Hill-Climbing SMT Processor Resource Distribution

SEUNGRYUL CHOI
Google
and
DONALD YEUNG
University of Maryland

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 online, learning time is crucial. We develop a *hill-climbing algorithm* that quickly learns the best distribution of resources by following the performance gradient within the resource distribution space. We also develop several ideal learning algorithms to enable deeper insights through limit studies.

This article conducts an in-depth investigation of hill-climbing SMT resource distribution using a comprehensive suite of 63 multiprogrammed workloads. Our results show hill-climbing outperforms ICOUNT, FLUSH, and DCRA (three existing SMT techniques) by 11.4%, 11.5%, and 2.8%, respectively, under the weighted IPC metric. A limit study conducted using our ideal learning algorithms shows our approach can potentially outperform the same techniques by 19.2%, 18.0%, and 7.6%, respectively, thus demonstrating additional room exists for further improvement. Using our ideal algorithms, we also identify three bottlenecks that limit online learning speed: local maxima,

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, the U.S. Government.
Authors' addresses: S. Choi, Google; D. Yeung (contact author), Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742; email: yeung@umd.edu.

phased behavior, and interepoch jitter. We define metrics to quantify these learning bottlenecks, and characterize the extent to which they occur in our workloads. Finally, we conduct a sensitivity study, and investigate several extensions to improve our hill-climbing technique.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Hill-climbing algorithm, limit study, SMT processor

ACM Reference Format:

Choi, S. and Yeung, D. 2009. Hill-climbing SMT processor resource distribution. ACM Trans. Comput. Syst. 27, 1, Article 1 (February 2009), 47 pages. DOI = 10.1145/1482619.1482620 http://doi.acm.org/10.1145/1482619.1482620

1. INTRODUCTION

Simultaneous MultiThreading (SMT) is an important architectural technique, as evidenced by the widespread attention it has received from academia [Cazorla et al. 2004; El-Moursy and Albonesi 2003; Tullsen et al. 1996; Tullsen and Brown 2001; Raasch and Reinhardt 2003], and by industry's willingness to incorporate it into commercial processors [Kalla et al. 2004; Pentium4 2002]. 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 resource sharing. SMT processors improve performance by allowing multiple threads to share hardware resources simultaneously. Sharing leads to increased utilization, and thus higher processor throughput. However, the actual performance gains of SMT processors depend greatly on the quality of the resource distribution decisions. High performance occurs only when resources are distributed to those threads that will use them most efficiently. Hence, the mechanisms for controlling resource distribution play a critical role in SMT processors.

Several resource distribution techniques have been studied in the past [Cazorla et al. 2004; El-Moursy and Albonesi 2003; Tullsen et al. 1996; Tullsen and Brown 2001; Raasch and Reinhardt 2003]. One shortcoming of these previous techniques is they optimize performance only *indirectly*. As illustrated in Figure 1(a), 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.

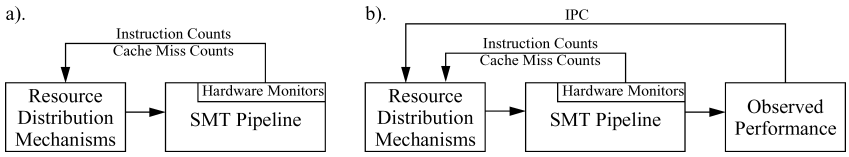

Fig. 1. (a) Existing resource distribution techniques optimize performance indirectly by making decisions based on hardware monitors only; (b) learning-based resource distribution examines actual performance to learn the best resource distribution.

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 1(b). 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 appropriate performance metric for feedback.

Since we perform learning online, learning speed 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, namely 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 high learning speeds.

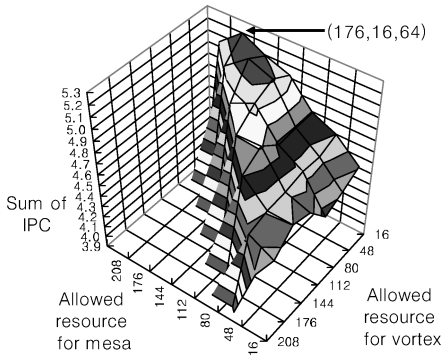

Fig. 2. IPC of mesa, vortex, and fma3d during a 32K-cycle time interval as the fraction of resources distributed to each thread is varied. The x- and y-axes show the resource distribution for mesa and vortex (fma3d receives the remaining resources). The arrow indicates the resource distribution with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

This article investigates SMT resource distribution techniques that employ hill-climbing to dynamically learn the best resource share. Specifically, we use hill-climbing to distribute three key SMT hardware structures, namely the issue queue, rename registers, and reorder buffer, across simultaneously executing threads. On a suite of 63 multiprogrammed workloads, we show our hill-climbing technique outperforms ICOUNT [Tullsen et al. 1996] by 11.4%, FLUSH [Tullsen and Brown 2001] by 11.5%, and DCRA [Cazorla et al. 2004] by 2.8% when comparing performance under the weighted IPC (WIPC) metric. Hill-climbing's advantage over ICOUNT, FLUSH, and DCRA is as high as 23.7%, 13.6%, and 5.9%, respectively, when comparing performance under either the average IPC or harmonic mean of weighted IPC metrics.

This article also investigates several ideal SMT resource distribution algorithms that perform learning offline, achieving the best (or near-best) performance any learning-based SMT resource distribution technique can achieve. Our results show offline learning outperforms ICOUNT, FLUSH, and DCRA by 19.2%, 18.0%, and 7.6%, respectively, under the WIPC metric, demonstrating there exists additional room to improve our online hill-climbing technique. Offline learning also enables us to study the impact of workload behavior on learning speed. Using our offline algorithms, we identify three major bottlenecks that limit learning speed, namely local maxima, phased behavior, and interepoch jitter, and characterize the extent to which they occur in our 63 multiprogrammed workloads. Finally, in addition to studying our basic hill-climbing technique, we also investigate its sensitivity to processor configuration assumptions, and explore extensions for further improvements.

The rest of this article is organized as follows. We begin by discussing related work in Section 2. Section 3 presents our hill-climbing SMT resource distribution technique, and Section 4 evaluates its performance. Next, Section 5 introduces our offline learning algorithms, and conducts a study on the limits of hill-climbing SMT resource distribution. Then, Section 6 investigates the workload behaviors that impede learning, and characterizes the extent to which they occur in our workloads. This is followed by Section 7 which presents our

sensitivity study and hill-climbing extensions. Finally, Section 8 concludes the article.

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 [Tullsen et al. 1996] and FPG [Luo et al. 2001] 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 mispredictions) 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 [Tullsen and Brown 2001] triggers fetch-lock when a load remains outstanding beyond some threshold number of cycles; DG [El-Moursy and Albonesi 2003] triggers fetch-lock when the number of cache-missing loads exceeds some threshold; and PDG [El-Moursy and Albonesi 2003] uses a cache-miss predictor to trigger fetch-lock.

One problem with fetch-locking is that 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 [Tullsen and Brown 2001]. 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 [Tullsen and Brown 2001]) 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 [Goncalves et al. 2001; Marr et al. 2002; Raasch and Reinhardt 2003], but these techniques cannot adapt to changing workload behavior. In contrast, DCRA [Cazorla et al. 2004] partitions dynamically based on memory performance. Threads with frequent L1 cache misses are given large partitions, allowing them to exploit parallelism beyond stalled memory operations. 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 those threads that can use resources most efficiently.

Hill-climbing SMT resource distribution, like static partitioning and DCRA, is a partitioning technique. The idea was first introduced in our previous work [Choi and Yeung 2006]; this article conducts a more thorough investigation, including an expanded suite of workloads, a new methodology for studying learning speed limitations, and a new sensitivity study and investigation of extensions. Compared to previous techniques, hill-climbing 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 hill-climbing 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 feedback 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 workload behavior.

Finally, our approach borrows from program phase analysis [Sherwood et al. 2003, 2002]. 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) [Latorre et al. 2004] 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 online fashion.

3. LEARNING SMT RESOURCE DISTRIBUTIONS VIA HILL-CLIMBING

Figure 3 shows a block diagram of the SMT processor we assume in this article. The solid boxes in the figure illustrate the main hardware structures common to most SMT pipelines (the dotted boxes represent the additional hardware needed for our technique, explained later in Section 3.3). Except for the per-thread program counters, labeled "PC," all of the main hardware structures in Figure 3 are shared amongst simultaneously executing threads. The goal of SMT resource distribution is to allocate these shared hardware structures to threads such that they are utilized as efficiently as possible, thus leading to high processor throughput.

This article investigates learning-based techniques for partitioning the shared hardware structures found in SMT processors. Like other partitioning

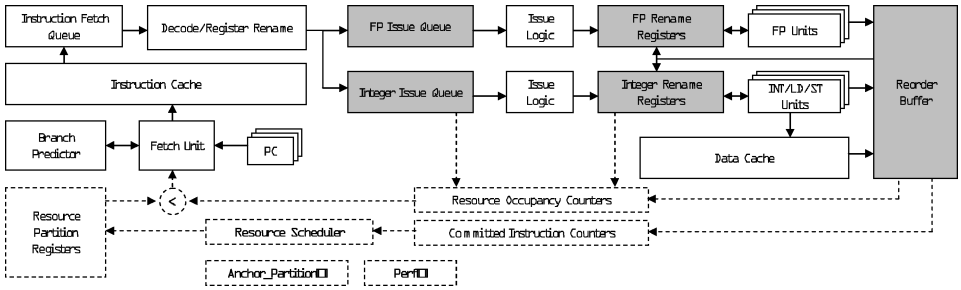

Fig. 3. Block diagram of our SMT processor model. Shaded boxes indicate shared hardware structures that are partitioned by learning-based resource distribution. Dotted boxes indicate additional hardware needed for our hill-climbing algorithm.

techniques (e.g., DCRA), we apply partitioning to several key hardware structures that significantly impact SMT performance: the issue queues (IQ), the rename registers, and the reorder buffer (ROB).[1] These hardware structures are shaded gray in Figure 3. By allowing simultaneously executing threads to consume up to, but no more than, the shared resources allotted within per-thread partitions, no single thread can monopolize all of the shared resources. Hence, partitioning increases SMT throughput by preventing resource clog, as discussed in Section 2. In addition to partitioning, distributing fetch bandwidth effectively is also crucial to SMT performance. We rely on the ICOUNT fetch policy [Tullsen et al. 1996] to distribute fetch bandwidth across threads on a cycle-by-cycle basis (more on this later in Section 3.3).

In this section, we present our hill-climbing technique for partitioning the IQ, rename registers, and ROB in SMT processors. We begin by introducing the basic idea of learning resource partitionings at epoch boundaries (Section 3.1). Then, we describe our hill-climbing algorithm for intelligently guiding learning to achieve high learning speeds (Section 3.2). Finally, we discuss implementation issues (Section 3.3).

3.1 Epoch-Based Learning

Hill-climbing SMT resource distribution performs learning based on *epochs*, an idea illustrated in Figure 4. Along the x-axis, Figure 4 shows a timeline of SMT execution broken down into fixed-size intervals, called epochs. Along the y-axis, Figure 4 shows all possible partitionings of the shared SMT hardware resources (e.g., the IQs, rename registers, and ROB). At the beginning of each epoch, a learning algorithm is invoked to choose a single resource partitioning out of all the possible partitionings. The SMT processor executes for one epoch using this chosen partitioning, after which the achieved performance is measured (we assume the measurement is taken using hardware performance counters resident in the SMT processor), thus providing direct feedback on how well

[1]Most SMT processors implement private ROBs to simplify per-thread commit. As illustrated in Figure 3; we assume a shared ROB to be consistent with DCRA [Cazorla et al. 2004]. Our approach would still work for private ROBs; we would ignore the ROBs, and partition the remaining shared resources only.

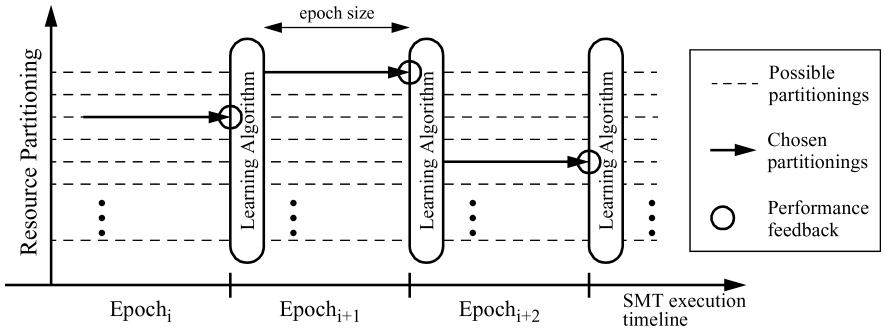

Fig. 4. Epoch-based learning. A learning algorithm is invoked at epoch boundaries to evaluate performance feedback. The learning algorithm's goal is to quickly find the best partitioning out of all possible partitionings.

the chosen partitioning performed. Then a new partitioning is chosen, and the entire process repeats. Over time, a significant number of partitionings can be sampled, allowing the learning algorithm to determine which partitionings exhibit the best performance.

Because epoch-based learning samples different partitionings to find the best one, an important question is: How large is the sample space? Unfortunately, the number of unique partitionings of the SMT hardware resources can be extremely large. Given S shared structures, E_i entries for structure i, and T threads, the number of ways to partition the resources is on the order of $\Pi_{i=1}^{S} E_i^{(T-1)}$. To reduce the size of this sample space, we observe that a thread's use of different hardware resources is *not* independent; instead, the number of entries of each resource type a thread occupies is 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 sampling. We exploit this observation in two ways. First, we assume the number of IQ entries, rename registers, and ROB entries occupied by a thread are *in proportion to one another*.[2] Rather than partition every resource independently, we partition a single resource only, and then apply the same partition proportionally to the other resources. Second, we do not explicitly partition the floating point IQ and rename registers. By partitioning the integer IQ, integer rename registers, and ROB, we indirectly control how many floating point resources each thread consumes, making learning for these resources less critical.

With these simplifications, the number of unique partitionings reduces significantly, and is on the order of ${E_{max}}^{(T-1)}$, where $E_{max} = max_{i=1}^{S}(E_i)$, making epoch-based learning more tractable. However, this sample space is still very large, especially for large T. Given our large sample spaces, it is critical to intelligently choose the partitionings to sample from epoch to epoch (e.g., random

[2]In reality, a thread's utilization of the IQ, rename registers, and ROB is probably not in exact proportion, but we make this assumption to simplify the learning problem. It is possible that allowing nonproportional allocation of these structures to threads would result in higher performance, but we leave that study to future work.

```
1.  #define Epoch_Size    64k
2.  #define T             Total number of threads
3.  #define Delta         4
4.  #define eval_perf(X)  Evaluate SMT performance during 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 each epoch boundary
7.    perf[epoch_id % T] = eval_perf(epoch_id);  // evaluate the performance of
                                                 // the previous epoch
8.    if (epoch_id % T == (T - 1)) {
        // move the anchor_partition every T-th epoch                    a).
9.      gradient_thread = max(perf, T);
10.     for (i = 0; i < T; i++)
11.       if (i == gradient_thread)
            // move the anchor_partition in favor of gradient_thread
12.         anchor_partition[i] += Delta * (T - 1);
13.       else
14.         anchor_partition[i] -= Delta;
15.   }
16.   epoch_id++;
17.   for (i = 0; i < T; i++)                                             b).
18.     if (i == epoch_id % T)
          // try giving favor to thread (epoch_id % T)
19.       trial_partition[i] += anchor_partition[i] + Delta * (T - 1);
20.     else
21.       trial_partition[i] -= anchor_partition[i] - Delta;
22. }
```

Fig. 5. Hill-climbing algorithm pseudocode. Shaded box (a) chooses a new partitioning based on samples acquired by shaded box (b) along all possible directions from the current best partitioning.

or exhaustive sampling is unlikely to lead to the best partitioning quickly). This is the job of the learning algorithm, and is the topic of the next section.

3.2 Hill-Climbing Algorithm

As discussed in Section 1, the variation of performance across different partitionings of SMT hardware resources is not random, but instead often follows a hill-shape. Hence, we propose using a hill-climbing algorithm to decide which partitionings to sample across epochs. Our hill-climbing algorithm samples those partitionings whose performance lies along the direction of the *positive gradient*, or maximal performance increase, within the sample space. By following the positive gradient, we can potentially find the globally optimal partitioning (i.e., the peak of the hill) after sampling only a small fraction of the entire resource distribution space.

Figure 5 presents our hill-climbing algorithm. The body of the algorithm (lines 7–21 in Figure 5) is executed once at the end of each epoch, and consists of two parts: selection of the next partitioning to sample (lines 16–21), and identification of the best-performing partition amongst those sampled (lines 8–15). An array variable, called `anchor_partition`, stores the best-performing partition currently found.[3] During sample selection, the performance of several partition settings near `anchor_partition` are sampled to determine the local shape of the performance hill. This is done by shifting the next partition to

[3]At the very beginning of program execution, `anchor_partition` defaults to an equal partition for every thread.

sample, called `trial_partition`, 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 5, we assume *Delta* specifies the number of shifted integer rename registers; a proportional number of IQ and ROB entries are also shifted.)

After T samples are taken (i.e., T invocations of lines 7–21 in Figure 5), every thread in the workload is given a chance to execute with additional resources, and their resulting performance values are stored in the `perf` array variable (line 7). Our algorithm then identifies the best-performing partitioning among the T samples (line 9). This best partition setting lies along the direction of the positive gradient from the `anchor_partition`. Our algorithm moves in this direction by setting `anchor_partition` to the best-performing partitioning found (lines 10–14). Then, the process repeats to determine the positive gradient direction for the next `anchor_partition`.

Several design considerations affect the performance of our hill-climbing algorithm. In the remainder of this section, we discuss two especially important issues, epoch size and performance feedback metrics. Later, in Section 7.2, we will propose some extensions to the basic hill-climbing algorithm that address other performance issues.

Epoch size. The `Epoch_Size` parameter in Figure 5 controls how frequently the hill-climbing algorithm is invoked, as illustrated in Figure 4. Hence, epoch size determines the adaptivity of learning. If the epoch size is too large, then the learning may not adapt quickly enough to changes in the workload's resource demands. If the epoch size is too small, then interepoch behavior may become too dynamic, making learning difficult. (With smaller epochs, the time spent processing performance feedback information also grows, potentially increasing runtime overhead. This is an important issue for software implementations of the hill-climbing algorithm, discussed in Section 3.3.) We ran several experiments to measure the sensitivity of hill-climbing performance to epoch size. Based on these experiments, we found a 64K-cycle epoch consistently yields good performance. So, in Figure 5, we set `Epoch_Size` = 64K cycles for all the experiments in this article.

Performance feedback metrics. The `eval_perf` function (line 7 of Figure 5) provides performance feedback to the hill-climbing algorithm. As discussed in Sections 1 and 2, an advantage of our hill-climbing technique is the ability to optimize for a specific performance goal by simply changing the metric assessed during performance feedback. In this article, we consider three performance metrics for the `eval_perf` function: average IPC, average weighted IPC [Snavely et al. 2002], and harmonic mean of weighted IPC [Luo et al. 2001]. These metrics can be computed as

$$\text{Average\_IPC} = \frac{\sum IPC_i}{T} \tag{1}$$

$$\text{Average\_Weighted\_IPC} = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{T} \tag{2}$$

$$\text{Harmonic\_Mean\_of\_Weighted}\_IPC = \frac{T}{\sum \frac{SingleIPC_i}{IPC_i}} \quad (3)$$

where IPC_i is the *ith* thread's IPC assuming SMT execution, $SingleIPC_i$ is the *ith* thread's IPC assuming stand-alone execution, and T is the number of threads.

Our metrics quantify different aspects of SMT performance. Average IPC quantifies how well all threads perform collectively; hence, it reflects overall throughput. Average weighted IPC quantifies how much slower each thread executes relative to stand-alone execution; hence it reflects per-thread slowdown. Similar to average weighted IPC, harmonic mean of weighted IPC considers per-thread slowdown as well, so it is also a performance (slowdown) metric. But instead of computing the arithmetic mean, it computes the harmonic mean of per-thread slowdown. Since the harmonic mean accentuates small values more than the arithmetic mean, the harmonic mean of IPC is highly sensitive to large slowdowns, and hence effectively reflects fairness. By choosing one of these three performance metrics for the `eval_perf` function, the user can specify a different performance goal to optimize via hill-climbing: either maximum throughput, minimum per-thread slowdown, or maximum performance/fairness.

Note, the average weighted IPC and harmonic mean of weighted IPC metrics (Eqs. (2) and (3)) both require the $SingleIPC_i$ values. One way to obtain these values is to measure them offline, and then provide them to the hill-climbing algorithm. The disadvantage of this approach is that it requires a separate profiling run for each program in the multithreaded workload. Another way to obtain the $SingleIPC_i$ values is for the hill-climbing algorithm itself to acquire them online. In this approach, 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. Our experiments will evaluate both the offline and online approaches.

3.3 Implementation

Figure 3 illustrates the implementation of our hill-climbing SMT resource distribution technique. In particular, the modules in dotted lines show the additional necessary hardware on top of an SMT processor. First, we require committed instruction counters per thread (these counters are available in most SMT processors already) as well as counters for the number of shared hardware resources, namely 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 modifying the basic Icount fetch policy as follows. Every cycle, we compare each thread's resource occupancy counters against its partitioning registers. If a

thread has reached its partition limit in any one of the partitioned shared resources, we fetch-lock the thread. The Icount policy distributes fetch bandwidth only across those threads that have not been fetch-locked. Hence, our technique enforces per-thread partitions by applying back-pressure to the ICOUNT policy via the partitioning registers.

Lastly, our technique requires implementing the hill-climbing algorithm in Figure 5 for updating the resource partitioning registers every epoch. This can be done either in hardware or software. A hardware implementation requires special registers to store the `anchor_partition` and `perf` variables from Figure 5. It also requires hardware to implement the performance evaluation function, `eval_perf`, discussed in Section 3.2. We anticipate the latter can become costly from a hardware implementation standpoint, especially for the average weighted IPC and harmonic mean of weighted IPC metrics. In contrast, a software implementation simply requires delivering an interrupt to the SMT processor every epoch, and executing the hill-climbing algorithm in a software handler. Our experiments will evaluate both the hardware and software implementations.

The hardware support for hill-climbing SMT resource distribution is slightly less than what's needed for other dynamic resource allocation techniques. For example, in DCRA, the same counters, partitioning registers, and fetch-lock logic shown in Figure 3 are also required, but instead of the 4 counters per thread required by our technique, DCRA requires 8 counters per thread. In addition, like our technique, DCRA also requires implementing a partitioning policy (DCRA's policy is based on occupancy counters, whereas ours is based on the hill-climbing algorithm in Figure 5). However, DCRA's partitioning policy is more challenging to implement because it is invoked *every cycle*; in comparison, we execute the hill-climbing algorithm only once per epoch. This relaxed timing constraint allows for a less aggressive hardware implementation of our hill-climbing algorithm or even implementation in software, as described before.

4. HILL-CLIMBING SMT RESOURCE DISTRIBUTION PERFORMANCE

Having presented our hill-climbing SMT resource distribution technique, we now evaluate its performance. This section presents our evaluation in two parts. First, we discuss the methodology used to conduct our experiments (Section 4.1). Then, we present the performance results (Section 4.2).

4.1 Experimental Methodology

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 [Madon et al. 1999], an extension of the out-of-order processor model from the SimpleScalar toolset [Burger and Austin 1997]. For our evaluation, we model an aggressive 8-way issue SMT processor with 4 hardware contexts and a 512-entry reorder buffer. The detailed processor and memory system settings for our simulations are listed in Table I. We chose this aggressive machine as our baseline to eliminate the artificial resource constraints imposed by less aggressive hardware. Later, in Section 7.1, we will evaluate our technique on a smaller CPU.

Table I. SMT Processor and Memory System Settings Used in Our Experiments

Processor Parameters	
Contexts	4
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 II. SPEC CPU2000 Benchmarks Used to Create Multiprogrammed Workloads

App	Type		Skip	App	Type		Skip
bzip2	Int	ILP	1.1B	fma3d	FP	ILP	1.9B
vortex	Int	ILP	0.1B	wupwise	FP	ILP	3.4B
gap	Int	ILP	0.2B	mcf	Int	MEM	2.1B
perlbmk	Int	ILP	1.7B	twolf	Int	MEM	2.0B
gzip	Int	ILP	0.2B	vpr	Int	MEM	0.3B
crafty	Int	ILP	0.5B	lucas	FP	MEM	0.8B
eon	Int	ILP	0.1B	applu	FP	MEM	0.8B
parser	Int	ILP	1.0B	equake	FP	MEM	0.4B
gcc	Int	ILP	2.1B	ammp	FP	MEM	2.6B
apsi	FP	ILP	2.3B	art	FP	MEM	0.2B
mesa	FP	ILP	0.5B	swim	FP	MEM	0.4B

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). We use our hill-climbing algorithm to explore different partitionings of the 256 integer rename registers; the integer IQ and ROB partitions are set in proportion to the integer rename register partition at all times. In addition to resource partitioning, we also use ICOUNT to select the threads from which to fetch every cycle.

Our experiments are driven by 63 multiprogrammed workloads created from 22 SPEC CPU2000 benchmarks. Table II lists our benchmarks. We use the pre-compiled Alpha binaries from the SimpleScalar Web site[4] which are built with the highest level of compiler optimization. All of our benchmarks use the reference inputs provided by SPEC. From the benchmarks, we

[4]http://www.simplescalar.com/benchmarks.html.

Table III. Multiprogrammed Workloads Used in the Experiments

ILP2	ILP3	ILP4
apsi eon	gcc eon gap	apsi eon fma3d gcc
fma3d gcc	gcc apsi gzip	apsi eon gzip vortex
gzip vortex	crafty perlbmk wupwise	fma3d gcc gzip vortex
gzip bzip2	mesa vortex fma3d	gzip bzip2 eon gcc
wupwise gcc	fma3d vortex eon	mesa gzip fma3d bzip2
fma3d mesa	parser apsi wupwise	crafty fma3d apsi vortex
apsi gcc	gap mesa perlbmk	apsi gap wupwise perlbmk
MIX2	MIX3	MIX4
applu vortex	twolf eon vortex	ammp applu apsi eon
art gzip	lucas gap apsi	art mcf fma3d gcc
wupwise twolf	equake perlbmk gcc	swim twolf gzip vortex
lucas crafty	mcf apsi fma3d	gzip twolf bzip2 mcf
mcf eon	art applu wupwise	mcf mesa lucas gzip
twolf apsi	swim craft parser	art gap twolf crafty
equake bzip2	bzip2 mesa swim	swim fma3d vpr bzip2
MEM2	MEM3	MEM4
applu ammp	mcf twolf vpr	ammp applu art mcf
art mcf	swim twolf equake	art mcf swim twolf
swim twolf	art twolf lucas	ammp applu swim twolf
mcf twolf	equake vpr swim	art twolf equake mcf
art vpr	art ammp lucas	vpr lucas swim applu
art twolf	vpr swim ammp	lucas swim art ammp
swim mcf	art applu swim	ammp equake lucas vpr

These workloads are based on the number of threads in each workload (2, 3, or 4) and the behavior of the individual benchmarks in the workload (ILP, MIX, and MEM).

created multiprogrammed workloads by following the methodology in Cazorla et al. [2004] and Tullsen and Brown [2001]. We first categorized the SPEC benchmarks into either high-ILP or memory-intensive programs, labeled "ILP" and "MEM," respectively, in Table II. Then, we created 3 groups of 2-, 3-, and 4-thread workloads. Table III lists our multiprogrammed workloads. The ILP2, ILP3, and ILP4 workloads group high-ILP benchmarks; the MEM2, MEM3, and MEM4 workloads group memory-intensive benchmarks; and the MIX2, MIX3, and MIX4 workloads group both high-ILP and memory-intensive benchmarks.

We selected simulation regions for our multithreaded workloads in the following way. First, we used SimPoint [Sherwood et al. 2001] to analyze the first 16 billion instructions of each benchmark (or the entire execution, whichever is shorter), 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 II reports the number of skipped instructions in each benchmark during fast-forwarding, in the columns labeled "Skip." Then, we turn on detailed multithreaded simulation, and simulate for 1 billion instructions.

Lastly, as discussed earlier, our results consider both offline measurement and online sampling of the $SingleIPC_i$ values (see Section 3.2) as well as both hardware and software implementations of the hill-climbing algorithm (see Section 3.3). We first evaluate our hill-climbing technique assuming the most

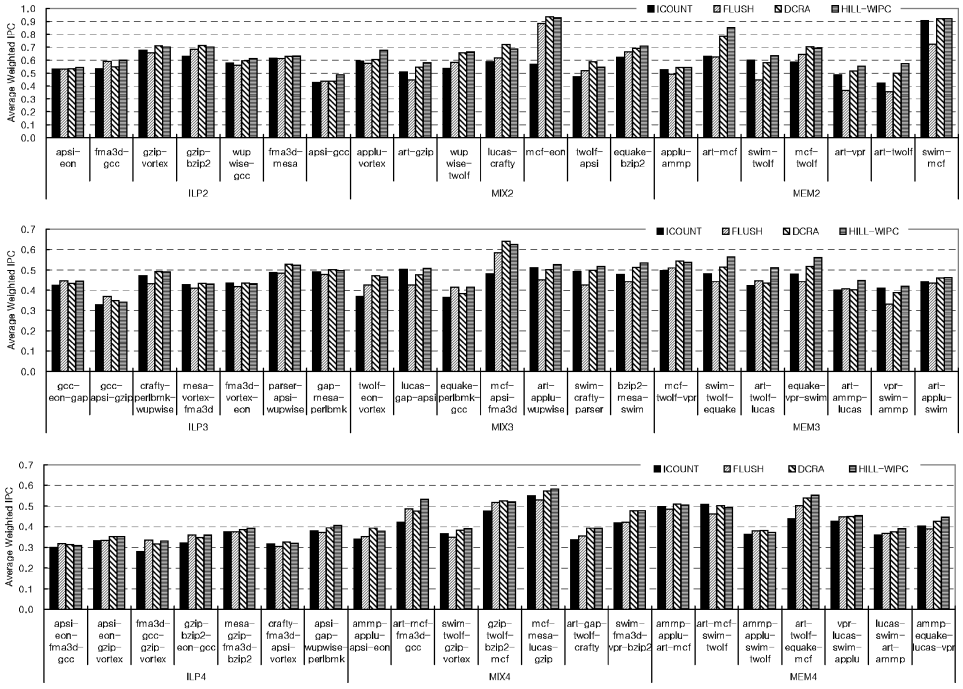

Fig. 6. HILL-WIPC vs. ICOUNT, FLUSH, and DCRA under the average weighted IPC metric for all 63 multiprogrammed workloads.

aggressive implementation, that is, offline measurement of $SingleIPC_i$ and hardware implementation of hill-climbing, to illustrate the best performance our technique can achieve. Then we study the performance of the less aggressive implementations.

4.2 Experimental Results

Figures 6, 7, and 8 present our hill-climbing SMT resource distribution performance results. Each graph compares our hill-climbing technique against Icount, Flush, and DCRA on all 63 multiprogrammed workloads under different performance metrics: average weighted IPC (Figure 6), average IPC (Figure 7), and harmonic mean of weighted IPC (Figure 8). The performance feedback metric our hill-climbing technique uses in each graph is different, and is always matched to the metric under which performance is evaluated for that graph. (For example, when evaluating performance under the average weighted IPC metric in Figure 6, hill-climbing also uses average weighted IPC as the performance feedback metric.) These different versions of hill-climbing are labeled "HILL-WIPC," "HILL-IPC," and "HILL-HWIPC" in Figures 6–8, respectively.

We first examine performance under the average weighted IPC metric evaluated by Figure 6. Comparing the HILL-WIPC, ICOUNT, and FLUSH bars in Figure 6, we find hill-climbing outperforms ICOUNT and FLUSH in 61 and 57 workloads, respectively (i.e., hill-climbing is better in 90–96% of the workloads). Averaged across all 63 workloads, HILL-WIPC achieves a gain of 11.4% and

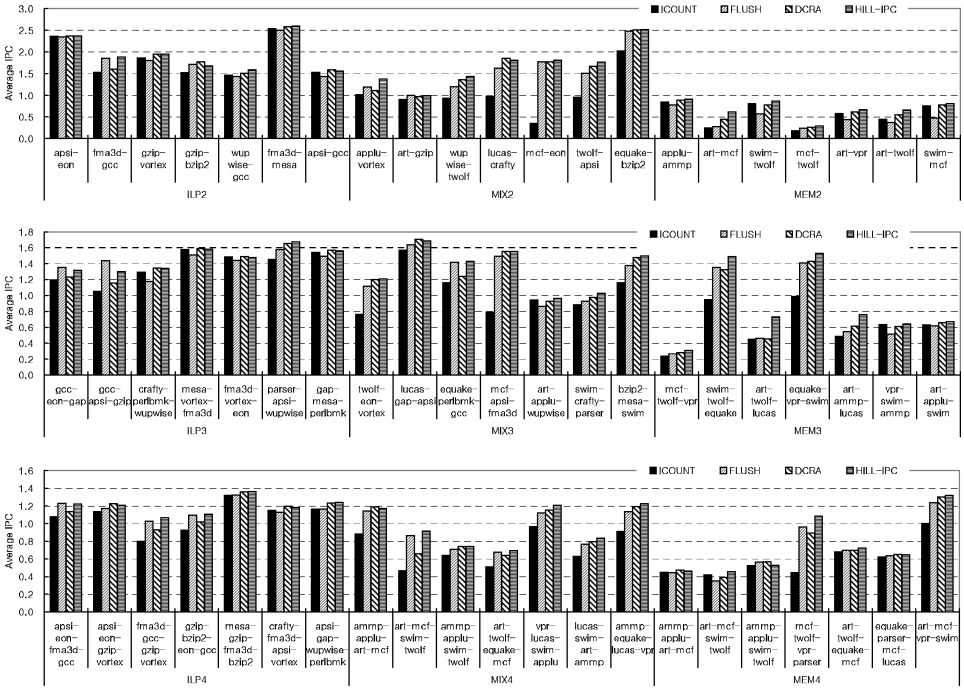

Fig. 7. HILL-IPC vs. ICOUNT, FLUSH, and DCRA under the average IPC metric for all 63 multiprogrammed workloads.

11.5% over ICOUNT and FLUSH, respectively, under average weighted IPC. Comparing the HILL-WIPC and DCRA bars in Figure 6, we find hill-climbing outperforms DCRA in 37 workloads (i.e., hill-climbing is better in 58% of the workloads). Averaged across all 63 workloads, HILL-WIPC achieves a more modest gain of 2.8% over DCRA.

The results are similar, if not slightly better, for the other two performance metrics. Comparing the HILL-IPC, ICOUNT, and FLUSH bars in Figure 7, we find hill-climbing outperforms ICOUNT and FLUSH in 62 and 57 workloads, respectively. Averaged across all 63 workloads, HILL-IPC achieves a gain of 23.7% and 10.7% over ICOUNT and FLUSH, respectively, under average IPC. Comparing the HILL-IPC and DCRA bars in Figure 7, we find hill-climbing outperforms DCRA in 47 workloads. Averaged across all 63 workloads, HILL-IPC achieves a gain of 5.9% over DCRA. Finally, comparing the HILL-HWIPC, ICOUNT, and FLUSH bars in Figure 8, we find hill-climbing outperforms ICOUNT and FLUSH in 61 and 62 workloads, respectively. Averaged across all 63 workloads, HILL-HWIPC achieves a gain of 18.3% and 13.6% over ICOUNT and FLUSH, respectively, under harmonic mean of weighted IPC. Comparing the HILL-HWIPC and DCRA bars in Figure 8, we find hill-climbing outperforms DCRA in 38 workloads. Averaged across all 63 workloads, HILL-HWIPC achieves a gain of 2.5% over DCRA.

Not only does hill-climbing achieve performance gains across our entire workload suite, but its performance advantage is robust in the face of diverse

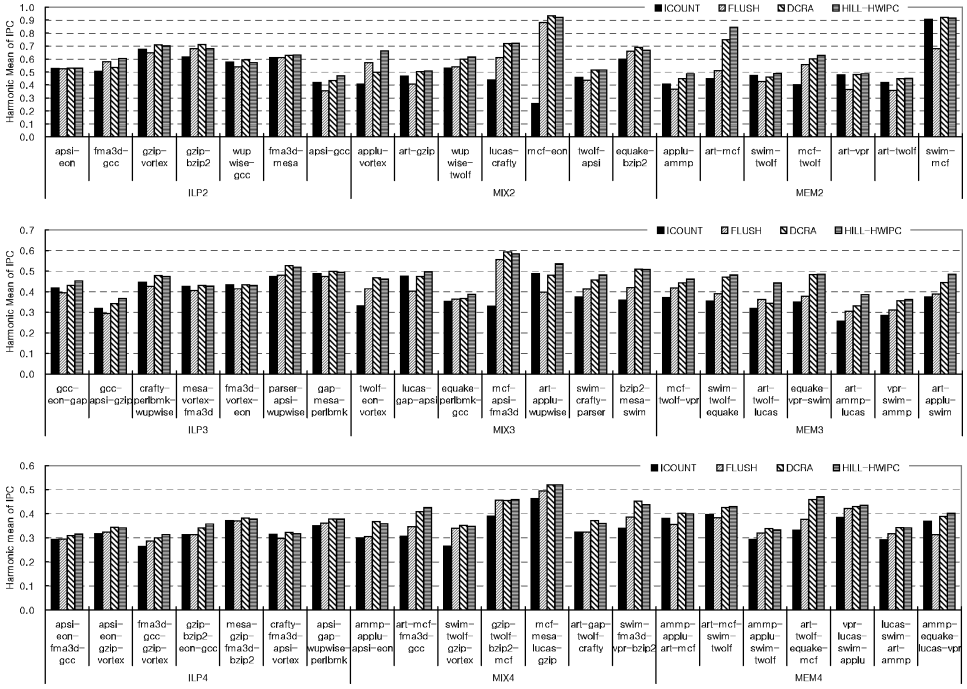

Fig. 8. HILL-HWIPC vs. ICOUNT, FLUSH, and DCRA under the harmonic mean of average weighted IPC metric for all 63 multiprogrammed workloads.

workload behaviors. To illustrate this point, Figure 9 reports performance averaged across different groups of workloads from Figures 6–8. Specifically, the groups labeled "ALL" in Figures 9(a)–(c) report average performance across all 63 workloads in Figures 6–8, respectively. The groups labeled "2-Thd," "3-Thd," and "4-Thd" in Figures 9(a)–(c) report average performance across the 2-, 3-, and 4-thread workloads in Figures 6–8, respectively. And the groups labeled "ILP," "MIX," and "MEM" in Figures 9(a)–(c) report average performance across the ILP, MIX, and MEM workloads in Figures 6–8, respectively. As Figure 9 shows, hill-climbing outperforms ICOUNT, FLUSH, and DCRA in *all* 21 groups. Regardless of the workload's thread count, the workload's memory behavior, or the metric under which performance is considered, hill-climbing consistently maintains a performance advantage.[5] This is a fairly positive result, given the diversity of workloads and metrics represented by the different groups in Figure 9. Overall, based on the results presented in Figures 6–9, we conclude that hill-climbing SMT resource distribution outperforms existing techniques across a large and diverse suite of workloads, and for a diverse set of performance goals.

As mentioned earlier, an important feature of our hill-climbing technique is its ability to employ different performance feedback metrics. Figure 10 studies the importance of this feature. In Figure 10, we compare the HILL-WIPC,

[5]Hill-climbing and DCRA achieve essentially the same performance for 4-thread workloads under the harmonic mean of weighted IPC metric, but even in this case hill-climbing holds a 0.3% performance advantage.

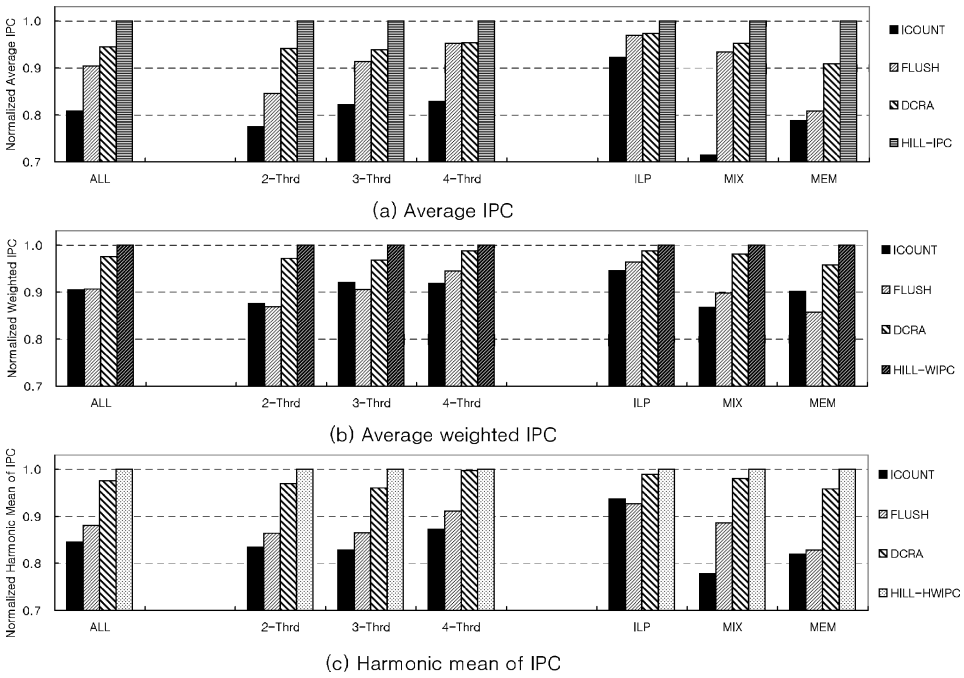

Fig. 9. Performance under: (a) average weighted IPC; (b) average IPC; and (c) harmonic mean of weighted IPC for different groups of workloads in Figures 6, 7, and 8, respectively. In each graph, the first group, labeled ALL, reports average performance across all 63 workloads. The remaining 6 groups report average performance across 2-, 3-, and 4-thread workloads and ILP, MIX, and MEM workloads.

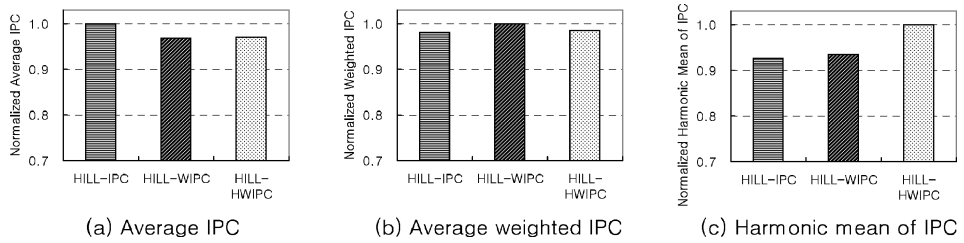

Fig. 10. Comparison of HILL-WIPC, HILL-IPC, and HILL-HWPIC under the: (a) average weighted IPC; (b) average IPC; and (c) harmonic mean of weighted IPC metrics.

HILL-IPC, and HILL-HWIPC bars from the ALL groups in Figures 9(a)–(c) against each other under the average weighted IPC, average IPC, and harmonic mean of weighted IPC metrics. As Figure 10 shows, hill-climbing achieves its best performance under a given metric when using the same metric for performance feedback. When both evaluation and feedback metrics are matched, hill-climbing performs 5.9% better then when they are not matched. This ability to *directly optimize* for a particular performance goal is crucial to hill-climbing's performance. In contrast, existing techniques cannot optimize for a particular performance goal.

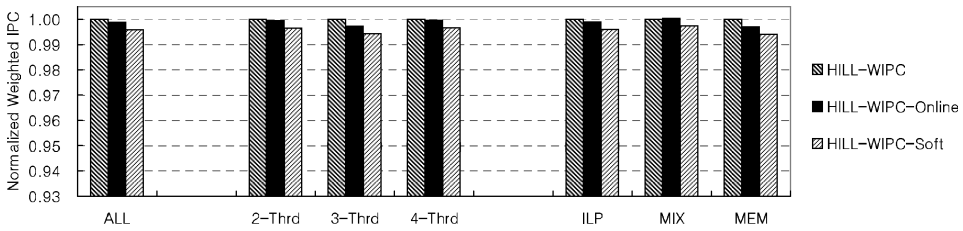

Fig. 11. Runtime overhead of online $SingleIPC_i$ sampling and software implementation of the hill-climbing algorithm. The HILL-WIPC bars are identical to the corresponding experiments in Figure 9(a). The HILL-WIPC-Online bars sample the stand-alone IPC values online. The HILL-WIPC-Soft bars in addition execute the hill-climbing algorithm in software handlers. All bars are normalized against the Hill-WIPC bars.

4.2.1 *Online Stand-Alone IPC Sampling.* The results presented in this section thus far optimistically assume: (a) offline measurement of the stand-alone IPC ($SingleIPC_i$) values necessary for the average weighted IPC and harmonic mean of weighted IPC performance feedback metrics, and (b) hardware implementation of the hill-climbing algorithm. We now investigate the impact that relaxing these assumptions has on performance.

First, we evaluate online $SingleIPC_i$ sampling using the approach described in Section 3.2. By sampling stand-alone IPC online, we remove the need for separate profiling runs, which are impractical for most general-purpose applications. Figure 11 shows the results of our experiments. In Figure 11, we compare HILL-WIPC (these are identical to the HILL-WIPC bars in Figure 9(a)) against an implementation of HILL-WIPC that performs $SingleIPC_i$ sampling online, labeled "HILL-WIPC-Online." HILL-WIPC-Online samples each thread's stand-alone IPC by disabling the other $T - 1$ threads for an entire epoch every 40 epochs (see Section 3.2). To exclude the impact of cache warmup and pipeline flush effects, we measure the stand-alone IPC during the second half of the sampling epoch only. All bars in Figure 11 are normalized to the HILL-WIPC bars, and are presented using the same groups of bars as Figure 9. (We only report results for HILL-WIPC since the results for Hill-HWIPC, are almost identical).

Figure 11 shows that HILL-WIPC-Online essentially matches HILL-WIPC in performance. Averaged across all 63 workloads, there is only a 0.12% performance degradation due to online $SingleIPC_i$ sampling. At worst, for the MEM workloads, there is a 0.29% performance degradation. These small performance reductions are due to the runtime overheads incurred by online sampling. One source of overhead comes from disabling $T - 1$ threads during the sampling epochs; another source of overhead arises from discrepancies between the sampled and actual stand-alone IPCs. Fortunately, Figure 11 demonstrates these overheads are extremely small. Based on these results, we conclude online sampling of stand-alone IPC has a negligible impact on performance.

4.2.2 *Software Implementation of Hill-Climbing.* In addition to offline measurement of $SingleIPC_i$, another assumption we have made is that the hill-climbing algorithm from Figure 5 is implemented in hardware, and incurs no runtime overhead. We now evaluate the software implementation described

in Section 3.3, which delivers an interrupt to the SMT processor every epoch and executes the hill-climbing algorithm in a software handler. Our experiments account for the software handler's runtime cost by stalling the entire SMT processor for 200 cycles at the end of every epoch. We found each 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. Moreover, our study is particularly conservative when one considers the fact that an actual software implementation would stall only a single thread, not the entire machine.

We add the overhead of a software hill-climbing algorithm to the online $SingleIPC_i$ sampling technique discussed in Section 4.2.1. In Figure 11, the bars labeled "HILL-WIPC-Soft" report the performance of this combined technique. Comparing the HILL-WIPC-Online and HILL-WIPC-Soft bars, we see that software implementation incurs a 0.3% performance penalty. This performance reduction is constant across workload groups since the 200-cycle per-epoch stall is fixed regardless of workload type. Based on these results, we conclude that software implementation of the hill-climbing algorithm has a negligible impact on performance, and is an attractive alternative to our hardware implementation. Comparing the HILL-WIPC and HILL-WIPC-Soft bars, we see the aggregate runtime overhead for both online $SingleIPC_i$ sampling and software implementation is only 0.42%. Hence, HILL-WIPC-Soft (the least aggressive of our implementations) essentially matches the performance of HILL-WIPC evaluated in Figures 6–10.

5. LIMITS OF HILL-CLIMBING SMT RESOURCE DISTRIBUTION

Section 4 presented an in-depth evaluation of hill-climbing SMT resource distribution; however, it did not provide any detailed insight into the source of hill-climbing's performance advantage. The next two sections investigate how hill-climbing achieves its performance gains. We begin by introducing several offline learning algorithms that always achieve the best (or near-best) resource partitionings with zero overhead, that is, they have infinite learning speed (Section 5.1). Using these offline algorithms, we conduct a limit study of learning-based techniques like hill-climbing under ideal conditions (Sections 5.2 and 5.3), and investigate the source of observed performance gains (Section 5.4). Then in Section 6, we will study why online hill-climbing cannot achieve the performance limit defined by our offline algorithms. Together, these two sections will provide the deeper understanding we seek of hill-climbing's performance reported in Section 4.

5.1 Offline Learning Algorithms

Hill-climbing chooses a resource partitioning at the beginning of each epoch based on performance feedback acquired from previously executed epochs, as illustrated in Figure 4. In contrast, our offline learning algorithms choose partitionings based on performance feedback from the *currently executing epoch*. At the beginning of each epoch, we try several partitionings (i.e., the dashed lines in Figure 4) for the current epoch offline, and identify the one with the highest

Table IV. Different Offline Learning Algorithms

	2 Threads	3 Threads	4 Threads
Space Size	256	32,896	2,862,206
OFF-LINE-Exhaust	128	—	—
OFF-LINE-Uniform	—	496 (S = 8)	680 (S = 16)
OFF-LINE-Hill	—	128	128
OFF-LINE-UniformHi (every 64th epoch)	-	2,016 (S = 4)	5,456 (S = 8)

This table includes the number of partitionings of the 256 integer rename registers they sample offline per epoch for 2-, 3-, and 4-thread workloads. The step size parameters for uniform sampling are indicated in parentheses. The row labeled "Space Size" indicates the total number of unique partitionings of the rename registers.

measured performance. Using this best sampled partitioning, we advance the machine state to the next epoch. The execution time of the best partitioning is charged to wall-clock time, while the cost of sampling all other partitionings offline is ignored. Then the process repeats for subsequent epochs. (Such ideal offline learning is impractical for real machines but is feasible through simulation, as we will describe in Section 5.2.)

Our limit study employs three offline learning algorithms that differ in how they explore the resource partitioning space. The most aggressive algorithm, OFF-LINE-Exhaust, tries *all* possible partitionings at every epoch. Because its exploration is exhaustive, OFF-LINE-Exhaust always achieves the best partitioning. Moreover, its exhaustive samples fully characterize performance variation across the entire resource partitioning space. (As we will see in Sections 5.4 and 6, such performance variation information can provide enormous insight into hill-climbing's performance.) It is important to note, however, that OFF-LINE-Exhaust is not an optimal algorithm because it does not find the best sequence of partitionings globally across all epochs; it only finds the best partitioning locally at each epoch. Nonetheless, OFF-LINE-Exhaust is a reasonable upper bound for our limit study, since hill-climbing itself only optimizes locally as well.

Ideally, we would conduct our entire limit study using OFF-LINE-Exhaust. Unfortunately, this is impossible due to the high cost of exhaustive simulation. In Table IV, the row labeled "Space Size" reports the number of unique ways that 256 rename registers can be partitioned across 2, 3, and 4 threads. (Like hill-climbing, our offline algorithms explicitly partition the rename registers, and then apply the same partitioning to the IQ and ROB proportionally.)[6] While it is feasible to simulate OFF-LINE-Exhaust for 2 threads, the size of the resource partitioning spaces for 3 and 4 threads, as shown in Table IV, is far too large to explore exhaustively in simulation.

[6]The space size values in Table IV for $T = 3$ and 4 are not equal to $256^{(T-1)}$, the expression given in Section 3.1 (after substituting $E_{max} = 256$), for two reasons. First, $256^{(T-1)}$ does not take into consideration the constraint that the number of rename registers allocated to the T threads must sum to 256. In geometric terms, this constrains the feasible partitionings in the resource partitioning space to be confined within a triangle in 2-space (for 3 threads) and a tetrahedron in 3-space (for 4 threads). The size of these feasible partitioning spaces is $\frac{256^2}{2}$ and $\frac{256^3}{6}$ for 3 and 4 threads, respectively. Second, our expression also does not account for a small number of corner cases that arise due to the discrete nature of the resource partitioning spaces.

To study 3- and 4-thread workloads, we employ two additional offline algorithms that explore only a subset of the resource partitioning space via sampling to mitigate simulation time. The simplest, OFF-LINE-Uniform, performs sampling uniformly with some step size, S. OFF-LINE-Uniform tries every S^{th} partitioning along each dimension of the resource partitioning space (i.e., for each thread), thus down-sampling in total by a factor $S^{(T-1)}$. The advantage of OFF-LINE-Uniform is that it covers the whole resource partitioning space, so its samples can be used to characterize performance variation across the entire space. However, OFF-LINE-Uniform may miss the best partitioning in some epochs, since down-sampling skips over many partitionings.

In addition to OFF-LINE-Uniform, another algorithm, OFF-LINE-Hill, performs sampling using randomized hill-climbing. OFF-LINE-Hill invokes multiple hill-climbing passes within the same epoch to search for the best partitioning. Each pass executes the hill-climbing algorithm in Figure 5. However, instead of acquiring the performance feedback samples across different epochs online, all the performance feedback samples are acquired from the same epoch offline (i.e., iterations of the outer-loop in line 6 from Figure 5 are performed on the same epoch). Furthermore, when a hill-climbing pass reaches a peak, a new hill-climbing pass is initiated from a randomly chosen point in the resource partitioning space. Lastly, sampling for a particular epoch terminates after a predetermined number of samples have been acquired. By performing multiple hill-climbing passes from random points, OFF-LINE-Hill has a high likelihood of finding the best partitioning, even when the resource partitioning space is complex and contains multiple local maxima (see Section 6.1). However, unlike OFF-LINE-Uniform, OFF-LINE-Hill does not explore the entire resource partitioning space, so its samples cannot be used to characterize performance variation.

5.2 Limit Study Methodology

We modified the SMT simulator described in Section 4.1 to support our offline learning algorithms. The primary modification is to add checkpointing of simulator state at epoch boundaries. At the beginning of each epoch, we checkpoint the entire simulator (including architectural registers and main memory, as well as microarchitecture state such as rename registers, pipeline registers, branch predictors, caches, etc.), and allow the simulator to roll-back to this checkpoint after executing the epoch. This capability enables multiple executions of each epoch using different resource partitionings, which is the main requirement for offline learning.

In addition, we implemented all three offline learning algorithms described in Section 5.1. We always use OFF-LINE-Exhaust to study 2-thread workloads. For 3- and 4-thread workloads, we use OFF-LINE-Hill to find the best partitioning in each epoch, and we use OFF-LINE-Uniform to study performance variation across the resource partitioning space. Table IV reports the number of samples per epoch acquired by each offline algorithm. For OFF-LINE-Exhaust, we acquire 128 samples per epoch instead of 256. We found that sampling every other partitioning has no measurable impact on OFF-LINE-Exhaust, so

Table V. Ratios of OFF-LINE-Hill Weighted IPC to OFF-LINE-UniformHi Weighted IPC

# Threads	# Sample Epochs	Average	Std. Dev.
2	238	0.9998	0.0002
3	151	0.9999	0.0002
4	133	0.9999	0.0001
Overall	522	0.9999	0.0002

Weighted IPC ratios are computed across all 63 workloads.

we use the lower sampling rate to conserve simulation time. For OFF-LINE-Hill, we sample in each epoch using randomized hill-climbing, as described in Section 5.1, until 128 total samples are acquired. For OFF-LINE-Uniform, we down-sample by a factor of 8 for 3-thread workloads, and by a factor of 16 for 4-thread workloads. This results in 496 and 680 total samples per epoch for 3- and 4-thread workloads, respectively.

Our limit study is driven by the 63 multiprogrammed workloads listed in Table III. We use the same method described in Section 4.1 for identifying and fast-forwarding to our simulation regions. However, once in detailed simulation mode, we simulate for 100 million instructions instead of 1 billion instructions. In addition, we conduct our limit study for the average weighted IPC metric only. These changes were necessary, given the extremely high cost of simulating offline learning compared to hill-climbing.

Finally, an important question for our limit study is: How good are the partitionings found by OFF-LINE-Hill? To help address this question, we created another offline learning algorithm based on OFF-LINE-Uniform, called OFF-LINE-UniformHi. As shown in Table IV, OFF-LINE-UniformHi has a higher sampling rate than OFF-LINE-Uniform, down-sampling by a factor of 4 for 3-thread workloads and 8 for 4-thread workloads. This higher sampling rate explores 2,016 and 5,456 partitionings per epoch for 3- and 4-thread workloads, respectively. Unfortunately, OFF-LINE-UniformHi is too expensive to run end-to-end; however, we ran it on select epochs analyzed by OFF-LINE-Hill. For every 64th epoch visited by OFF-LINE-Hill in the 3- and 4-thread workloads, we also ran OFF-LINE-UniformHi, and then compared the best partitioning found by both algorithms.

Table V reports the results of this experiment. In total, we compared OFF-LINE-Hill and OFF-LINE-UniformHi in 522 epochs under average weighted IPC. Averaged across all 522 epochs, the ratio of weighted IPC achieved by the best partitionings from the two algorithms is 0.9999, and the standard deviation is 0.0002. In other words, OFF-LINE-Hill and OFF-LINE-UniformHi find partitionings of essentially identical performance, despite using completely different offline algorithms. This independent validation provides some confidence that OFF-LINE-Hill (as well as OFF-LINE-UniformHi) is in fact finding partitionings of very high quality.

5.3 Limit Study Results

Figure 12 compares our ideal offline learning algorithms against ICOUNT, FLUSH, DCRA, and HILL-WIPC under the average weighted IPC metric. To

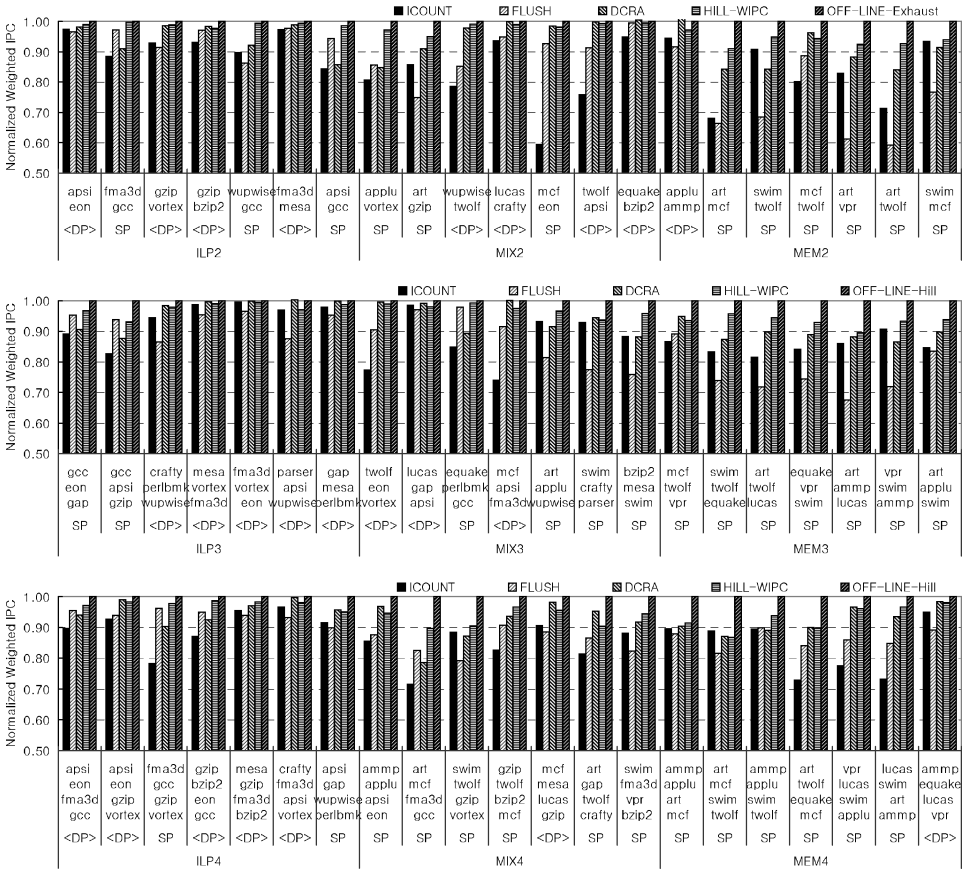

Fig. 12. Average weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC normalized against OFF-LINE-Exhaust (2-thread workloads) or OFF-LINE-Hill (3- and 4-thread workloads). The labels "SP" and "DP" indicate workloads with sharp peaks versus dull peaks, respectively.

find the performance limit, we use OFF-LINE-Exhaust for 2-thread workloads and OFF-LINE-Hill for 3- and 4-thread workloads, as discussed in Section 5.2. All bars from the same workload have been normalized to the offline algorithm from that workload (the "SP/DP" labels appearing underneath each set of bars will be explained later). Notice, the results for ICOUNT, FLUSH, DCRA, and HILL-WIPC in Figure 12 are qualitatively the same as the results in Figure 6, despite the fact that these two sets of experiments use differently sized simulation windows (100M and 1B instructions, respectively). Because we use SimPoint to pick representative simulation windows, the difference in window size has very little impact on our results. This implies the insights we are about to gain from our limit study are also relevant for explaining the hill-climbing performance we observed in Section 4.

Comparing offline learning against ICOUNT and FLUSH in Figure 12, we see offline learning outperforms both techniques in all 63 workloads, providing an average performance gain of 16.5% and 17.2%, respectively. Comparing offline learning against DCRA, we see offline learning outperforms DCRA in 59

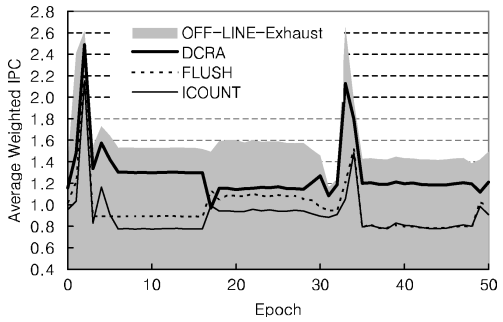

Fig. 13. Synchronized time-varying performance of offline learning, DCRA, FLUSH, and ICOUNT from the art-mcf workload.

out of 63 workloads (all but equake-bzip2, applu-ammp, parser-apsi-wupwise, and mcf-apsi-fma3d), providing a performance gain of 7.4% on average. Finally, comparing offline learning against HILL-WIPC, we see offline learning outperforms HILL-WIPC in all 63 workloads, providing a performance gain of 4.4% on average.

In addition to end-to-end performance, we also compared performance across individual epochs to see how pervasively (within each workload) offline learning achieves its performance gains. Because the different SMT techniques in Figure 12 execute at different rates, it is impossible to compare per-epoch performance in our end-to-end simulations. To enable our study, we "synchronized" all the SMT techniques using the checkpoints acquired by offline learning. At the beginning of every epoch, we simulate ICOUNT, FLUSH, and DCRA for 1 epoch starting from the same checkpoint used by OFF-LINE-Exhaust and OFF-LINE-Hill. This yields a time-varying performance profile, as illustrated in Figure 13. Comparing WIPCs from the same epoch in Figure 13 is meaningful because all the SMT techniques are synchronized to a common execution point at all times. (We also verified that synchronization does not noticeably alter the end-to-end performance of ICOUNT, FLUSH, and DCRA compared to Figure 12.) We performed synchronized versions of all the experiments for the existing SMT techniques in Figure 12, and then compared performance in every epoch. Across all 63 workloads, we found offline learning outperforms ICOUNT and FLUSH in 100% of the epochs. Offline learning also outperforms DCRA in 97.2% of the epochs.

Based on Figure 12 and the synchronized experiments, we make two important observations. First, our results demonstrate both OFF-LINE-Exhaust and OFF-LINE-Hill achieve extremely high-quality resource partitionings, consistently outperforming ICOUNT, FLUSH, DCRA, and HILL-WIPC when considering end-to-end performance. Moreover, our synchronized experiments show offline learning's performance advantage over existing SMT techniques occurs in essentially every epoch. Hence, we conclude that while our offline algorithms are not optimal, they nevertheless provide a reasonable target against which our limit analyses can be conducted.

Second, our results also demonstrate there exists additional room to improve online hill-climbing. The performance advantage of our offline algorithms

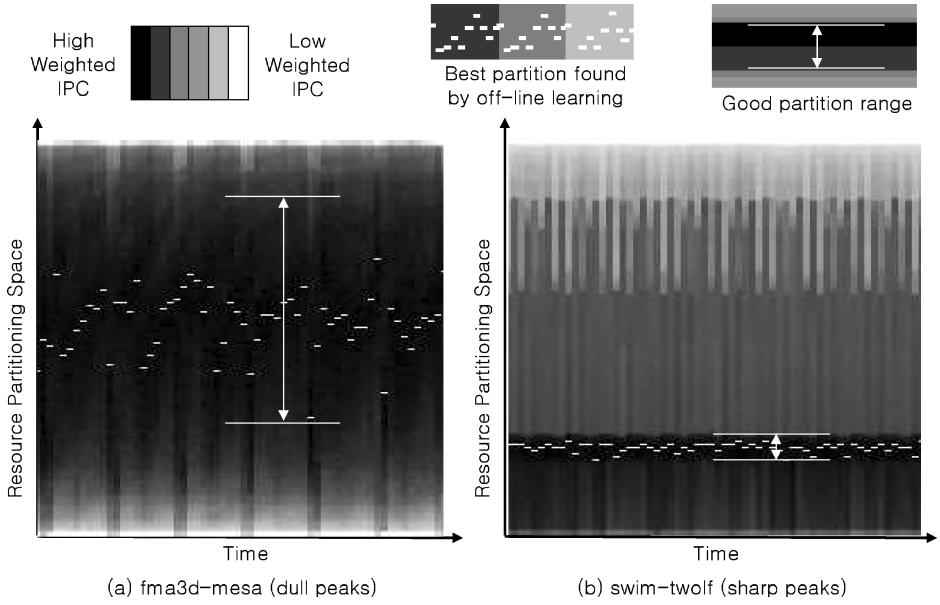

Fig. 14. (a) fma3d-mesa exhibits dull peaks; (b) swim-twolf exhibits sharp peaks.

over ICOUNT, FLUSH, and DCRA (16.5%, 17.2%, and 7.4%, respectively) are larger than the corresponding performance gains observed for our hill-climbing technique in Section 4 (11.4%, 11.5%, and 2.8%, respectively). The discrepancy is most pronounced for DCRA, where the additional "performance headroom" is almost twice the actual performance achieved by hill-climbing. Based on these results, we conclude that overheads are indeed incurred when learning online, preventing our hill-climbing technique from enjoying all the benefits that learning-based SMT resource distribution has to offer. In the next section, we investigate the source of offline learning's performance advantage in greater depth. Later in Section 6, we will provide more insight into the online learning overheads.

5.4 Hill Peak Analysis: Finding the Source of Performance Gains

As mentioned in Section 5.1, offline learning not only finds the best (or near-best) partitioning, it also characterizes how performance varies with resource partitioning because it samples a large number of different partitionings. Figure 14(a) illustrates such performance variation information for fma3d-mesa, one of our 2-thread workloads. In Figure 14(a), the white dots plot the best partitioning found by offline learning, in this case OFF-LINE-Exhaust, as a function of epoch ID (time). In addition, for every epoch, we also plot the weighted IPC for all other partitionings visited by offline learning using a gray scale: Lighter shades represent lower performance while darker shades represent higher performance. By following the change in gray scale along any vertical line in Figure 14(a), we can determine the shape, of the performance

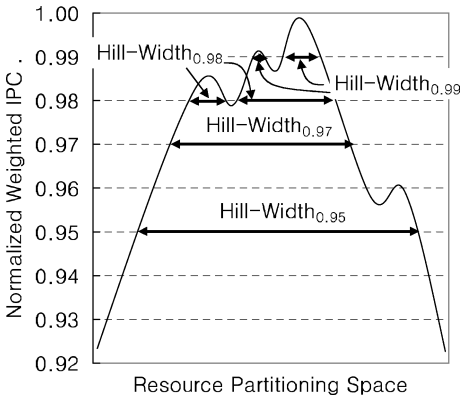

Fig. 15. Hill-width$_{0.99}$, hill-width$_{0.97}$, and hill-width$_{0.95}$ for a single epoch from a hypothetical 2-thread workload.

variation within the corresponding epoch. Typically, intra-epoch performance varies as a hill, transitioning from light regions to dark regions, and then back to light regions again. Enormous insight into offline learning's performance (and hence, hill-climbing's performance) can be gained by studying these performance hills, including their shape, the number of peaks (i.e., local maxima), and their variation over time.

In this section, we analyze the sharpness of the performance peaks. Peak sharpness is important because it determines a workload's sensitivity to different resource partitionings. For example, fma3d-mesa in Figure 14(a) exhibits *dull peaks* because the maximum performance (darkest shade) spans a wide region of resource partitionings, as indicated in the figure. So, many resource partitionings achieve similar performance to the best partitioning. This low performance sensitivity implies it is easy to do well in fma3d-mesa (almost any partitioning will do). As a result, Figure 12 shows that existing SMT techniques almost match offline learning's performance. The situation is quite different for swim-twolf, whose performance variation is illustrated in Figure 14(b). In contrast to fma3d-mesa, swim-twolf exhibits *sharp peaks* because the maximum performance spans a narrow region of resource partitionings, as indicated in the figure. So, the best partitioning achieves much higher performance than most other resource partitionings. This high performance sensitivity implies it is difficult to do well in swim-twolf (only the best partitioning will do). As Figure 12 shows, existing SMT techniques are unable to find the best partitioning as frequently as does offline learning, so they cannot match offline learning's performance.

Peak sharpness strongly indicates the performance opportunities afforded learning-based techniques; hence, we characterize our workloads using sharpness analysis. To enable this study, we introduce the *hill-width$_N$ metric* for quantifying peak sharpness. Hill-width$_N$ is the fraction of resource partitionings whose performance is better than some specified performance level, N. Graphically speaking, hill-width$_N$ is the width of all hills at some altitude. Figure 15 illustrates this metric for a single epoch from a hypothetical 2-thread

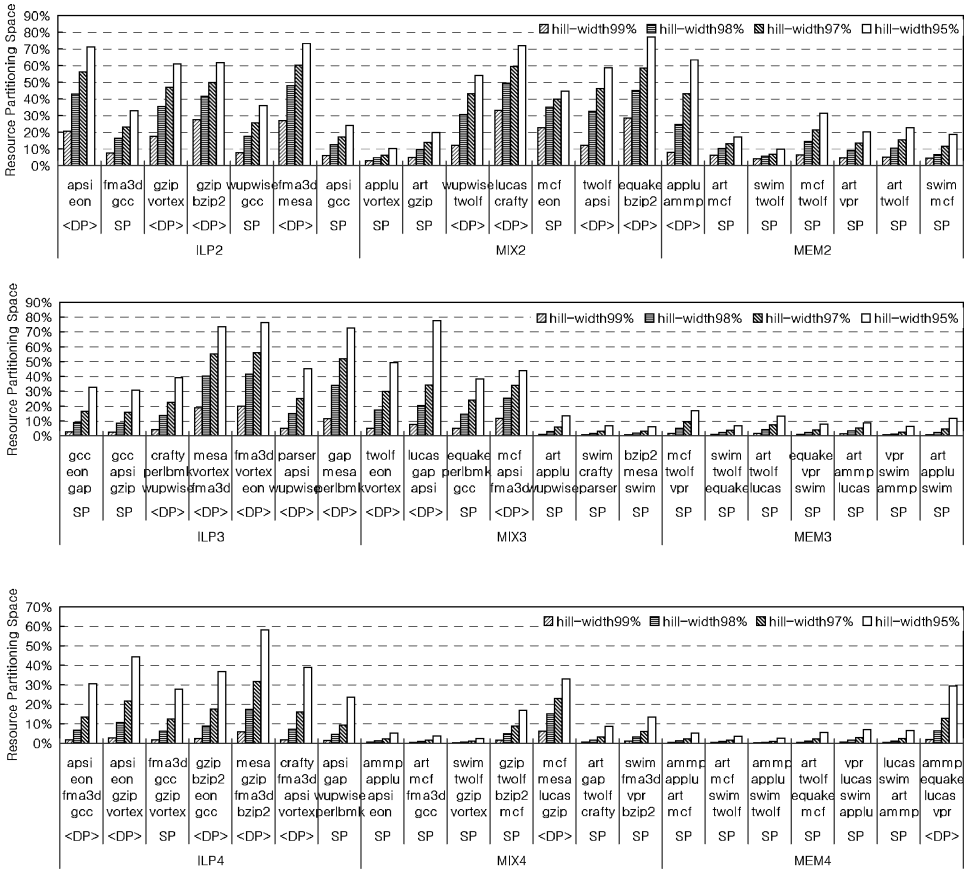

Fig. 16. Hill-width$_{0.99}$, hill-width$_{0.98}$, hill-width$_{0.97}$, and hill-width$_{0.95}$ values across our workloads. The "SP" and "DP" labels indicate the sharp peak and dull peak workloads, respectively.

workload. In Figure 15, we plot WIPC normalized to the maximum WIPC as a function of resource partitioning normalized to the total number of partitionings. Hill-width$_{0.99}$, hill-width$_{0.98}$, hill-width$_{0.97}$, and hill-width$_{0.95}$ are indicated in the figure.[7] While hill-width$_N$ quantifies peak sharpness, we also wish to summarize our sharpness analysis qualitatively. From our experience, low performance sensitivity occurs in workloads with hill-width$_{0.95} > 0.5$, 0.4, and 0.3 for 2-, 3-, and 4-thread workloads, respectively. So, we use these thresholds to qualitatively identify dull- and sharp-peak workloads.

Figure 16 reports the four hill-width$_N$ values illustrated in Figure 15 across our entire suite. Each bar represents a hill-width$_N$ value averaged across all epochs from its corresponding workload. Using the hill-width$_N$ thresholds discussed previously, we identify each workload as exhibiting either sharp peaks or dull peaks, labeled "SP" or "DP," respectively. In Figure 16, we see 39 out of

[7]Figure 15 illustrates the hill-width$_N$ metric for a 2-thread workload. The metric can easily be extended to 3- and 4-thread workloads by performing the same analysis for 3D and 4D hills.

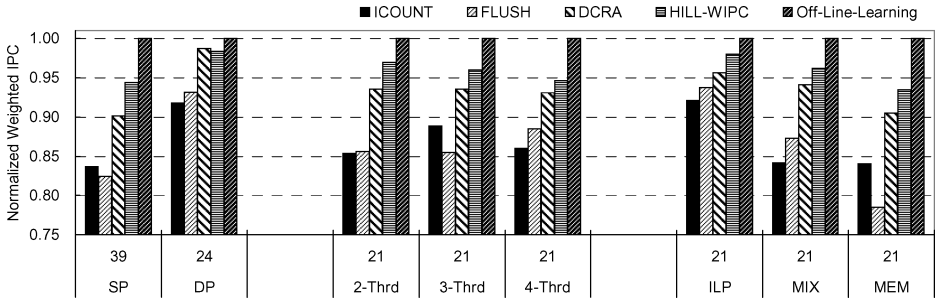

Fig. 17. The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC normalized against offline learning. Results are reported for sharp- and dull-peak workloads, 2-, 3-, and 4-thread workloads, and ILP, MIX, and MEM workloads.

63 workloads exhibit sharp peaks, while the remaining 24 workloads exhibit dull peaks. We copy the corresponding "SP" and "DP" labels from Figure 16 into Figure 12, thus enabling a side-by-side comparison of these two results.

Figures 12 and 16 show there exists a strong correlation between peak sharpness and performance potential across our entire suite. For the sharp-peak workloads, offline learning tends to outperform existing SMT techniques by significant margins; for the dull-peak workloads, offline learning and existing SMT techniques tend to achieve similar performance. To further illustrate this point, Figure 17 reports performance averaged across the sharp- and dull-peak workloads from Figure 16. The groups of bars labeled "SP" in Figure 17 show that offline learning outperforms Icount, Flush, and DCRA by 19.5%, 21.3%, and 10.9% respectively, for sharp-peak workloads, while the groups of bars labeled "DP" show the performance advantage reduces to 8.9%, 7.4%, and 1.3% for dull-peak workloads. This result confirms our earlier intuition: Certain workloads achieve their highest performance only for a small number of resource partitionings. Offline learning's ability to find these best partitionings is the main source of its performance advantage.

This intuition seems to also apply to our hill-climbing technique. The groups of bars labeled "SP" in Figure 17 show HILL-WIPC outperforms ICOUNT, FLUSH, and DCRA by 12.8%, 14.4%, and 14.7% for sharp-peak workloads, while the groups of bars labeled "DP" show the performance advantage reduces to 7.2%, 5.6%, and −0.4% (i.e., a 0.4% degradation) for dull-peak workloads. Not only are certain workloads more performance sensitive to high-quality partitionings, but hill-climbing is capable of finding, at the very least, good partitionings in these cases. Its ability to do so is the main source of its performance advantage reported in Section 4.

5.4.1 *Sharp Peak Sources.* An important question is: What causes the high performance sensitivity responsible for the performance opportunities in the sharp-peak workloads? And why can hill-climbing exploit these opportunities while existing SMT techniques cannot? Figure 16 contains some of the clues. In Figure 16, we see a majority of the sharp-peak workloads consist of at least one memory-bound application. Practically all MEM workloads and almost

two-thirds of MIX workloads are sharp-peak workloads. In contrast, only one-third of ILP workloads are sharp-peak workloads. To further illustrate this point, the groups of bars labeled "2-Thrd," "3-Thrd," "4-Thrd," "ILP," "MIX," and "MEM" in Figure 17 report the average performance across the 2-, 3-, and 4-thread workloads and ILP, MIX, and MEM workloads, respectively, from Figure 12. As the last three groups of bars in Figure 17 show, offline learning outperforms Icount, Flush, and DCRA by 18.9%, 20.6%, and 8.3% in the MIX and MEM workloads, while the performance advantage reduces to 8.6%, 6.7%, and 4.6% in the ILP workloads. Moreover, this trend persists across different numbers of threads: Offline learning's advantage over ICOUNT, FLUSH, and DCRA is roughly constant across the 2-, 3-, and 4-thread workloads in Figure 17. These results indicate that optimizing resource partitioning for memory-bound workloads offers the largest potential performance gains, and constitutes the main source of sharp peaks in Figure 16.

Upon closer examination of the MIX and MEM workloads, we found learning-based techniques (i.e., offline learning and hill-climbing) achieve large performance gains for memory-bound workloads because they exploit memory parallelism via *cache-miss clustering* more effectively than do existing SMT techniques. Cache-miss clustering occurs whenever multiple memory loads from the same thread appear in the instruction window and trigger cache misses 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. Learning-based techniques learn the best action, that is, to contract a thread's partition to prevent clogging or aggressively increase a thread's partition to exploit memory parallelism, individually for each workload. Existing techniques conservatively try to prevent resource clogging, possibly missing performance in epochs with memory parallelism.

While MIX and MEM workloads offer the greatest opportunities, we also carefully examined the ILP workloads for which offline learning and hill-climbing outperform existing SMT techniques. We found learning-based techniques exploit *compute-intensive low-ILP threads* more effectively in these cases. ICOUNT, FLUSH, and DCRA tend to distribute resources to threads that cache-miss infrequently, namely 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 learning 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 noncache-missing threads the same, leading to subpeak performance.

6. ONLINE LEARNING OVERHEADS

While hill-climbing SMT resource distribution can find higher-quality partitionings than existing SMT techniques for sharp peak workloads, it still cannot find *the best* partitionings, as demonstrated by the superior performance of offline learning in Section 5.3. This performance discrepancy suggests our hill-climbing technique incurs overheads while learning online. In this section, we identify several types of online learning overheads (Section 6.1). We also introduce several metrics (Section 6.2) for characterizing the extent to which the online learning overheads occur in our workloads (Section 6.3).

6.1 Learning Speed Limitations

Our hill-climbing technique cannot match the performance of offline learning because online learning has limited learning speed, thus preventing our hill-climbing algorithm from always tracking the optimal partitionings. The speed at which hill-climbing learns the optimal partitionings depends largely on how performance varies across the resource partitioning space. Hence, insight into the learning speed limitations can be gained by studying the same performance variation information presented in Section 5.4 for peak sharpness analysis. Figure 18 shows performance variation information for several 2-thread workloads in a format similar to Figure 14. As in Figure 14, Figure 18 plots the best partitioning found by offline learning (in this case, OFF-LINE-Exhaust) as white dots, and indicates the weighted IPC for all other partitionings visited by OFF-LINE-Exhaust as a gray scale, thus showing the shape of the performance variation in each epoch. In addition, Figure 18 also plots the partitionings explored by our hill-climbing technique, HILL-WIPC, as "+" symbols. Both HILL-WIPC and OFF-LINE-Exhaust have been synchronized using the technique from Section 5.3.[8] So, by comparing the + symbols with the white dots, we can determine how closely HILL-WIPC tracks the optimal partitionings.

Figure 18(a) shows the performance variation information for a portion of swim-mcf where hill-climbing's learning speed, while limited, is sufficient to achieve high performance. In Figure 18(a), performance varies in a simple hill shape across the resource partitioning space, increasing monotonically from the top of the graph to the bottom. Since there is a clear performance gradient, our hill-climbing algorithm immediately moves towards the peak, and after a short time, achieves the optimal partitionings (i.e., the + symbols merge with the white dots). Moreover, the hill shape hardly changes over time. Hence, our hill-climbing algorithm closely tracks the optimal partitionings once it reaches the peak of the hill, thus maintaining the highest possible performance. For cases like Figure 18(a), hill-climbing can closely match the performance of offline learning.

While Figure 18(a) shows an ideal case, unfortunately, not all workloads permit such effective learning. Instead, some workloads exhibit more complex

[8]In Section 5.3 (in particular Figure 13), we synchronized existing techniques against OFF-LINE-Exhaust. Here, we synchronize OFF-LINE-EXHAUST against HILL-WIPC instead. By doing so, the + symbols in Figure 18 represent the actual path taken by hill-climbing during an execution of the workload, allowing us to study how hill-climbing performs learning.

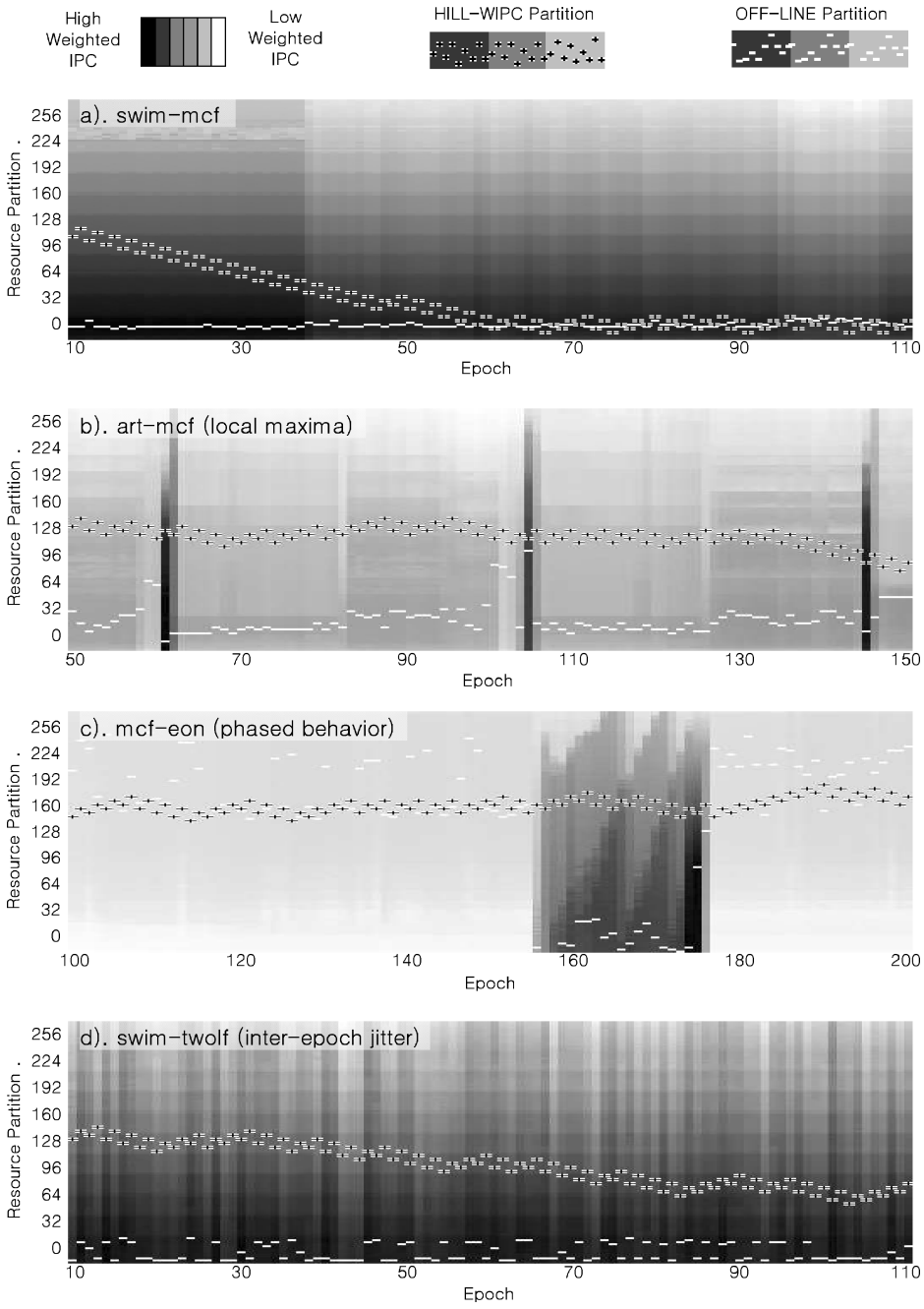

Fig. 18. Performance variation information for four 2-thread workloads: (a) swim-mcf exhibits fast learning speed; (b) art-mcf exhibits local maxima; (c) mcf-eon exhibits phased behavior; (d) swim-twolf exhibits interepoch jitter.

performance variation that results in performance degradation due to hill-climbing's finite learning speed. After carefully examining the performance variation information across all our workloads, we found three major types of performance variation that cause problems: local maxima, phased behavior, and interepoch jitter. Figure 18(b) shows the performance variation information for art-mcf, a workload that exhibits *local maxima*. In art-mcf, there are multiple performance peaks in the resource partitioning space, as indicated by the multiple bands of nonmonotonically varying gray scales in Figure 18(b). Hill-climbing gets "stuck" on one of the nonmaximal peaks, and hence does not move towards the optimal partitionings. We find most local maxima eventually dissipate as the workload changes behavior, allowing hill-climbing to move off the nonmaximal peaks. But the formation of local maxima, however ephemeral, impedes the progress of our hill-climbing technique towards the optimal partitionings, thus sacrificing performance relative to offline learning.

Figure 18(c) shows the performance variation information for mcf-eon, a workload that exhibits *phased behavior*. In Figure 18(c), we see mcf-eon experiences a long period of low performance, indicated by light gray scales, followed by a short period of high performance, indicated by dark gray scales. (These two program phases repeat throughout the workload.) Hill-climbing effectively tracks the optimal partitionings in the low-performing phase due to its long duration. However, the high-performing phase does not persist long enough for hill-climbing to be able to adapt; hence, hill-climbing cannot reach the optimal partitionings and misses significant performance opportunities. Because hill-climbing's finite learning speed limits its ability to track optimal partitionings during short-lived program phases, it cannot match offline learning performance for workloads like mcf-eon.

Finally, Figure 18(d) shows the performance variation information for swim-twolf, a workload that exhibits *interepoch jitter*. In Figure 18(d), we see swim-twolf has a single performance peak (no local maxima), and the optimal partitionings are fairly stable (no phased behavior). Nevertheless, hill-climbing still has trouble moving towards the optimal partitionings. Although the positive gradient within each epoch always points towards the maximal peak, the performance hills *between* adjacent epochs are slightly offset from one another in a jittery fashion. This creates transient positive gradients across epochs that temporarily point away from the maximal peak. Such bogus gradients fool the hill-climber, causing it to reverse course occasionally and move away from the optimal partitionings.[9] Consequently, learning speed becomes limited, and hill-climbing is unable to match the performance of offline learning.

[9]Both phased behavior and interepoch jitter exhibit variation across epochs, but they do so in very different ways. In phased behavior, the performance hills across epochs (from adjacent phases) differ significantly. However, in interepoch jitter, the performance hills across epochs are essentially the same; the hill-climber is fooled only because the performance hills (from adjacent epochs) shift slightly relative to each other. Due to the difference between these two cases, we decided to distinguish them in two separate categories.

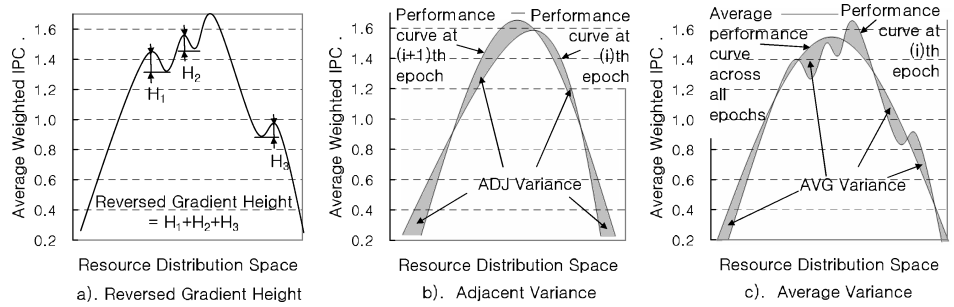

Fig. 19. (a) Reversed gradient height; (b) adjacent variance; and (c) average variance metrics.

6.2 Metrics

Having identified the different types of online learning overheads, we now introduce several metrics that characterize the extent to which the overheads occur in our workloads. Figures 19(a)–(c) illustrate the metrics, one for each learning speed problem discussed in Section 6.1. Similar to Figure 15, each graph in Figure 19 shows the performance variation information for a single epoch (normalized WIPC versus resource partitioning) from a hypothetical 2-thread workload. Labels indicate the features in the performance variation information that each metric measures.

Reversed gradient height. Figure 19(a) illustrates the Reversed Gradient Height (RGH) metric. RGH characterizes local maxima by measuring their total height. For example, in Figure 19(a), the performance variation information exhibits 3 nonmaximal peaks; H_1, H_2, and H_3 indicate the height of each nonmaximal peak's hillside facing the maximal peak (we refer to the slope of each such hillside as a *reverse gradient*, since it points away from the maximal peak). The RGH for this epoch is the sum $H_1 + H_2 + H_3$. A large RGH value signifies either a large number of local maxima, in which case there is a good chance of getting stuck on a nonmaximal peak, or very tall local maxima, in which case it becomes difficult to leave a nonmaximal peak after getting stuck. Overall, RGH quantifies the severity of the local maxima problem.

While the RGH metric is easily explained for 2-thread workloads, it's more complex for 3- and 4-thread workloads. Eq. (4) that follows expresses the RGH metric for the general case, regardless of the number of threads. This equation contains vectors defined within the resource partitioning space.[10] In particular, $\overrightarrow{P}$ is the vector defined by the optimal partitioning $\overrightarrow{C_i}$ is the vector defined by some partitioning i, and $\overrightarrow{C_{BN(i)}}$ is the vector defined by $BN(i)$, the partitioning with the largest performance amongst the neighbors of i. $WIPC_i$ and $WIPC_{BN(i)}$ are the weighted IPCs achieved by partitionings i and $BN(i)$, respectively.

[10]Note that the resource partitioning space is 1D for 2-threads, 2D for 3-threads, and 3D for 4-threads.

RGH =

$$\sum_{for\ all\ i} max(0, ((-1) \times \frac{(\overrightarrow{P} - \overrightarrow{C_i}) \cdot (\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i})}{|(\overrightarrow{P} - \overrightarrow{C_i})| \cdot |\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i}|} \times (WIPC_{BN(i)} - WIPC_i))) \quad (4)$$

Eq. (4) computes RGH by summing over all $(WIPC_{BN(i)} - WIPC_i)$, the performance increments at different partitionings, whose gradients $(\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i})$ point away from the direction to the optimal partitioning $(\overrightarrow{P} - \overrightarrow{C_i})$. The direction of the gradient relative to the optimal partitioning (either reverse or forward) is determined by the sign of the cosine of the angle (i.e., dot product of two vectors divided by their norm) between the gradient $(\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i})$ and the direction to the optimal partitioning $(\overrightarrow{P} - \overrightarrow{C_i})$. In particular, a negative cosine value signifies a reversed gradient. Also note, by multiplying the cosine value into Eq. 4, we incorporate a sense of *how* opposing the two vectors are.

Adjacent variance. Figure 19(b) illustrates the adjacent variance (ADJV) metric. ADJV characterizes interepoch jitter by measuring the change in performance variation information between adjacent epochs. For example, in Figure 19(b), we show the performance variation information for two back-to-back epochs, i and $i+1$, one offset from the other. The ADJV metric quantifies the performance variation change by measuring the *area* in between the two curves. A large ADJV value signifies a big performance variation change, or large interepoch jitter, while a small ADJV value signifies a small performance variation change, or small interepoch jitter.

Average variance. Figure 19(c) illustrates the average variance (AVGV) metric. Like ADJV, AVGV also measures the area in between two performance variation curves. However, instead of considering two adjacent epochs, AVGV considers a performance curve from an epoch i, and the *average* performance variation information curve (i.e., averaged across all epochs), as illustrated in Figure 19(c). By comparing a workload's AVGV with its ADJV, we can characterize phased behavior. A large AVGV in comparison to ADJV indicates changes in performance variation information do not occur between adjacent epochs, but rather across groups of similar epochs. This suggests phased behavior. On the other hand, a large ADJV in comparison to AVGV indicates changes in performance variation information occur predominantly between adjacent epochs, indicating jitter.

6.3 Workload Characterization

Figures 20 and 21 characterize the occurrence of online learning overheads in all 63 of our multithreaded workloads. For each workload, we analyze every epoch and compute its RGH, ADJV, and AVGV. Then, we average the computed values across all epochs from the workload. Figure 20 reports each workload's average RGH, while Figure 21 reports each workload's average ADJV and AVGV (in the bars labeled "ADJ Variance" and "AVG Variance," respectively). In addition, we summarize our learning overheads analysis. After examining numerous epochs across all our workloads, we found each learning problem tends to limit the learning speed of hill-climbing when the corresponding metric exceeds some

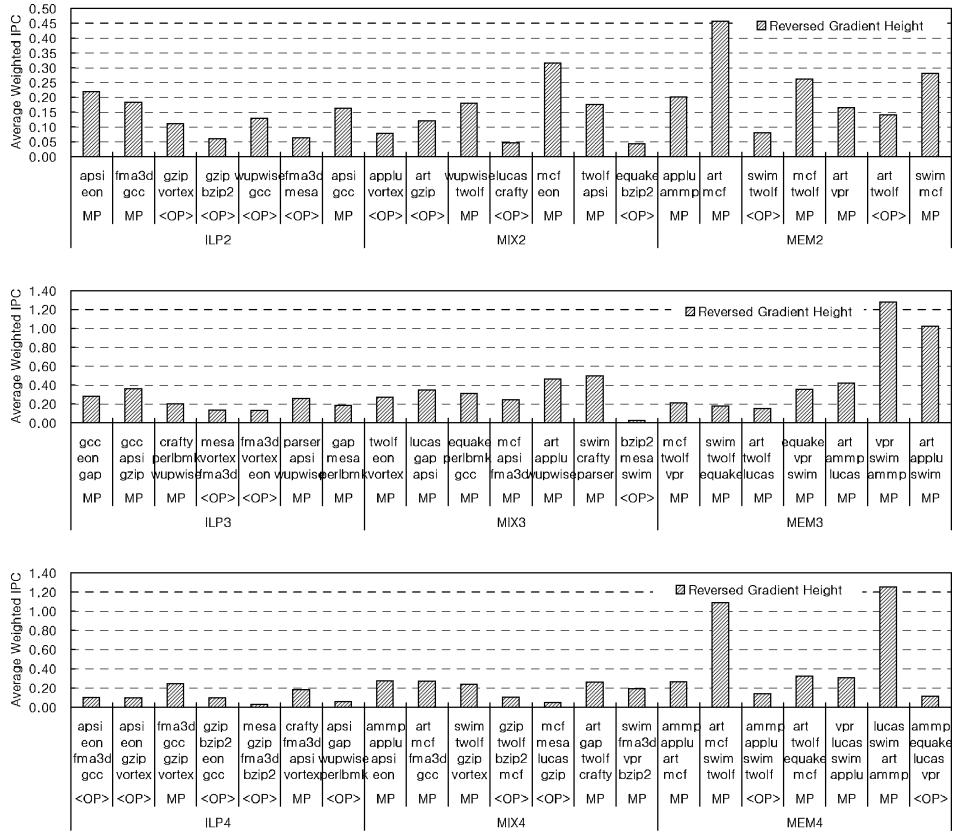

Fig. 20. Reversed gradient height for our 63 workloads. Labels characterize each workload as having epochs dominated by either one peak (OP) or multiple peaks (MP).

threshold. In particular, workloads with an average RGH $\geq$ 0.15 tend to be dominated by epochs with significant local maxima; workloads with an average ADJV $\geq$ 0.05 tend to exhibit significant interepoch jitter; and when a workload's AVGV is equal to or greater than twice its ADJV, the workload tends to exhibit phased behavior. Hence, based on these threshold comparisons, we label each workload as either "OP" (for one peak) or "MP" (for multiple peaks), "LJ" (for low jitter) or "HJ" (for high jitter), and "NP" (for not phased) or "P" (for phased), as indicated in Table VI. These labels appear below each workload in Figures 20 and 21 (labels corresponding to a lower-than-threshold comparison are inside brackets: "<>").

Our workload characterization shows that a large number of workloads exhibit learning speed limitations. In Figure 20, we see 41 out of 63 workloads are labeled MP and thus exhibit local maxima; in Figure 21, we see 44 out of 63 workloads are labeled HJ, and thus exhibit interepoch jitter. Phased behavior is the only learning speed problem that does not occur in a majority of the workloads: Only 17 out of 63 workloads are labeled P in Figure 21. Worse yet, practically *all* workloads incur at least one learning speed limitation. Only 4

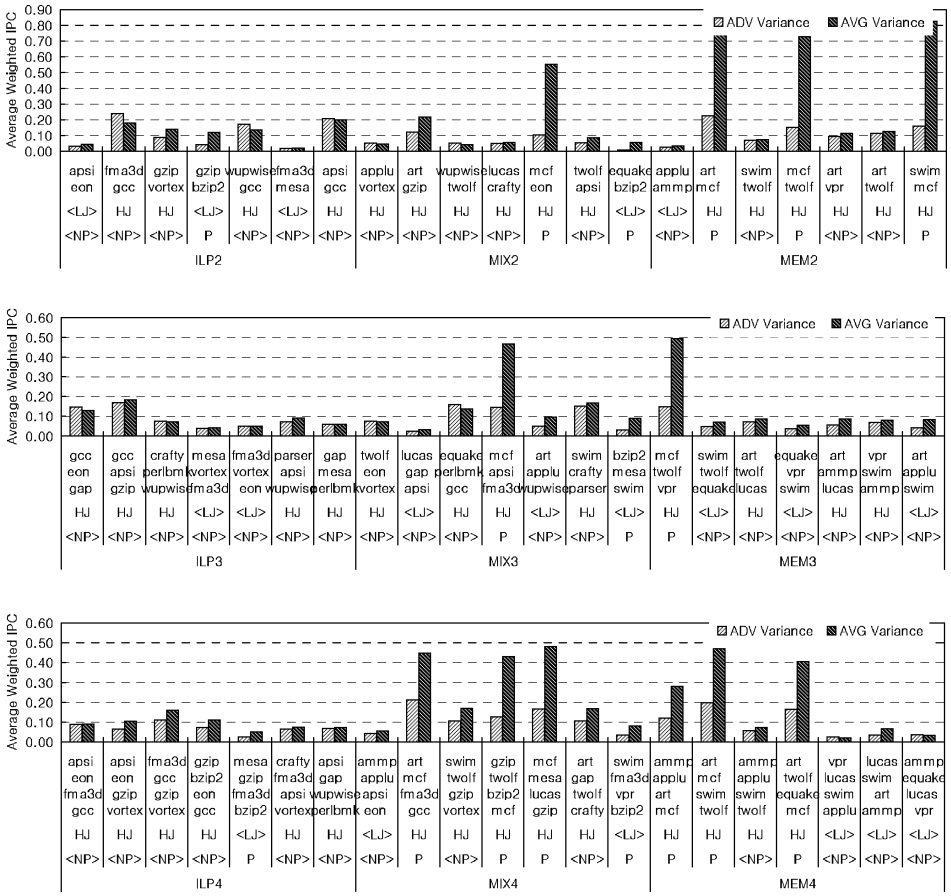

Fig. 21. Adjacent and average variance for our 63 workloads. Labels characterize each workload based on its ADJV value as being either low jitter (LJ) or high jitter (HJ) and based on its AVGV value as being either not phased (NP) or phased (P).

Table VI. Thresholds Used to Characterize Learning Speed Problems in Our Multithreaded Workloads

Condition	Label for True Condition	Label for False Condition
Reversed Gradient Height < 0.15	OP (One Peak)	MP (Multiple Peaks)
ADJ variance < 0.05	LJ (Low Jitter)	HJ (High Jitter)
AVG variance < 2× ADJ variance	NP (Not Phased)	P (Phased)

out of 63 workloads have no learning problems at all (i.e., labeled OP, LJ, and NP in Figures 20 and 21). Based on these results, we conclude learning speed limitations are ubiquitous in our workloads.

Not only do a large number of workloads exhibit learning speed limitations, but, perhaps more importantly, the limitations are prevalent in the sharp peak workloads. To illustrate this point, Table VII lists for each workload its OP/MP

label from Figure 20 and its LJ/HJ and NP/P labels from Figure 21 underneath its SP/DP label from Figure 16. This permits each workload's learning overheads analysis to be viewed simultaneously with its peak sharpness analysis. As Table VII shows, every one of the 39 sharp peak workloads (labeled SP) exhibits at least one learning speed limitation, and 24 of them exhibit two or three limitations. Recall from Section 5.4 that effective learning has the greatest potential to improve performance in the sharp peak workloads. The fact that learning speed limitations are prevalent in these critical workloads is the key reason our hill-climbing technique cannot match the performance of offline learning.

Finally, to illustrate the fact that learning speed limitations have an impact on performance, Figure 22 presents the limit results from Figure 12 based on our workload characterization. In Figure 22, the different bars report the performance of ICOUNT, FLUSH, DCRA, and HILL-WIPC from Figure 12 normalized against offline learning. The bars labeled SP and DP report results for the sharp and dullpeak workloads, respectively, and are identical to the corresponding bars in Figure 17. The remaining bars in Figure 22 show the performance of the 39 sharp peak workloads based on the workload characterization labels in Table VII. Groups of bars labeled SP-OP and SP-MP report the performance of the SP workloads that exhibit few and many local maxima, respectively; groups of bars labeled SP-LJ and SP-HJ report the performance of the SP workloads that exhibit low and high jitter, respectively; and groups of bars labeled SP-NP and SP-P report the performance of the SP workloads that exhibit no phased and phased behavior, respectively.

Figure 22 shows HILL-WIPC always performs better in workloads without a particular learning speed limitation. Specifically, HILL-WIPC performs 2.0% higher in workloads with few local maxima compared to workloads with many local maxima, 1.1% higher in workloads with low jitter compared to workloads with high jitter, and 2.0% higher in workloads with no phased behavior compared to workloads with phased behavior. In addition, HILL-WIPC's performance gain over ICOUNT, FLUSH, and DCRA is also generally larger when a particular learning speed limitation is absent. Note, however, that the change in performance across different workload categories is small, and HILL-WIPC never matches offline learning performance, even in the good cases (SP-OP, SP-LJ, or SP-NP). As mentioned earlier, no sharp peak workload is completely free of learning speed limitations. Hence, while the SP-OP, SP-LJ, and SP-NP workloads do not exhibit local maxima, interepoch jitter, and phased behavior, respectively, they still incur at least one other learning speed limitation that prevents them from matching offline learning. Nevertheless, Figure 22 demonstrates performance is generally higher when a particular learning speed limitation is not present.

7. SENSITIVITY STUDY AND EXTENSIONS

The performance evaluation and in-depth analysis of hill-climbing SMT resource distribution is now complete. In this section, we address two final issues. First, we examine the sensitivity of our hill-climbing technique to different

Table VII. Combining Peak Sharpness and Learning Overheads Analyses

ILP2	apsi	fma3d	gzip	gzip	wupwise	fma3d	apsi
	eon	gcc	vortex	bzip2	gcc	mesa	gcc
	DP	SP	DP	DP	SP	DP	SP
	MP	MP	OP	OP	OP	OP	MP
	NP	NP	NP	P	NP	NP	NP
	LJ	HJ	HJ	LJ	HJ	LJ	HJ
MIX2	applu	art	wupwise	lucas	mcf	twolf	equake
	vortex	gzip	twolf	crafty	eon	apsi	bzip2
	SP	SP	DP	DP	SP	DP	DP
	OP	OP	MP	OP	MP	MP	OP
	NP	NP	NP	NP	P	NP	P
	HJ	HJ	HJ	HJ	HJ	HJ	LJ
MEM2	applu	art	swim	mcf	art	art	swim
	ammp	mcf	twolf	twolf	vpr	twolf	mcf
	DP	SP	SP	SP	SP	SP	SP
	MP	MP	OP	MP	MP	OP	MP
	NP	P	NP	P	NP	NP	P
	LJ	HJ	HJ	HJ	HJ	HJ	HJ
ILP3	gcc	gcc	crafty	mesa	fma3d	parser	gap
	eon	apsi	perlbmk	vortex	vortex	apsi	mesa
	gap	gzip	wupwise	fma3d	eon	wupwise	perlbmk
	SP	SP	DP	DP	DP	DP	DP
	MP	MP	MP	OP	OP	MP	MP
	NP	NP	NP	NP	NP	NP	NP
	HJ	HJ	HJ	LJ	LJ	HJ	HJ
MIX3	twolf	lucas	equake	mcf	art	swim	bzip2
	eon	gap	perlbmk	apsi	applu	crafty	mesa
	vortex	apsi	gcc	fma3d	wupwise	parser	swim
	DP	DP	SP	DP	SP	SP	SP
	MP	MP	MP	MP	MP	MP	OP
	NP	NP	NP	P	NP	NP	P
	HJ	LJ	HJ	HJ	LJ	HJ	LJ
MEM3	mcf	swim	art	equake	art	vpr	art
	twolf	twolf	twolf	vpr	ammp	swim	applu
	vpr	equake	lucas	swim	lucas	ammp	swim
	SP	SP	SP	SP	SP	SP	SP
	MP	MP	MP	MP	MP	MP	MP
	P	NP	NP	NP	NP	NP	NP
	HJ	LJ	HJ	LJ	HJ	HJ	LJ
ILP4	apsi	apsi	fma3d	gzip	mesa	crafty	apsi
	eon	eon	gcc	bzip2	gzip	fma3d	gap
	fma3d	gzip	gzip	eon	fma3d	apsi	wupwise
	gcc	vortex	vortex	gcc	bzip2	vortex	perlbmk
	DP	DP	SP	DP	DP	DP	SP
	OP	OP	MP	OP	OP	MP	OP
	NP	NP	NP	NP	P	NP	NP
	HJ	HJ	HJ	HJ	LJ	HJ	HJ
MIX4	ammp	art	swim	gzip	mcf	art	swim
	applu	mcf	twolf	twolf	mesa	gap	fma3d
	apsi	fma3d	gzip	bzip2	lucas	twolf	vpr
	eon	gcc	vortex	mcf	gzip	crafty	bzip2
	SP	SP	SP	SP	DP	SP	SP
	MP	MP	MP	OP	OP	MP	MP
	NP	P	NP	P	P	NP	P
	LJ	HJ	HJ	HJ	HJ	HJ	LJ

Continued

Table VII. Continued

MEM4	ammp applu art mcf SP MP P HJ	art mcf swim twolf SP MP P HJ	ammp applu swim twolf SP OP NP HJ	art twolf equake mcf SP MP P HJ	vpr lucas swim applu SP MP NP LJ	lucas swim art ammp SP MP NP LJ	ammp equake lucas vpr DP OP NP LJ

"SP/DP" indicates sharp peak vs. dull peak workloads. "OP/MP" indicates one peak vs. multiple peak-workloads. "LJ/HJ" indicates low-jitter vs. high-jitter workloads. "NP/P" indicates unphased vs. phased workloads.

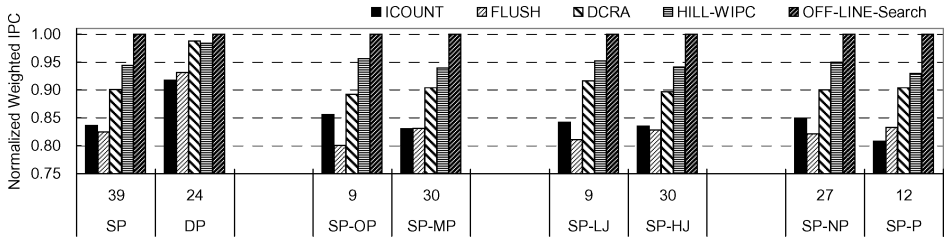

Fig. 22. The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC normalized against offline learning. Results are reported for sharp and dull peak workloads. Among sharp peak workloads, results are reported for single- and multipeak workloads, low- and high-jitter workloads, and workloads with unphased and phased behavior.

system assumptions (Section 7.1). Then, we propose several extensions to our basic hill-climbing technique, and conduct a preliminary evaluation of their effectiveness (Section 7.2).

7.1 Sensitivity Study

Throughout this article, we have assumed the SMT processor parameters listed in Table I. An important question is: How sensitive are our results to these specific parameters? This section investigates sensitivity with respect to the processor parameters.

To study processor resource sensitivity, we ran experiments with different SMT processor configurations. In particular, we reduced by half and doubled the number of entries in the processor queues (e.g., the IFQ, IQ, LSQ, and ROB) as well as the number of rename registers compared to the default configuration used throughout this article. In Table VIII, the columns labeled "Half," "Default," and "Double" report the parameters used by these different configurations. Figure 23 shows the performance observed for each configuration. In Figure 23, we report the average weighted IPC for ICOUNT, FLUSh, DCRA, and HILL-WIPC normalized against HILL-WIPC for the half, default, and double configurations. The first group of bars, labeled "ALL," reports performance averaged across all 63 workloads, while the remaining 6 groups report performance averaged across the 2-, 3-, and 4-thread workloads and ILP, MIX, and MEM workloads, respectively. These experiments (along with all other experiments

Table VIII. Processor Queue Sizes and Number of Rename Registers Used for the Processor Resource Sensitivity Study

Configuration	Half	Default	Double
IFQ	16	32	64
IQ	40-Int/40-FP	80-Int/80-FP	160-Int/160-FP
LSQ	128	256	512
ROB	256	512	1024
Rename register	128-Int/128-FP	256-Int/256-FP	512-Int/512-FP

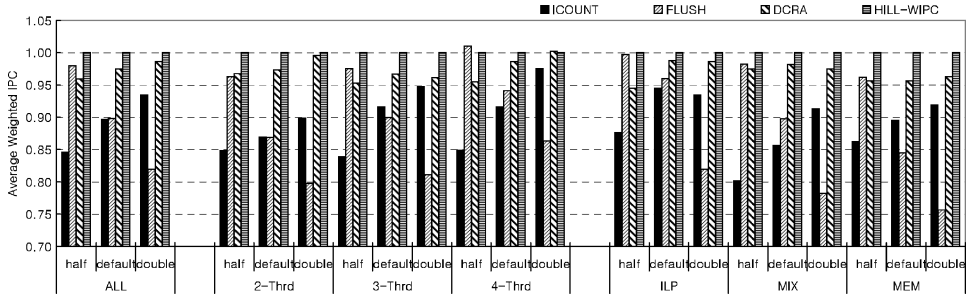

Fig. 23. The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC as we vary processor resources between the half and double configurations. All bars are normalized against HILL-WIPC.

in the remainder of this article) use our larger one-billion instruction simulation windows.

Across all configurations, HILL-WIPC achieves the best performance compared to ICOUNT, FLUSH, and DCRA, as illustrated by the ALL bars in Figure 23. With the exception of the half configuration running under FLUSH and the double configuration running under DCRA for the 4-thread workloads, HILL-WIPC always achieves the best performance. This shows the effectiveness of our hill-climbing technique is robust with respect to different amounts of processor resources. Figure 23 also shows FLUSH improves significantly relative to HILL-WIPC as processor resources are reduced. With fewer resources, the benefits provided by hill-climbing are less important. In particular, there is almost no opportunity for hill-climbing to exploit memory-level parallelism within a single thread for the half configuration, since back-to-back L2 cache misses are unlikely to appear simultaneously in a small instruction window. At the same time, FLUSH effectively frees resources whenever a thread suffers a cache miss, which is increasingly important as resources become scarce. Nevertheless, HILL-WIPC still maintains a performance advantage over FLUSH at all SMT processor configurations.

7.2 Hill-Climbing Extensions

7.2.1 *Thread Priority.* We now examine several mechanisms that extend the functionality and/or try to improve the performance of our basic hill-climbing SMT resource distribution technique. Our first extension is support for thread priorities. Thus far, we have treated all threads in our multithreaded workloads equally, which is appropriate for throughput-oriented applications.

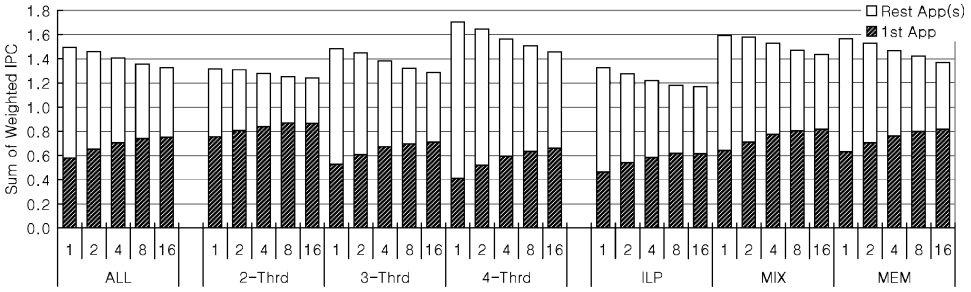

Fig. 24. Performance of hill-climbing with thread priority control. The shaded component of each bar shows the IPC of thread 0 as P_0 is varied from 1–16. The white components show the IPC of all other threads in the workload.

However, for latency-sensitive applications, it is desirable to prioritize critical threads, giving them more resources than the noncritical threads. Researchers have studied mechanisms for controlling thread priority in SMT processors to support latency-sensitive applications [Raasch and Reinhardt 1999; Dorai et al. 2003]. We propose providing a similar capability by embedding a notion of priority in our hill-climber's performance feedback metric.

The performance feedback metrics we've studied thus far, namely Eqs. (1)–(3) from Section 3.2, enforce equal priority across all threads. Eq. (5), given next, shows an alternate metric that incorporates thread priority control. Like Eq. (1), Eq. (5) sums the IPC from every thread, but in addition, it weights each thread's IPC with a priority value P_i. By giving a larger weight to a particular thread, we can increase its relative importance to the sum, thus increasing its priority. Since the hill-climbing algorithm tries to maximize the performance metric, it will naturally steer more resources to the threads with highest priority.

$$\text{Sum\_of\_Prioritized\_IPC} = \sum IPC_i \times P_i \tag{5}$$

Figure 24 shows our thread priority results. In Figure 24, we report the average weighted IPC for our hill-climbing technique as the priority value for thread 0, P_0, is varied from 1–16 in powers of 2. Priority values for all other threads, P_i where $i \neq 0$, are set to 1. To show the impact of the priority value on thread 0's performance, we break down each bar into two components: thread 0's contribution (shaded components) and all other threads' contributions (white components) to the total weighted IPC. The first group of bars, labeled ALL, reports performance averaged across all 63 workloads, while the remaining 6 groups report performance averaged across the 2-, 3-, and 4-thread workloads and ILP, MIX, and MEM workloads, respectively.

As the shaded components in Figure 24 show, Eq. (5) is effective in controlling thread priority. When P_0 is increased to 16, thread 0's performance increases by 14.8%, 35.2%, and 61.0% relative to the equal priority case ($P_0 = 1$) for the 2-, 3-, and 4-thread workloads, respectively. Averaged across all the workloads, thread 0's performance goes up by 29.8% compared to having no priority control. In addition to these results, we also compared thread 0's performance in each

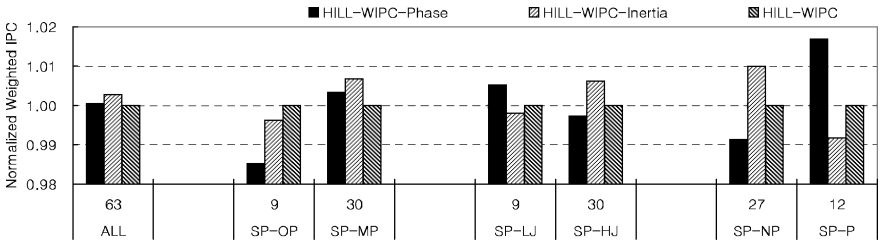

Fig. 25. Performance of hill-climbing extensions. The bars labeled HILL-WIPC-Phase and HILL-WIPC-Inertia report the performance of hill-climbing with phase-based learning and inertia, respectively. All bars are normalized against HILL-WIPC.

multithreaded workload to its single-threaded performance on a uniprocessor with the same hardware resources as our SMT processor. We find the maximum performance achieved by thread 0 for the 2-, 3-, and 4-thread workloads comes within 86.4%, 71.1%, and 65.7%, respectively, of its single-threaded performance. Figure 24 also shows that optimizing thread priority comes at the expense of overall throughput. Relative to equal priority, total weighted IPC degrades by 5.6%, 13.4%, and 14.6% for the 2-, 3-, and 4-thread workloads, respectively, when $P_0 = 16$. Not surprisingly, some throughput must be sacrificed in order to improve latency.

7.2.2 *Improving Learning Speed.* In addition to supporting thread priorities, we also try to improve the performance of hill-climbing SMT resource distribution. As discussed in Section 6.1, our hill-climbing technique suffers from three learning speed limitations: local maxima, phased behavior, and interepoch jitter. We propose two extensions to address these limitations. The first of these extensions is *phase-based learning* for addressing phased behavior. Phase-based learning employs existing phase detection techniques [Sherwood et al. 2001] to detect when two or more epochs exhibit similar behavior. To reduce learning overhead, we apply the best partitioning learned for an earlier epoch to later epochs that are similar, without incurring additional learning time. In addition to phase detection, phase-based learning also employs phase prediction techniques [Sherwood et al. 2003] to predict each epoch's phase type. This allows a previously learned partitioning to be applied at the *beginning* of each epoch, prior to its execution.

We implemented Sherwood's Basic Block Vector (BBV) signature analysis technique [Sherwood et al. 2001] for performing phase detection in single-threaded programs. We extended Sherwood's basic technique to handle multithreaded workloads by simply concatenating the BBVs from simultaneously executing threads, thus forming a multithreaded BBV. In all our experiments, we use per-thread BBVs consisting of 64 entries. We also implemented Sherwood's phase prediction technique [Sherwood et al. 2003] to predict the phase type at the beginning of each epoch. Our phase predictor can track 128 unique phases, and uses a 2048-entry run-length encoded Markov predictor. Figure 25 compares the performance of phase-based learning in the bars labeled HILL-WIPC-Phase against HILL-WIPC. The groups of bars labeled ALL report

the average performance achieved across all 63 multiprogrammed workloads, while the remaining groups of bars labeled SP-OP, SP-MP, SP-LJ, SP-HJ, SP-NP, and SP-P report the average performance achieved across the different workload categories defined in Section 6.3. All bars are normalized against the HILL-WIPC bars.

Unfortunately, phase-based learning does not provide a noticeable performance gain across our workloads. As the ALL bars in Figure 25 show, phase-based learning boosts hill-climbing performance by only 0.05% averaged across all the workloads. However, phase-based learning does address phased behavior. As the SP-P bars in Figure 25 show, the HILL-WIPC-Phase bars provide a 1.7% performance improvement over the HILL-WIPC bars. The reason this does not translate into an overall performance gain is because phase-based learning degrades the performance of workloads without phased behavior, as shown by the SP-NP bars in Figure 25.

Our second extension to improve learning speed is *learning with inertia* to address local maxima and interepoch jitter. Inertia causes the hill-climber to continue its movement over the top of a hill when it reaches the peak, thus permitting the algorithm to explore the downward slope on the other side of the hill. While this leads to overshoot for maximal peaks, it enables the hill-climbing algorithm to potentially escape from nonmaximal peaks. Moreover, inertia also prevents the hill-climber from following every single change in the performance gradient. This allows the hill-climbing algorithm to filter out interepoch jitter, and move more steadily towards performance peaks. To implement inertia, we replaced line 7 of the hill-climbing algorithm in Figure 5 with the following statement.

$$\text{perf}[\text{epoch\_id} \,\%\, \text{T}] = (\text{perf}[\text{epoch\_id} \,\%\, \text{T}] + \text{eval\_perf}(\text{epoch\_id}))/2;$$

This expression computes the current epoch's performance as a function of both the new performance sample and previous samples accumulated into the `perf` array variable. Hence, the choice of the `gradient_thread`, computed by line 9 in Figure 5, is determined not only by the current performance gradient, but also by the performance gradients from previous epochs. The incorporation of such history information adds inertia to the hill-climber's movement.

The bars labeled HILL-WIPC-Inertia in Figure 25 report the performance of our hill-climbing technique with inertia. As the ALL bars in Figure 25 show, inertia boosts hill-climbing performance by a modest 0.28% averaged across all the workloads. This performance advantage comes from addressing local maxima and interepoch jitter, as shown by the SP-MP and SP-HJ bars in Figure 25 where inertia provides a 0.62% and 0.67% performance gain, respectively. Unfortunately, inertia also decreases the performance of workloads with phased behavior by 0.83%, as shown by the SP-P bars, thus reducing its overall benefit. Because inertia delays the impact of performance gradient changes on the hill-climber's movement, it diminishes the quickness with which the hill-climber can respond to phase changes.

Our preliminary evaluation of phase-based learning and inertia show both techniques have the potential to address the learning limitations in our hill-climbing technique. Unfortunately, the overall performance benefit is

currently very small, in part due to performance degradation to those workloads that are not directly targeted by the techniques. Further research is needed to address this shortcoming.

8. CONCLUSION

This article proposes hill-climbing SMT resource distribution. Our technique observes the actual impact that partitioning key SMT hardware resources has on performance at runtime, and makes future partitioning decisions based on this feedback. Over time, many partitionings can be visited, allowing us to learn which ones are the best. Since we perform learning online, we develop a hill-climbing algorithm that exploits the hill-shaped nature of performance variation in the resource distribution space to quickly find the best partitionings. Compared to existing techniques our approach has several advantages. First, our approach customizes resource distribution decisions to the actual performance bottlenecks in a workload, thus reducing missed performance opportunities. Second, whenever learning succeeds, our approach finds the best resource distribution for a particular workload behavior. Lastly, our approach can optimize for a specific performance goal by simply using the appropriate performance feedback metric.

On a comprehensive suite of 63 multiprogrammed workloads, our hill-climbing technique outperforms ICOUNT, FLUSH, and DCRA by 11.4%, 11.5%, and 2.8%, respectively, when comparing performance under the weighted IPC metric. Hill-climbing's advantage over ICOUNT, FLUSH, and DCRA is as high as 23.7%, 13.6%, and 5.9%, respectively, when comparing performance under either the average IPC or harmonic mean of weighted IPC metrics. Using two ideal offline learning algorithms, OFF-LINE-Exhaust and OFF-LINE-Hill, we conduct a limit study. Our limit study shows offline learning outperforms ICOUNT, FLUSH, and DCRA by 19.2%, 18.0%, and 7.6%, respectively, under the weighted IPC metric, demonstrating there exists additional room to improve our hill-climbing technique. In addition to performance limits, our offline algorithms also provide insightful performance variation information. Analysis of this information shows the gains achieved by offline learning (and hill-climbing) come from exploiting the performance opportunities in sharp peak workloads. In addition, the performance variation information also reveals three bottlenecks that prevent high learning speed in our hill-climber: local maxima, phased behavior, and interepoch jitter. We propose the reversed gradient, adjacent variance, and average variance metrics to quantify these learning limitations, and find they are ubiquitous in our workloads. Finally, we conduct a sensitivity study that shows our hill-climbing technique is robust to changes in processor resources. We also study extensions to our basic hill-climbing technique, including thread priority control, phased-based learning, and learning with inertia.

Based on our experience, we believe hill-climbing SMT resource distribution is a promising technique for managing SMT hardware resources. Furthermore, we also find learning-based resource management is quite general. Hence, we believe the basic techniques and analyses presented in this article can

be applied to manage other types of resources in multithreaded and multicore architectures.

REFERENCES

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. CS TR 1342, University of Wisconsin-Madison. June.

CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNANDEZ, E. 2004. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, 171–182.

CHOI, S. AND YEUNG, D. 2006. Learning-Based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. IEEE Computer Society, 239–250.

DORAI, G. K., YEUNG, D., AND CHOI, S. 2003. Optimizing SMT processors for high single-thread performance. *J. Instruction-Level Parallel. 5*, 1–35.

EL-MOURSY, A. AND ALBONESI, D. H. 2003. Front-End policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture*. IEEE Computer Society, 31–40.

GONCALVES, R., AYGUADE, E., VALERO, M., AND NAVAU, P. O. A. 2001. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. In *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*.

KALLA, R. N., SINHAROY, B., AND TENDLER, J. M. 2004. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro 24*, 2, 40–47.

LATORRE, F., GONZALEZ, J., AND GONZALEZ, A. 2004. Back-End assignment schemes for clustered multithreaded processors. In *Proceedings of the 18th Annual International Conference on Supercomputing*. ACM Press, 316–325.

LUO, K., FRANKLIN, M., MUKHERJEE, S. S., AND SEZNEC, A. 2001. Boosting SMT performance by speculation control. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society.

LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 164–171.

MADON, D., SANCHEZ, E., AND MONNIER, S. 1999. A study of a simultaneous multithreaded processor implementation. In *Proceedings of EuroPar'99*. Springer, 716–726.

MARR, D. T., BINNS, F., HILL, D., HINTON, G., KOUFATY, D., MILLER, J. A., AND UPTON, M. 2002. Hyper-Threading technology architecture and microarchitecture. *Intel Technol. J. 6*, 1, 4–15.

PENTIUM4. 2002. Intel Pentium 4 processor. http://www.intel.com/design/Pentium4/index.htm.

RAASCH, S. E. AND REINHARDT, S. K. 1999. Applications of thread prioritization in SMT processors. In *Proceedings of the Multithreaded Execution, Architecture, and Compilation Workshop*.

RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 15–25.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. 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*. IEEE Computer Society, 3–14.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 45–57.

SHERWOOD, T., SAIR, S., AND CALDER, B. 2003. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 336–347.

SNAVELY, A., TULLSEN, D. M., AND VOELKER, G. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. ACM, 66–76.

TULLSEN, D. M. AND BROWN, J. A. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 318–327.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, 191–202.

Received August 2007; accepted December 2008