	swim twolf gzip vortex	581
lucas crafty	189	gzip twolf bzip2 mcf	436
mcf eon	179	mcf mesa lucas gzip	354
twolf apsi	310	art gap twolf crafty	693
quake bzip2	173	swim fma3d vpr bzip2	536
MEM			
applu ammp	285	ammp applu art mcf	558
art mcf	274	art mcf swim twolf	670
swim twolf	396	ammp applu swim twolf	681
mcf twolf	281	mcf twolf vpr parser	550
art vpr	356	art twolf quake mcf	558
art twolf	360	quake parser mcf lucas	351
swim mcf	310	art mcf vpr swim	666

Table 3. Multiprogrammed workloads used in the experiments.

OFF-LINE and DCRA, we see OFF-LINE outperforms DCRA in all but two workloads (quake-bzip2 and applu-ammp), providing a performance gain of 7.6% on average. Comparing OFF-LINE, FLUSH, and ICOUNT, we see OFF-LINE outperforms ICOUNT and FLUSH in all 21 workloads, providing an average performance gain of 19.2% and 18.0%, respectively. The largest performance gains are achieved for the MEM workloads where OFF-LINE outperforms ICOUNT, FLUSH, and DCRA by 21.9%, 39.4%, and 13.2%, respectively. Figure 4 demonstrates there is significant headroom for learning-based SMT resource distribution to improve performance over existing techniques. Moreover, this performance potential exists across a wide range of workloads.

Because each multithreaded workload in Figure 4 executes at a different rate across the different techniques, it is hard to compare them directly. To facilitate a more thorough side-by-side comparison, we “synchronized” all the techniques by exploiting the checkpoints acquired in OFF-LINE for the exhaustive trials. At the beginning of every epoch, we simulate ICOUNT, FLUSH, and DCRA for 64K cycles starting from the same checkpoint used by OFF-LINE, and record the resulting IPCs. This yields a time-varying performance profile for each technique, as illustrated in Figure 5. Comparing IPCs from the same epoch in Figure 5 is meaningful because all the techniques are synchronized to a common execution point. (We also verified that synchronization does not noticeably alter the end-to-end performance of ICOUNT, FLUSH, and DCRA compared to Figure 4).

We performed synchronized versions of the experiments in Figure 4, and compared time-varying performance across differ-

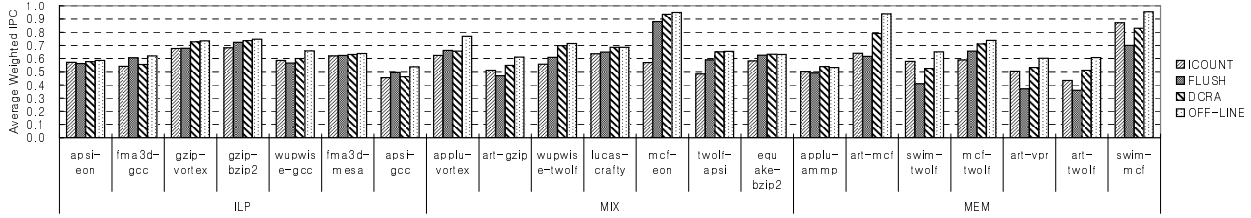


Figure 4. Comparison of off-line exhaustive learning against ICOUNT, FLUSH, and DCRA.

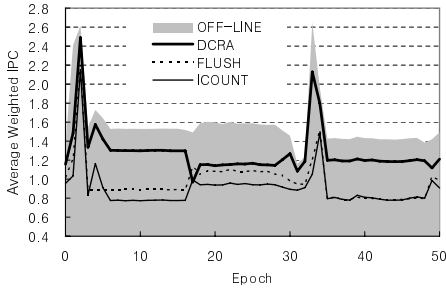


Figure 5. Synchronized time-varying performance of OFF-LINE, DCRA, FLUSH, and ICOUNT from the art-mcf workload.

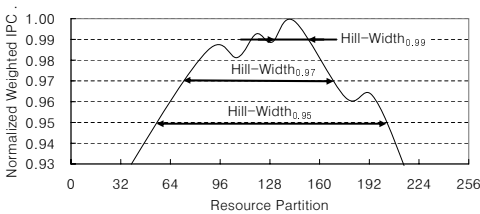


Figure 6. Variation of IPC versus partitioning in a hypothetical epoch. Three hill-width_N values are indicated on the curve.

ent techniques. For all 21 workloads, OFF-LINE outperforms ICOUNT and FLUSH in 100% of the epochs. OFF-LINE also outperforms DCRA in 97.2% of the epochs averaged across all the workloads. OFF-LINE is effective all the time. Even though OFF-LINE uses a fixed resource partitioning over each 64K-cycle epoch (the other techniques update resource distribution decisions every cycle), it still achieves higher performance in practically every epoch. These results show phase-based learning is very general, and has the potential to consistently make higher quality resource distribution decisions compared to existing techniques.

3.3.1. Hill Peak Analysis

In this and the following section, we investigate the source of OFF-LINE’s performance gains. We begin by studying performance sensitivity inside individual epochs. Since OFF-LINE exhaustively searches over all resource partitionings, we not only know the best partitioning, but we also know exactly how performance varies with partitioning for every epoch. Figure 6 shows this relationship for a hypothetical epoch, plotting IPC (normalized to the maximum IPC) as a function of different partitionings of the integer rename registers. As illustrated in Figure 6, the performance variation is typically hill shaped, with one or more peaks. (Notice the maximal peak may not occur at the middle of the resource

partition space). Insights can be gained by quantifying the “sharpness” of the performance peak containing the best partitioning. We define *hill-width_N* to be the width of the hill containing the maximal peak at some performance level, *N*. In Figure 6, we indicate *hill-width_N* for *N* = 0.95, 0.97, and 0.99. Peak sharpness can be assessed by examining *hill-width_N* across different *N*: a small *hill-width_N* value for a small *N* indicates a sharp peak, while a large *hill-width_N* value for a large *N* indicates a dull peak.

Figure 7 reports *hill-width_N* across several *N* (between 0.99 and 0.90) for our 2-thread multiprogrammed workloads; each bar represents a *hill-width_N* value averaged across all epochs from its corresponding workload. In Figure 7, we see 5 workloads (quake-bzip2, mcf-eon, fma3d-mesa, gzip-bzip2, and lucas-crafty) exhibit very dull peaks. For these workloads, partitionings that deviate by 32–64 registers or $\frac{1}{8}$ - $\frac{1}{4}$ th of the total integer rename registers away from the best partitioning still achieve 99% of peak performance (*hill-width_{0.99}* > 32), and partitionings that deviate by roughly 96 registers or $\frac{3}{8}$ th of the integer rename registers away from the best still achieve 98% of peak performance (*hill-width_{0.98}* ≈ 96). Due to dull peaks, these 5 workloads are insensitive to non-optimal partitionings. Hence, as illustrated in Figure 4, OFF-LINE achieves comparable performance to existing techniques (e.g., DCRA) in these workloads because there is very little performance advantage for learning the best partitioning. Two other workloads (gzip-vortex and apsi-eon) exhibit moderately dull peaks, and demonstrate similarly small OFF-LINE performance gains in Figure 4.

In contrast, the remaining 14 workloads in Figure 7 exhibit sharp peaks. To achieve 99% of peak performance for these workloads, we cannot deviate by more than 8 integer rename registers from the best partitioning (*hill-width_{0.99}* ≤ 8), and for 8 of these 14 workloads, we lose 5% of peak performance when we deviate by roughly 48 registers from the best (*hill-width_{0.95}* ≈ 48). In Figure 4, we see OFF-LINE achieves its largest performance gains for these 14 workloads. The hill peak analysis in Figure 7 shows that the performance of most workloads is sensitive to small resource partitioning changes due to sharp performance peaks within individual epochs. Learning-based SMT resource distribution techniques that can find the best partitioning in these performance-sensitive epochs will achieve the majority of the potential performance gains reported in Figure 4.

3.3.2. Qualitative Analysis

The previous section provides a quantitative analysis of OFF-LINE’s performance gains. An important question is, *qualitatively*, what is the source of the performance variations within epochs, and why do existing techniques miss opportunities to find the performance peaks? We found several important cases where existing techniques miss performance that OFF-LINE exploits.

First, OFF-LINE exploits *cache-miss clustering*. Cache-miss clustering occurs whenever multiple memory loads from the same thread appear in the instruction window and trigger cache misses

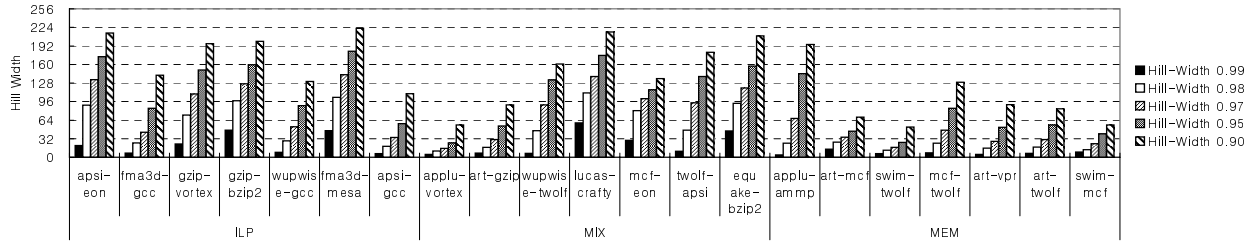


Figure 7. Hill-width measurements.

simultaneously. Existing techniques rarely exploit cache-miss clustering because they avoid fetching too far past a cache miss to prevent clogging resources (*e.g.*, FLUSH flushes after each cache miss and DCRA prevents fetch into other threads’ partitions). However, aggressively fetching past a cache miss is desirable if independent cache-missing loads can be brought into the instruction window to exploit memory parallelism. OFF-LINE learns the best action (contract a thread’s partition to prevent clogging or aggressively increase a thread’s partition to exploit memory parallelism) via performance feedback. Existing techniques conservatively try to prevent resource clogging, possibly missing performance in epochs with memory parallelism.

Second, OFF-LINE exploits *compute-intensive low-ILP threads*. ICOUNT, FLUSH, and DCRA tend to distribute resources to threads that cache-miss infrequently, *i.e.* that are compute-intensive. Existing techniques naively assume such threads always exhibit high ILP and will efficiently use the resources given to them. However, some compute-intensive threads exhibit low ILP even though they incur very few cache misses. We found two examples in our workloads: threads with long instruction dependence chains, and threads with poor branch prediction. OFF-LINE contracts partitions containing compute-intensive low-ILP threads because it learns that doing so does not reduce their performance, freeing up larger partitions for threads that can gainfully exploit them. Existing techniques provide too many resources to compute-intensive low-ILP threads because they treat all non-cache-missing threads the same, leading to sub-peak performance.

4. Hill-Climbing SMT Resource Distribution

The limit study from Section 3 shows phase-based performance-feedback learning applied to SMT resource distribution has the potential to outperform existing SMT techniques. In this section, we try to realize these potential performance gains by developing a technique that performs learning on-line. A key point motivating our approach is illustrated in Figure 2 and discussed in Section 3.3.1: within epochs, SMT performance varies with partitioning in a hill-shaped manner. Hence, we use a *hill-climbing algorithm* to follow the slope of performance hills to directly reach the performance peaks.

4.1. Hill-Climbing Algorithm

Our hill-climbing algorithm builds on top of the off-line exhaustive algorithm presented in Section 3.1. Like the off-line algorithm, hill-climbing performs learning based on epochs, and partitions SMT hardware resources using the approach from Section 3.1.2. However, instead of relying on perfect off-line information to choose the best partitioning for an epoch, hill-climbing

```

1. #define Epoch_Size 64k
2. #define N Total number of running threads
3. #define Delta 4
4. #define eval_perf(X) Evaluate the performance of SMT during the epoch X.
5. #define max(A, n) Get the index of the maximum value in the array A[0 : n]

6. For every Epoch_Size cycles { // invoked at the epoch boundary
7.   perf[epoch_id % N] = eval_perf(epoch_id); // evaluate the performance of the previous epoch (a)
8.   if (epoch_id % N == (N - 1)) { // move the anchor_partition every N-th epochs
9.     gradient_thread = max(perf, N);
10.    for (i = 0 ; i < N ; i++)
11.      if (i == gradient_thread) // move the anchor_partition in favor of gradient_thread
12.        anchor_partition[i] += Delta * (N - 1);
13.    else
14.      anchor_partition[i] -= Delta;
15.  }
16.  epoch_id++;
17.  for (i = 0 ; i < N ; i++) (b)
18.    if (i == epoch_id % N) // try giving favor to thread (epoch_id % N)
19.      trial_partition[i] += anchor_partition[i] + Delta * (N - 1);
20.    else
21.      trial_partition[i] -= anchor_partition[i] - Delta;
22. }

```

Figure 8. Hill-climbing algorithm pseudo-code. Shaded box (a) chooses a new partitioning based on samples acquired by shaded box (b) along all possible directions from the current best partitioning.

guides partitioning using performance samples acquired on-line during the execution of previous epochs.

Figure 8 presents the hill-climbing algorithm. The algorithm consists of two parts: a sampling sequence, called a “round” (lines 16-21), and partition selection at the end of every round (lines 7-15). An array variable, called `anchor_partition`, stores the best-performing partition setting currently found.³ During each round, the performance of several partition settings “near” `anchor_partition` are sampled to determine the local shape of the performance hill. For each sample, we shift the partitioning away from `anchor_partition` slightly by giving a single thread some resources from the other $T - 1$ threads (lines 17-21). The amount taken from each of the $T - 1$ threads, *Delta*, determines how far each sample shifts away from `anchor_partition`; we use $Delta = 4$. (In Figure 8, we assume *Delta* specifies the number of shifted integer rename registers; a proportional number of IQ and ROB entries are also shifted). In total, T samples are taken, allowing each of the T threads to take turns receiving additional resources.

At the end of a round, the best-performing partitioning among the T samples is identified (line 9). This best partition setting lies along the direction of the *positive gradient* (*i.e.*, maximal performance increase) from the `anchor_partition`. Our

³In the very first round, `anchor_partition` defaults to an equal partition for every thread.

algorithm moves in this positive gradient direction by setting `anchor_partition` to the best-performing partitioning found (lines 10-14). Then, the process repeats as another round begins to determine the positive gradient direction for the new `anchor_partition`.

Compared to off-line exhaustive learning, our hill-climbing SMT resource distribution has two limitations. First, hill-climbing incurs *learning time* to find the best partition settings. During learning, non-optimal partitionings are used, sacrificing performance opportunities. Second, hill-climbing may be limited by *local maxima*. When performance hills contain multiple peaks, it may be possible for hill-climbing to reach a non-optimal peak and become trapped. In Section 4.4, we will study the effect of both learning time and local maxima on the performance of hill-climbing.

4.2. Implementation

The modules in dotted lines from Figure 3 show the additional hardware on top of an SMT processor needed to implement our hill-climbing SMT resource distribution technique. First, our technique requires committed instruction counters per thread (these counters are available in most SMT processors already) as well as the number of shared resources—integer IQ entries, integer rename registers, and ROB entries—occupied by each thread. Second, our technique requires a set of resource partitioning registers that specify the size of each thread’s partition in each of the three partitioned shared resources. These partitioning registers are updated every epoch by the hill-climbing algorithm. Third, our technique requires fetch logic that compares the resource occupancy counters against the partitioning registers, and fetch-locks any thread that reaches its partition limit in one or more of the partitioned shared resources.

In addition, our technique also requires implementing the hill-climbing algorithm in Figure 8 for updating the resource partitioning registers every epoch. Because the hill-climbing algorithm is invoked only once per epoch, we believe it can be performed in software. For a software implementation, we envision using a hardware counter to deliver an interrupt to one of the application threads at the end of each epoch, and use its context to execute the hill-climbing algorithm. In this paper, we account for the software implementation’s runtime cost by stalling the entire SMT processor for 200 cycles. We found a single invocation of the hill-climbing algorithm costs roughly 26 cycles, so 200 cycles should be sufficient, even when factoring in the time to interrupt and save/restore the few registers needed by the hill-climbing algorithm. In any case, our accounting is conservative because we need not stall the entire machine, only one thread.

Finally, of the 3 performance metrics discussed in Section 3.1.1, average weighted IPC and harmonic mean of weighted IPC (Equations 2 and 3) require the stand-alone IPC of each thread, $SingleIPC_i$. Because $SingleIPC_i$ values are not known a priori, the hill-climbing algorithm must learn them along with the best partitioning. We continuously sample the stand-alone IPC of each thread by periodically disabling the other $T - 1$ threads for a single epoch and measuring the resulting IPC. To minimize overhead, we acquire a sample every 40 epochs only; hence, each thread’s $SingleIPC_i$ is sampled once every $40 * T$ epochs. This sampling cost is included in all of our experiments.

4.3. Methodology Issues

Our evaluation of hill-climbing SMT resource distribution uses the SMT simulator described in Section 3.2. The simulator is augmented with the hardware and runtime support for hill-climbing described in Section 4.2. To drive our simulations, we use both the 2- and 4-thread multiprogrammed workloads from Table 3. We pick simulation windows using the methodology described in Section 3.2, but we extend their duration to 1 billion instructions. However, comparisons against off-line learning (e.g., Section 4.4.1) use the smaller simulation windows of 100 million instructions due to the complexity of simulating OFF-LINE. Lastly, we evaluate performance using all 3 performance metrics from Section 3.1.1. When calculating end performance with a metric that requires $SingleIPC_i$, we use the $SingleIPC_i$ value from an end-to-end run of each application. When learning with a metric that requires $SingleIPC_i$, hill-climbing algorithm uses a dynamically sampled $SingleIPC_i$ value, as described in Section 4.2.

Part of our evaluation compares hill-climbing to ideal off-line learning algorithms. For the 2-thread workloads, we compare against OFF-LINE from Section 3. For the 4-thread workloads, we develop a new off-line algorithm based on hill-climbing, called RAND-HILL. Like OFF-LINE, RAND-HILL uses check-pointing (see Section 3.2) to search the current epoch’s resource distribution space with zero overhead to find a partition setting for the *same* epoch. Instead of exhaustive search, however, RAND-HILL performs hill-climbing multiple times. Each hill-climbing pass executes the algorithm in Figure 8 starting from the checkpoint and terminates when a peak is found. Every outer-loop iteration of the algorithm (line 6 in Figure 8), we restore machine state to the checkpoint so that the search optimizes for the current epoch only. When a peak is found, we start a new hill-climbing pass from a randomly chosen `anchor_partition`. By performing multiple hill-climbing passes initiated from random points in the resource distribution space, RAND-HILL can find good partitioning solutions even when multiple peaks and local maxima exist. The search for the current epoch ends after 128 total iterations of the algorithm’s outer-loop (line 6).

4.4. Hill-Climbing Results

Figure 9 compares hill-climbing (labeled “HILL-WIPC”) against ICOUNT, FLUSH, and DCRA on our 42 workloads. The comparison is made using the weighted IPC metric; hill-climbing also uses weighted IPC as the performance feedback metric for learning. Comparing HILL-WIPC, ICOUNT, and FLUSH, we see HILL-WIPC outperforms both ICOUNT and FLUSH in all but 4 of our 42 workloads, providing an average performance boost of 12.4% and 11.3%, respectively. Comparing HILL-WIPC and DCRA, we see HILL-WIPC outperforms DCRA by 2.4% averaged over the 42 workloads. This overall performance gain is achieved non-uniformly across the different workload groups. Performance gains are larger for the 2-thread workloads (3.7%) compared to the 4-thread workloads (0.4%). Furthermore, within 2-thread workloads, performance gains are larger for the MEM category (5.1%) compared to the ILP and MIX categories (3.4% and 2.7%, respectively). However, HILL-WIPC outperforms or matches DCRA across *all* (6 total) categories in Figure 9, which is a positive result given the size and diversity of our workload set.

Figure 10 compares all the techniques using different metrics both for measuring performance and for learning. The three graphs report performance in terms of (a) weighted IPC, (b) average IPC, and (c) harmonic mean of weighted IPC. Within

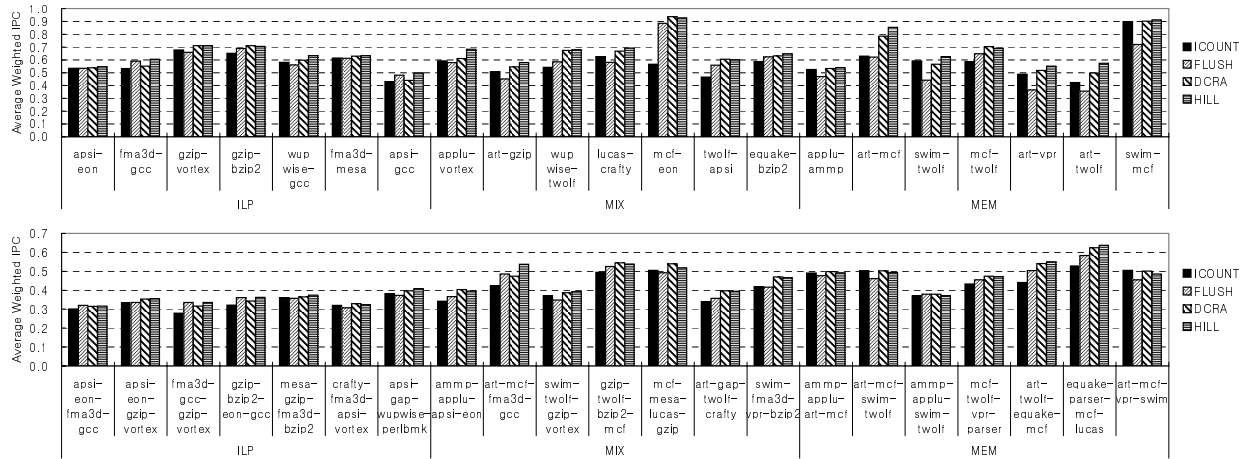


Figure 9. Hill-Climbing versus ICOUNT, FLUSH, and DCRA under the weighted IPC metric. Hill-Climbing uses weighted IPC as the performance feedback metric.

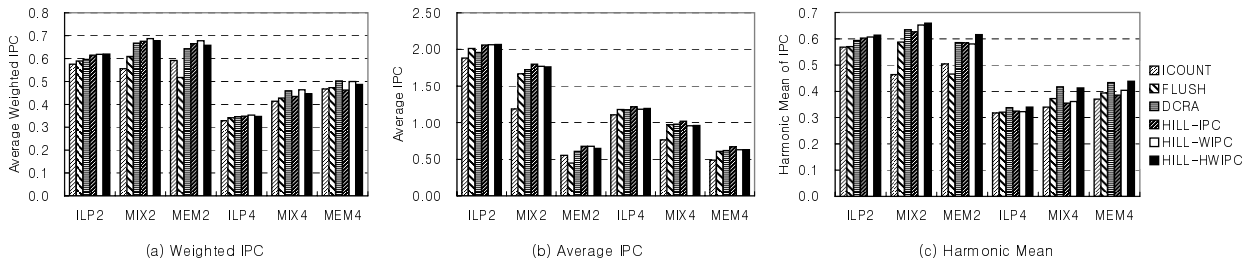


Figure 10. Hill-Climbing versus ICOUNT, FLUSH, and DCRA under the (a) weighted IPC, (b) average IPC, and (c) harmonic mean of weighted IPC metrics. Hill-Climbing uses average IPC (HILL-IPC), weighted IPC (HILL-WIPC), and harmonic mean of weighted IPC (HILL-HWIPC) as the performance feedback metric.

each graph, hill-climbing uses either average IPC (HILL-IPC), weighted IPC (HILL-WIPC), or harmonic mean of weighted IPC (HILL-HWIPC) as the performance feedback metric for learning. Results are summarized by workload group to conserve space. Comparing HILL-IPC, HILL-WIPC, and HILL-HWIPC across the graphs, we see hill-climbing achieves its best performance under a given metric when using the same metric to drive learning. When both evaluation and learning metrics are matched, hill-climbing performs 5.9% better than when they are not matched. This demonstrates one of the strengths of learning-based SMT resource distribution: the ability to *directly optimize* the performance metric most important to the user. Existing techniques cannot optimize for a particular performance goal.

Figures 10b and c show hill-climbing achieves a performance gain under the average IPC and harmonic mean of weighted IPC metrics in addition to the gains already demonstrated under the weighted IPC metric in Figure 9. Comparing HILL-IPC against ICOUNT and FLUSH in Figure 10b, we see hill-climbing outperforms ICOUNT and FLUSH under average IPC in all the workload groups, providing an average performance boost of 24.2% and 7.7%, respectively. Comparing HILL-HWIPC against ICOUNT and FLUSH in Figure 10c, we see hill-climbing outperforms ICOUNT and FLUSH under harmonic mean of weighted IPC in all the workload groups as well, providing an average performance boost of 19.9% and 13.3%, respectively. Comparing HILL-IPC and DCRA in Figure 10b, we see hill-climbing outperforms DCRA by 5.1% under average IPC, and comparing HILL-

HWIPC and DCRA in Figure 10c, we see hill-climbing outperforms DCRA by 2.3% under harmonic mean of weighted IPC.

4.4.1. Comparing Against Off-Line Learning Algorithms

Figure 11 compares HILL-WIPC against our ideal off-line learning algorithms, *i.e.* OFF-LINE for 2-thread workloads and RAND-HILL for 4-thread workloads, under the weighted IPC metric. We note that while OFF-LINE represents the best that hill-climbing can do for 2-thread workloads, a similar performance upper bound does not exist for 4-thread workloads since RAND-HILL does not search exhaustively. To quantify how well RAND-HILL does, we include results for DCRA in the bottom half of Figure 11. Comparing RAND-HILL and DCRA, we see RAND-HILL outperforms DCRA in all but one 4-thread workload, achieving a 7.4% performance boost on average. We also ran synchronized time-varying experiments similar to Figure 5, and found RAND-HILL beats DCRA in 96.4% of all epochs simulated. Consequently, we find RAND-HILL consistently performs very well.

Comparing HILL-WIPC and OFF-LINE in Figure 11, we see hill-climbing achieves 96.6% of ideal performance, and comparing HILL-WIPC and RAND-HILL, we see hill-climbing achieves 94.1% of RAND-HILL’s performance, averaged across all workloads. The largest performance differences occur in the MEM workloads, with some in the MIX workloads as well. For ILP workloads, HILL-WIPC performs very close to the off-line algorithms.

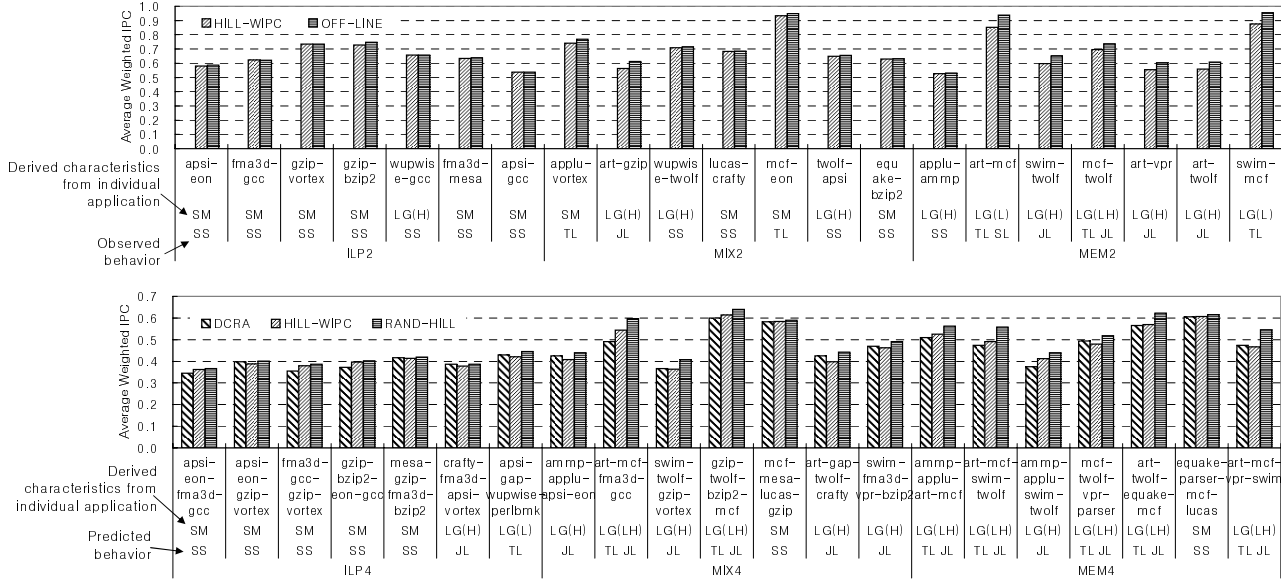


Figure 11. Comparison of HILL-WIPC and OFF-LINE for 2-thread workloads (top graph). Comparison of DCRA, HILL-WIPC, and RAND-HILL for 4-thread workloads (bottom graph).

To gain further insight, we compared the time-varying behavior of HILL-WIPC and OFF-LINE across all of the 2-thread workloads. Figure 12 illustrates 5 representative cases across all of the epochs. This data was generated by running synchronized experiments using the methodology from Section 3.3. However, instead of synchronizing existing techniques to OFF-LINE (Figure 5), we synchronize OFF-LINE to HILL-WIPC. Each graph in Figure 12 plots the partitioning found for the integer rename registers (the integer IQ and ROB are partitioned proportionally) by HILL-WIPC (“+” symbols) and OFF-LINE (white dots) as a function of epoch ID. In addition, for every epoch, we plot the weighted IPC for all possible partitionings (these are visited by OFF-LINE’s exhaustive search) using a gray scale: lighter shades represent lower performance while darker shades represent higher performance. Hence, by following the change in gray scale along any vertical line, we can determine the shape of the performance hill within the corresponding epoch.

Figure 12a shows the *temporally-stable* (TS) case. In TS, OFF-LINE partitioning doesn’t change over time, and there are no bottlenecks that limit hill-climbing’s movement; hence, after a short time, hill-climbing reaches the best partitioning and remains there to enjoy the highest possible performance. Figure 12b shows the *spatially-stable* (SS) case. In SS, OFF-LINE partitioning changes rapidly over time, so fast that hill-climbing cannot track it. As a result, hill-climbing settles in between the fluctuating best partitionings. However, in Figure 12b, the different best partitionings perform similarly, as indicated by the “band” of similar gray scales that encompass the best partitionings. In other words, these are wide hills, and have a large hill-width_N value (see Section 3.3.1). Hence, HILL-WIPC and OFF-LINE achieve similar performance even though hill-climbing cannot find the absolute best.

Figure 12c shows the *temporally-limited* (TL) case. In TL, OFF-LINE partitioning is stable over relatively short periods of time, experiencing sudden large changes occasionally. For example, Figure 12c shows a long period of low performance followed by a short period of high performance. Hill-climbing effectively tracks the best partitioning in the low-performing period due to

its long duration. When the best partitioning changes, it does not remain stable long enough for hill-climbing to adjust; hence, hill-climbing misses significant performance opportunities during the high-performing period. The TL case illustrates the limitations of finite learning time in hill-climbing. Figure 12d shows the *spatially-limited* (SL) case. In SL, OFF-LINE partitioning is relatively stable over time; however, there are multiple peaks, as indicated by the multiple bands of non-monotonically varying gray scales. Hill-climbing gets “stuck” on one of the non-maximal peaks, again missing performance opportunities. The SL case illustrates the limitations of local maxima in hill-climbing.

Finally, Figure 12e shows the *jitter-limited* (JL) case. In JL, OFF-LINE partitioning is relatively stable. Furthermore, in Figure 12e, there is a single maximal peak (*i.e.*, no local maxima). However, hill-climbing has trouble moving towards the best partitioning because of inter-epoch jitter. Although the positive gradient within each epoch always points towards the maximal peak, inter-epoch jitter creates transient positive gradients *between* epochs that temporarily point away from the maximal peak. These bogus gradients fool the hill-climber, causing it to reverse course occasionally and move away from the best partitioning.

In Figure 11, we label each of the 2-thread workloads with the case(s) from Figure 12 that dominate the workload’s time-varying behavior in the row marked “Observed behavior.” Figure 11 shows HILL-WIPC closely matches OFF-LINE in TS and SS workloads, where hill-climbing finds very good partitionings. Fortunately, SS is quite common, allowing HILL-WIPC to perform well in many workloads. In contrast, Figure 11 shows a noticeable performance difference between HILL-WIPC and OFF-LINE in TL, SL, and JL workloads, where hill-climbing has trouble finding good partitionings. Interestingly, SL appears in only one workload, *art-mcf*. Further investigation revealed that local maxima do occur in many of our workloads; however, they rarely *persist* at the same partition setting for more than 3 or 4 epochs. We find there is some level of jitter in all our workloads that allows the hill-climber to escape from local maxima.

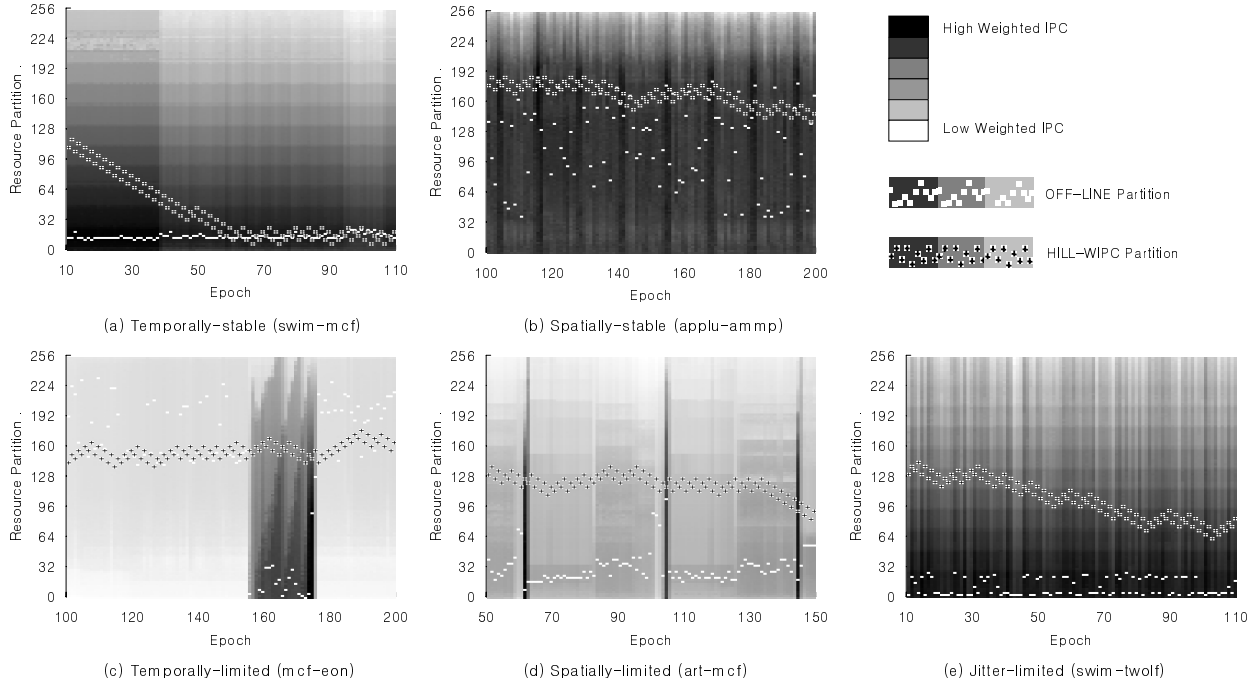


Figure 12. Five representative time-varying behaviors of HILL-WIPC and OFF-LINE from the 2-thread workloads: (a) temporally-stable (TS), (b) spatially-stable (SS), (c) temporally-limited (TL), (d) spatially-limited (SL), and (e) jitter-limited (JL).

4.4.2. Per-Application Analysis

Unfortunately, we cannot perform Figure 12’s analysis in the 4-thread workloads due to the intractability of simulating OFF-LINE for more than 2 threads. In this section, we *predict* the 4-thread workload behaviors by analyzing individual applications.

Two application characteristics affect hill-climbing performance: *resource requirement* and its *time variation*. We quantify resource requirements for our benchmarks by executing them on the SMT simulator stand-alone (without other threads), and measuring IPC as we vary the number of integer rename registers from 100% down to 10%. In the column labeled “Rsc” in Table 2, we report the number of integer rename registers needed to achieve 95% of the maximum single-thread performance for each benchmark. Then, in the column labeled “Rsc” in Table 3, we report the sum of the per-application integer rename register requirements across each workload. This estimates each workload’s resource requirement to perform “well.” To quantify resource requirement time variation, we perform the same experiment, but we record the resource requirement periodically—every 64K cycles—and identify changes between epochs. This analysis reveals 3 behaviors: high-frequency variation (a change every 1 or 2 epochs), low-frequency variation (a change after several epochs), or no appreciable time variation. Each benchmark’s time variation behavior is indicated in the column labeled “Freq” in Table 2.

For the 2-thread workloads in Figure 11, we label each workload at the row marked “Derived characteristics from individual application” based on its composite application characteristics. Workloads whose resource requirements are ≤ 256 are small, labeled “SM;” workloads whose resource requirements exceed 256 are large, labeled “LG.” Large workloads are further labeled with “H” or “L” whenever a high-frequency or low-

frequency benchmark, respectively, participates in the workload. We find good correlation between the labels and workload behaviors. Small workloads (SM) almost always exhibit SS behavior. These workloads “fit” within the SMT’s 256 rename registers and 512-entry ROB. Such resource slack leads to similar performance between widely varying partitionings, as in Figure 12b. In contrast, large workloads exhibit either TL or JL behavior. We find high-frequency workloads (LG(H)) exhibit JL behavior because the frequent inter-epoch resource requirement changes cause the jitter in Figure 12c; we find low-frequency workloads (LG(L)) exhibit TL behavior because the periodic resource changes in mcf (the only “Low” benchmark in Table 2) lead to the TL case in Figure 12c. Exceptional cases include *wupwise-gcc*, *wupwise-twolf*, *twolf-apsi*, and *applu-ammp*, which are large (LG) but exhibit SS behavior. After close examination, we found *wupwise*, *apsi*, and *applu* are insensitive to partitioning across a wide range of partition settings. Even in small partitions, these benchmarks achieve close to 90% of their maximum single-thread performance. Hence, they effectively have smaller resource needs than indicated in Table 2.

We label the 4-thread workloads in Figure 11 at the row marked “Derived characteristics from individual application” in a similar fashion. We choose the SM or LG labels based on resource requirements from Table 3. However, instead of using the 256 threshold, we increase the threshold to 400 to reflect the larger number of threads in each workload. Then, we add the time variation labels (“H” and “L”) to large workloads based on the participating benchmarks. Finally, from the “Derived characteristics from individual application” labels in Figure 11, we *predict* the workload behavior in the row marked “Predicted behavior:” SM workloads yield SS behavior, LG(H) workloads yield JL behavior, and LG(L) workloads yield TL behavior. We find the predicted

workload behaviors correlate to observed performance. In workloads with SS behaviors, HILL-WIPC closely matches RAND-HILL. In workloads with TL or JL behaviors, HILL-WIPC does not achieve all of the potential performance exhibited by RAND-HILL, just like in the 2-thread workloads.

5. Phase Detection and Prediction

One of the limitations of hill-climbing is finite learning time. A natural approach to attack the finite learning time problem is to exploit existing phase detection and prediction techniques. Phase detection [15] can be used to determine which epochs are similar. Instead of re-learning a partitioning for such an epoch, we can simply use a previously learned partitioning to save the learning time. Phase prediction [17] can be used to predict a future epoch so that we can apply a previously learned partitioning.

We implemented Sherwood's Basic Block Vector (BBV) signature analysis technique [15] to perform phase detection on our epochs. We use a BBV with 64 entries per SMT context. We also implemented Sherwood's phase prediction technique [17] to predict the phase ID of the next epoch. Our phase predictor stores 128 unique phase IDs, and uses a 2048-entry run-length encoded (RLE) Markov predictor. With phase detection and prediction, we are able to boost hill-climbing performance by only 0.4% across all 42 of our multiprogrammed workloads. Interestingly, almost all of the performance benefit comes from speeding up workloads exhibiting the TL behavior, the one that exposes hill-climbing's learning time problem. Considering only TL workloads, we see a 2.1% performance boost. We believe this is a promising approach to improve hill-climbing, and plan on pursuing it as future work.

6. Conclusion

This paper investigates learning-based SMT resource distribution techniques. We present an ideal off-line exhaustive search technique that enables a limit study. Our limit study shows learning-based SMT techniques have the potential to improve performance by 19.2% over ICOUNT, 18.0% over FLUSH, and 7.5% over DCRA. We also present a novel hill-climbing SMT resource distribution technique which varies the resource share of multiple threads towards the direction that improves end performance. Our evaluation shows hill-climbing improves performance by 12.4% over ICOUNT, 11.3% over FLUSH, and 2.4% over DCRA. Because our approach learns based on actual performance, the resource distribution decisions it makes are customized to the performance bottlenecks of the workload. Moreover, whenever learning for a particular behavior succeeds, our approach finds the best resource distribution for that behavior. Finally, our approach can optimize for a specific performance goal by using the appropriate performance feedback metric. Due to these advantages, we believe feedback-based learning is a promising approach for SMT resource distribution.

References

[1] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.
 [2] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 171–182. IEEE Computer Society, December 2004.

[3] G. K. Dorai and D. Yeung. Transparent Threads: Resource Allocation in SMT Processors for High Single-Thread Performance. In *Proceedings of the 11th Annual International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, September 2002.
 [4] A. El-Moursy and D. H. Albonese. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture*, February 2003.
 [5] R. Goncalves, E. Ayguade, and a. P. O. A. N. M. Valero. Performance Evaluation of Decoding and Dispatching Stages in Simultaneous Multithreaded Architectures. In *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, September 2001.
 [6] <http://www.intel.com/design/Pentium4/index.htm>. Intel Pentium 4 Processor. 2002.
 [7] R. N. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.
 [8] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002.
 [9] F. Latorre, J. Gonzalez, and A. Gonzalez. Back-end Assignment Schemes for Clustered Multithreaded Processors. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 316–325, July 2004.
 [10] K. Luo, M. Franklin, S. S. Mukherjee, and A. Sez nec. Boosting SMT Performance by Speculation Control. In *Proceedings of the International Parallel and Distributed Processing Symposium*, San Francisco, CA, April 2001.
 [11] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, November 2001.
 [12] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*, pages 716–726, Toulouse, France, August 1999. Springer-Verlag.
 [13] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading Technology Architecture and Microarchitecture. In *Intel Technology Journal*, 6(1), February 2002.
 [14] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.
 [15] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.
 [16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002. ACM.
 [17] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 336–347, June 2003.
 [18] A. Snaveley, D. M. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.
 [19] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 318–327. IEEE Computer Society, 2001.
 [20] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.