ABSTRACT

Title of dissertation: HILL-CLIMBING SMT PROCESSOR RESOURCE DISTRIBUTION

Seungryul Choi, Doctor of Philosophy, 2006

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

The key to high performance in SMT processors lies in optimizing the shared resources distribution among simultaneously executing threads. Existing resource distribution techniques optimize performance only indirectly. They infer potential performance bottlenecks by observing indicators, like instruction occupancy or cache miss count, 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, opportunities for performance gains are lost whenever the corrective actions do not effectively address the actual performance bottlenecks occurring in the SMT processor pipeline.

In this dissertation, we propose a different approach to SMT processor 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 successively applying and evaluating different resource distributions, our approach tries to *learn* the best distribution over time. Because we perform learning on-line, learning time is crucial. We develop a *hill-climbing SMT processor resource distribution technique* that efficiently learns the best resource distribution by following the performance gradient within the resource distribution space.

This dissertation makes three contributions within the context of learning-based SMT processor resource distribution. First, we characterize and quantify the time-varying performance behavior of SMT processors. This analysis provides understanding of the behavior and guides the design of our hill-climbing algorithm. Second, we present a hill-climbing SMT processor resource distribution technique that performs learning on-line. The performance evaluation of our approach shows a 11.4% gain over ICOUNT, 11.5% gain over FLUSH, and 2.8% gain over DCRA across a large set of 63 multiprogrammed workloads. Third, we compare existing resource distribution techniques to an ideal learning-based technique that performs learning off-line to show the potential performance of the existing techniques. This limit study identifies the performance bottleneck of the existing techniques, showing that the performance of ICOUNT, FLUSH, and DCRA is 13.2%, 13.5%, and 6.6%, respectively, lower than the ideal performance. Our hill-climbing based resource distribution, however, handles most of the bottlenecks of the existing techniques properly, achieving 4.1% lower performance than the ideal case.

HILL-CLIMBING SMT PROCESSOR RESOURCE DISTRIBUTION

by

Seungryul Choi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

ACKNOWLEDGMENTS

It is my honor to know so many good people who made my Ph.D. study at the University of Maryland enjoyable. First of all, I really thank my advisor, Dr. Donald Yeung. He not only advised my research but also was a role model in my academic life. In addition, the discussion with him inspired me and made me to aim at higher quality intellectual level. As a result, my graduate study was fun and a meaningful part of my life.

I specially thank my grandmother, father, mother, and brother for their endless love. I was able to finish my long graduate study in Maryland because of their enormous support and endless prayers to God for me. My cousins and cousin-in-laws, Wonsun, Sam, Moonsun, Andrew, and Taewook, always remind me of the meaning of the family. I really appreciate their help since I first came to the USA.

During my stay in Maryland, I was so lucky to have many good people around me. I thank my office mates in the University of Maryland for their help and discussion, and for sharing ideas: Dongkeun, Gautham, Deepak, Hameed, Sumit, Wanli, Xuanhua, Priyanka, Aamer, Sada, Zahran, Anashua, Vinod, Aneesh, Jayanth, Brinda, Renju, Barath, Sumesh, David, Kursad, Mike, Ohm, Sean, and all the other friends in SCA-Lab. I would like to acknowledge the Korean friends and their families in Maryland: Junesong, Jungmin, Kwangoon, Dongin, Eenjun, Hyunsang, Hwansoo, Kyungdong, Sungjoon, Kyongil, Jae-jung, Sangwook, Hyunmo, Pyonguk, Bongwon, Changha, Jihwang, Yooah, Seungjoon, Kyungnam, Beomseok, Jinwook, Minho, Bongshin, Jaeyong, Jaeyoon, Sungwook, Bohyung, Hyunook, Woochul, Joonhyuk, Seungjong, Jusub, Dongik, Soobum, and many others. I also appreciate the members of my Ph.D. defense committee: Dr. Manoj Franklin, Dr. Chau-Wen Tseng, Dr. Alan Sussman, and Dr. Sung Lee. In addition, I want to thank Daniel, Timoteo, Joseph, Jennifer, Kwanghoon, Wooyoung, Jaehwa, Jin-hyuk, Ilchul, Woochang, and TJ for their religious support and prayers for me. I apologize to those I have inadvertently left out.

Most importantly, I would like to thank my lovely Jiyoun, Andy, and Claire. Andy and Claire have been the source of happiness in my family. Jiyoun has always been the best friend, a great supporter, a trustful advisor as well as a sweet wife.

Lastly, I sincerely thank God for his endless love and care.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

1.1.1 Simultaneous Multithreading Processors

Shared memory multiprocessor design is a natural extension of the mass produced microprocessors and yields linear or sub-linear scale performance enhancement for well-designed parallel applications. In the past, small-scale to massively parallel multiprocessors were built to exploit thread level parallelism. Recently, single chip multi-threading processors have been actively studied. Single chip multi-threading processors allow concurrent execution of multiple threads with a single chip by maintaining multiple hardware contexts on a chip. This architecture effectively utilizes ever increasing available hardware budget in a single chip module, which is predicted by Moore's law. In addition, this architecture is a cost-effective way of exploiting thread-level parallelism because instead of having dedicated hardware resources per thread, it allows some of the on-chip hardware resources to be shared between concurrently running multiple threads thus increasing the resource utilization.

One implementation of a single chip multi-threading processor is the simultaneous multi-threading (SMT) processor [1, 2, 3], which executes multiple threads in a single chip by allowing fine grained sharing of most of the processor resources. Because a single thread typically cannot fully utilize all of a processor's available resources, fine-grained processor resource sharing improves overall resource utilization. The increased utilization directly translates into higher processor throughput. SMT is an important architectural technique, as evidenced by the widespread attention it has received from academia [4, 5, 1, 6, 7], and by industry's willingness to incorporate it into commercial processors [8, 9]. Given single chip multi-threading will continue to be an important architectural direction, research that improves SMT performance without increasing its cost, like power consumption or fabrication cost, will remain highly relevant in future systems.

1.1.2 Feedback-based SMT processor resource distribution

SMT processors improve performance by allowing multiple threads to share most of the hardware resources. However, the actual performance gain of SMT processors depends greatly on how resources are distributed to individual threads. High performance occurs only when resources are distributed to those threads that will use them efficiently. Otherwise, the shared resource may be monopolized by a thread that just holds the resource for a long time during the long latency operation, making other threads to wait for the release of the resource. Hence, the mechanism for controlling resource distribution among the simultaneously executing threads play a critical role in achieving good SMT performance.

Several resource distribution techniques have been studied in the past [4, 5, 1, 6, 7]; all of them try to increase performance by reducing the amount of time instructions stall in shared processor resources. While existing resource distribution techniques have demonstrated good performance gains, one shortcoming is they optimize performance *indirectly*. As illustrated in Figure 1.1(a), resource distribution decisions are made based on hardware monitors that indicate per-thread resource usage (for example, instruction occupancy or cache miss counts); the hardware monitors may not be correlated with the actual perfor-

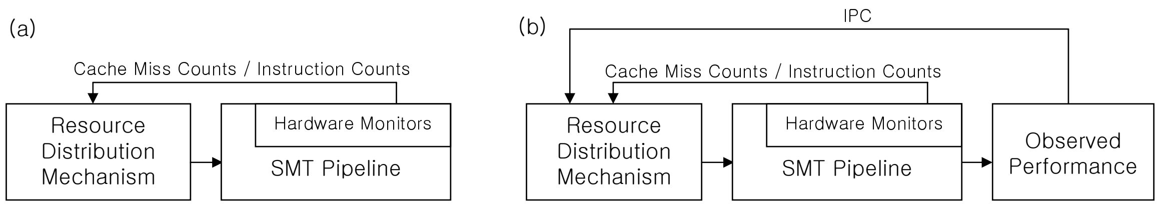

(a)
Cache Miss Counts / Instruction Counts
Resource Distribution Mechanism
Hardware Monitors
SMT Pipeline
(b)
IPC
Cache Miss Counts / Instruction Counts
Resource Distribution Mechanism
Hardware Monitors
SMT Pipeline
Observed Performance

Figure 1.1: (a) Existing resource distribution techniques optimize performance indirectly by making decisions based on hardware monitors only. (b) Learning-based resource distribution examines actual performance (*e.g.*, IPC) to learn the optimal resource distribution.

mance. From resource usage information, the resource distribution mechanisms can infer potential performance bottlenecks and take actions to try to alleviate them. For example, some of the techniques, such as ICOUNT [1], reduce fetch priority of a thread that consumes too many entries in the instruction fetch queue and issue queue. FLUSH [6] flushes instructions from the pipeline that belong to a thread with outstanding L2 cache miss. FPG [10] reduces fetch priority of a thread if the thread's branch prediction confidence is low. DCRA [4] reduces the resource partition of a thread if the thread has no L1 cache misses.

These actions are designed to improve performance; however, their actual performance impact on the application workload is unknown since the resource distribution mechanisms never re-evaluate their decisions at run-time by monitoring the end performance resulting from their 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 the application workload mix. Existing resource distribution mechanisms cannot possibly anticipate all bottlenecks for all workloads. Second, even for the anticipated performance bottlenecks, further performance gains might still be possible because they are designed to achieve the better–not the optimal–performance.

We propose a different approach to SMT processor resource distribution that optimizes end performance *directly*. Our approach observes the impact that resource distribution decisions have on performance at run-time and feeds this information back to the resource distribution mechanisms to improve future decisions, as illustrated in Figure 1.1(b). By successively applying and evaluating different resource distributions, our approach tries to search for the best distribution over time. Because our approach searches for the optimal resource distribution based on actual performance feedback, the resource distribution decisions it makes are customized to the specific performance bottlenecks of each workload, reducing missed performance opportunities.

While similar feedback-based optimizations have been applied to run-time hardware optimization studies [11, 12, 13, 14], we are the first to apply performance feedback to SMT processor resource distribution.

1.1.3 Hill-Shaped SMT Processor Performance Curve

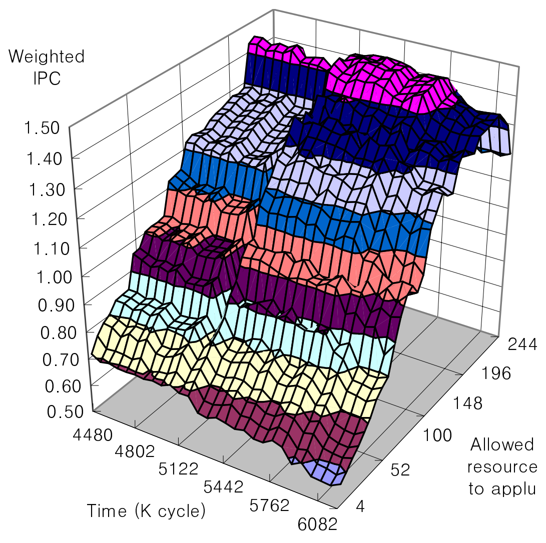
There are two ways of balancing SMT processor resource: *fetch policy* and *resource partitioning*. Fetch policy selects threads to fetch from every cycle. ICOUNT [1], FPG [10] STALL [6], and FLUSH [6] are examples of fetch policy. Resource partitioning balances the resource distribution across concurrently running threads by explicitly maintaining allowed resource share per thread. DCRA [4] and static partitioning [15, 16, 7] are examples of resource partitioning. Resource partitioning can rely on fetch policy to enforce its resource partition. For example, if a thread consumes beyond the given partition, fetch of the thread is stalled (STALL) or excessively used resources are preempted (FLUSH). We will detail fetch policy and resource partition in Section 3.2.

Our technique maintains resource partition to explicitly distribute the shared resources among the concurrent threads, and stalls fetching of a thread if the thread consumes resources up to its partition.

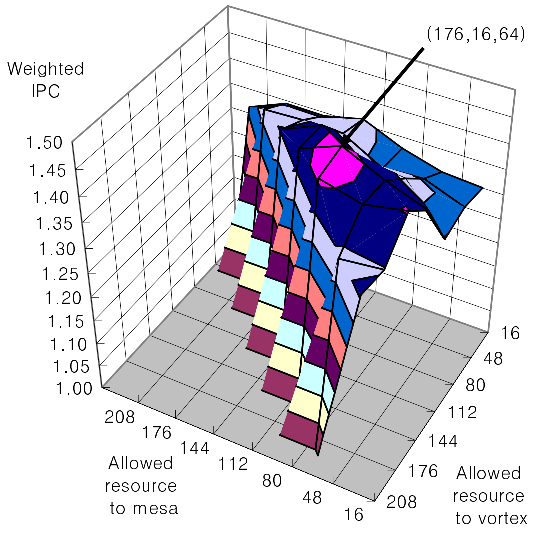
To collect the feedback information, we divide SMT execution into fixed intervals in time, which we call *epochs*. At the beginning of each epoch, we set a resource partition. Then, at the end of each epoch, we measure the performance of the SMT processor during the epoch. Based on the collected history information on different resource partitions and their corresponding performance feedback, our mechanism searches for the best resource partition for the current workload. The success of our approach depends on the searching speed because we can enjoy maximum performance benefit only after our mechanism reaches (or approaches to) the optimal resource distribution.

A key observation that enables fast searching is that performance, as a function of resource distribution, does not change randomly; instead, the performance sensitivity is often “hill-shaped.” In addition, the shape of the hill does not change randomly over time. Figure 1.2(a) illustrates this observation by showing the time-varying performance curve of two applications–applu and vortex–running simultaneously on an SMT processor for 30 epochs. Figure 1.2(b) shows the performance curve of three applications–mesa, vortex, and fma3d–running simultaneously on an SMT processor during an epoch. These graphs plot weighted IPC [17], one possible performance metric, as the resource partitions of individual threads are varied. In the figure, performance follows a well-defined hill shape, with a clear performance peak, and the hill shape is stable over time.

We exploit this behavior by using a *hill-climbing algorithm* [18] to search for the best resource distribution. Because searching is guided by the slope of the hill, our hill-



(a) applu–vortex performance curve



(b) mesa–vortex–fma3d performance curve

Figure 1.2: Performance, measured in weighted IPC metric, of (a) applu and vortex, and (b) mesa, vortex, and fma3d running simultaneously on an SMT processor, as the fraction of resources allowed to each application is varied. In (a), the Y-axis shows the amount of resources allowed to applu (vortex receives the remaining resources), and the X-axis shows the time. In (b), the X- and Y-axes show the amount of resources allowed to mesa and vortex (fma3d receives the remaining resources). The labeled arrow indicates the resource distribution that achieves peak performance.

climbing algorithm reaches the optimal resource distribution (*i.e.*, the peak of the hill) after sampling only a small portion of the resource distribution space, thus leading to low searching times.

One of the most important attributes that leads to the success of the hill-climbing algorithm is the *shape of the hill* (*i.e.*, the run-time behavior of the SMT processor performance curve). Several pitfalls related to the shape of the hill may defeat the hill-climbing algorithm. For example, if the hill has multiple humps, the hill-climbing algorithm may be trapped at one of local maxima. In addition, if the shape of the hill changes very frequently, the hill-climbing algorithm may not find an optimal resource distribution, or it may find a temporally optimized resource distribution that performs poorly in the near future. As the quality of resource distribution provided by the hill-climbing algorithm is highly affected by the shape of the hill, we conduct in-depth research on the time-varying shape of the hill in Chapter 4 before we design (in Chapter 5) and evaluate (in Chapter 6 and 7) our hill-climbing algorithm.

1.2 Contributions

This dissertation makes the following contributions.

Viewing SMT Processor Resource Distribution Problem as a Classical Optimization Problem

The performance of SMT processor is mainly determined by the resource distribution among the concurrently running threads. So, we view the SMT processor resource distribution problem as a search problem whose goal is finding a resource distribution that produces optimal performance. We believe this is a unique view in the SMT processor resource distribution study. This view makes us translate the resource distribution problem into a classical optimization problem, allowing us to apply general optimization problem solvers, hill-climbing algorithm, to SMT processor domain. In this dissertation, we first define the performance curve as a function of SMT processor resource distribution. Then, we design the hill-climbing algorithm that climbs up to the peak of the curve to search for the optimal resource distribution.

Development of SMT Processor Run-Time Performance Behavior Visualization Tool

The nature of SMT processor performance as a function of the resource distribution space is unknown prior to our research. In order to understand the time-varying behavior of this SMT processor performance curve, we built a visualization tool. Using this tool, we identified several workload characteristics. Some characteristics are problematic for hill-climbing algorithm like multiple humps or extremely frequent time-varying behavior. On the other hand, many workloads have favorable characteristics to the hill-climbing, like single hump and stable temporal behavior.

Quantitative Analysis of SMT Processor Run-Time Performance Behavior

Based on the knowledge acquired through the visualization tool, we developed four new metrics that quantitatively measure the shape of the performance curve. Two metrics quantify the static shape of the performance curve and two metrics measure the temporal variation of the performance curve. Using these metrics, we classify workloads. This classification helps understanding and analyzing the performance of prior SMT processor resource distribution techniques as well as our hill-climbing technique because of the strong correlation between run-time workload characteristics and its performance.

Hill-Climbing Resource Distribution Algorithm

We are the first to apply the hill-climbing algorithm to SMT processor resource distribution problem. The understanding of the time-varying performance curve from both the visualization tool and quantitative measurement enable us to customize the hill-climbing algorithm for the SMT processor resource distribution. We design our hill-climbing algorithm so that it can handle both problematic workload as well as the favorable workload characteristics.

Evaluation of the Hill-Climbing Resource Distribution

We faithfully evaluate the performance of the hill-climbing resource distribution technique across 63 workloads. Then, we compare the performance of hill-climbing algorithm against three prior SMT processor resource distribution techniques: ICOUNT, FLUSH, and DCRA. We suggest two improvements over the baseline hill-climbing resource distribution: phase based learning and hill-climbing with momentum term. In addition, we study hill-climbing resource distribution's sensitivity to three design parameters; memory latency, amount of processor resource, and thread priority.

SMT Processor Performance Limit Study

A performance comparison of existing resource distribution techniques against an ideal SMT processor can uncover performance bottlenecks and suggest ways to improve performance. However, figuring out the ideal performance limit of SMT processor is computationally infeasible because it is an NP-hard problem. For the first time in SMT study, we developed a heuristic that approximates the ideal performance limit of SMT processor. To make our heuristic computationally feasible, we assumed three simplifying constraints;

first, per-thread resource partition is maintained to distribute resources, second, updating resource partition is allowed only at every epoch boundary, and third oracle provides information only on the next epoch. Using the performance limit suggested by our heuristic, we re-evaluate four SMT processor resource distribution techniques (including ours), detail their performance potentials, and show our mechanism is the closest to the performance limit.

Extending Hill-Climbing technique

We show that hill-climbing is an effective mechanism for SMT processor resource distribution. Since hill-climbing algorithm is a general optimization problem solver, our technique can also be applied to more general problems, like run-time hardware optimization, which changes hardware parameters at run-time to achieve optimal performance or power consumption.

1.3 Road Map

The rest of the dissertation is organized as follows. Chapter 2 explains the background of our study including single chip multi-threading processors, issues on SMT processor resource distribution, and general hill-climbing algorithm. Chapter 3 lists the prior researches related to our study covering run-time hardware optimization study and SMT processor resource distribution techniques. In Chapter 4, we analyze the run-time behavior of SMT processors to better understand the time-varying performance curve. Based on this analysis, Chapter 5 presents the customized hill-climbing algorithm for SMT processor resource distribution problem domain. In Chapter 6, we show the performance of our heuristic that approximates the ideal performance limit of SMT processor and suggests the performance potential of the existing techniques. Chapter 7 evaluates the performance of hill-climbing resource distribution techniques and compares its performance against three existing techniques. Chapter 8 suggests directions to improve hill-climbing resource distribution and shows the sensitivity study results. Chapter 9 discusses the preliminary study on applying our hill-climbing technique to multi-treaded run-time system. Finally, Chapter 10 concludes the dissertation and suggests the future research directions.

Chapter 2

Background

2.1 Single Chip Multi-Threading Processors

Single chip multi-threading processors allow concurrent execution of multiple threads in a single chip by maintaining multiple on-chip hardware contexts. This architecture effectively utilizes ever increasing available hardware budget in a single chip module, which is predicted by Moore's law. In addition, this architecture is a cost-effective way of exploiting thread-level parallelism because it allows some of the on-chip hardware resources to be shared between concurrently running multiple threads, rather than dedicating them to each individual thread.

Depending on the design of the single chip multi-threading processors, the choice of the dedicated and shared hardware resources varies. Two extremes of single chip multi-threading processor design are *chip multiprocessor* (CMP) and *simultaneous multithreading processor* (SMT). CMP [19, 20, 21, 22] has multiple processor cores in a single chip. Each core has its own dedicated processor resources (including branch predictor, fetch queue, issue queue, functional unit, memory port, register file, and reorder buffer) to execute a thread. However, multiple cores share the on-chip L1 and(or) L2 caches. SMT [1, 2, 3] allows execution of multiple threads in a single core by allowing fine grained sharing of most of the processor resources, as well as the L1 cache and L2 cache between concurrently running threads. The only dedicated resources to each thread are the program counter and additional storage to maintain context information (*e.g.*, architected register file).

Since the multiple cores in CMP are duplicates of a single core, CMP can use single core design to implement multi-core CMP, thus simplifying the chip design. In addition, since each core in CMP is independent of each other, increasing the number of cores in a chip does not severely increase the complexity of the interconnections within a chip, making it scalable [20]. CMP may have either heterogeneous cores [21], with both powerful out-of-order processor core(s) mixed with simple small in-order processor core(s) in a chip, or homogeneous cores [19, 20]. Considering that the concurrently running threads may have diverse processor resource requirements, heterogeneous core CMP is an attractive design choice.

On the other hand, SMT can utilize the processor resource more efficiently because SMT allows one thread to use almost all of the shared resources when the other thread(s) cannot fully utilize them. In addition, SMT achieves higher per-core throughput by exploiting ILP between independent threads as well as within a single thread. The increased processor throughput provided by SMT, however, comes at the expense of single-thread performance. Because multiple threads share hardware resources at the same time, individual threads get fewer resources than they would have otherwise received had they been running alone. For threads with diverse characteristics, compared to heterogeneous CMP, SMT can give proper amount of resources to each thread dynamically by simply shifting the resource share between threads at run-time.

Due to these advantages of single chip multi-threading processor design, many CMP and SMT processors are commercially available these days. Intel Pentium4 with Hyper-threading [9] is an SMT product. IBM Power4, AMD Athlon64 dual core, Intel Pentium dual core, Intel Pentium quad core (Clovertown) are CMP products. IBM Power5 architecture [8] has two SMT cores in a single chip, making it an SMT and CMP hybrid. IBM Cell processor [23] has an SMT core, named PPE, and multiple CMP cores, named SPE, in a single chip, making it an SMT and heterogeneous CMP hybrid. On these commercial products, the detailed resource sharing structures are not well documented.

2.2 SMT Processor Resource Distribution

In SMT processors, resource sharing between concurrently running threads allows better resource utilization because a single thread cannot fully utilize the available resources all the times. This increased resource utilization directly translates into the improved throughput. However, the resource distribution may be unbalanced losing the performance opportunities. For example, if the first thread holds large amount of shared resource and waits for the data from memory, the second thread cannot get any more resource until the first thread gets the data from the memory and releases the resource, thus reducing the resource utilization.

There are three types of shared processor resources. The first type is called a “slot”, which includes fetch unit, issue unit, and functional units. A thread holds slot type resources for only one cycle. The second type is called a “queue”, which includes fetch queue, issue queue, rename register, and reorder buffer. A thread can hold queue type resources for many cycles until the thread voluntarily releases the resource. The third type is called a “memory”, which includes branch predictor tables and L1 cache. A thread can hold memory type resources for many cycles until the other threads claim them.

Figure 2.1 shows our classification of the shared resources. Among the three types of shared resources, only the second one, queue type, causes resource under-utilization problem because it may potentially participate in *hold-and-wait* condition and resource monopolization. Unbalanced resource distribution of slot type resource can be promptly fixed, if it happens and is detected. Unbalanced resource distribution of memory type

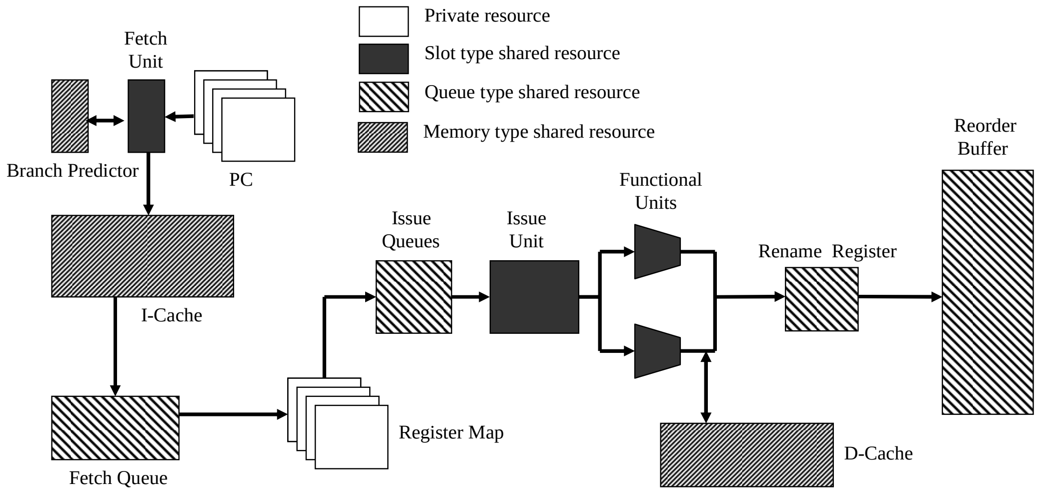


Figure 2.1: Block diagram of the SMT processor resources. The shared resources are classified into three types: slot type, queue type, and memory type. White boxes indicate private resources.

resource can be fixed after warming-up time.

Depending on the application characteristics and run-time phase, the amount of queue type resource requirement varies. Queue type resource is used to keep the in-flight instructions in the processor pipeline. In general, applications with high instruction level parallelism need many in-flight instructions to help find the parallelism, thus requiring many queue type resources. On the contrary, applications with long dependency chains can make progress with small amounts of queue type resources. Therefore, the balancing of the shared queue type resource distribution should take the application's current requests for the shared queue type resource into account. For example, L1 miss count (used by DCRA [4]), and the fetch / issue queue occupancy (used by ICOUNT [1]) can be used to *indirectly* figure out the application's requests for the shared queue type resource.

To correct unbalanced resource distribution, three fetch policies are used by the existing SMT resource distribution techniques. First, "fetch prioritizing" gives fetch priority to a thread, which deserves more resources. With this mechanism, after fetching from a high fetch priority thread, the low fetch priority thread can use the remaining fetch bandwidth. In addition, if the high fetch priority thread cannot use any of the fetch bandwidth due to branch misprediction or an instruction cache miss, the low fetch priority thread can fully utilize the fetch bandwidth. ICOUNT uses this mechanism. Second, “fetch stalling” stops fetching of a thread, which deserves fewer resources. Fetch-stalled threads cannot fetch any instructions, even if there are available fetch bandwidth. DCRA uses this mechanism. Third, “flushing” evicts instructions from the processor pipeline, whose thread deserves fewer resources. Flushing is the most timely way of balancing the resource distribution among the three existing mechanisms, but it is also the most costly as it needs to fetch the flushed instructions again. FLUSH [6] uses this mechanism. In Section 3.2, we will discuss the existing SMT processor resource distribution techniques, ICOUNT, FLUSH, and DCRA, in greater detail.

Our hill-climbing technique controls the allocation of the queue type resources as they affect the number of in-flight instructions in the pipeline and determines the achievable instruction level parallelism of each thread. In addition, our approach uses fetch stalling mechanism to maintain proper amount of queue type resource occupancy per thread. We will detail the ideal and implementation of our hill-climbing technique in Chapter 5.

2.3 Hill-Climbing Algorithm

Hill-climbing algorithm [18] is an optimization problem solver. Hill climbing attempts to maximize an evaluation function $f(x)$ by finding the optimal x. In discrete domains, the domain of f is typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing traverses the graph from

```
1.      #define   f            evaluation function

2.      current_vertex = initial_vertex;

3.      do until (there is no change in current_vertex) {
4.          for all (successor_vertex[i] = Get a successor of the current_vertex)
5.              successor_vertex[i].score = f(successor_vertex[i]);

6.          if (one of the successor_vertex[*].score is better than the current_state)
7.              current_vertex = the successor with the best score;
8.      }
```

(a) Hill-climbing algorithm on discrete space

```
9.      #define   ▽f           gradient of f
10.     #define   Delta        movement step

11.     current_state = initial_state;

12.     do until (there is no change in current_stae) {
13.         successor_state = current_state + ▽f(current_state) * Delta;
14.         successor_state.score = f(successor_state);

15.         if (successor_state.score is better than current_state's score)
16.             current_state = successor_state;
17.     }
```

(b) Hill-climbing algorithm on continuous space

Figure 2.2: Hill-climbing algorithm on discrete domain space (a) and continuous domain space (b). In discrete domain space, the next vertex is picked among the neighboring vertices whose f value is the best (line 4-7). In continuous domain space, the next state is directed by the gradient vector (line 13-16).

vertex to vertex, always heading towards the locally increasing value of f, until a local maximum is reached. Hill climbing can also operate on a continuous domain space: in that case, the algorithm is called *gradient ascent*. Hill climbing is guided by the gradient vector in choosing the next state. Hill climbing terminates when there are no successors of the current state which are better than the current state itself. Figure 2.2(a) and (b) show the hill-climbing algorithm on discrete and continuous domain spaces, respectively. Note, the algorithm does not attempt to exhaustively try every vertex and edge (or the entire search space in the continuous domain case), so no previously visited vertex list is maintained–the algorithm only tracks the current vertex being visited.

One problem with hill-climbing algorithm is *local maxima* in the search space, where the current state gets trapped causing the algorithm to terminate before finding the optimal state. There are several ways we can get around this problem with varying degree of success by extending the algorithm. We can use a limited amount of *backtracking* so that we record alternative reasonable looking paths which weren't taken and go back to them. Alternatively, we can use the *momentum term* by giving weight to the preceding movement direction to allow jumping over the small local maxima on the way to the optimal state. However, none of these solutions are perfect. Another extension is multiple restart stochastic hill-climbing (MRSH) [24], which simply runs an outer loop over hill-climbing. Each step of the outer loop chooses a random initial state to restart hill-climbing. The best state is kept: if a new run of hill-climbing produces a better state than the stored state, it replaces the stored state. Since at least one of the hill-climbing runs is likely to reach the optimal state successfully, MRSH is surprisingly effective in many cases.

Chapter 3

Related Work

3.1 Run-time Hardware Optimization

Processors are designed to achieve good average performance across various applications. At run-time, however, the usage of the processor resources is unbalanced; some resources are fully utilized and others are under-utilized. The unbalanced resource utilization happens because application's demand for the resource is diverse and the demand also changes over time even within an application.

To deal with the unbalanced resource utilization problem, run-time hardware optimization techniques have attempted to match the hardware configuration to the running application's resource demand by allowing some degree of flexibility in the hardware design. SMT processor resource distribution techniques [1, 10, 6, 5, 4] also tune multiple hardware parameters (*i.e.*, resource partition or fetch priority of each thread) at run-time to adapt to the time-varying application characteristics. Therefore, SMT processor resource distribution is a specific field of run-time hardware optimization study. In this section, we will compare several run-time hardware optimization studies in terms of three aspects: optimization goal, optimal configuration finding method, and configuration update frequency.

Optimization Goal

There are two goals of any run-time hardware optimization. The first goal is "achieving better performance." For better performance, under-utilized hardware budget must be shifted to support heavily used resources, thus removing the performance bottleneck. For example, part of the functional unit in a host processor [25, 26] or independent co-processor [27, 28, 29, 30] are reconfigured using FPGA to achieve optimal performance for the target application. Balasubramonian, *et al.* [13] use configurable cache organization to find the best on-chip memory partition between L1 and L2 cache for the current phase of the application. All SMT processor resource distribution techniques fix the unbalanced resource distribution by shifting the resource between concurrently running threads.

The second goal is "saving power consumption." To save power, under-utilized devices are dynamically turned off or slowed down. For example, Buyuktosunoglu, *et al.* [14] dynamically disable some of the issue queue entries based on the issue queue utilization. Manne, *et al.* [31] reduce the number of flushed instructions due to branch misprediction by preventing instruction fetching if the number of low-confidence branch predictions exceeds a threshold. Banasadi, *et al.* [32] gate the decode pipeline if the number of instructions to be decoded is less than the decode bandwidth. Karkhanis, *et al.* [12] dynamically control the maximum number of in-flight instructions in the pipeline to save power.

Even with two distinct goals in run-time hardware optimization, the techniques used for one goal can be easily applied to the other goal. For example, both studies in [31] and [10] use the confidence of the branch prediction to either save power or improve the performance.

Optimal Configuration Finding Method

To maximize the goal of run-time hardware optimization, three techniques have been studied to find the best configuration at run-time. The first technique "exhaustively tries” all possible configurations, and then picks the best performing one. This approach is useful if the search space volume is very small [12]. The second technique “searches” for the optimal configurations by trying a carefully chosen sequence of configurations and then picking the best performing one. The techniques used in [13, 14] carefully increment or decrement the L1 cache or issue queue size depending on the recent performance feedback. Our hill-climbing SMT processor resource distribution uses this technique to find the optimal resource distribution. This approach increases the search speed compared to the exhaustive trials, making it applicable to large search space. The third technique “loads” one of the pre-defined configurations based on indicators. All FPGA based systems load pre-defined configurations at start-up of an application or periodically during run-time [25, 26, 27, 28, 29, 30]. Prior SMT processor resource distribution techniques rely on indicators (*e.g.* L1 / L2 cache miss count, resource occupancy, or branch misprediction count) to change the resource partition or fetch priority at run-time.

The third approach is useful if the search space volume is huge or the configuration should be chosen promptly. On the contrary, the first two approaches use the *feedback* information from the trial configurations to pick the best performing one, thus delaying the decision making time. In addition, trying all the configuration or searching for the optimal configuration out of a huge search space may require too many trials before we find the optimal one. However, the third approach determines the configuration based on the indicators without actually trying and measuring the performance of the alternative configurations. Furthermore, the indicators may not be expressive enough to show the myriad resource requirement scenarios of the application. Therefore, there is a danger that the configuration picked by indicators may be a non-optimal one.

Configuration Update Frequency

There is a spectrum of configuration update frequencies. First, if the configuration update overhead is very high, the configuration is updated at the "application start-up time." Most of the FPGA based techniques load the pre-defined configuration at the application start-up time because loading the FPGA configuration (*i.e.*, netlist) takes at least a few minutes. Second, the configuration can be changed only when the "phase of the application shifts." Most of the cache reconfiguration techniques update the configuration when the phase of the application changes because the changing cache configuration requires invalidation of all the cache lines [13]. Third, the configuration can be updated "every fixed interval in time." Both adaptive clock frequency [33] and adaptive issue queue size [14] techniques update the configuration every fixed interval. Our hill-climbing SMT processor resource distribution also updates the resource partition every fixed interval. Fourth, the configuration is updated "every cycle." All the prior SMT processor resource distribution techniques update resource partition or fetch priority dynamically every cycle to adapt to the cycle-by-cycle changing application behavior.

There is a trade-off in determining the hardware configuration update frequency. As we increase the update frequency, the hardware adapts to the time-varying application characteristics quickly. However, with high frequency update, the run-time overhead increases due to the frequent configuration flushing operations, finding new configurations, and new configuration loading time. In addition, techniques to reduce the run-time overhead (specifically reducing "finding new configuration time") may lead to low-quality configurations.

The goal of our hill-climbing SMT processor resource distribution is to improve performance. Our approach's method of finding the optimal configuration is searching after attempting a few trial configurations. The configuration is updated at every fixed intervals. Compared to the other run-time hardware optimization techniques, the uniqueness of our approach is that the sequence of the trial configurations is *systematically defined based on the hill-climbing algorithm.* Therefore, our approach reduces the number of trials, minimizes the search overhead, is expandable to huge and multi-dimensional search spaces, and provides high quality configurations. Compared to prior SMT processor resource distribution techniques, the uniqueness of our approach is *searching* for the optimal configuration out of a large number of possible configurations using feedback information, and *fixed interval* granularity configuration update.

3.2 SMT Processor Resource Distribution

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 [1] and FPG [10] are examples of such SMT fetch policies. These techniques monitor indicators of resource usage, such as resource occupancy (ICOUNT) or branch prediction accuracy (FPG). Every cycle, the threads using their resources most efficiently (*e.g.*, with low occupancy or few branch miss-predicts) are given fetch priority. By favoring efficient 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-stall the threads when they suffer long-latency memory operations. Techniques in this category differ in how they detect the stall condition. STALL [6] triggers fetch-stall when a load remains outstanding beyond some threshold number of cycles; DG [5] triggers fetch-stall when the number of cache-missing loads exceeds some threshold; and PDG [5] uses a cache-miss predictor to trigger fetch-stall.

One problem with fetch-stalling is resource clog can still occur because the stall condition is detected either too late or unreliably. Instead of anticipating resource clog and fetch-stalling, a second approach allows resource clog to occur but immediately recovers it by flushing the stalled instructions. This is the approach taken by FLUSH [6]. 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 [6]) minimize the number of flushed instructions by first employing fetch-stall, and resorting to flushing only when resources are exhausted.

Finally, a third approach is to partition the processor resources. The simplest is static partitioning [15, 16, 7], but these techniques cannot adapt to changing workload behavior. In contrast, DCRA [4] 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 partition of the threads that can use resources most efficiently.

Compared to previous techniques, hill-climbing 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 our technique makes partitioning decisions based on performance *feedback*, thus optimizing end performance as illustrated in Figure 1.1. In contrast, DCRA and other previous techniques perform resource distribution based on hardware monitors like resource occupancy or cache miss counts. Hence, they optimize performance only indirectly, potentially missing opportunities for performance gains, as discussed in Chapter 1. An added benefit of exploiting feedback is we can optimize to a user-definable performance goal–like throughput, per-thread speedup, or fairness–by simply changing the performance metric used for the performance feedback. Previous techniques cannot tailor their optimizations to a specific performance goal. Lastly, because it takes time for our hill-climbing algorithm to process performance feedback, we update partitioning decisions at every *fixed interval*. Thus, our technique lies somewhere in between DCRA (update every cycle) and static partitioning (fixed) in terms of its responsiveness to dynamic runtime behavior.

Chapter 4

SMT Processor Run-Time Performance Behavior Analysis

The performance of SMT processor is mainly determined by the resource distribution among the concurrently running threads. Therefore, both of the existing techniques, fetch polices and resource partitioning techniques, attempt to implicitly and explicitly balance the resource distribution. To find out the best resource distribution, we propose a unique approach. We view the SMT processor resource distribution problem as the searching for the maximum performance in resource distribution space. This view allows us to translate the resource distribution problem into the classical optimization problem. Since the shape of the performance curve looks like hill, as illustrated in Figure 1.2, we apply a general optimization problem solver, hill-climbing algorithm, to the SMT processor resource distribution problem.

In this chapter, we define the performance curve on which we make our hill-climbing algorithm to climb up to search for the optimal resource distribution. Since, the nature of SMT processor performance curve is unknown as is defined and used in our research for the first time, in this chapter, we perform in-depth analysis of the performance curve and provide its look-and-feel. The analysis on the performance curve guides the design of our hill-climbing algorithm (in Chapter 5) and helps understanding the experimental results (in Chapter 6, 7, and 8).

4.1 Performance Curve

To distribute resources among the concurrently running threads, we maintain resource partition for each thread. During the execution, threads are allowed to consume up to (but no more than) the allotted resources within their partition. Hence, partitioning guarantees every thread to receive some fraction of each shared resource. All possible combinations of the resource partition among the threads constitute the *resource distribution space*. As the performance of a resource partition can change over time to adapt to the changing the application behavior, we define the *performance curve* f_i at a point in time i as Equation 4.1.

$$f_i : resource\ distribution\ space \longmapsto performance \tag{4.1}$$

This hypothetical performance curve maps a resource distribution at a point in time to its performance outcome. This performance curve is the target hill where our hill-climbing algorithm searches for the peak.

The nature of the hills, (*i.e.*, performance curve) in our study are different from the hills that traditional hill-climbing algorithms assume. As described in Section 2.3, traditional hill-climbing algorithms assume fixed hills. But, in our case, the shape of the hill changes over time as the characteristics of the applications running on SMT processor change. Therefore, the peak of the hill is a moving target for which our mechanism should search.

As a first step of our research, we perform in-depth analysis on the performance curve, named *OFF-LINE-Analysis*, because the performance curve is defined and used for the first time in our research. As we will show in Chapter 7, the advantage of hill-climbing resource distribution highly relies on the shape of the performance curve. OFF-LINE-

Analysis schedules resources off-line to get the perfect knowledge of the performance curve. So, this analysis provides an oracle view which any real implementation is unable to figure out at run-time and allows in-depth insight into the performance curve.

4.2 OFF-LINE-Analysis

The goal of OFF-LINE-Analysis is to provide the global view of the performance curve. To achieve this goal, OFF-LINE-Analysis tries to discover the whole mapping of the performance evaluation function f_i for all i's in Equation 4.1 by trying all data point in resource distribution space.

4.2.1 Implementation Issues

There are two issues in providing the global view of the time-varying hill-shaped performance curve.

Huge Volume of the Resource Distribution Space

One problem with providing the view of the performance curve is the intractably large resource distribution space. Each data point in the performance curve represents the performance of a resource distribution, which needs to be evaluated individually. So, given S shared resource types, E_i entries for resource type i, and T threads, the number of unique ways to distribute the resources is $O(\Pi_{i=1}^{S} E_i^{(T-1)})$. Even for modest values of S, E_i, and T, the size of the resource distribution space becomes intractably large to be evaluated using simulation. To reduce the number of unique resource distributions, we observe that a thread's usage of different hardware resources is *not* independent; instead, the number of entries of each resource type a thread occupies is often related (*e.g.*, a thread can never use more rename registers than the number of ROB entries it holds). So, many cases do not need to be explored by the OFF-LINE-Analysis. We exploit this observation in two ways. First, we assume the number of integer IQ entries, integer rename registers, and ROB entries occupied by a thread are *in proportion to one another*. Rather than distributing the three resources independently, we distribute integer rename registers only, and then apply the same distribution proportionally to all other resources. Second, we abstain from explicitly distributing the fetch queue, floating point IQ, floating point rename register, L1 cache, and branch predictor. For those resources, any thread that needs them grabs them, as long as there are available entries (the request for L1 cache and branch predictor evicts the existing entry), thus increasing the utilization of those resources. By explicitly partitioning the integer rename registers, integer IQ, and ROB for each thread, the number of in-flight instructions in the pipeline per thread is controlled. Please note that the three resources that we explicitly control–integer rename registers, integer IQ, and ROB–for each thread are the queue type resource (defined in Section 2.2) and controlling over queue type resource is the most important in balancing the resource distribution. This implicitly partitions the uncontrolled shared resources. These simplifying techniques reduce the number of unique resource distributions to $O({E_{integer\_rename\_register}}^{(T-1)})$, making the OFF-LINE-Analysis feasible.

However, the resource distribution space is still very large, especially when T is 2 or larger. Therefore, we uniformly down sample the resource distribution. Table 4.1 shows our OFF-LINE-Analysis settings. Our sample size is 128, 496, and 680 resource distribution configurations out of 256, 32,896, and 2,962,206 for 2-, 3-, and 4-threaded workloads, respectively. The sampling rate for 2 threads is essentially exhaustive providing a complete view of the performance curve throughout the entire resource distribution

# threads	1	2	3	4
space complexity	$O(N)$	$O(N)$	$O(N^2)$	$O(N^3)$
space volume	256	256	32,896	2,862,206
purpose	Analyzing the time varying performance behavior			
# samples / epoch	128	128	496	680
sampling rate	50.000%	50.000%	1.508%	0.024%
sampling method	uniform, step:2	uniform, step:2	uniform, step:8	uniform,step:16
sampling frequency	every epoch	every epoch	every epoch	every epoch

Table 4.1: OFF-LINE-Analysis simulation settings.

space. Even with the relatively small sampling rate for 3- and 4-threaded workloads, OFF-LINE-Analysis still allows us to make significant and consistent observations just like 2-threaded workloads. Table 4.1 also shows the step size of the sampling that allows uniform down sampling across the entire resource distribution space. As shown in Table 4.1, we conducted the 1-threaded workload experiment because the characteristics of the multi-programmed workloads depends on that of the individual applications belonging to each workload. For 1-threaded workloads, we varied the amount of resources allowed to the thread.

Performance Evaluation Frequency

For the OFF-LINE-Analysis, we choose to evaluate the performance of SMT processor periodically. We divide SMT execution into a linear sequence of *epochs* or fixed-size time intervals.

If epoch size is extremely small, inter-epoch behavior becomes too dynamic because the performance within an epoch is vulnerable to any small events like individual L2 misses or branch mispredictions. As a result, the performance of sequence of epochs will have large fluctuation. In order to get more meaningful performance result, we need to average out the effect of noise by increasing the epoch size. If epoch size is too large, the performance curve summarizes too much run-time performance behavior, thus loosing the detailed time-varying behavior information. In Section 5.3.1, we discuss the choice of the epoch size, and in Section 8.1.2, we present the sensitivity of the choice of the epoch size to the performance of our hill-climbing resource distribution implementation. Based on our study, we picked 64K cycle for the granularity of the performance evaluation period (*i.e.*, epoch size).

4.2.2 Probe-Based Simulation Methodology

Normal simulator executes the workload's simulation window only once and collects the statistics. For the OFF-LINE-Analysis simulation, we developed a new simulation methodology, called *probe-based simulation*, which executes each epoch multiple times to explore the alternative configurations that the simulator can potentially choose, before moving on to the next epoch. Such off-line probing is impractical for real machines. But, its evaluation via simulation provides a oracle view of SMT processor resource distribution space.

At the end of each epoch, probe-based simulation check-points every processor memory structure (register file, pipeline registers, branch predictors, caches, etc.), the contents of main memory, as well as the simulator internal data structures. This check-point is a fresh copy of simulator state from which all trials of the subsequent epoch begins. Probe-based simulation repeatedly simulates the subsequent epoch with variety of resource distributions, thus discovering the performance of multiple alternative resource distributions. We call each repeated test a *probing*. Each probing begins its simulation by restoring the check-point to avoid any side effects from the previous probings. A probing simulate its assigned resource distribution for one epoch only.

To simplify the implementation of check-pointing, we use a UNIX system call, *fork()*. The fork() generates a child process, which is a clone of the parent simulator process. The clone is exactly same as the parent process except for its process id and its assigned resource distribution configuration. All probings are done by the child simulator processes (*i.e.*, clone). Since the child simulator process is allowed to update only its own memory, the parent process is unaffected by the child process. After the child simulator process simulates the assigned resource distribution configuration for an epoch, it reports its simulation result to the parent process and terminates itself, discarding all the changes made by the child simulator process. The parent process forks the child process again to probe another resource distribution configuration from the saved simulator status thus making all probings independent of each other. After trying all the probings, parent process updates its configuration into one of the resource distributions based on their reported performance result, executes itself for an epoch again, and initiates another set of probings.[1]

Figure 4.1 illustrates the probe-based simulation timing diagram. Only parent process is allowed to simulate the whole simulation window. Child processes probes diverse resource distribution configurations for an epoch.

4.2.3 Probe-Based Simulation Algorithm

Figure 4.2 shows the detailed probe-based simulation algorithm. Probing begins after the parent process finishes execution of an epoch (line 7 and 12). The parent process picks a sample resource distribution configuration (line 14), forks a child process (line 16),

[1]In this analysis, the parent process chooses one of the resource distributions for its execution using the hill-climbing algorithm. But this choice is less related to the analysis of the run-time behavior of the performance curve.

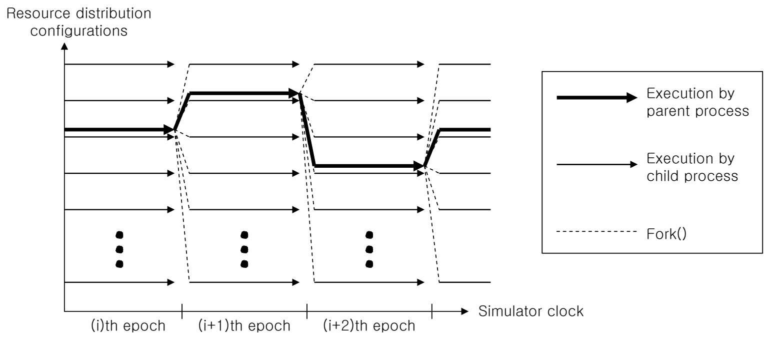

Resource distribution configurations
Execution by parent process
Execution by child process
Fork()
Simulator clock
(i)th epoch
(i+1)th epoch
(i+2)th epoch

Figure 4.1: Probe-based simulation timing diagram. Parent simulator process forks multiple child processes to investigate the various alternatives of the future (*i.e.*, the next epoch). Only the parent process is allowed to execute the end-to-end simulation.

```
1.  #define compute_performance()   compute the performance of the previous epoch
2.  #define send_to_parent(X)       send X to parent process
3.  #define receive_from_child()    receive data from child process
4.  #define sample_a_conf()         pick a configuration based on sampling method and (or)
                                    earlier sample configurations and their performance outcome
5.  #define conf[]                  resource sharing configuration of child processes
6.  #define perf[]                  performance of child processes

7.  if (end_of_an_epoch) {
8.      if (I_am_child_process) {   // child process finished execution of an epoch
9.          p = compute_performance();
                                    // compute child process's performance for the previous epoch
10.         send_to_parent(p);      // inform the parent process of the performance of the child process
11.         exit();                 // terminate itself (i.e. child process)
12.     } else if (I_am_parent_process) {
13.         for (i = 0 ; i < num_samples ; i++) {
14.             current_conf = sample_a_conf(sampling_method, conf[0 : i-1], perf[0 : i-1]);
                                    // pick a new configuration for the child process to investigate
15.             conf[i] = current_conf;   // save the new configuration
16.             fork();             // create a child process which begins simulation
                                    // from the parent process's current machine state

17.             if (I_am_parent_process) {
18.                 wait();         // wait for the completion of the child process
19.                 perf[i] = receive_from_child();
                                    // save the child process's performance outcome
20.             } else if (I_am_child_process)
21.                 break;          // do not allow child process to iterate this loop
22.         }

23.         if (I_am_parent_process)    // set the best configuration for the parent process
24.             current_conf = conf[m], where perf[m] is the best;
25.     }
26. }
```

Figure 4.2: Probe-based simulation algorithm pseudo code. Child processes investigate the alternative configurations that the parent process can potentially choose from (line 14).

and suspends itself (line 18). After the child process executes an epoch (line 8), the child process reports its performance measured during the simulated epoch to the parent process (line 10), and terminates itself (line 11). The completion of the child process wakes up the parent process (line 18). The parent process receives the probing result from the child process and records it (line 19). The parent process iterates these steps until it finishes trying all the alternative configurations.

Note, there are three metrics to compute the performance evaluation function, compute_performance() (line 9): average IPC, average weighted IPC [34], and harmonic mean of weighted IPC [35]. Equation 4.2, 4.3, and 4.4 define the three metrics, where IPC_i is the IPC of the ith thread in SMT processor, $SingleIPC_i$ is the IPC of the stand-alone execution of the ith thread, and T is the number of simultaneously running threads.

$$\mathsf{Average\_IPC} = \frac{\sum IPC_i}{T} \tag{4.2}$$

$$\mathsf{Average\_Weighted\_IPC} = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{T} \tag{4.3}$$

$$\mathsf{Harmonic\_Mean} = \frac{T}{\sum \frac{SingleIPC_i}{IPC_i}} \tag{4.4}$$

Each metric presents the performance of the SMT processor based on its own goal. Average IPC quantifies throughput improvement (*e.g.*, the number of finished programs); average weighted IPC quantifies execution time reduction (*e.g.*, the response time); and harmonic mean of weighted IPC quantifies both performance and fairness improvement (*e.g.*, the completion time of the given group of jobs).

For our OFF-LINE-Analysis, we will use the *average weighted IPC metric* only. Later, when we evaluate hill-climbing resource distribution in Section 7.2.2, we will use all three performance metrics.

4.2.4 Probe-Based Simulation Overhead

The fork() system call is implemented with copy-on-write paging technique to reduce the fork overhead. Therefore, initially each virtual memory page is shared between parent and child process without making any extra copies, until the page is written by any one of the processes. We observe that the kernel mode execution and page copying overhead due to the fork system call is less than 5% of the simulation time. The biggest overhead comes from the repeated simulation of the epochs even after the down sampling (As shown in Table 4.1, 128, 496, and 680 times of repetition, for 2-, 3-, and 4-threaded workloads, respectively).

4.2.5 Experimental Methodology

Our experiments are performed on a detailed event-driven SMT processor simulator that models the processor pipeline as illustrated in Figure 4.3. The simulator is derived from sim-ssmt [36], an extension of the out-of-order processor model in SimpleScalar [37], and has been used previously to study several SMT techniques [38, 39, 40, 41]. For our evaluation, we model an 8-way issue SMT processor with up to 4 hardware contexts and a 512-entry reorder buffer. The processor and memory system settings for our simulations are listed in Table 4.2.

Figure 4.3 illustrates our processor model. Like other techniques that explicitly control resource distribution (*e.g.*, DCRA [4]), we dynamically partition several shared hardware resources in SMT pipeline. Specifically, we target the integer issue queue, integer rename registers, and reorder buffer (ROB), which are shaded gray in Figure 4.3. We simply keep per-thread occupancy counters for three resources and allow fetching of a thread as long as its resource occupancy hasn't exceeded any of the three resource

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

Table 4.2: SMT processor simulator settings.

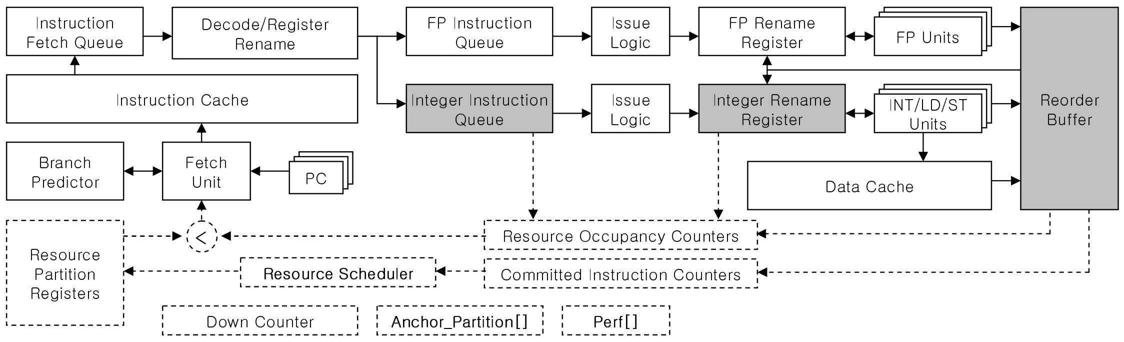


Figure 4.3: Block-level diagram of our SMT processor model. Shaded boxes indicate shared hardware structures that are partitioned by the OFF-LINE-Analysis. Dashed boxes indicate additional hardware needed for the implementation of our hill-climbing resource distribution implementation, which we will detail in Section 5.4.

ILP			MEM		
App	Skip Ins	Type	App	Skip Ins	Type
bzip2	1,100M	Int	vpr	300M	Int
perlbmk	1,700M	Int	mcf	2,100M	Int
eon	100M	Int	twolf	2,000M	Int
vortex	100M	Int	equake	400M	FP
gzip	200M	Int	art	200M	FP
parser	1,000M	Int	lucas	800M	FP
gap	200M	Int	ammp	2,600M	FP
crafty	500M	Int	swim	400M	FP
gcc	2,100M	Int	applu	800M	FP
apsi	2,300M	FP			
fma3d	1,900M	FP			
wupwise	3,400M	FP			
mesa	500M	FP			

Table 4.3: SPEC CPU2000 benchmarks used to create our multi-programmed workloads.

partitions. If one or more partitions become exhausted, the corresponding thread is fetch-stalled until it releases some of its entries in the exhausted partition(s). In addition to resource partitioning, we also use the ICOUNT policy [1] in the fetch stage to select the threads from which to fetch every cycle. On top of this baseline SMT processor simulator, we added a probe driving routine, which implements the algorithm in Figure 4.2, to conduct probe-based simulation for the OFF-LINE-Analysis.

Our study is driven by 63 multiprogrammed workloads created from 22 SPEC CPU2000 benchmarks. Table 4.3 lists our benchmarks. We use the pre-compiled alpha binaries from Chris Weaver[2], which were built with the highest level of compiler optimization. All of our benchmarks use the reference input set provided by SPEC. From the benchmarks, we created multiprogrammed workloads by following the methodology

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

in [4, 6]. We first categorized the SPEC benchmarks into either high-ILP or memory-intensive applications, labeled “ILP” and “MEM,” respectively, in Table 4.3. Then, we created 3 groups of 2-, 3-, and 4-threaded workloads. Table 4.4 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 multi-programmed workloads in the following way. First, we used SimPoint [42] to analyze the first 16 billion instructions (or the entire execution, whichever is shorter) of each benchmark, and picked the earliest representative region reported by SimPoint. In our SMT simulations, we fast-forward each benchmark in the multi-programmed workload to its representative region. Table 4.3 reports the number of skipped instructions (“Skip Ins” column) in each benchmark during fast forwarding. Finally, we turn on detailed multi-programmed simulation, and simulate for 100M “on-line” instructions executed by the parent simulator process (*i.e.*, not counting the “off-line” probings needed for OFF-LINE-Analysis).

Due to the cost of OFF-LINE-Analysis simulation, we are unable to simulate more instructions; however, the regions we simulate are representative thanks to the SimPoint analysis. Note, we use 100M instruction window only for the analysis of the workload in this chapter and for the limit study in Chapter 6. When evaluating the performance of our hill-climbing algorithm, reporting the end-to-end performance, and comparing against other techniques in Chapter 7, we use larger simulation regions of 1B instructions.

TYPE	2-threaded	3-threaded	4-threaded
ILP	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
MIX	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 crafty parser	art gap twolf crafty
	equake bzip2	bzip2 mesa swim	swim fma3d vpr bzip2
MEM	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

Table 4.4: Multiprogrammed workloads used in the experiments.

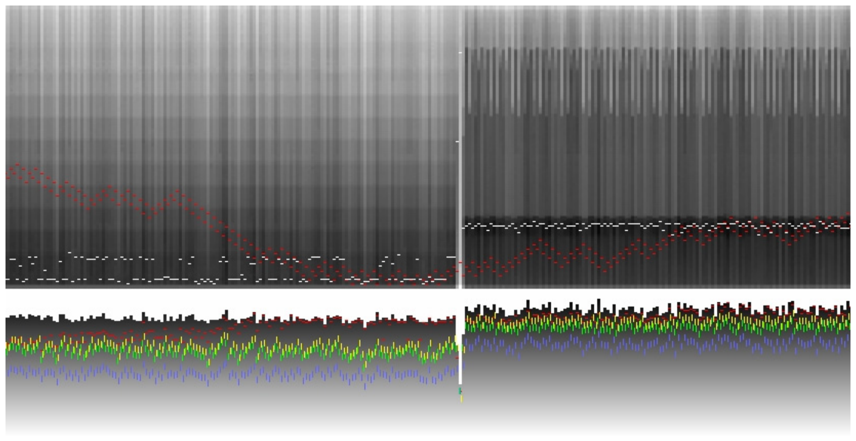
4.2.6 SMT Processor Run-Time Performance Behavior Analysis

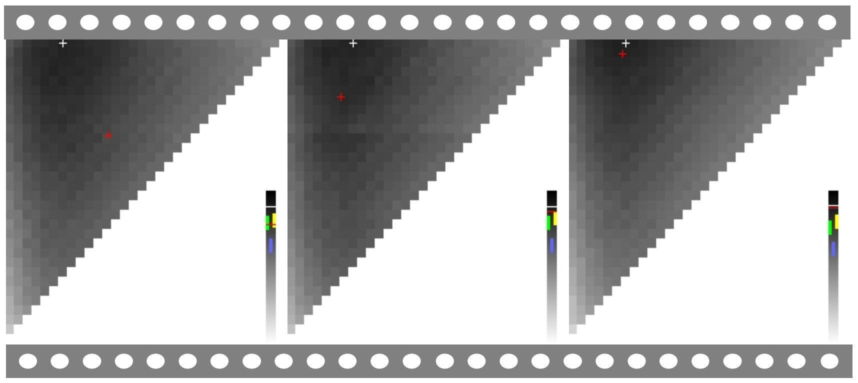
The performance curve provided by OFF-LINE-Analysis contains the SMT processor's run-time performance changes represented as a function of both time and resource distribution. But the volume of the raw data is very large as the simulation window and the number of applications in the workload increase thus making the interpretation of the data difficult. Therefore, we designed methodologies for both qualitative and quantitative analysis of the performance curve. Our qualitative analysis visualizes the performance curve into a still image (for 2-threaded workload) or into a motion picture (for 3- and 4-threaded workload). The qualitative analysis allows us to visually inspect the time-varying performance behavior. This inspection–albeit manual–provides intuition and in-depth understanding of the relationship between the SMT processor performance and the resource distribution. In addition, the qualitative analysis enables the development of the quantitative analysis methodology. We detail the qualitative and quantitative analysis in the following sections.

4.3 Qualitative Analysis of the Run-Time Performance Behavior

First, we built the workload performance behavior visualization tool to display the raw data in an intuitive way and provide qualitative understanding of the performance behavior of the workloads.

Like a topographical terrain map, our visualization tool displays the performance curve graphically as shown in Figure 4.4. For 2-threaded workloads, the X-axis represents the simulation time (in epochs) and the Y-axis represents the resource distribution of the first application. (The second application gets the remaining resources.) For 3-threaded workloads, the X- and Y-axis are the resource distribution of the first two applications (the


(a) 2–threaded workload (swim–twolf)


(b) 3–threaded workload (equake–vpr–swim)

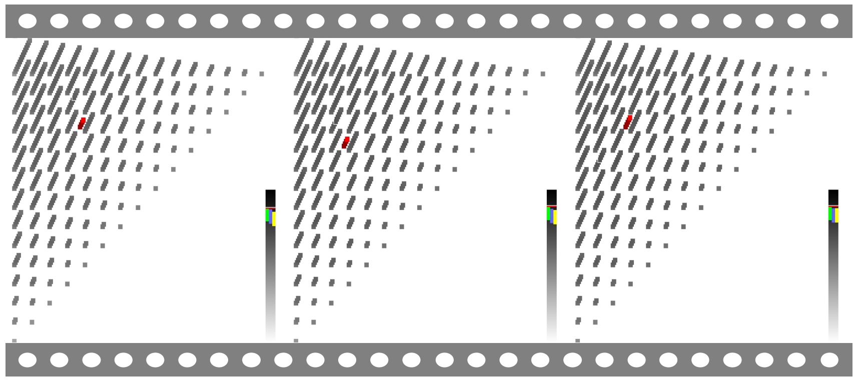

(c) 4–threaded workload (mesa–gzip–fma3d–bzip2)

Figure 4.4: Snapshot of SMT processor run-time behavior visualization tool for (a) 2-threaded, (b) 3-threaded, and (c) 4-threaded workloads. The run-time behavior of a 2-threaded workload is displayed as a still picture frame, and those of 3- and 4-threaded workload are displayed as motion pictures.

third applications gets the remaining resources.), and each frame in the motion picture represents the simulation time (in epochs). For 4-threaded workloads, X-, Y-, and height of the bar represents the resource distribution of the first three applications, and each frame in the motion picture represents the simulation time (in epochs). In each image, the weighted IPC is reported as a gray scale. Dark colored areas denote high weighted IPC while light colored areas denote low weighted IPC. In 2-threaded workloads, by following the change in gray scale along any vertical line, we can determine the shape of the performance hill within the corresponding epoch. The white dots indicate the position of the peak of the hill (*i.e.*, optimal resource distribution). Our visualization tool provides more information than this dissertation covered up to this chapter. For example, this visualization tool shows the synchronized performance of ICOUNT (green), FLUSH (blue), DCRA (yellow), and HILL-WIPC (red), which we will detail in Section 6.2.1.

This visualization tool allows us to identify several key patterns of the performance curve. The patterns of the time-varying shape of the performance curve include random changing pattern, fine/coarse grain alternating pattern, and stable pattern. The patterns of the performance curve within an epoch include single/multiple hump(s), deep/shallow valley, and sharp/dull peak. Among these patterns, random changing pattern, fine grain alternating pattern, multiple humps, and deep valley are potentially hostile to our hill-climbing algorithm because these patterns delay (or prevent from) finding the peak of the hill.

One of the sources of the random changing or fine grain alternating pattern of the multi-threaded workloads is the frequent function calls in the integer applications. For example, gcc and parser makes call to a function appropriate to handle each token found in the input file sequence. Therefore, many small functions with diverse resource requirement are called frequently causing temporal variations in the performance curve.

One source of the multiple humps and deep valley in multi-threaded workloads is a parallel loop with a long latency load instruction in each loop iteration. As we increase the amount of the resource given to a thread with the parallel loop, the processor pipeline contains more in-flight instructions from the parallel loop iterations. However, the performance of the thread does not improve until the long latency load instruction in the loop is included in the pipeline and overlap multiple memory operations. So, as we increase the amount of resource given to the thread, we get step-shaped performance improvement. Before the performance of the step-shaped thread jumps up, the overall performance of the multi-threaded workload decreases as the other thread(s) gets less amount of resources. But the overall performance jumps up when the processor pipeline includes the long latency instruction in the step shaped thread, thus generating a local hump.

These patterns that we observe via visualization tool guide us to develop metrics to quantitatively measure the characteristics of the workloads.

4.4 Quantitative Analysis of the Run-Time Performance Behavior

In this section, we present the workload characterization metrics to quantify the run-time performance behavior of the workloads. And then, we classify workloads based on the measured metrics.

4.4.1 Workload Characterization

Here, we define 4 workload characterizing metrics.

Reversed Gradient Area

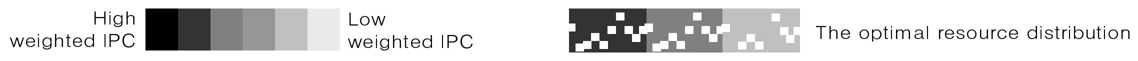
The reversed gradient area metric measures the total depth of the slope whose gradient vector points in the opposite direction of the optimal position. The bigger the reversed gradient area, the more local maxima there are in the performance curve.

Figure 4.5(a) presents the definition of the reversed gradient area, where Y-axis shows the performance in average weighted IPC and X-axis shows the resource distribution space. We do not directly measure the volume of the local maxima because the definition of the local maxima becomes less clear in 3 or higher dimensional space.

To extend this definition to 3- and 4-threaded workloads, we formally defined the reversed gradient area in Equation 4.5. In this equation, $\overrightarrow{P}$ is the optimal resource distribution, $\overrightarrow{C_i}$ is i's resource distribution, $BN(i)$ is one of i's neighbors in resource distribution space, which has the maximum performance among all i's neighbors, $\overrightarrow{C_{BN(i)}}$ is $BN(i)$'s resource distribution, $WIPC_i$ is i's weighted IPC, and $WIPC_{BN(i)}$ is the $BN(i)$'s weighted IPC.

$$\mathsf{RGA} = \sum_{for\ all\ i} max(0, ((-1) \times \frac{(\overrightarrow{P} - \overrightarrow{C_i}) \cdot (\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i})}{|(\overrightarrow{P} - \overrightarrow{C_i})| \cdot |\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i}|} \times (WIPC_{BN(i)} - WIPC_i))) \tag{4.5}$$

The reversed gradient area, RGA, is the sum of the performance difference (*i.e.*, $WIPC_{BN(i)} - WIPC_i$), whose gradient vector (*i.e.*, $\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i}$) is opposite (*i.e.*, -1) to the optimal position (*i.e.*, $\overrightarrow{P} - \overrightarrow{C_i}$). The cosine (*i.e.*, dot product of two vectors and the division by their norm) of the angle between the gradient vector (*i.e.*, $\overrightarrow{C_{BN(i)}} - \overrightarrow{C_i}$) and vector to the optimal position (*i.e.*, $\overrightarrow{P} - \overrightarrow{C_i}$) is added to incorporate how opposite the two vectors are. Figure 4.5(b) and (c) are the snapshot from the visualization tool, whose depth of the valley is small and large, respectively. In Figure 4.5(c), the local maxima is represented as high contrast horizontal lines.



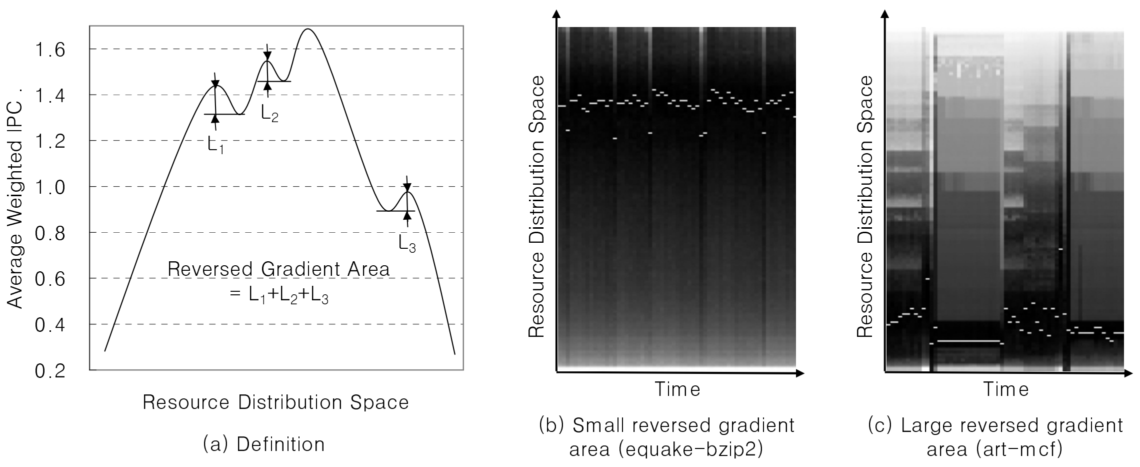


Figure 4.5: Reversed gradient area metric measures the amount of local maxima in the performance curve.

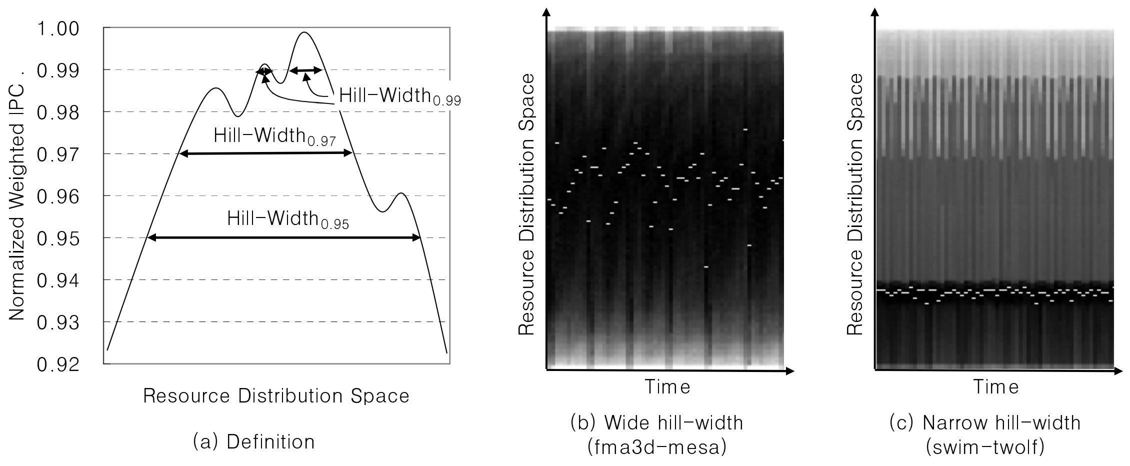


Figure 4.6: Hill-width$_X$ metric measures the fraction of the resource distributions whose performance outcome is better than X of the optimal one.

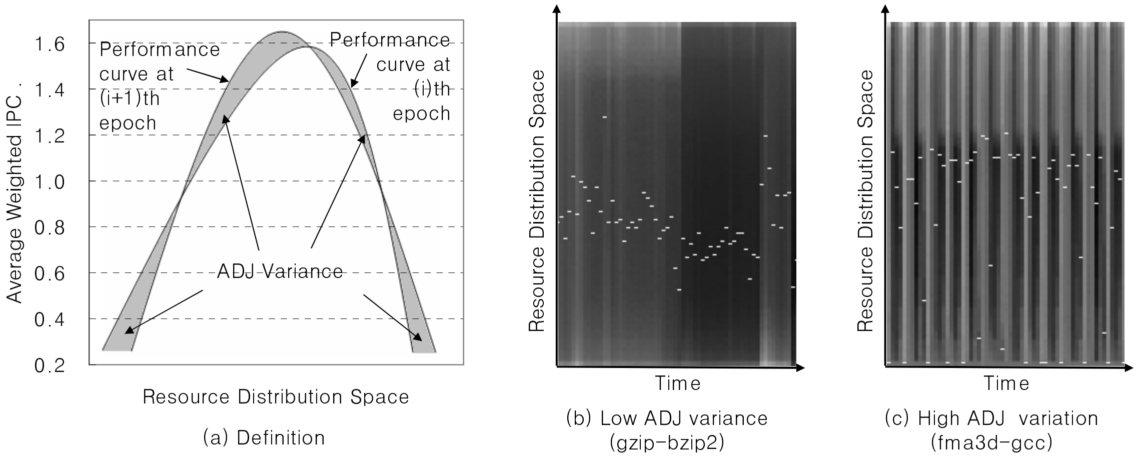


Figure 4.7: ADJ variance metric measures the performance difference between two adjacent epochs by diff-ing two performance curves.

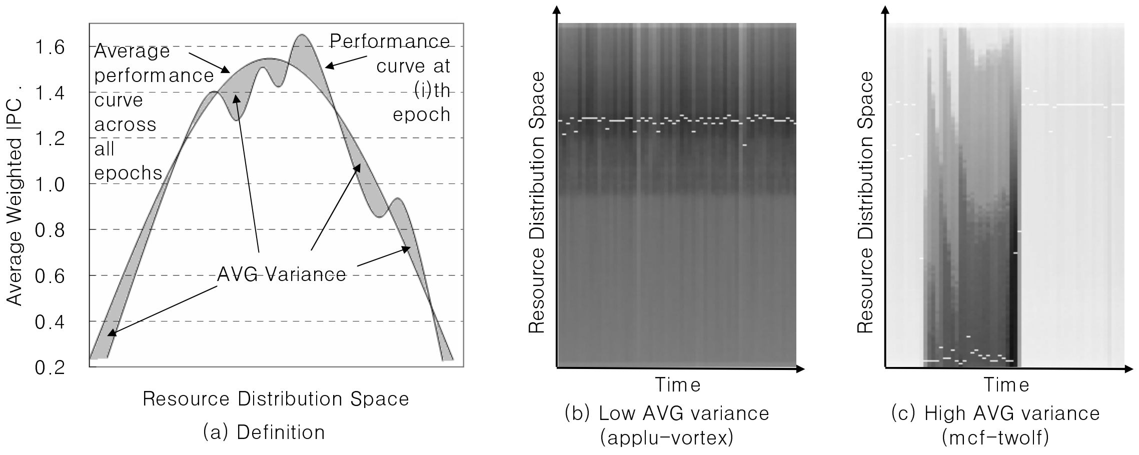


Figure 4.8: AVG variance metric measures the performance difference between the current curve and the averaged one.

Hill-Width

Figure 4.6(a) illustrates the definition of the hill-width metric. This metric quantifies the "sharpness" of the performance peak. We define *hill-width*$_X$ to be the fraction of the resource distribution whose performance outcome is better than X of the optimal performance. In Figure 4.6(a), we indicate hill-width$_{0.99}$, hill-width$_{0.97}$, and hill-width$_{0.95}$ on our hypothetical performance curve. Peak sharpness can be assessed by examining hill-width$_N$: a small hill-width$_N$ value indicates a sharp peak, while a large hill-width$_N$ value indicates a dull peak. Figure 4.6(b) and (c) show the workload examples with wide and narrow hill, respectively.

ADJ Variance

Figure 4.7(a) shows the definition of the ADJ variance metric. The ADJ variance metric measures the performance changes between two adjacent epochs, which represents the "short term" temporal variation in the performance curve. ADJ variance is computed by measuring the area difference in the performance curves between two adjacent epochs. Figure 4.7(b) and (c) show the workload examples with low and high ADJ variance, respectively.

AVG Variance

Figure 4.8(a) shows the definition of the AVG variance metric. The AVG variance metric measures every epoch's performance variation compared to the averaged performance curve, which represents combination of "short and long term" temporal variation in the performance curve. AVG variance is computed by measuring the area difference between the individual epoch's performance curve and the averaged one across all epochs.

Figure 4.8(b) and (c) show the workload examples with low and high AVG variance, respectively.

4.4.2 Workload Classification

Now we classify workloads based on the measured value of the four metrics, averaged across all epochs in the simulation window. Table 4.5 shows the conditions for our classification. This classification of the workloads discretizes the quantity measured using four metrics just for the convenience of explaining the characteristics of their performance curve. Therefore, the thresholds in the conditions are empirically chosen to balance the classification of the workloads between two extreme performance curve characteristics.

If a workload's reversed gradient area is less than 0.15, we call it SH (Single-Hump). Otherwise, we call it MH (Multiple-Humps). If a workload's hill-width$_{0.95}$ is more than 0.6, 0.5, 0.4, and 0.3 for 1-, 2-, 3-, and 4-threaded workloads, we call it WH (Wide-Hill). Otherwise, we call it NH (Narrow-Hill). We picked only hill-width$_{0.95}$ for our classification condition because hill-width$_X$ values for all X's are proportional to each other in most of the workloads. If a workload's ADJ variance is less than 0.05, we call it TS (Temporally-Stable). Otherwise, we call it TU (Temporally-Unstable). If a workload's AVG variance minus two times ADJ variance is less than 0, we call it TC (Temporally-Consistent)[3]. Otherwise, we call it TU (Temporally-Phased).

Figure 4.9 shows the reversed gradient area of our 63 workloads and the classification of the workloads based on SH and MH. The X-axis shows the classification of the workloads as well as the name of the workloads. Figure 4.10 reports hill-width$_N$ across several N (between 0.99 and 0.95) and classifies the workloads based on WH and NH;

[3]The intuition behind this condition is (short and long term temporal variation) - (short term temporal variation) makes (long term temporal variation).

each bar represents a hill-width$_N$ value averaged across all epochs from its corresponding workload. For the workloads labeled WH, 50% (2-threaded), 40% (3-threaded), and 30%(4-threaded) of all possible resource distribution's performance is as good as 95% of peak performance. As WH-labeled workloads have dull peaks, they are insensitive to non-optimal resource partitions. Figure 4.11 shows the ADJ and AVG variance of the workloads and the classification based on TS/TU, and TC/TP.

The 1-threaded workloads included in Figure 4.10 and Figure 4.11 are good references because the characteristics of the multi-threaded workloads are mainly determined by the individual characteristics of the participating applications. For example, in Figure 4.10, both fma3d and mesa have wide hill characteristics. So, the fma3d-mesa workload also exhibits wide hill characteristics. In addition, in Figure 4.11, gcc has high ADJ variance. Therefore, any workloads that include gcc also have high ADJ variance. 1-threaded workloads are excluded in Figure 4.9 because their performance curves are monotone non-decreasing and do not have any local maxima.

In the subsequent chapters, we will use these classifications and show the strong correlation between run-time characteristics of the workloads and SMT processor performance.

Condition	Label for true condition	Label for false condition
(Reversed Gradient Area)<0.15	SH (Single-Hump)	MH (Multiple-Humps)
1-thread: (Hill-Width$_{0.95}$)>0.6 2-thread: (Hill-Width$_{0.95}$)>0.5 3-thread: (Hill-Width$_{0.95}$)>0.4 4-thread: (Hill-Width$_{0.95}$)>0.3	WH (Wide-Hill)	NH (Narrow-Hill)
(ADJ variance)<0.05	TS (Temporally-Stable)	TU (Temporally-Unstable)
(AVG variance)−2×(ADJ variance)<0	TC (Temporally-Consistent)	TP (Temporally-Phased)

Table 4.5: The workload classifications based on the OFF-LINE-Analysis.

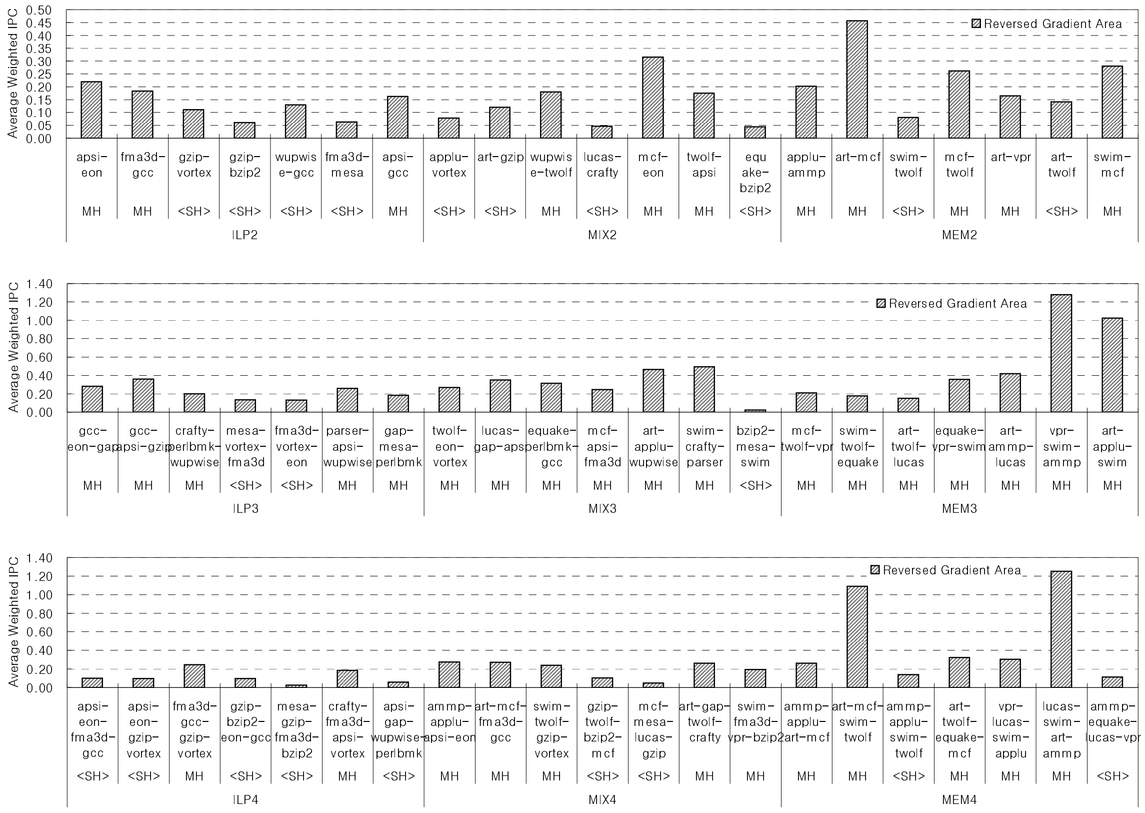


Figure 4.9: Reversed gradient area (a local maxima metric for OFF-LINE-Analysis, as illustrated in Figure 4.5) of our 63 workloads. Large reversed gradient area means many local maxima. Based on the condition in Table 4.5, workloads are classified as either single hump (SH) or multiple humps (MH).

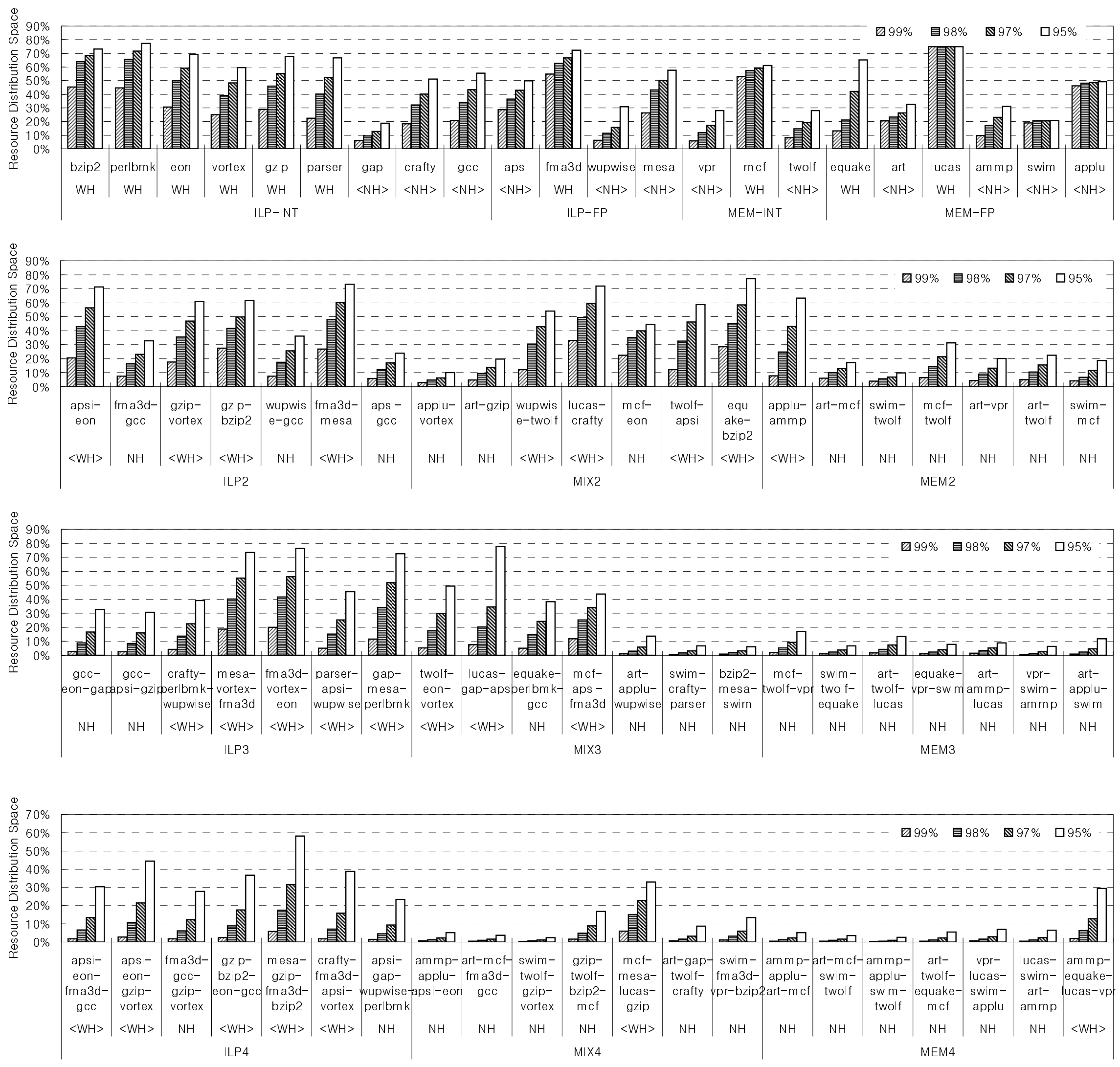


Figure 4.10: Hill-width (a metric illustrated in Figure 4.6) of our 63 workloads. Wide hill width means that we can achieve good performance from a wide range of less optimal resource distribution. Based on the condition in Table 4.5, workloads are classified as either wide hill (WH) or narrow hill (NH).

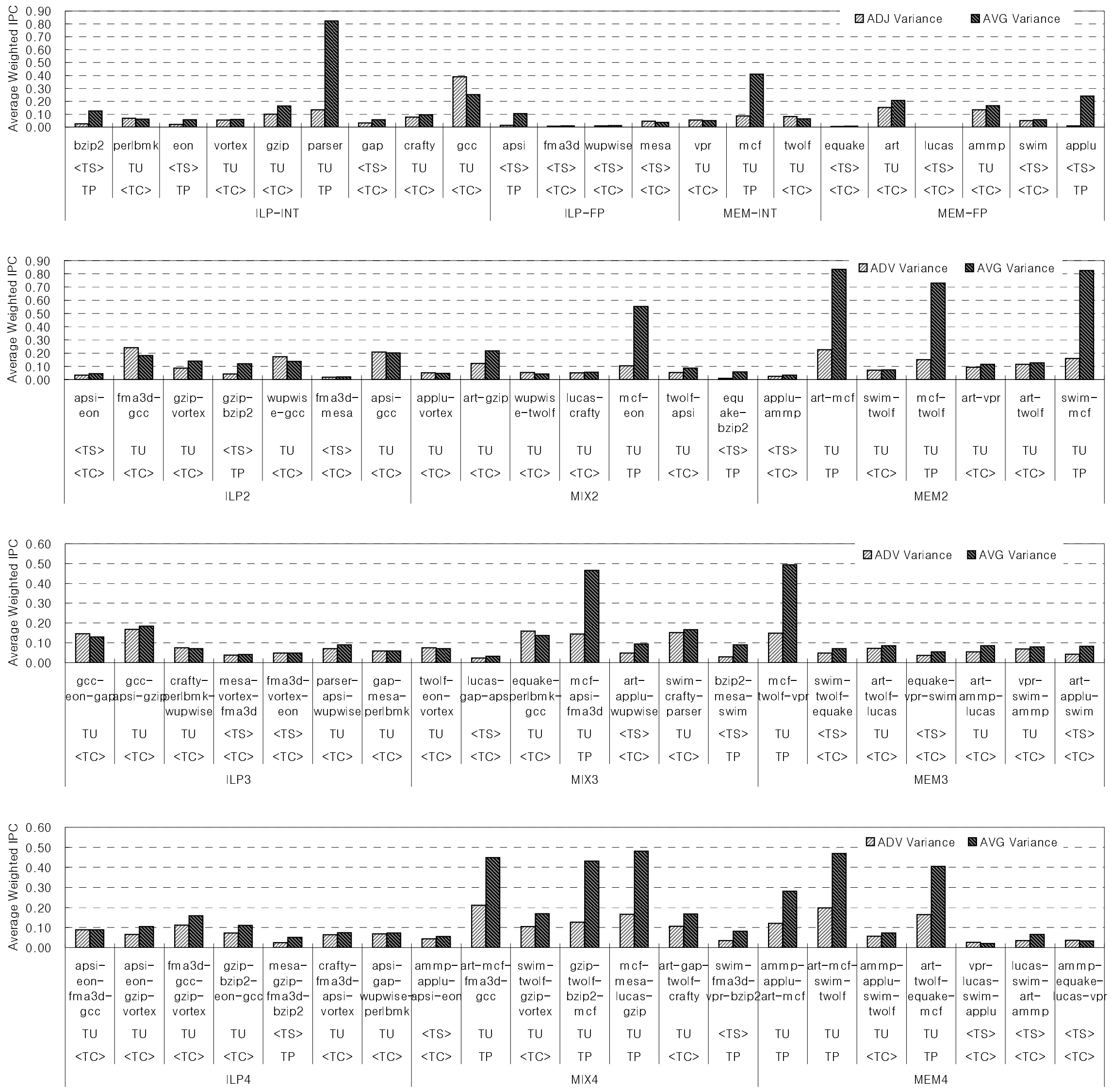


Figure 4.11: ADJ and AVG variance (metrics illustrated in Figure 4.7 and 4.8) of our 63 workloads. High ADJ variance means high frequency performance variation over time. Based on the condition in Table 4.5, ADJ variance metric classifies workloads into temporally stable (TS) and temporally unstable (TU). High AVG variance means either high or(and) low frequent performance variation over the execution time. Based on the condition in Table 4.5, AVG variance metric classifies workloads into temporally consistent (TS) and temporally phased (TP).

Chapter 5

Hill-Climbing SMT Processor Resource Distribution Algorithm

In this chapter, we realize the *hill-climbing algorithm* that searches for the optimal resource distribution on-line. The insights provided by the OFF-LINE-Analysis in Chapter 4 guides the design of the hill-climbing algorithm.

5.1 Locality of Performance

The design of our hill-climbing algorithm first focuses on the workloads with “good” run-time performance curve characteristics, which we call *temporal and spatial locality of performance*. As we will show in Section 6.3, our hill-climbing algorithm achieves near optimal performance on workloads with temporal and spatial locality of performance. In addition, we discuss “bad” situations in Section 5.2 and attempt to hand them for better performance in Section 5.3.

5.1.1 Temporal Locality of Performance

Temporal locality of performance of a multi-programmed workload on an SMT processor is detected by observing long sequence of stable performance. For multi-programmed workloads that exhibit temporal locality of performance, the shape of the performance curve changes “slowly.” Therefore, simple hill climbing algorithm which assumes static hill is likely to be successful in finding the optimal resource distribution. The workloads classified as TS (Temporally-Stable) and TC (Temporally-Consistent) have good temporal locality of performance.

5.1.2 Spatial Locality of Performance

Spatial locality of performance of a multi-programmed workload on an SMT processor is detected when the performance of a resource distribution D is similar to a resource distribution close to D at a fixed point in time. For multi-programmed workloads that exhibit spatial locality of performance, the shape of the performance curve is "smooth;" therefore, the performance curve is less likely to have local maxima, allowing hill-climbing algorithm to find the optimal resource distribution quickly. The workloads classified as WH (Wide-Hill) and SH (Single-Hump) have good spatial locality of performance.

In Chapter 7, we will show that temporal and spatial locality of performance and the performance of our hill-climbing algorithm has very string correlation.

5.2 Algorithm Design Issues

We first focus on the workloads with the locality of performance characteristics in designing our hill-climbing algorithm. However, for better performance, our algorithm should be able to deal with the situation where we cannot rely on these characteristics. Before we detail our hill-climbing algorithm, we discuss the hill-climbing algorithm design issues which we should address for better performance. And then we suggest the algorithm design guidelines.

5.2.1 Lack of Temporal Locality of Performance

In this section, we discuss one approach to handle workloads that exhibit temporal instability. Temporal instability makes it difficult for our hill-climbing algorithm to find the optimal resource distribution. Unlike the OFF-LINE-Analysis, our hill-climbing implementation cannot use perfect oracle information. Instead, it must find the resource distribution using only the performance samples acquired on-line during the execution of past epochs. Since we do not have the global view of the shape of the hill, we must compute the movement direction on the hill (*i.e.*, the gradient vector) based on past movement trails. However, the quality of the information on the past movement trail depends on the temporal stability of the workload. If the workload is not temporally stable, the history information is less reliable in determining the current hill shape, making it difficult to find the optimal resource distribution.

In addition, temporal instability makes our hill-climbing algorithm to deviate from the optimal partition frequently. Hill-climbing needs *learning time* to reach the optimal partition settings. During learning, non-optimal partitions are used, sacrificing performance opportunities. For temporally unstable workloads, the optimal resource distribution changes frequently, causing our hill-climbing algorithm to perform learning frequently as well. This can result in performance loss.

To handle the temporal instability of the workloads, we developed the following guidelines for the hill-climbing algorithm design. First, one way to mitigate learning time effects is to increase epoch size since larger epochs "smooth out" the noise in workload behavior. Second, to minimize the effect of stale information, we design our hill-climbing algorithm to compute the movement direction after executing only a small fraction of the resource distribution samples. This increases learning speed. Third, to deal with the moving target, we design our hill-climbing algorithm to *continuously* chase the optimal resource distribution.

5.2.2 Lack of Spatial Locality of Performance

In addition to temporal instability, we also considered spatial instability in designing our hill-climbing algorithm. Spatial instability can potentially cause two problems. First, hill-climbing may be limited by local maxima in the resource distribution performance curve. As suggested by Figure 4.5(a), the performance curve within an epoch often contains multiple humps. Hence, it is possible for hill-climbing to reach a non-optimal hump and become trapped. If performance on separate peaks differs considerably, potential performance gains may be sacrificed. Second, narrow peaks can make it difficult for our hill-climbing algorithm to pin-point the optimal resource distribution. Since small fluctuations away from optimal resource distribution setting can cause large reduction in performance.

To handle the spatial instability of the workloads, we developed the following hill-climbing algorithm design guideline. We pick the movement step size large enough to jump over small local maxima and to accelerate the movement speed towards the peak of the hill. At the same time, to deal with narrow peaks, the movement step size is small enough to limit the deviations from the optimal resource distribution.

5.3 Algorithm Description

Like the OFF-LINE-Analysis algorithm, hill-climbing performs learning at epoch granularity, and partitions three types of resources–integer issue queue, integer rename register, and reorder buffer–proportionally as described in Section 4.2.5. Figure 5.1 presents our hill-climbing algorithm. The algorithm consists of two parts: a sampling sequence, called a "round" (lines 16-21 in Figure 5.1), and partition selection at the end of every round (lines 8-15). An array variable, called `anchor_partition`, stores the best-performing partition setting currently found.[1] During each round, the performance of several partition settings "near" `anchor_partition` are sampled to determine the local shape of the performance curve. For each sample, we slightly shift the partition away from `anchor_partition` 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`. (In Figure 5.1, we assume *Delta* specifies the number of shifted integer rename register entries; a proportional number of integer IQ entries and ROB are also shifted). In total, T samples are taken, allowing each of the T threads to take turns receiving additional resources.

At the end of a round, the best-performing partition among the T samples is identified (line 9). This best partition setting lies along the direction of the *positive gradient* (*i.e.*, maximal performance increase) from the `anchor_partition`. Our algorithm moves in this positive gradient direction by setting `anchor_partition` to the best-performing partition found (lines 10-14). Then, the process repeats as another round begins to determine the positive gradient direction for the new `anchor_partition`.

Figure 5.2 illustrates an example of the hill-climbing algorithm on a hypothetical performance curve. From the initial anchor_partition (1), we evaluate the trial_partition **a** for an epoch (2), and record its performance outcome (3). And then, we try another trial_partition **b** for an epoch (4), and record its performance outcome (5). Since performance outcome of the trial_partition **b** is better than that of trial_partition **a**, anchor_partition is moved to trial_partition **b** (6). We repeat another round for the trial_partition **c** and trial_partition **d** (7-10), and move the anchor_partition to trial_partition **d** because the performance of trial_partition **d** is better than that of trial_partition **c** (11).

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

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

For every Epoch_Size cycles {
  perf[epoch_id % N] = eval_perf(epoch_id);    // evaluate the performance of the previous epoch
                                                                                              (a)
  if (epoch_id % N == (N – 1)) {               // move the anchor_partition every N-th epochs
    gradient_thread = max(perf, N);            // find the best performing trial_partition for the past N epochs
    for (i = 0 ; i < N ; i++)
      if (i == gradient_thread)                // move the anchor_partition in favor of gradient_thread
        anchor_partition[i] += Delta * (N – 1);
      else
        anchor_partition[i] -= Delta;
  }

  epoch_id++;                                                                                 (b)
  for (i = 0 ; i < N ; i++)
    if (i == epoch_id % N)                     // try giving favor to thread (epoch_id % N)
      trial_partition[i] += anchor_partition[i] + Delta * (N – 1);
    else
      trial_partition[i] -= anchor_partition[i] – Delta;
}
```

Figure 5.1: Hill-climbing algorithm pseudo-code. Shaded box (a) chooses a new partition based on samples acquired by shaded box (b) among all possible directions from the currently best partition, anchor_partition.

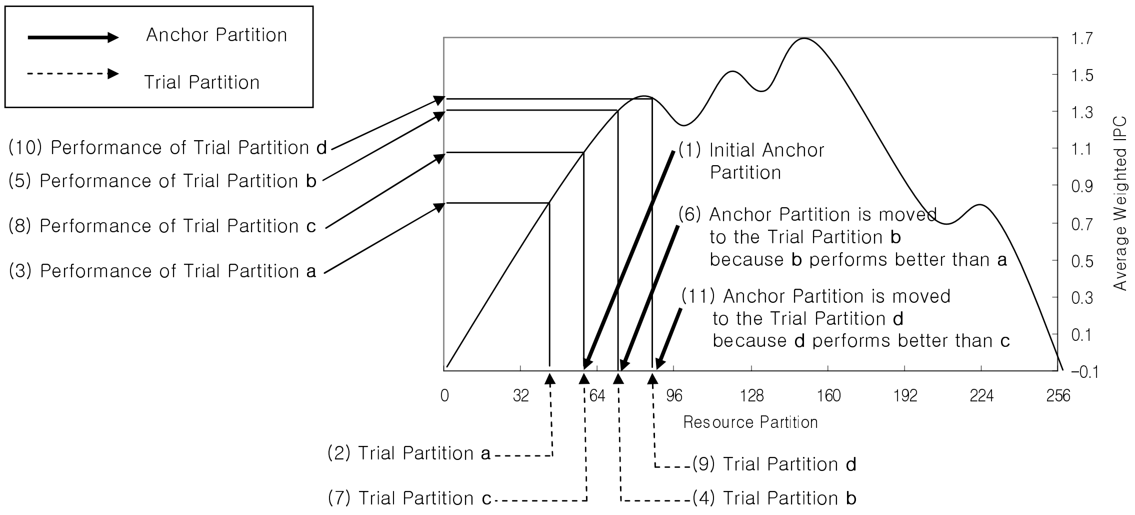


Figure 5.2: Hill-climbing algorithm working example. The position of the anchor_partition is determined by the performance outcome of the earlier trial_partitions.

Having presented the basic hill-climbing algorithm, we now discuss several algorithm design issues raised in Section 5.2 in greater depth.

5.3.1 Epoch Size

The choice of the epoch size affects the performance of the hill-climbing algorithm in three ways. First, the scheduling overhead of the software implementation of the algorithm decreases as we increase the epoch size because the hill-climbing algorithm performs scheduling at every epoch boundary. Second, the performance of the workload becomes more temporally stable as we increase the epoch size because with large epoch size frequent fluctuations in the resource distribution performance curve are "smoothed out" due to averaging. And third, the search speed will be increased as we decrease the epoch size because the hill-climber gets performance feedback more frequently, thus adapting to the changes more quickly. In Section 8.1.2, we show the sensitivity of the epoch size to the performance of the hill-climbing algorithm. Based on this sensitivity study, we pick a 64K cycle epoch size. An alternative design choice is an adaptive epoch size, which finds the optimal epoch size at run-time. But we did not experiment with this approach due to its increased complexity.

5.3.2 Movement Direction

Our hill-climbing algorithm samples performance along all possible directions from the current best partition setting (*i.e.*, T directions around the anchor_partition) before moving the anchor_partition. This approach provides complete information for making the movement decision, but comes at the expense of sampling many directions, especially when a large number of threads run simultaneously. We experimented with more expedient approaches that sample less comprehensively in the hope of reducing learning time. One possible approach is to greedily move the anchor_partition in the first sampled direction that provides a performance gain compared to that of the previous anchor_partition; another approach is to sample and move in without changing the direction as long as it continues to provide a performance gain. While these approaches acquire fewer samples per partition visited, our experiments show they result in longer overall learning times due to increased search path length. We find the sampling approach in Figure 5.1 provides the best performance.

5.3.3 Continuous Search for the Peak

Our hill-climbing algorithm runs continuously even after the hill-climbing algorithm finds the optimal resource distribution and the workload becomes temporally stable because the behavior of the workload may suddenly change at any time. As a result, our hill-climbing algorithm changes the resource distribution continuously around the anchor_partition to sense the behavior changes.

5.3.4 Step Size

Once a movement direction is chosen, our hill-climbing algorithm moves the current partition setting by *Delta* units (measured in integer rename register entries). Clearly, *Delta* affects learning time since it controls the speed at which the algorithm moves towards the optimal partition. In addition, *Delta* controls the effect of the local maxima. By choosing a large *Delta*, some local maxima–those whose peaks are narrower than *Delta*– are skipped, saving the algorithm from becoming trapped. While a large *Delta* can address both learning time and local maxima, unfortunately, a large *Delta* causes overshoot past the optimal partition setting, which is a serious problem when the hill width is narrow. We choose $Delta = 4$ (line 3 in Figure 5.1) to balance these conflicting factors. Note, we experimented with adapting *Delta* dynamically, but found a carefully chosen fixed *Delta* provides higher performance due to the difficulty of managing adaptation.

5.4 Implementation Cost

Our hill-climbing algorithm can be implemented either in software or in hardware. Software implementation needs less hardware support and hardware implementation has less run-time overhead. In this section, we estimate the cost of both implementations.

5.4.1 Software Implementation

The software implementation needs minimal hardware support. The dashed boxes with normal face labels in Figure 4.3 show the additional hardware on top of the baseline SMT processor needed for the software implementation of the hill-climbing algorithm. (The dashed boxes with bold face labels are required only for the hardware implementation.) First, our technique requires hardware statistics counters to track both the number of committed instructions per thread (these counters are available in most SMT processors already), labeled "Committed Instruction Counters" in Figure 4.3, as well as the number of shared resources–integer IQ entries, integer rename registers, and ROB entries–occupied by each thread, labeled "Resource Occupancy Counters." These statistics counters are updated every cycle by the processor pipeline. Second, our technique also requires a set of resource partition registers, labeled "Resource Partition Registers," that specify the size of each thread's partition in each of the three partitioned shared resources. These partition registers implement the trial_partition variable in Figure 5.1, and are updated every epoch by the hill-climbing algorithm. Third, our technique requires fetch stalling logic, labeled as a circled "<" sign in Figure 4.3, that compares the resource occupancy counters against the resource partition registers, and fetch-locks any thread that reaches its partition limit in one or more of the partitioned shared resources. Fourth, our technique needs a count-down timer to trigger the hill-climbing algorithm, labeled "Down Counter." This is similar to a conventional timer-interrupt used by operating systems, but we need a devoted counter just for our technique. Fifth, the software implementation of our technique needs special instructions to control the fetch stalling logic.

The resource scheduling procedure of the software implemented hill-climber is as follows. Every 64K cycles, the down counter triggers the interrupt to a randomly chosen victim thread, causing (1) context switch from the victim thread to the scheduling thread. The scheduling thread (2) reads the committed instruction counters, (3) computes the performance of the previous epoch, and (4) updates the resource partition registers. And then, (5) switches back to the victim thread. The process from step (2) to (4) takes only 58 machine instructions in Alphas binary requiring 26 cycles on the simulator with the processor model of Section 4.2.5.

In Section 8.1.2, we evaluate the overhead of the software implementation of the hill-climbing algorithm and show that the software overhead is only 0.3% of total execution time.

5.4.2 Hardware Implementation

The dashed boxes with bold face labels, as well as those with normal face labels, in Figure 4.3 show the additional devices required for the hardware implementation of our hill-climbing algorithm. Hardwire implementation requires two more hardware components on top of what is required for the software implementation. First, it requires local variables to implement the anchor_partition and perf variable from Figure 5.1. Second, the hardware implementation requires control logic to implement the scheduler, labeled "Resource Scheduler." The special instructions required for the software implementation are not necessary for the hardware implementation. Among the control logic, the most costly part is the performance evaluation function (line 4 in Figure 5.1) which implements one of the performance metrics from Section 4.2.3 (Equations 4.2-4.4).[2] However, because the hill-climbing algorithm is invoked infrequently–only once per epoch–the hardware need not be fast, potentially simplifying its design.

5.4.3 Single-Threaded IPC Computation Overhead

Of the 3 performance metrics discussed in Section 4.2.3, average weighted IPC and harmonic mean of weighted IPC (Equations 4.3 and 4.4) require the stand-alone IPC of each thread, $SingleIPC_i$. Because the $SingleIPC_i$ values are not known a priori, the hill-climbing algorithm must learn them along with the best partition. 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 its performance impact, we acquire a sample every 40 epochs only; hence, each thread's $SingleIPC_i$ is sampled once every $40 * T$ epochs. The required hardware to implement the stand-alone IPC computation is the fetch stalling logic. We will show the effect of run-time stand-alone IPC computation on SMT processor performance in Section 8.1.1.

[2]For the harmonic mean of the weighted IPC metric, the hardware can be simplified by modifying the hill-climbing algorithm to minimize the inverse of the metric rather than maximizing it.

Chapter 6

Performance Limit of SMT Processor

The performance limit of SMT processor will show the performance goal that any SMT processor may potentially achieve. In addition, the performance gap between the performance limit and the real implementation will reveal the source of the performance bottleneck, thus suggesting a way to improve the performance.

To our knowledge, no one has successfully shown the performance limit of SMT processor. One contribution of this dissertation is the development of the methodology that *approximates* the SMT processor performance limit. In this chapter, we list the simplifying assumptions that enables our limit study, validate our limit study methodology, and show the performance limit of SMT processor suggested by our approximation.

Note, the limit suggested by our approximation it less than the ideal performance limit of the SMT processor. Several simplifying assumptions make our limit study computationally feasible at the cost of its bounded performance result. However, our approximation is still useful in providing in-depth understanding of the existing techniques as we will show in Section 6.3 and 6.4.

6.1 Limit Study Methodology

In this section, we first discuss the problems associated with evaluating the performance limit of the SMT processor. Then, we present our simplifying assumptions that enable the limit study. Finally, we detail our limit study methodology.

		2	3	4
# threads		2	3	4
space complexity		$O(N)$	$O(N^2)$	$O(N^3)$
space volume		256	32,896	2,862,206
OFF-LINE-Search	purpose	Finding the performance limit of SMT processor		
	# samples / epoch	128	128	128
	sampling rate	50.000%	0.389%	0.004%
	sampling method	Multiple applications of the hill-climbing algorithm		
	sampling frequency	every epoch	every epoch	every epoch
OFF-LINE-Greedy	purpose	Validating the performance of OFF-LINE-Search		
	# samples / epoch	256	2,016	5,456
	sampling rate	100.000%	6.128%	0.191%
	sampling method	uniform, step:1	uniform, step:4	uniform, step:8
	sampling frequency	every 64 epoch	every 64 epoch	every 64 epoch

Table 6.1: OFF-LINE-Search and OFF-LINE-Greedy simulation settings.

6.1.1 Issues on SMT Processor Performance Limit Study

Finding the optimal resource distribution is NP-hard even with the epoch granularity resource distribution. The resource distribution of an epoch is not independent between that of other epochs because the configuration chosen in one epoch will affect the stream of instructions of every thread in subsequent epochs, thus affecting the optimum configuration for those subsequent epochs. Therefore, the number of all possible combinations of resource distribution is $O(S^N)$, where S is the resource distribution space volume and N is the total number of epochs during the execution of the workload. As presented in Table 6.1, S is $O(E^{T-1})$, where T is the number of concurrently running threads and E is the number of entries in the resource type that we partition. Even after we reduce S using the down sampling technique we presented in Section 4.2.1, $O(S^N)$ still implies that the problem is NP-hard. Therefore, we cannot find the optimal epoch-granularity resource distribution unless we try all $O(S^N)$ combinations, which is extremely large.

6.1.2 The Heuristics: OFF-LINE-Greedy and OFF-LINE-Search

We developed two polynomial time heuristics that approximate the optimal epoch-granularity resource distribution: *OFF-LINE-Greedy* and *OFF-LINE-Search*.

OFF-LINE-Greedy

OFF-LINE-Greedy is an approximation of the optimal epoch-granularity SMT processor resource distribution. It finds the sequence of resource distribution with the assumption that *the best performing resource distribution for the current epoch leads to finding the best resource distribution for the future epochs.* This assumption is equivalent to the common belief that *doing one's best today is the best for one's future, too.* With this assumption, we can safely find the optimal resource distribution locally within each epoch, and repeat this sequentially from the first to the last epoch. Therefore, the computation complexity of the OFF-LINE-Greedy is reduced to $O(S \times N) = O(E^{T-1} \times N)$. However, if T is 3 or more, OFF-LINE-Greedy is still not computationally feasible. For this reason, we designed OFF-LINE-Search.

OFF-LINE-Search

To find the best performing resource distribution, OFF-LINE-Greedy tries all possible combinations of resource distributions within each epoch. Instead of exhaustively trying every possible combination, OFF-LINE-Search uses the technique from multiple restart stochastic hill-climbing (MRSH) [24] to performs hill-climbing multiple times *off-line.* Each hill-climbing pass executes the algorithm in Figure 5.1 starting from the optimal resource distribution of the previous epoch and continues over the same epoch until it finds a peak. When a peak is found, we start a new hill-climbing pass from a randomly chosen `anchor_partition`. By performing multiple hill-climbing passes initiated from random points in the resource distribution space, OFF-LINE-Search can find good partitioning solutions even when local maxima exist. After trying L unique resource distributions, OFF-LINE-Search stops searching and picks the best performing resource distributions among L trials. The computational complexity of OFF-LINE-Search is further reduced to $O(L \times N) = O(N)$, as L is constant.

6.1.3 Implementation

We use probe-based simulation methodology (as presented in Section 4.2.3) to implement both OFF-LINE-Greedy and OFF-LINE-Search simulator. The implementation follows the algorithm shown in Figure 4.2. At the end of an epoch, the parent process forks child processes to probe the resource distributions defined by either OFF-LINE-Greedy or OFF-LINE-Search. After all the probings, the parent process picks the best performing resource distribution among all the probings and uses it as the parent process's resource distribution for the next epoch. Therefore, the parent process can always make the best choice and produces the performance close to the optimal since it knows the consequences of the alternative resource distributions for the next epoch. We only consider weighted IPC for this limit study; the same insights apply under other performance metrics as well.

Table 6.1 shows our OFF-LINE-Search and OFF-LINE-Greedy simulation settings. OFF-LINE-Search probes the next epochs up to 128 times. OFF-LINE-Greedy probes the next epochs 256, 2016, and 5456 times for 2-, 3-, and 4-threaded workloads uniformly across the entire resource distribution space. The large number of samples in OFF-LINE-Greedy is intended to make it close to exhaustively trying the entire resource distribution space. (But still the sampling rate is very low because of the huge search space volume.)

We ran 100M instructions counted only by the parent process. Even with this small simulation window, OFF-LINE-Greedy needs excessive simulation time. So, we run OFF-LINE-Greedy only once every 64 epochs.

Note, OFF-LINE-Analysis presented in Chapter 4 uses the same methodology as that used in OFF-LINE-Greedy as both of them samples the resource distribution uniformly. OFF-LINE-Analysis is intended to show the characteristics of the workloads on SMT processor. So, we do not report end performance of OFF-LINE-Analysis. Instead, we run OFF-LINE-Analysis every epoch to show the time-varying behavior of the workload. On the contrary, OFF-LINE-Greedy is intended to show its performance and approximate the performance limit. So, we increase the sampling rate of OFF-LINE-Greedy to make it close to exhaustively trying the entire resource distribution space. (Actually, we exhaustively try the entire resource distribution space for 2-threaded workloads.) To limit the simulation time with increased sampling rate, we ran OFF-LINE-Greedy once every 64 epochs.

6.2 Quality of the Limit Study Heuristics

At best, OFF-LINE-Greedy and OFF-LINE-Search provides optimal resource distribution of the SMT processor with three constraints; first, per-thread *resource partition* is maintained to distribute resources, second, updating resource partition is allowed only at every *epoch boundary*, and third oracle provides only the *next epoch information*. The three constraints reduce the degree of freedom of our limit study thus potentially making our limit study to suggest less meaningful performance goal. But, due to the bounded computation time, we were able to figure only limited amount of oracle information resulting in three constraints in our limit study.

Therefore, we need to validate the quality of our OFF-LINE-Greedy and OFF-LINE-Search implementation to make our approximations useful. However, the quality of our approximations cannot be validated by directly comparing against the ideal optimal resource distribution because the ideal optimal is computationally infeasible, thus unknown. So, we validate the quality of our two heuristics indirectly. Because of the above three constraints, the performance of both OFF-LINE-Greedy and OFF-LINE-Search is "lower" than the ideal optimal. We plan to show that the performance of two of our heuristics is "higher" than almost all the existing techniques. So, the bottom line is that we can show our two heuristics are good upper bound of the performance of the existing techniques.

6.2.1 OFF-LINE-Search vs. Existing Techniques

We compare the performance of OFF-LINE-Search and existing SMT processor resource distribution techniques in two ways: end-to-end performance comparison and side-by-side performance comparison. First, for the end-to-end performance comparison, we execute all techniques, including ICOUNT, FLUSH, DCRA, and OFF-LINE-Search for 100M instructions and compare their performance outcome. Out of 63 workloads, OFF-LINE-Search outperforms DCRA for 59 workloads, and ICOUNT and FLUSH for all workloads.

Second, for the side-by-side performance comparison, we "synchronized" the execution of all the techniques using the probe-based simulation methodology. At the end of each epoch, we sequentially fork three child processes, which control the shared resources using one of ICOUNT, FLUSH, or DCRA techniques. After that, we fork child processes to perform OFF-LINE-Search simulation and find the best performing resource distribution configuration. For each epoch, the performance of the OFF-LINE-Search is compared against the three existing techniques. This shows a time-varying performance profile for each technique, as illustrated in Figure 6.1. Comparing the performance from the same epoch in Figure 6.1 is meaningful because all the techniques are synchronized to a common execution point. (We also verified that synchronization does not noticeably alter the end-to-end performance of ICOUNT, FLUSH, and DCRA.) For all 63 workloads, OFF-LINE-Search outperforms ICOUNT and FLUSH in 100% of the epochs. OFF-LINE-Search also outperforms DCRA in 97.2% of the epochs averaged across all the workloads. This result suggests that OFF-LINE-Search is at least a good upper bound. Note, this is the only way of validating our heuristics considering that there is no way we can show that a heuristic is close to an *unknown* ideal performance limit.

6.2.2 OFF-LINE-Greedy vs. OFF-LINE-Search

Since we increased the number of samples of OFF-LINE-Greedy to make it close to trying the entire resource distribution space, running OFF-LINE-Greedy for our simulation window becomes computationally infeasible. Therefore, we cannot get the end performance of OFF-LINE-Greedy–we run OFF-LINE-Greedy only once every 64 epochs. Instead, we use realizable heuristic, OFF-LINE-Search. In order to validate the use of OFF-LINE-Search, we compare its performance against OFF-LINE-Greedy only during the epochs when OFF-LINE-Greedy runs.[1]

[1]There is a chance that OFF-LINE-Greedy's performance is inferior to that of OFF-LINE-Search because OFF-LINE-Greedy's uniform sampling may not pin point the optimal resource distribution, thus making this comparison less useful. But still this comparison is statistically meaningful because OFF-LINE-Greedy's performance comes from the unbiased samples, while there is a chance OFF-LINE-Search's performance may be affected by the shape of the performance curve. Here is another indirect but significant argument. Let a set S be an unbiased random sample from a set M. Let $max(X)$ be the maximum value among the elements in a set X. Let $|X|$ be the cardinality of a set X. Let $E(k)$ be the expected value of a

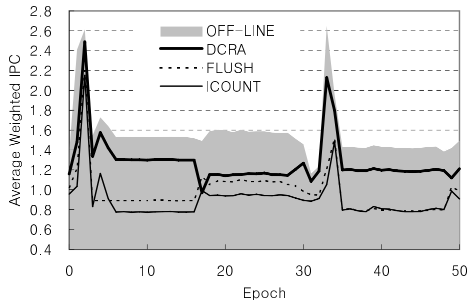


Figure 6.1: Synchronized time-varying performance of OFF-LINE-Search, DCRA, FLUSH, and ICOUNT from the art-mcf workload.

# 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

Table 6.2: Performance of OFF-LINE-Search compared to OFF-LINE-Greedy. (Performance of OFF-LINE-Search) / (Performance of OFF-LINE-Greedy) is computed across all 63 workloads.

As shown in Table 6.2, during the 522 sampled epochs, OFF-LINE-Search's performance is 0.01% worse than that of OFF-LINE-Greedy and the the standard deviation of the performance difference is 0.02%. This means that the performance of OFF-LINE-Search and OFF-LINE-Greedy is almost identical all the time.

It is important to note that even though OFF-LINE-Search performs resource distribution at epoch granularity, which uses a fixed resource partition during the 64K-cycle epoch (the other techniques update resource distribution decisions every cycle), it still achieves higher performance in practically every epoch.

random variable k. Then, $E(max(S)) = (1 - 1/|S|) \times E(max(M))$, if M's values follow uniform distribution. In our case, we believe that the distribution of the performance from the entire resource distribution space is not far away from the uniform distribution. (The performance curve study in Chapter 4 shows this.) So, our OFF-LINE-Greedy's performance for 3- and 4-threads will be close to $1 - 1/2016 = 0.9995$ and $1 - 1/5456 = 0.9998$, respectively, of that of the exhaustively trying (*i.e.*without down sampling) OFF-LINE-Greedy.

6.3 Results of the Limit Study

Using the limit study methodology presented in Section 6.1, we measured the performance of OFF-LINE-Search for our 63 multi-programmed workloads for 100M instructions. Figure 6.2 compares OFF-LINE-Search against ICOUNT, FLUSH, DCRA, and HILL-WIPC. HILL-WIPC is the hardware implementation of the hill-climbing algorithm described in Chapter 5, which uses weighted IPC as the performance evaluation function (*i.e.*, performance feedback). This figure plots the weighted IPC, normalized against OFF-LINE-Search, versus different resource distribution techniques applied to the 2-, 3-, and 4-threaded workloads. On the bottom of each graph, we added the classification of the workload from the OFF-LINE-Analysis presented in Chapter 4 (*i.e.* SH/MH, WH/NH, TC/TP, and TS/TU labels) to correlate the run-time workload characteristics and the performance.

Figure 6.3 plots the same performance data as Figure 6.2, but categorizes the workloads by the classifications. X-axis shows the classification and the number of workloads that belongs to the class. OFF-LINE-Search outperforms ICOUNT by 16.5%, FLUSH by 17.2%, DCRA by 7.4% and HILL-WIPC by 4.4%. This implies that the epoch-granularity resource distribution has the potential to consistently make higher quality resource distribution decisions compared to existing techniques. In addition, the performance gap between the limit suggested by OFF-LINE-Search and the real techniques shows the performance opportunities that the real techniques can potentially achieve.

Below, we investigate the performance opportunities of the prior resource distribution techniques as well as our hill-climbing resource distribution by analyzing the Figure 6.2 and Figure 6.3. In addition, we identify the bottleneck of each technique.

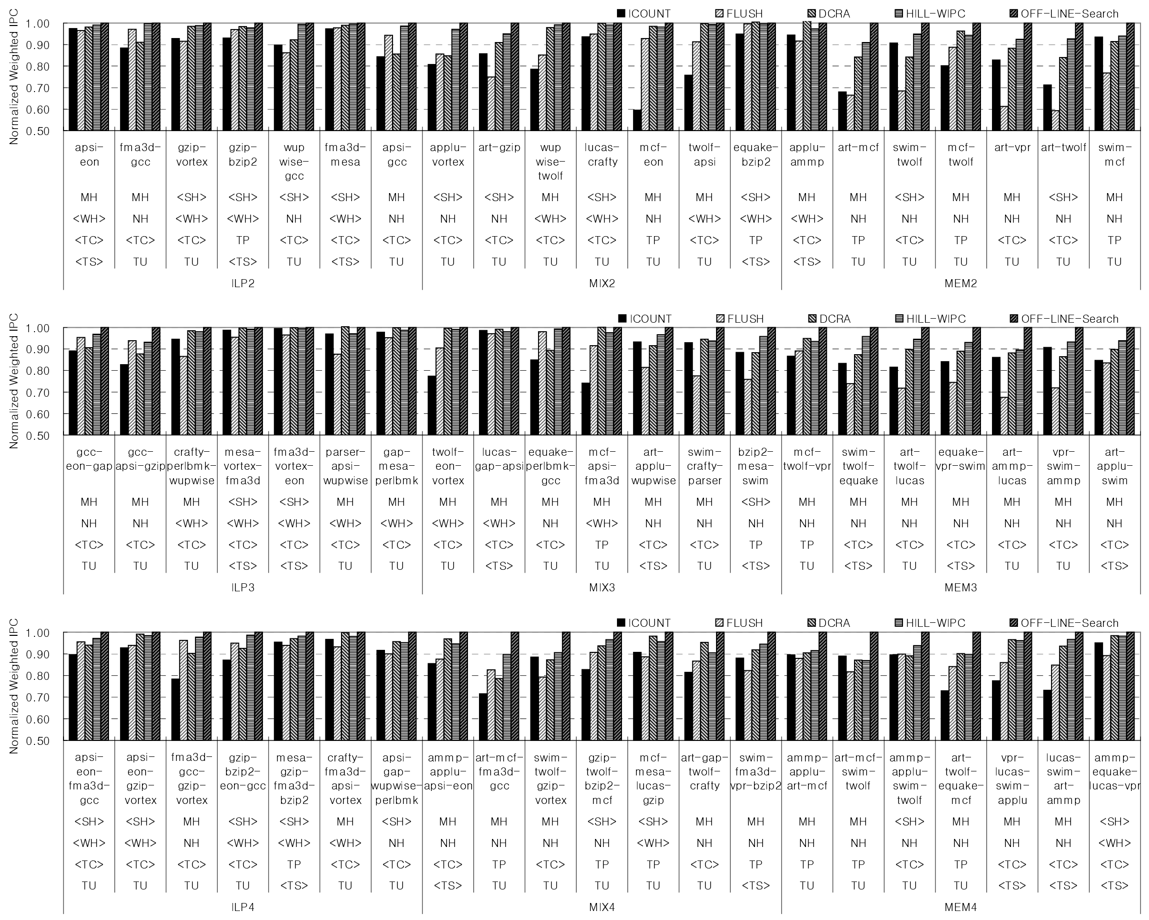


Figure 6.2: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC normalized against OFF-LINE-Search. The labels (SH/MH, WH/NH, TC/TP, and TS/TU) from the OFF-LINE-Analysis are added to each workload to present the correlation between SMT performance and workload characteristics.

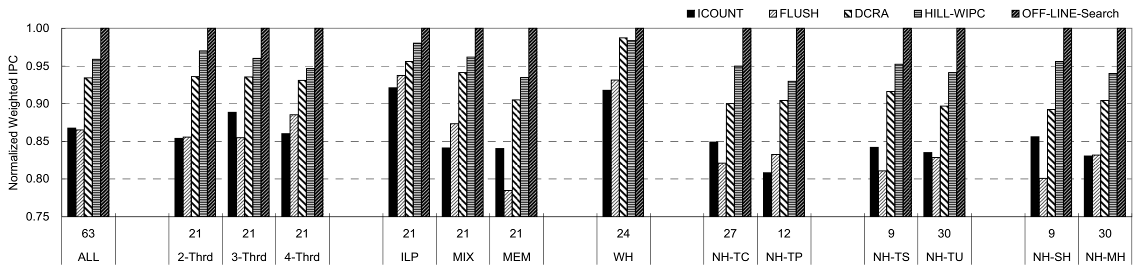


Figure 6.3: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC normalized against OFF-LINE-Search by the workload type and characteristics. The first X-axis label indicates the number of workloads that belongs to the workload type/characteristics specified in the second X-axis label.

6.3.1 Performance Opportunities of ICOUNT and FLUSH

As shown in "ALL" bars in Figure 6.3, the average performance of ICOUNT and FLUSH is as good as 86.8% and 86.5% of OFF-LINE-Search, respectively. Except for the ILP and WH workloads, both ICOUNT and FLUSH cannot fully exploit the potential performance of SMT processor. The following two observations can explain the performance opportunities lost by ICOUNT and FLUSH techniques.

First, both ICOUNT and FLUSH have difficulty in dealing with long latency memory operations. ICOUNT achieves 84.1% and 84.0% of OFF-LINE-Search for MIX and MEM workloads, respectively, as shown in "MIX" and "MEM" bars in Figure 6.3. FLUSH achieves 78.5% of OFF-LINE-Search on MEM workloads. ICOUNT slows down fetching of an application with large pre-decoded instruction count to give advantage to an application which uses the resource more efficiently. But, with ICOUNT, an application may still *hold* shared resources during the L2 cache miss and just *wait* for the resolution of the cache miss, decreasing the resource utilization. Because of this hold-and-wait condition, ICOUNT cannot achieve good performance for MIX workloads (84.1% of OFF-LINE-Search), since the MEM application may prevent the progress of the ILP applications during the time MEM application waits for the L2 miss resolution. FLUSH has difficulty in exploiting memory parallelism because after an L2 cache miss, all the instructions next to the load instruction are flushed. Therefore, any subsequent load instructions in the instruction queue, which can potentially overlap their cache miss with the current outstanding cache miss, are also flushed, losing the opportunity to exploit the memory parallelism. For example, the benchmark "art" is one of the applications with significant memory level parallelism. So, FLUSH achieves only 74.9%, 66.5%, 61.3%, and 59.3% of OFF-LINE-Search for the art-gzip, art-mcf, art-vpr, and art-twolf workloads, respectively, as shown in Figure 6.2 because FLUSH cannot fully exploit memory parallelism of art.

Second, both ICOUNT and FLUSH achieve good performance only for WH workloads showing 91.8% and 93.1% of OFF-LINE-Search performance because WH tolerates less accurate resource distributions. Unfortunately, both techniques are unable to distribute the resources properly for NH workloads.

6.3.2 Performance Opportunities of DCRA

DCRA achieves fairly good performance. (93.4% of OFF-LINE-Search on average as shown in "ALL" bars in Figure 6.3) Especially for WH workloads, the performance of DCRA is almost as good as OFF-LINE-Search (98.7% of OFF-LINE-Search) since WH workloads tolerate less accurate resource distribution. On NH workloads, however, DCRA misses two performance opportunities: memory level and instruction level parallelism.

Memory Level Parallelism

DCRA achieves significantly better performance compared to ICOUNT and FLUSH for MIX and MEM workload by giving more resources to memory-intensive applications. As a result, memory-intensive applications are allowed to overlap the independent long latency memory operations thus achieving memory parallelism. But, there is still room for performance improvement by exploiting more memory level parallelism.

For example, the benchmark "art" iterates over the independent array elements[2] and "mcf" iterates over the pointer chains[3]. Both applications cause one L2 cache miss per iteration making them memory intensive. However, mcf's memory operations should be serialized due to the dependencies in the pointer chain. In contrast, art has abundant

[2]The inner most loop in scan_recognize() is an example.

[3]The loop in refresh_potential() is an example.

memory parallelism. For the art-mcf workload, DCRA tends to give the same amount of resources to both art and mcf because both applications are memory intensive. However, giving more resources to art improves the performance by overlapping more L2 misses. OFF-LINE-Search “learns” this behavior and provides high performance for art-mcf. As a result, DCRA achieves only 84.3% of OFF-LINE-Search performance on the art-mcf workload.

Instruction Level Parallelism

The resource distribution behavior of DCRA converges to ICOUNT if there is no outstanding cache miss. Therefore, DCRA’s performance of ILP workloads is similar to that of ICOUNT. (On ILP workloads, DCRA and ICOUNT have 95.6% and 92.1%, respectively. On non-ILP workloads, DCRA and ICOUNT have 92.3% and 84.1%, respectively.) ICOUNT tries to keep the number of pre-decoded instruction count balanced across all the simultaneously running threads. However, applications with long instruction dependence chains or those with poor branch prediction accuracy should receive small amount of instruction issue queue resources. On the contrary, applications whose independent instructions can fit only within large instruction window can utilize a large amount of instruction issue queue resources. But ICOUNT does not consider the per-application demand for the instruction issue queue resources, losing performance opportunities.

For example, in the apsi-gcc workload, if we give more fetch bandwidth to apsi than what ICOUNT normally gives, we can achieve better performance because apsi can find more independent instructions when it receives a large instruction window. As a result, ICOUNT and DCRA only achieve 84.4% and 85.7% of OFF-LINE-Search on the apsi-gcc workload, respectively.

6.3.3 Performance Opportunities of Hill-Climbing Resource Distribution

Compared to OFF-LINE-Search, HILL-WIPC performs searching for the optimal resource distribution on-line thus increasing the run-time scheduling overheads discussed in Section 5.1. However, as shown in Figure 6.3, HILL-WIPC achieves the closest performance to OFF-LINE-Search among all the techniques implying that HILL-WIPC faithfully approximates OFF-LINE-Search (95.9% of OFF-LINE-Search). In addition, HILL-WIPC handles DCRA's memory and instruction level parallelism problems properly by using the feedback information to find the optimal resource distribution.

One drawback of HILL-WIPC, however, is that its performance becomes worse as the number of simultaneously running threads increases, while all the other techniques' performance is independent of the number of threads (HILL-WIPC has 97.0%, 96.0%, and 94.7% of OFF-LINE-Search performance for 2-, 3-, and 4-threaded workloads, respectively). This is because the number of epochs to determine the next movement direction increases as the number of threads increases, thus delaying the search speed.

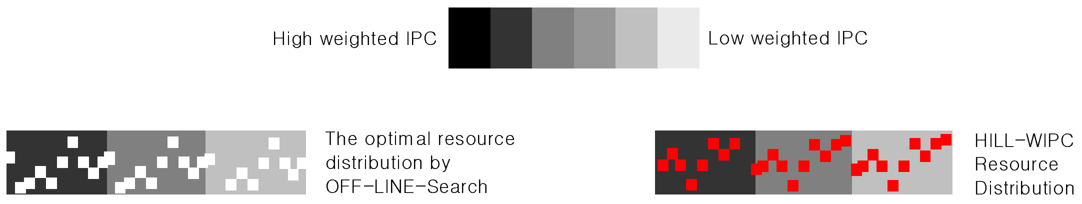
Like all the other techniques, HILL-WIPC performs well on WH workloads, achieving 98.4% of OFF-LINE-Search. Among the NH workloads, HILL-WIPC has good performance for the workloads with temporal and spatial locality, which we defined in Section 5.1. As hill climbing can enjoy the locality of performance property, HILL-WIPC has better performance for workloads labeled NH-TC, NH-TS, and NH-SH than for those labeled NH-TP, NH-TU, and NH-MH, respectively. In the next section, we will detail the conditions under which HILL-WIPC suffers performance loss.

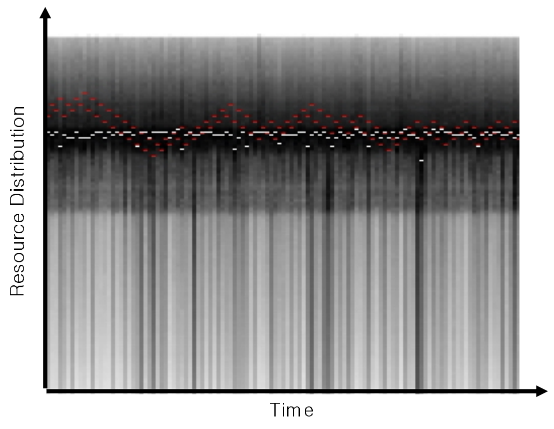
6.4 Performance Hazards of the Hill-Climbing Resource Distribution

Figure 6.4 shows how our hill-climbing algorithm searches for the optimal resource distribution at run-time on 2-threaded workloads. In the figure, the white dots indicate the optimal resource distribution that OFF-LINE-Search finds, and the red dots indicate the resource distribution that our hill-climbing algorithm finds. If the red dots and white dots are close to each other, our hill-climbing algorithm finds the optimal resource distribution. If the red dots are far away from the white dots, the hill-climbing algorithm has difficulty in finding the optimal resource distribution.

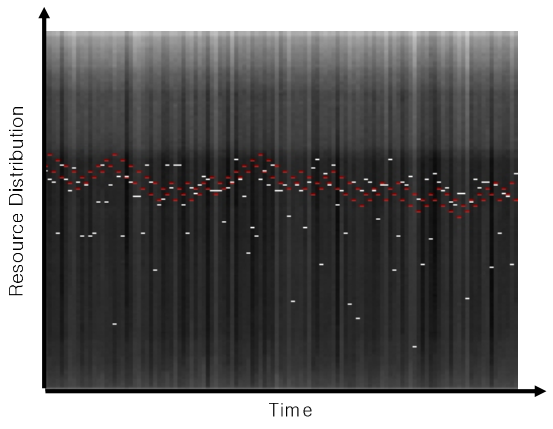
Figure 6.4(a), (b), and (c) show examples of workloads that exhibit rich spatial and temporal locality of performance. Figure 6.4(a) shows a single hump example, Figure 6.4(b) shows a wide hill example, and Figure 6.4(c) shows a temporally stable example. For these three examples, our hill-climbing algorithm successfully finds the optimal resource distributions. There are no bottlenecks that limit hill-climbing's movement; hence, hill-climbing moves along the positive gradient, and after a short time reaches the optimal partition, remaining there to enjoy the highest possible performance. (Note that the red dots and the white dots overlap in these three examples.)

On the other hand, Figure 6.4(d), (e), and (f) show examples of workloads without locality of performance. Figure 6.4(d) shows a multiple hump example, Figure 6.4(e) shows a temporally phased example, and Figure 6.4(f) shows a temporally unstable example. For these three examples, hill-climbing has difficulty in reaching the optimal resource distributions. (Note that the red dots are far away from the white dots.) We refer to the causes of our hill-climbing algorithm's performance loss *performance hazards*.

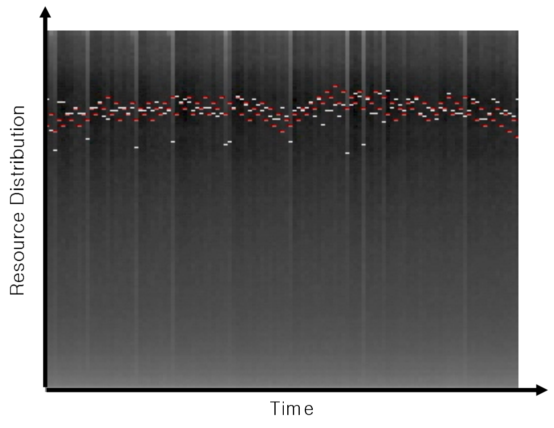

High weighted IPC
Low weighted IPC
The optimal resource distribution by OFF-LINE-Search
HILL-WIPC Resource Distribution



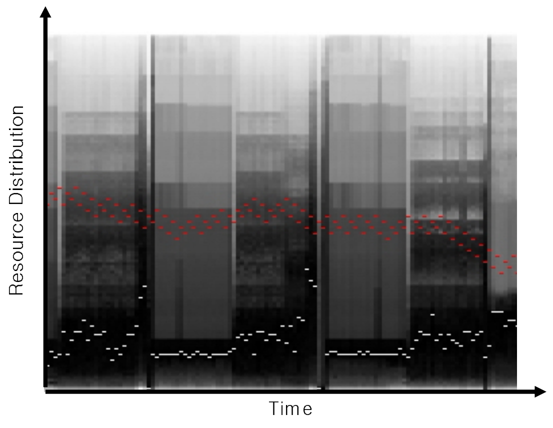
(a) Single-Hump (SH)
(applu-vortex)



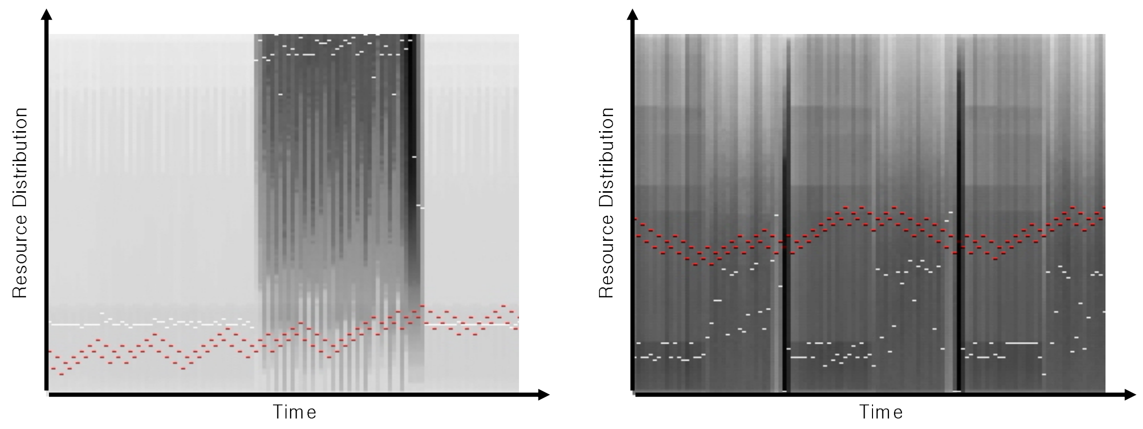
(b) Wide-Hill (WH)
(wupwise-twolf)



(c) Temporally-Stable (TS)
(equake-bzip2)



(d) Multiple-Humps (MH)
and Narrow-Hill (NH)
(art-mcf)



(e) Temporally–Phased (TP)
and Narrow–Hill (NH)
(swim–mcf)

(f) Temporally–Unstable (TU)
and Narrow–Hill (NH)
(art–twolf)

Figure 6.4: Three workload examples with rich locality of performance: (a) single hump, (b) wide hill, and (c) temporally stable, and three workload examples with performance hazards: (d) multiple humps, (e) temporally phased, and (f) temporally unstable. The white dots indicate the optimal resource distribution, and the red dots indicate the resource distribution that HILL-WIPC finds.

6.4.1 Spatial Hazards

A *Spatial hazard* is the condition that our hill-climbing algorithm suffers from the performance loss due to the lack of spatial locality of performance. Here are the specific cases.

Narrow Hill Width

As shown in Figure 6.3, the performance of HILL-WIPC is 94.4% of OFF-LINE-Search, if hill-width is narrow. (For workloads with wide hills (WH), HILL-WIPC's performance is 98.4% of OFF-LINE-Search.) On narrow hill workloads, small deviations from the optimal resource distribution suffers significant performance loss. The other performance hazards listed below have narrow hill width condition as well because workloads with wide hills allow the performance of any resource distribution to be as good as OFF-LINE-Search.

Multiple Humps

Figure 6.4(d) illustrates the situation where hill-climbing has difficulty in finding the peak of the hill due to multiple humps in the performance curve. (Note that the red dots linger around the local maxima shown as horizontal high contrast strips.) However, hill-climbing is not permanently trapped at the local maxima because the shape of the hill changes over time and so does the local maxima. But still, multiple humps slow down the searching speed, making hill-climbing to sacrifice its performance opportunities.

6.4.2 Temporal Hazards

A *Temporal hazard* is the condition that our hill-climbing algorithm suffers from the performance loss due to the lack of temporal locality of performance. Here are the specific cases.

Phased Behavior

Figure 6.4(e) shows the example of phased behavior (the sudden hill shape and optimal resource distribution changes). If the optimal resource distribution changes in phases, hill-climbing does not have enough time to track the changes, thus losing performance opportunities. The example in Figure 6.4(e) shows two behaviors, a long period of low performance followed by a short period of high performance. Hill-climbing effectively tracks the optimal partition in the low-performing period due to its long duration. When the optimal partition changes, it does not remain stable long enough for hill-climbing to adjust; hence, hill-climbing misses significant performance opportunities during the high-performing period.

Temporally Unstable Behavior

Figure 6.4(f) shows the example of temporally unstable behavior (the fine grained vertical lines). If the performance changes frequently over time, hill-climbing has difficulty in deciding the movement direction because hill-climbing algorithm uses the history information to pick the movement direction. The "jittered" performance curve makes the history information less consistent and confuses the hill-climbing algorithm in finding the optimal resource distribution. The example in Figure 6.4(f) shows that the positive gradient within each epoch always points towards the maximal peak. But, inter-epoch jitter creates transient positive gradients *between* epochs that temporarily point away from the maximal peak. These bogus gradients fool the hill-climbing algorithm, causing it to reverse course occasionally and move away from the optimal partition.

Chapter 7

Performance Evaluation of the Hill-Climbing Resource Distribution

So far, we have analyzed the performance characteristics of the workloads (in Chapter 4), designed the hill-climbing algorithm (in Chapter 5), and studied the performance limit of SMT processor (in Chapter 6). In this chapter, we evaluate the performance of our hill-climbing resource distribution implementation. We will first show the experimental methodology and the performance results. And then, we will present improved hill-climbing algorithm.

7.1 Experimental Methodology

Our evaluation of hill-climbing resource distribution uses SMT simulator described in Section 4.2.5 of Chapter 4. However, this experiment has two different simulator settings compared to what we used for the OFF-LINE-Analysis (Chapter 4) and OFF-LINE-Search (Chapter 6). First, we pick simulation windows using the methodology described in Section 4.2.5, but we extend their duration to 1 billion instructions to get results that may closely reflect the whole benchmark run. Second, instead of using probe-based simulation, we use end-to-end simulation throughout the simulation window, which executes the simulation window only once.

Note that there are two run-time overheads associated with the hill-climbing resource distribution simulation which we do not account for. First, at every epoch boundary, the software implementation of the hill-climbing resource distribution triggers an interrupt to invoke the resource scheduling thread that computes the resource distribution for the next epoch. The performance that we present in this chapter assumes hardware implementation of the hill-climbing resource distribution, thus including no run-time resource distribution overhead. In Section 8.1.2, we evaluate the run-time overhead of software implementation of the hill-climbing resource distribution in detail. Second, when calculating the performance with a metric that requires stand-alone IPC (*i.e.*, $SingleIPC_i$), the stand-alone IPC should be either computed off-line or on-line. For on-line computation, we sample the stand-alone IPC once every 40 epochs. During the sampling epoch, we stall fetching of all threads except one and measure the IPC of the live thread. The performance that we present in this chapter assumes the stand-alone IPC is computed off-line. Section 8.1.1 details the analysis of the on-line stand-alone IPC computation overhead.

7.2 Performance Results

Using the experimental methodology presented in the previous section, we conduct comprehensive experiments to show the baseline performance of our hill-climbing resource distribution.

7.2.1 Baseline Performance Results

Figure 7.1 compares hill-climbing resource distribution (labeled "HILL-WIPC") against ICOUNT, FLUSH, and DCRA on our 63 workloads. The comparison is made using the weighted IPC metric; hill-climbing also uses weighted IPC as the performance-feedback function for learning. Figure 7.2 shows the same performance data as what is shown in Figure 7.1, but categorized by the workload type and run-time characteristics. Comparing Figure 6.3, which uses 100M instruction window, and Figure 7.2, we cannot find any noticeable differences in general performance trends. This is expected because

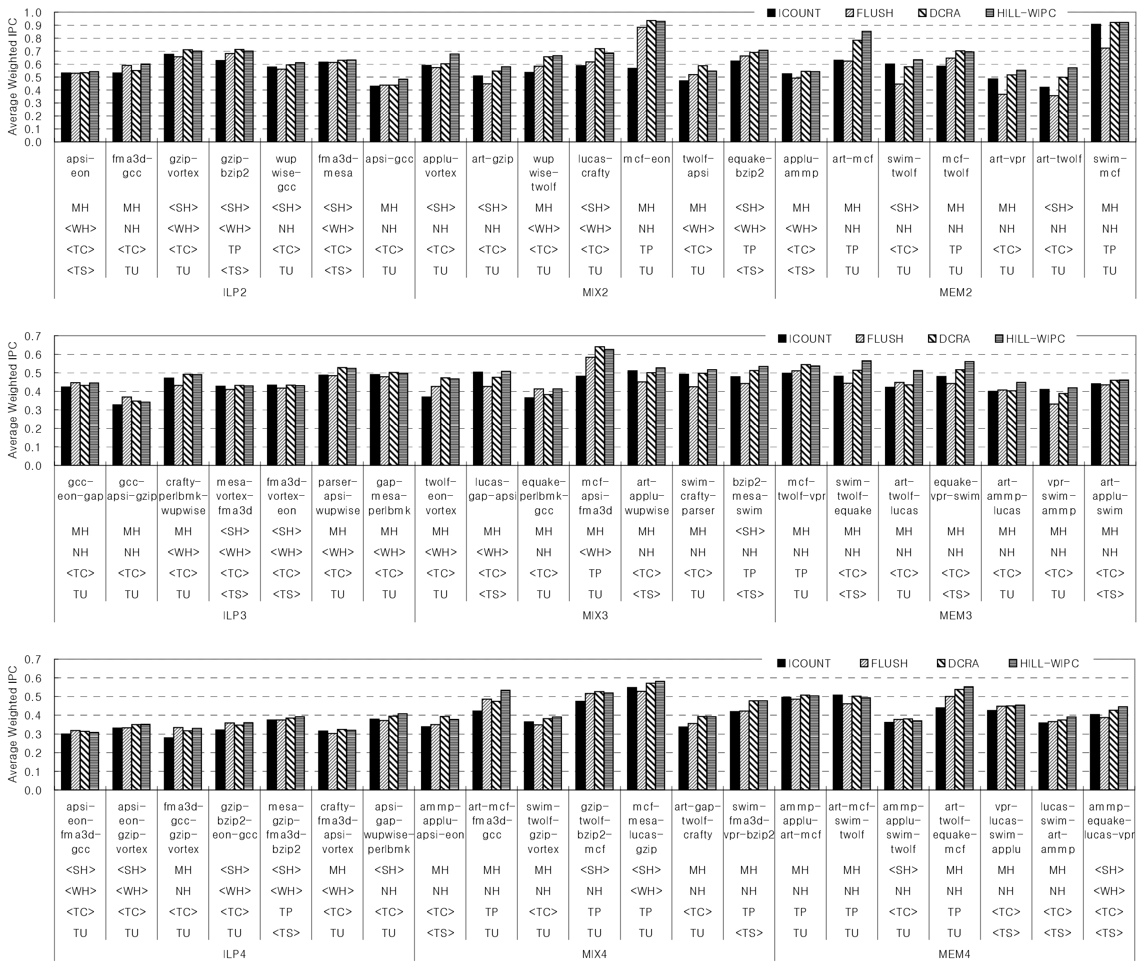


Figure 7.1: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC for all 63 workloads. Run-time characteristics and type are labeled on each workload.

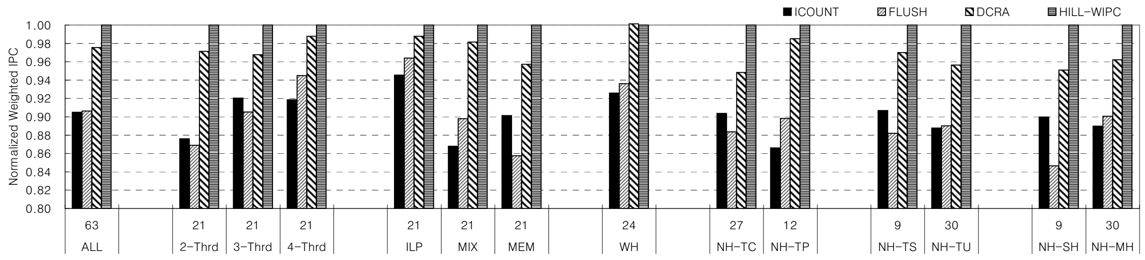


Figure 7.2: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC by the workload type and characteristics. All bars are normalized against HILL-WIPC.

we used SimPoint to identify the representative simulation window. This implies that the workload classification from the OFF-LINE-Analysis and the limit study from the OFF-LINE-Search using 100M instruction window is still useful for understanding the 1B instruction window simulation results.

Comparing HILL-WIPC, ICOUNT, and FLUSH, we see HILL-WIPC outperforms ICOUNT and FLUSH in all but 2 and 6 out of our 63 workloads, providing an average performance boost of 11.4% and 11.5%, respectively.

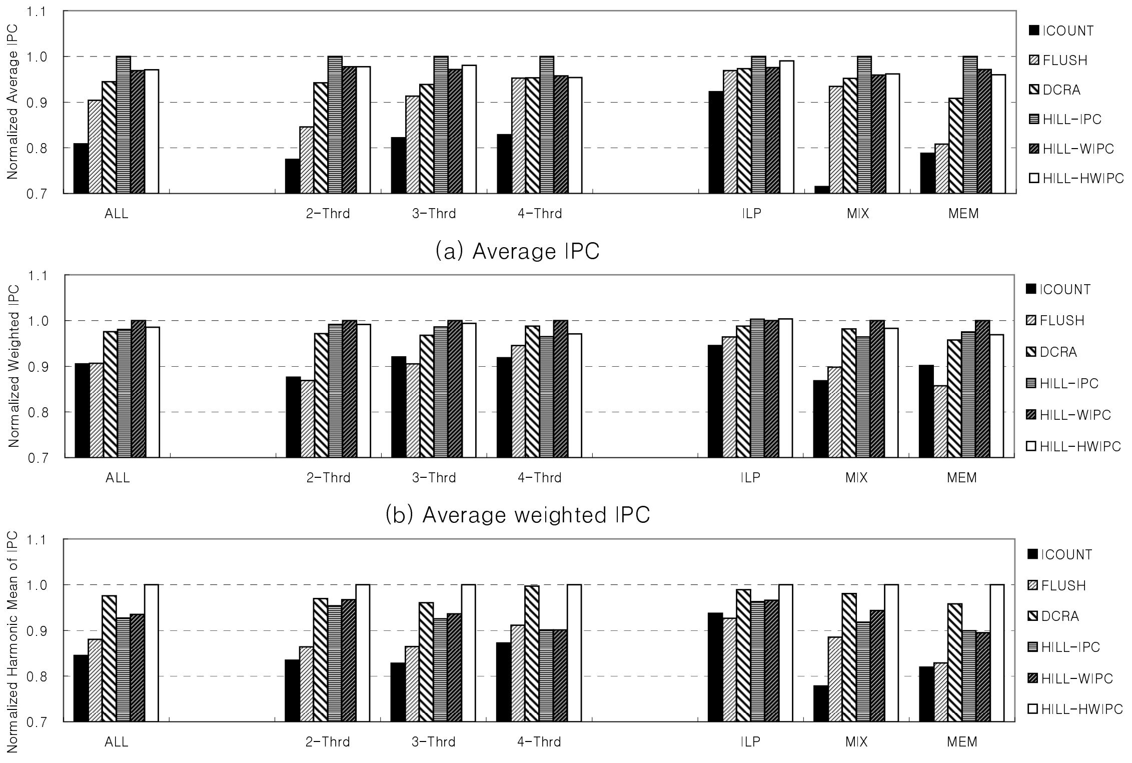
Comparing HILL-WIPC and DCRA, we see HILL-WIPC outperforms DCRA by 2.8% averaged over the 63 workloads. This overall performance gain is achieved non-uniformly across the different workload type and run-time characteristics. As shown in "2-Thrd", "3-Thrd", and "4-Thrd" bars of Figure 7.2, performance gains are larger for the 2- and 3-thread workloads (3.3% and 3.6%, respectively) compared to the 4-thread workloads (1.3%) because the number of epochs to determine the next anchor_partition (*i.e.*, the number of epochs for a round) increases as the number of running threads increases. Another observation is that the performance gain becomes larger for the MEM category (4.8%) compared to the ILP and MIX categories (1.3% and 2.1%, respectively), because of the HILL-WIPC's ability to exploit memory parallelism effectively. More importantly, HILL-WIPC outperforms or matches DCRA independent of the workload type (2-, 3-, and 4-threaded workload, and ILP, MIX, and MEM) in Figure 7.2, which we believe is a positive result given the size and diversity of our workload set.

As we expected, HILL-WIPC works well on workloads with temporal or spatial locality of performance property. For WH (Wide-Hill) workloads, the performance of DCRA is slightly better than that of HILL-WIPC because WH allows wide range of resource distribution to achieve good performance and cycle-by-cycle scheduling of DCRA makes their performance slightly better. However, on NH-TC and NH-SH workloads, HILL-WIPC can enjoy the locality of performance property achieving 5.5% and 5.2% performance gain, respectively compared to DCRA. On the other hand, for NH-TP and NH-MH workloads, the performance advantage of HILL-WIPC becomes small because of the performance hazards that we described in Section 6.4. However, HILL-WIPC still outperforms DCRA by 1.5% and 3.9% for NH-TP and NH-MH workloads, respectively.

7.2.2 Adaptive Optimization Goals

Figure 7.3 compares all the techniques using different metrics for both measuring the performance and feedback-based learning. Three graphs, labeled (a)-(c), report performance in terms of (a) average IPC, (b) average weighted IPC, and (c) harmonic mean of weighted IPC. Within each graph, hill-climbing uses either average IPC (HILL-IPC), weighted IPC (HILL-WIPC), or harmonic mean of weighted IPC (HILL-HWIPC) as the performance-feedback metric for learning. The bars in (a) are normalized against HILL-IPC bar, (b) are normalized against HILL-WIPC bar, and (c) are normalized against HILL-HWIPC bar. Results are summarized by workload group to conserve space.

Comparing HILL-IPC, HILL-WIPC, and HILL-HWIPC across the graphs, we see hill-climbing achieves its best performance when using the same metric for both driving feedback-based learning and measuring the performance. Figure 7.3(a) and (c) show hill-climbing achieves a performance gain under the average IPC and harmonic mean of weighted IPC metrics in addition to the gains already demonstrated under the weighted IPC metric in Figure 7.2. Comparing HILL-IPC against ICOUNT and FLUSH in Figure 7.3(a), we see hill-climbing outperforms ICOUNT and FLUSH under average IPC in all the workload groups, providing an average performance boost of 23.7% and 10.7%,



(a) Average IPC

(b) Average weighted IPC

(c) Harmonic mean of IPC

Figure 7.3: Hill-Climbing versus ICOUNT, FLUSH, and DCRA under the (a) average IPC, (b) weighted IPC, and (c) harmonic mean of weighted IPC metrics. Hill-Climbing uses average IPC (HILL-IPC), weighted IPC (HILL-WIPC), and harmonic mean of weighted IPC (HILL-HWIPC) as the performance-feedback metric.

respectively. Comparing HILL-HWIPC against ICOUNT and FLUSH in Figure 7.3(c), we see hill-climbing outperforms ICOUNT and FLUSH under harmonic mean of weighted IPC in all the workload groups as well, providing an average performance boost of 18.3% and 13.6%, respectively. Comparing HILL-IPC and DCRA in Figure 7.3(a), we see hill-climbing outperforms DCRA by 5.9% under average IPC, and comparing HILL-HWIPC and DCRA in Figure 7.3(c), we see hill-climbing outperforms DCRA by 2.5% under harmonic mean of weighted IPC.

This demonstrates one of the strengths of hill-climbing resource distribution: the ability to *directly optimize* the performance metric most important to the user. Existing techniques cannot optimize for a particular performance goal.

7.3 Improving Hill-Climbing Resource Distribution

In Section 6.4, we studied the performance hazards of hill-climbing resource distribution that lead to performance lose in HILL-WIPC. In addition, in Section 8.2, we witnessed that the performance hazards decrease the advantage of hill-climbing resource distribution. In this section, we present two techniques to overcome the performance hazards of the hill-climbing resource distribution.

7.3.1 Phase-Based Learning

A natural approach to attack the performance hazard and make the search speed fast is to exploit existing phase detection and prediction techniques. Phase detection [42] can be used to determine which epochs are similar to each other. Instead of re-learning a resource distribution for such an epoch, we can simply reuse a previously learned resource distribution for the similar phase to save the learning time. Phase prediction [43] can

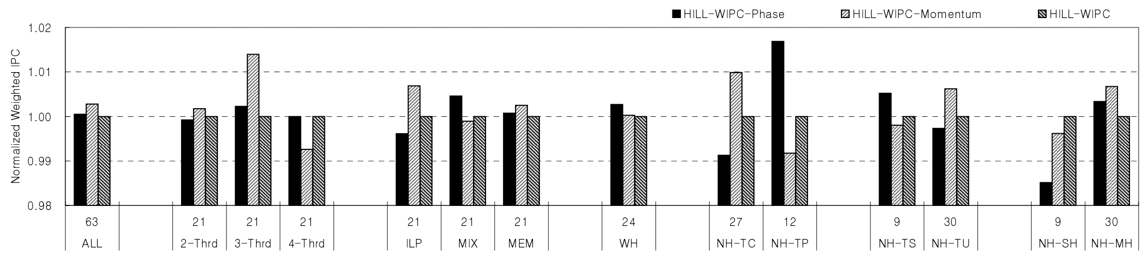


Figure 7.4: Improving the baseline hill-climbing resource distribution. Hill-WIPC-Phase trains the resource distribution per phase basis. HILL-WIPC-Momentum uses momentum term to jump over the small jitters. All bars are normalized against HILL-WIPC.

be used to predict a future phase so that we can apply a previously learned resource distribution to the next epoch.

We implemented Sherwood's Basic Block Vector (BBV) signature analysis technique [42] to perform phase detection on the epochs. We use a BBV with 64 entries per SMT thread. We also implemented Sherwood's phase prediction technique [43] to predict the phase ID of the next epoch. Our phase predictor stores 128 unique phase IDs, and uses a 2048-entry run-length encoded (RLE) Markov predictor.

Figure 7.4 shows the performance result of the phase based learning (labeled HILL-WIPC-Phase). In Figure 7.4, the X-axis shows the workload type and class, and the Y-axis shows the weighted IPC normalized against the baseline HILL-WIPC. With phase detection and prediction, we are able to boost hill-climbing performance by only 0.05% across our 63 workloads, on average. Interestingly, almost all the performance benefit comes from speeding up workloads exhibiting phased behavior (NH-TP). Considering only NH-TP workloads, we see a 1.7% performance boost. We believe this is a promising approach to improving hill-climbing, especially for dealing with the performance hazard caused by phased behavior.

7.3.2 Hill-Climbing with Momentum Term

Since the momentum term allows jumping over small local maxima, the traditional hill-climbing algorithm should have the momentum term. Otherwise, the hill-climber may be trapped at any small local maxima forever. However, our target hill is dynamic and time varying. Therefore, our hill-climber may not be trapped at the local maxima forever because the local maxima at an epoch may not be the local maxima in future epochs. Instead, local maxima may impede the speed of searching for the optimal resource distribution.

In the baseline hill-climbing algorithm in Figure 5.1, we do not include a momentum term. To implement the momentum term, we modified line 7 of the hill-climbing algorithm in Figure 5.1 in the following manner.

```
perf[epoch_id % N] = (perf[epoch_id % N] + eval_perf(epoch_id)) / 2;
```

This equation makes the performance of the current epoch a function of both the new information and previous history information. So, the choice of the gradient_thread is determined not only by the most recent performance behavior, but also by the past performance trends. To simplify the implementation of the modified line, we can use shift and add operation, rather than a division operation.

Figure 7.4 shows the performance result of the momentum term (labeled HILL-WIPC-Momentum). The Y-axis is the weighted IPC normalized against the baseline HILL-WIPC. With momentum term, the overall performance is improved by 0.28%. However, the momentum term boosts the performance of workloads with temporal jitter (NH-TU) and spatial jitter (NH-MH) by 0.62% and 0.67%, respectively. This is because the momentum term tends to keep its movement direction and this allows jumping over noisy within each epoch (NH-MH) and across adjacent epochs (NH-TU). However, the momentum term decreases the performance of workloads with phased behavior (NH-TP) by 0.83% because momentum term delays the prompt adaptation to sudden phase changes.

Chapter 8

Overhead Analysis And Sensitivity Study

So far, we evaluate the performance of hill-climbing resource distribution by comparing against prior techniques. To better understand our technique, we perform more experiments. First, we investigate the run-time overhead of our approach. Then, we study the sensitivity of the hill-climbing resource distribution's performance to various design parameters.

8.1 Run-time Overhead Analysis

There are two run-time overheads in the implementation of hill-climbing resource distribution: stand-alone IPC computation and resource distribution scheduling overhead. To measure the run-time overhead, we incorporated them into the simulator and experimented with them.

8.1.1 Run-time Stand-Alone IPC Computation Overhead

Of the 3 performance metrics discussed in Section 4.2.3, average weighted IPC and harmonic mean of weighted IPC (Equation 4.3 and 4.4) require the stand-alone IPC of each thread, $SingleIPC_i$. Because the $SingleIPC_i$ values are not known a priori, the hill-climbing algorithm must learn them along with the best resource distribution. 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 its performance impact, we acquire a sample every 40 epochs only; hence, each thread's

$SingleIPC_i$ is sampled once every $40 \times T$ epochs. To warm up the cache and clear out the instructions belonging to the stalled threads from the pipeline, we measure the stand-alone IPC only during the second half of the $SingleIPC_i$ sampling epoch.

Our study shows that there are three ways that the stand-alone IPC computation affects performance. First, there is performance loss due to the fetch stalling of $T - 1$ threads during the $SingleIPC_i$ sampling epoch. Second, if the application is not temporally stable, the run-time sampled stand-alone IPC is not representative. Third, if the application needs large amount of cache, the run-time sampled stand-alone IPC is lower than the actual stand-alone IPC because the cache is not fully warmed up during the sampling period.

Figure 8.1 shows the overhead of the run-time stand-alone IPC computation. The bars labeled HILL-WIPC-Online present the weighted IPC normalized against the baseline HILL-WIPC, which uses the off-line computed $SingleIPC_i$. As a reference, we included the DCRA bars.

On NH-TP and NH-TU workloads, the HILL-WIPC-Online has 0.68% and 0.42% worse performance compared to HILL-WIPC. This performance loss is caused by the temporal instability of the workloads. On workloads that exhibit less temporal locality of performance, the sampled stand-alone IPC becomes less representative. Therefore, the hill-climbing algorithm is guided by an inaccurate performance evaluation function, thus losing the performance. On MEM workloads, the HILL-WIPC-Online has 0.29% worse performance compared to HILL-WIPC because the cache is not sufficiently warmed up during the sampling epoch. As a result, the sampled stand-alone IPC becomes less accurate on MEM workloads. Interestingly, on WH and NH-TS workloads, HILL-WIPC-Online shows slightly better performance than HILL-WIPC (0.09% and 0.35%, respectively). On

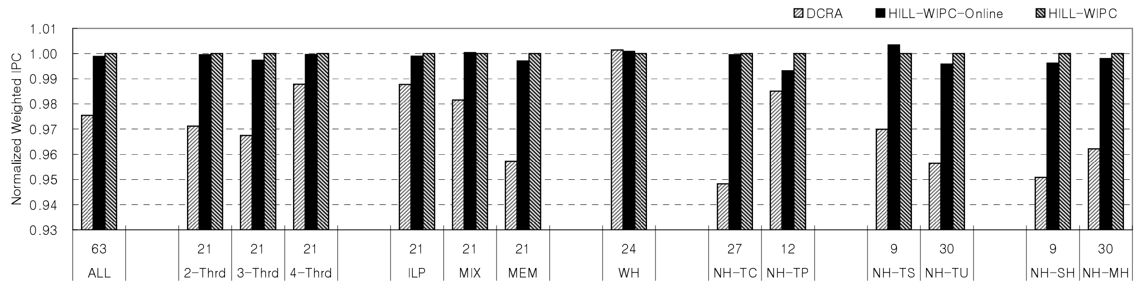


Figure 8.1: Stand-alone IPC computation overhead. HILL-WIPC-Online computes the stand-alone IPC at run-time. All bars are normalized against the HILL-WIPC.

workloads with temporal locality of performance, the sampled stand-alone IPC is more representative of the current application characteristics than the off-line computed stand-alone IPC, which is averaged across the entire application run. Therefore, on WH and NH-TS workloads, even after counting the run-time stand-along IPC computation overhead, HILL-WIPC-Online has better performance. On average, performance of HILL-WIPC-Online is 0.12% worse than that of HILL-WIPC. Considering the hill-climbing resource distribution's 2.8% performance gain over DCRA, this performance loss does not significantly reduce the performance advantage of the hill-climbing resource distribution over existing techniques.

8.1.2 Resource Distribution Overhead vs. Epoch Size

We varied the epoch size to study how epoch size affects the performance of software and hardware implementation of the hill-climbing algorithm. For the software implementation of the hill-climbing algorithm, we conservatively stall not only the victim thread but the whole processor for 200 cycles to account for the time to interrupt and save/restore the few registers needed by the hill-climbing algorithm. Considering that the resource distribution thread consumes only 26 cycles for its computation and no operating system is involved in this type of interrupt, we believe 200 cycle stall is a conservative setting.

Figure 8.2 shows the performance of software and hardware implementation of HILL-WIPC as we increase the epoch size from 8K to 256K cycle. All bars are normalized against the performance of the hardware implementation with 64K epoch size.

The epoch size affects the performance of the hill-climbing resource distribution in three ways. First, the resource distribution overhead of the software implementation of the algorithm decreases as we increase the epoch size because the hill-climbing algorithm performs resource distribution at the epoch boundary. When the epoch size is 8K, 16K, 32K, 64K, 128K, and 256K cycles, the resource distribution overhead of the software implementation is as large as 2.4%, 1.2%, 0.6%, 0.3%, 0.15%, and 0.08%, repectively, of the total execution time. Second, the performance of the workload becomes temporally stable as we increase the epoch size because large epoch size averages out the effect of noise caused, for example, by the L2 misses or branch mispredictions. So, hill-climbing benefits by reducing the unnecessary movement towards false peaks. On WH workloads, large epoch size reduces the unnecessary resource distributions around the optimal one, thus increasing performance. Third, hill-climbing's search speed will be increased as we decrease the epoch size because the hill-climber gets the performance feedback more frequently thus adapting to the changes more quickly. So, NH-TU and NH-MH workloads prefers small epoch size because the fast learning can help avoid their performance hazards.

Overall, the influence of the epoch size on the performance is small, especially for the hardware implementation. However, in software implementation, small epoch size significantly degrades the performance due to the run-time resource distribution overhead. As "ALL" bars in Figure 8.2 shows, 64K epoch size is in the middle of the stable range of both software and hardware implementation of the hill-climbing resource distribution. Therefore, we picked 64K cycle epoch size for our experiments throughout the dissertation.

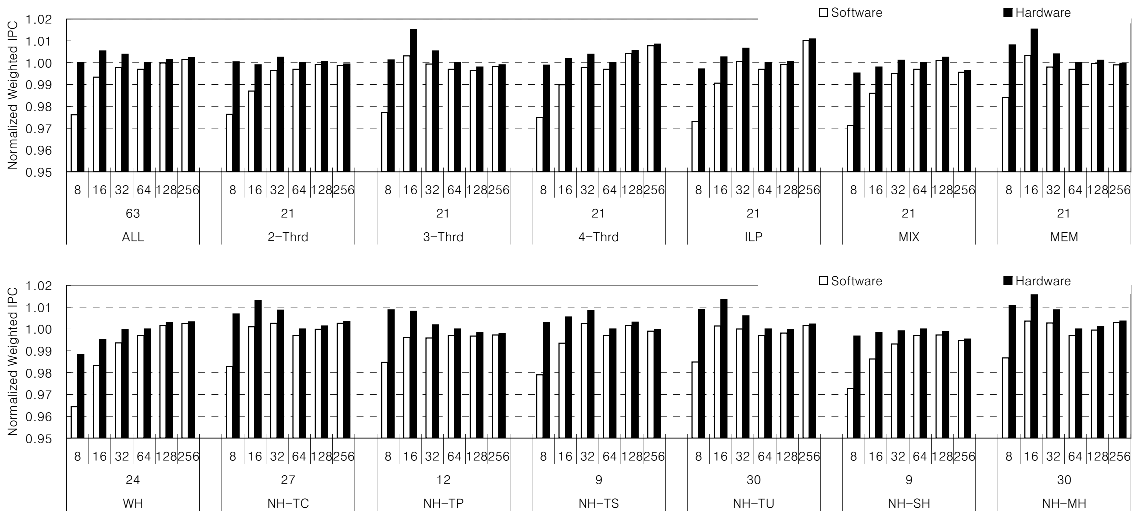


Figure 8.2: The resource distribution overhead of software and hardware implementation as we vary the epoch size from 8K to 256K cycle. All bars are normalized against 64K epoch size performance of hardware implementation.

At this epoch size, the run-time overhead of the software implementation is only 0.3%.

8.2 Sensitivity Study

To understand the effect of varying design parameters on the hill-climbing resource distribution performance, we conducte three sensitivity studies: varying memory latency, amount of pipeline resources, and priority of threads.

8.2.1 Memory Latency

We investigate the sensitivity of the HILL-WIPC's performance on the memory latency by varying memory latency between these settings–100, 300, and 500 cycles. Figure 8.3 shows the result of our memory latency study. In the figure, all bars are normalized against HILL-WIPC.

In this figure, we make two observations. First, within the range of the memory latencies that we experimented with, the HILL-WIPC achieves the best performance con-

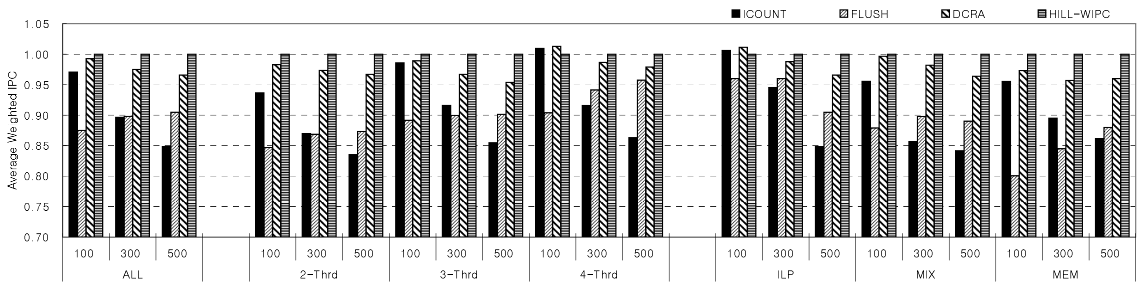


Figure 8.3: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC as we vary the memory latency from 100 to 500 cycle. All bars are normalized against HILL-WIPC.

sistently. This is a promising result, implying hill-climbing resource distribution can be applied to future platforms, where the processor-memory performance gap is larger. Second, HILL-WIPC's performance improvement over ICOUNT and DCRA increases as the memory latency increases by achieving performance advantage over ICOUNT by 3.0%, 11.4%, and 17.8%, and over DCRA by 0.8%, 2.8%, and 3.5% for 100, 300, and 500 cycle memory latencies, respectively. As the memory latency gets larger, the hill-climbing resource distribution's ability to exploit the memory level parallelism becomes more important.

8.2.2 Amount of Processor Resource

To understand the performance of the hill-climbing resource distribution on diverse platforms, we varied SMT processor simulator settings as shown in Table 8.1. In the table, the configurations labeled "half" and "double" have half and double the amount of queue type resources (we explained queue type in Section 2.2) in the processor compared to the "normal", respectively. The half configuration is similar to current modern processors, and the double configuration forcasts future platforms. Figure 8.4 shows our result on varying the amount of processor resources. In the figure, all bars are normalized against HILL-WIPC.

configuration	half	normal	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

Table 8.1: Simulator settings for the sensitivity study on the amount of the processor resource.

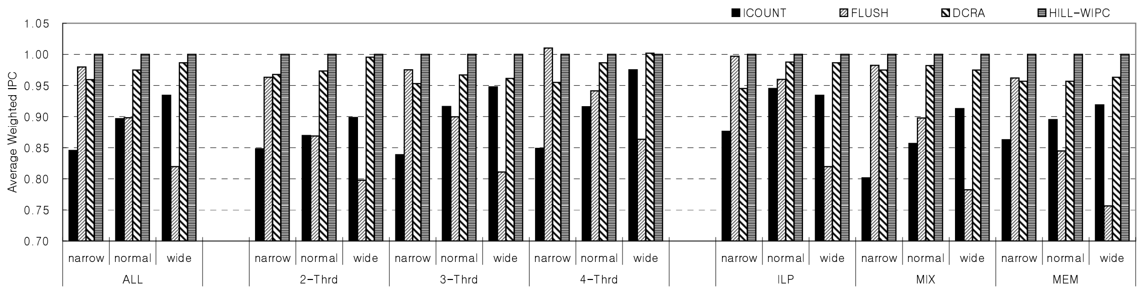


Figure 8.4: The weighted IPC of ICOUNT, FLUSH, DCRA, and HILL-WIPC as we vary processor resource budget to "half" and "double." All bars are normalized against HILL-WIPC.

Across all configurations, HILL-WIPC achieves the best performance compared to ICOUNT, FLUSH, and DCRA. For the half configuration, the performance of FLUSH gets better because FLUSH allows better resource utilization which is crucial as resources become scarce. In addition, with the half configuration, there is not much opportunity to exploit memory level parallelism within a thread. This is because back-to-back L2 cache misses are unlikely to fit inside the small instruction window of the half configuration. Therefore, in half the configuration, intra-thread memory parallelism generally cannot be exploited by any technique. Among all techniques, FLUSH exploits inter thread memory parallelism the best, thus achieving good performance in the half configuration.

For the double configuration, the performance of DCRA improves because the hill-width becomes wider, which benefits DCRA.

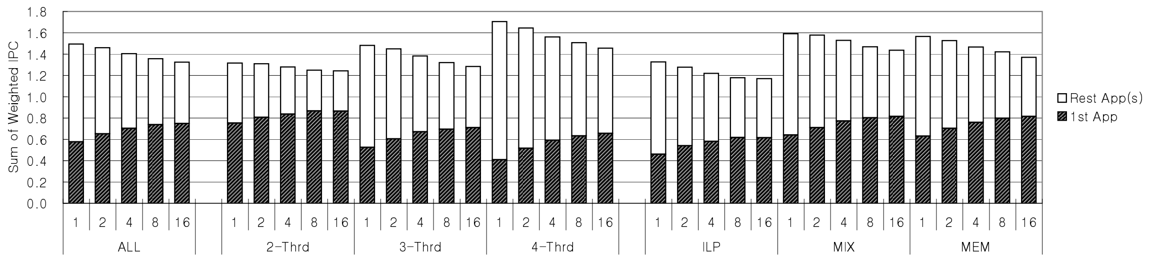


Figure 8.5: The effect of prioritizing the first thread. Shaded bars show the weighted IPC of the first thread and white bars show the sum of the weighted IPC of the rest of the thread(s), as we vary the priority of the first thread from 1 to 16.

8.2.3 Thread Priority

Enforcing the priority among multiple threads in SMT processor was studied by [44, 45]. We experimented with the possibility of prioritizing threads by simply modifying the performance evaluation function of hill-climbing resource distribution. Equation 8.1 shows the modified performance evaluation function for prioritizing threads, where P_i is the externally given priority of thread$_i$.

$$\mathsf{Sum\_of\_Prioritized\_IPC} = \sum IPC_i \times P_i \tag{8.1}$$

Figure 8.5 shows our results when P_0 is set to be 1, 2, 4, 8, and 16, while P_i $(i \neq 0)$ is set to 1. With a large value of P_0, small IPC_0 increases result in bigger improvements of the sum of prioritized IPC. Therefore, the hill-climbing resource distribution tends to improve IPC_0 to maximize sum of prioritized IPC. As a result, thread$_0$ gets the highest priority.

In Figure 8.5, two segments in the bar represent the weighted IPC of the first thread and the sum of weighted IPC of the rest of the threads, making the height of the stack the sum of weighted IPC of all the threads. The P_0 value of each bar is shown in X-axis.

When the priority of 16 is given to thread$_0$, the performance of thread$_0$ reaches 86.4%, 71.1%, and 65.7% of the single threaded execution, degrading the overall performance by 5.6%, 13.4%, and 14.6% compared to equal priority, for 2-, 3-, and 4-threaded workloads, respectively.

Chapter 9

Case Study: Optimizing Multi-Threaded Run-Time System

So far we investigate our hill-climbing SMT processor resource distribution for multiple independent applications. Recently, modern programming language environments provide rich run-time services to support flexibility, performance, security, and the correctness of the program. For example, Java (from Sun Microsystems) and C# (from Microsoft) provide just-in-time compilation, garbage collection, dynamic binding, and authentication service to the applications. These run-time services can potentially exploit SMT processor's support for multiple threads by running the services concurrently with the application, thus reducing the overall execution time. Compared to other multi-threaded applications, multi-threaded run-time system has advantage as it needs no programmer intervention to extract thread level parallelism.

In this chapter, we conduct preliminary study to show the benefit and potential of the multi-threaded run-time system running on SMT processor.

9.1 Kaffe–Multi-Threaded Run-Time System

Kaffe is a complete Java run-time environment, consisting of a Java virtual machine and a set of class libraries necessary to execute Java programs. This section details our target multi-threaded run-time system, Kaffe.

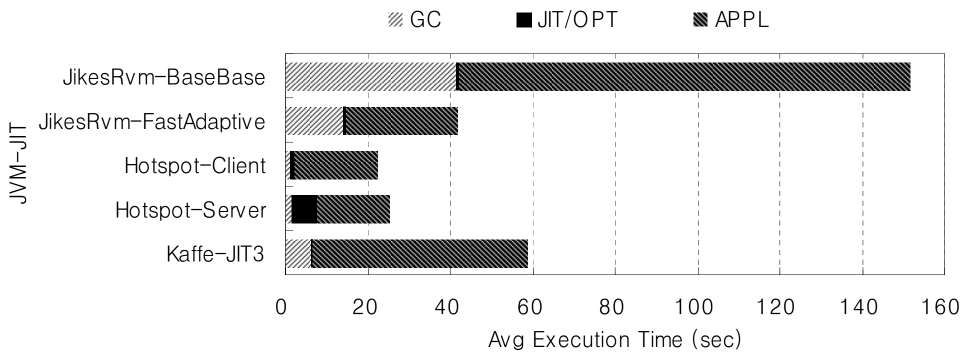


Figure 9.1: The fraction of execution time spent by garbage collection (GC), just-in-time compilation/optimization (JIT/OPT), and application execution (APPL) on 5 popular JVMs. Average execution time of 6 applications from SPECjvm98 is measured on Pentium III 500MHz processor using the profiling tool embedded in each JVM.

9.1.1 Kaffe

Kaffe has a JIT compiler that dynamically translates Java bytecodes into the native machine code. In addition, Kaffe supports automatic memory management using a non-copying mark-sweep garbage collection. The entire Kaffe is publicly available and has been ported to several platforms. To maintain compatibility with our simulator (see Section 4.2.5), we use the Alpha port of Kaffe version 1.0.7. We modify Kaffe to run on our SMT processor simulator. Our modified version of Kaffe is equipped with a concurrent garbage collection thread and multiple compiler threads.

Figure 9.1 shows the average execution time of SPECjvm98 applications on several popular JVMs; JikesRvm, HotSpot, and Kaffe. The execution time is divided into application execution time, labeled “APPL”, garbage collection time, labeled “GC”, and JIT compilation/optimization time, labeled “JIT/OPT”. As Figure 9.1 indicates, garbage collection overhead is bigger than JIT compilation overhead because JIT compilation is one-time service per method and garbage collection service is constantly invoked as applications consume heap space during the execution. Therefore, we investigate the effect of concurrent execution of garbage collection thread with the application thread on SMT processor.

9.1.2 Garbage Collection

The garbage collector in the original Kaffe runs sequentially with the application thread, even though features for the concurrent execution is in place for the future improvement. To make the garbage collector run concurrently with the application thread, we added write barrier synchronization.

Parallel execution of the garbage collection and application threads introduces a race condition: the application thread, or "mutator," may alter the references to an unmarked object in such a way that hides it from the collector. This problem can be addressed using Dijkstra's Tricolor formulation [46]. As the name implies, objects take on one of three colors during marking. All objects start white. When an object is marked, its color becomes gray. A gray object becomes black after all its children have also been marked (and thus colored gray). The JVM must ensure that a black object never directly references a white object. This would constitute an invalid state because the white object, while still reachable, may never be marked since the collector assumes all children of black objects have been processed.

To maintain this invariant in a parallel garbage collector, a check or *write barrier* is necessary every time the application thread writes an object reference into a heap object. The check tests the color of the written object. If it is black, the object is recolored gray, forcing the collector to reprocess the object's children.

In our modified JVM, write barriers are necessary in two places: Java code and non-Java JVM code (*i.e.*, C code). Instrumenting Java code is straight forward because only a limited number of bytecodes write object references and all bytecodes are translated.

We modified the JIT compiler to insert a write barrier whenever it translates one of the relevant bytecodes. Instrumenting the JVM C code is more challenging because there are hundreds of places in the JVM where object references are written. Rather than identifying these sites via code analysis, we identified them via profiling. We used our simulator to examine the contents of the data register each time a store instruction executes. Since we know the range of the heap in memory, we can recognize when a store instruction is writing a possible heap pointer. All static store instructions in the JVM C code meeting this condition were instrumented with write barriers (stores to the stack were excluded). Note, our approach, while complete, instruments too many write barriers since our test for heap pointer writes can falsely identify some store instructions.

9.2 Experimental Methodology

We use SPECjvm98 applications as listed in Table 9.1 with the problem size of 10.[1] We begin our simulation after loading and compiling all the classes, and simulate applications to the completion. We pick three applications out of 7 SPECjvm98 applications, which successfully run to the completion on our simulator. In Table 9.1, the column labeled "app insn" shows the number of simulated instructions by the application thread, "gc insn" shows the number of simulated instructions by the garbage collection thread, and "heap size" shows the maximum heap size allowed to the application.[2]

For concurrent garbage collection, the garbage collection triggering time affects the overall performance. Too early triggering makes garbage collector to collect small amount

[1]SPECjvm98 provides problem size of 10 and 100. 100 is for reporting the performance result and 10 is for testing. We chose the smaller one because we need to simulate the application to the completion.

[2]The number of instructions slightly varies depending on the SMT processor resource distribution techniques.

application	app insn	gc insn	heap size	gc trig
compress	2,431M	7M	13M	90%
jess	727M	131M	9M	30%
db	502M	409M	11M	90%

Table 9.1: Description of applications from SPECjvm98 benchmark suite. "app insn" shows the number of simulated instructions from the application thread, "gc insn" shows the number of simulated instructions gc thread, "heap size" shows the maximum heap size allowed to the application, "gc trig" indicates the garbage collection thread triggering time.

of garbage per invocation, thus increasing the run-time overhead. Too late triggering makes the application thread to be blocked because the insufficient amount of available memory may not support the memory request from the application. However, finding the proper garbage collection triggering time is beyond the scope of our research. In fact, operating system or run-time system should be designed to deal with this issue. Therefore, we tried 10 different garbage collector triggering time and pick the best performing one. We triggered garbage collector when application consumes 10%, 20%, 30%, ... 100% of the available heap space, which is measured right after the previous garbage collection, and we picked the best performing garbage collection triggering time for each application. The column labeled "gc trig" in Table 9.1 indicates the garbage collection thread triggering time of each application.

9.3 Results

To measure the performance impact of the concurrent execution of the garbage collection, we use ICOUNT, FLUSH, DCRA, and our hill-climbing resource distribution. We used two performance evaluation functions for our hill-climbing resource distribution; sum of IPC (used in Section 7.2.2) and sum of prioritized IPC (used in Section 8.2.3). Figure 9.2 shows the normalized execution time of sequential garbage collection (labeled "SEQ") and parallel garbage collections (labeled "ICOUNT", "FLUSH", "DCRA", "HILL-IPC", and

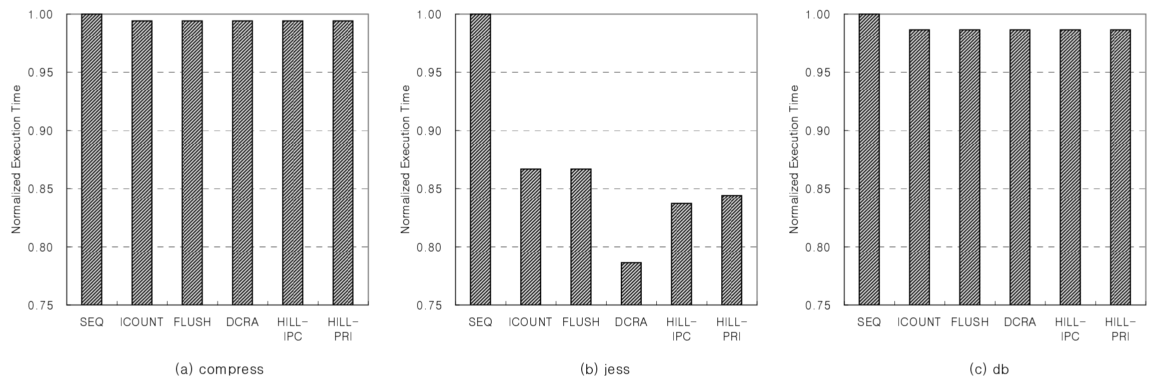


Figure 9.2: The normalized execution time of three SPECjvm98 applications with sequential garbage collection (labeled "SEQ") and parallel garbage collections (labeled "ICOUNT", "FLUSH", "DCRA", "HILL-IPC", and "HILL-PRI").

"HILL-PRI").

A result shows that concurrent garbage collection improves the performance over sequential garbage collection by exploiting the parallelism between application thread and garbage collection thread. The performance gain of concurrent garbage collection in compress is small because the number of instruction executed by garbage collection thread is only 0.29% of the application thread. (See "app insn" and "gc insn" column of compress in Table 9.1.) The performance gain of concurrent garbage collection in db is small because the concurrent execution of the garbage collection thread and application thread increases the number of instruction due to the write barriers. Therefore, garbage collection thread triggering time of 90% performs the best as it reduces the parallelism between two threads, thus reducing the write barrier overhead. As a result, db's performance of concurrent garbage collection becomes close to that of sequential garbage collection.

Among the concurrent garbage collection, the performance of most of the SMT resource distribution techniques are almost same except for jess. In compress, the garbage collection overhead is very small (refer to "gc insn" column of Table 9.1) making the performance of any resource distribution technique same. In jess, our hill-climbing algorithm does not have enough time to find the optimal the resource distribution considering that hill-climbing algorithm is active only when garbage collection thread is running. In db, our hill-climbing algorithm has enough time to find the optimal resource distribution, thus achieves as good performance as all the other resource distribution techniques.

Chapter 10

Conclusion

10.1 Summary and Conclusion

In this dissertation, we propose a new approach to SMT processor 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 successively applying and evaluating different resource distributions, our approach tries to learn the best resource distribution over time. Because we perform learning on-line, learning time is crucial. We develop a hill-climbing SMT processor resource distribution technique that efficiently learns the best distribution of resources by following the performance gradient within the resource distribution space.

From this research, we draw following four conclusions. First, as shown in Chapter 4, we found that the performance curve is not random. Instead, the performance curve is hill-shaped and is stable over time for many workloads. Second, our heuristic that approximates performance limit of SMT processor shows that prior resource distribution techniques have missed many performance opportunities, which we discussed in Chapter 6. This limit study shows that the performance of ICOUNT, FLUSH, and DCRA is 13.2%, 13.5%, and 6.6%, respectively, lower than our approximated performance limit. Third, as shown in Chapter 7, hill-climbing resource distribution technique achieves the best performance compared to the prior techniques. The performance evaluation of our approach provides 11.4% gain over ICOUNT, 11.5% gain over FLUSH, and 2.8% gain over

DCRA across a large set of 63 multiprogrammed workloads. Fourth, we showed that the traditional hill-climbing algorithm works well not only on the static hills, but also on the time-varying hills, which opens the possibility of applying the hill-climbing algorithm to a variety of adaptive optimization problems.

10.2 Contributions

This dissertation makes six contributions within the context of learning-based SMT processor resource distribution.

1. The performance of SMT processor is mainly determined by the resource distribution among the concurrently running threads. So, we view the SMT processor resource distribution problem as a search problem whose goal is finding a resource distribution that produces the maximum performance. We believe this is a unique view in the SMT processor resource distribution study. This view makes us translate the resource distribution problem into the classical optimization problem, allowing us to apply general optimization problem solvers, hill-climbing algorithm, to SMT processor domain. In this dissertation, we define the performance curve as a function of SMT processor resource distribution. Then, we design the hill-climbing algorithm to climb up the performance curve to search for the optimal resource distribution.

2. The nature of SMT processor performance as a function of the resource distribution space is unknown prior to our research. In order to understand the time-varying behavior of this SMT processor performance curve, we built a visualization tool. Using this tool, we identified several workload characteristics. Some characteristics are hostile to hill-climbing algorithm by having multiple humps or extremely frequent time-varying behavior. On the other hand, many workloads have favorable

characteristics to the hill-climbing, like single hump and stable temporal behavior.

3. Based on the knowledge acquired through the visualization tool, we developed four new metrics that quantitatively measure the shape of the performance curve. Two metrics quantify the static shape of the performance curve and two metrics measure the temporal variation of the performance curve. Using these metrics, we classify workloads. This classification helps understanding and analyzing the workload's performance of prior SMT processor resource distribution techniques as well as our hill-climbing technique.

4. We are the first to apply the hill-climbing algorithm to SMT processor resource distribution. The understanding of the time-varying performance curve from both the visualization tool and quantitative measurement enable us to customize the hill-climbing algorithm for the SMT processor resource distribution problem. We design our hill-climbing algorithm so that it can handle both problematic and favorable workload characteristics, making it applicable to a diverse set of workloads.

5. We faithfully evaluate the performance of the hill-climbing resource distribution technique across 63 workloads. Then, we compare hill-climbing performance against three prior SMT processor resource distribution techniques. We suggest two improvements over the baseline hill-climbing algorithm; phase based learning and hill-climbing with momentum term. In addition, we study hill-climbing resource distribution's sensitivity to three design parameters: memory latency, amount of processor resource, and thread priority.

6. A performance comparison of existing resource distribution techniques against an ideal SMT processor can uncover performance bottlenecks, and suggest ways to

improve performance. However, figuring out the ideal performance limit of SMT processor is computationally infeasible because it is an NP-hard problem. For the first time in SMT study, we developed a heuristic that approximates the ideal performance limit of SMT processor. To make our heuristic computationally feasible we made three simplifying constraints; first, per-thread resource partition is maintained to distribute resources, second, updating resource partition is allowed only at every epoch boundary, and third oracle provides information only on the next epoch. Using the performance limit suggested by our approximation, we re-evaluate four SMT processor resource distribution techniques (including ours) and detail their performance potential/bottleneck. This limit study shows that hill-climbing algorithm is the closest to the performance limit because our technique handles most of the bottlenecks of the existing techniques properly.

10.3 Future Directions

In this dissertation, we show that our hill-climbing SMT processor resource distribution is effective in achieving the best performance on multi-programmed workloads compared to the prior techniques. We believe that the idea presented in this dissertation can be extended to wider range of problems.

First, we can apply our technique to multi-threaded applications. Compared to the multi-programmed workload that we used in this dissertation, all threads in multi-threaded application belong to an application cooperating to accomplish a common job. So, the concurrently running threads interact each other via synchronization mechanisms for communication making some threads to be blocked waiting for the signal from other threads. These interactions make the resource distribution decision more complicated because any thread that may potentially delay other thread's execution should get the priority. To optimize the execution of the multi-threaded applications, the operating system should be aware of the dynamic criticality among the threads. Based on the criticality information, operating system schedules the threads to reduce the end-to-end execution time. To achieve better performance of a multi-threaded application on an SMT processor, the SMT processor resource distribution mechanism should adapt to dynamically changing demands from the operating system. Since our hill-climbing resource distribution technique is able to adaptively pursue any performance goal by just changing the performance evaluation function, our technique has advantage over any of the prior SMT resource distribution techniques. In Chapter 9, we opened this problem by conducting the preliminary study on executing parallel garbage collection thread and application thread simultaneously on SMT processor. For the complete experiment, we need more comprehensive experimental environment that includes the criticality aware operating system, multi-threaded benchmark applications, and our hill-climbing resource distribution technique.

Second, our idea can be used to deal with more general problems. In this dissertation, we designed a hill-climbing algorithm that searches for the peak on the time-varying hills, and showed its effectiveness on SMT processor resource distribution problem domain. However, our technique can be used for more general optimization problems, if we can translate the problem into "the chasing the moving target on the time-varying hill problem." For example, applying our hill-climbing algorithm to run-time hardware optimization problem requires the following steps. First, design the performance evaluation function that consists of any metrics that people care about in the specific problem domain. These metrics need to be easily looked up at run-time. An example of the performance evaluation function can be the weighted sum of (throughput), 1/(response time), 1/(power consumption), and 1/(power density). Then, we define the configuration space consisting of all valid settings of the multiple parameters which we tune to maximize the performance evaluation function. Third, we deploy our hill-climbing algorithm to search for the configuration that produces the maximum value of the performance evaluation function. We believe that our hill-climbing technique is useful for the class of optimization problems that needs to search for the optimal configuration out of large search space.

For example, using our hill-climbing algorithm, we can search for the proper size of activated cache entries for optimizing either throughput or power consumption. For optimizing the throughput, small cache reduces hit latency but increases the miss rate. Our hill-climbing technique measures the throughput of the application during an epoch (*i.e.*, end performance). So, our technique searches for the optimal cache size that maximize the throughput considering both hit latency and miss rate. For optimizing the power consumption, small cache reduces the power consumption by the cache. But it may increases the traffic to the memory, thus increasing the power consumption by the other part of the memory system. Our hill-climbing technique measures the total power consumption of the whole memory system during an epoch (*i.e.*, end power consumption). So, our technique finds the optimal cache size that minimizes the power consumption considering power consumption by both cache and rest of the memory system.

BIBLIOGRAPHY

[1] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 1996 International Symposium on Computer Architecture*, (Philadelphia), May 1996.

[2] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 392–403, ACM, June 1995.

[3] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, pp. 12–18, September/October 1997.

[4] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in SMT Processors," in *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pp. 171–182, IEEE Computer Society, 2004.

[5] A. El-Moursy and D. H. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," in *Proceedings of the 9th International Conference on High Performance Computer Architecture*, February 2003.

[6] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 318–327, IEEE Computer Society, 2001.

[7] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[8] R. N. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, 2004.

[9] http://www.intel.com/design/Pentium4/index.htm, "Intel Pentium 4 Processor." 2002.

[10] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec, "Boosting SMT Performance by Speculation Control," in *Proceedings of the International Parallel and Distributed Processing Symposium*, (San Francisco, CA), April 2001.

[11] S. Hu, M. Valluri, and L. John, "Effective Adaptive Computing Environment Management via Dynamic Optimization," in *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, (San Jose, CA), March 2005.

[12] T. Karkhanis, P. Bose, and J. E. Smith, "Saving Energy with Just in Time Instruction Delivery," in *In Proceedings of the International Symposium on Low Power Electronics and Design*, 2002.

[13] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," in *Proceedings of the 33th Annual International Symposium on Microarchitecture*, (Monterey, California), pp. 245–257, IEEE Computer Society TC-MICRO and ACM SIGMICRO, 2000.

[14] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy Efficient Co-Adaptive Instruction Fetch and Issue," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03)*, pp. 147–156, 2003.

[15] R. Goncalves, E. Ayguade, and a. P. O. A. N. M. Valero, "Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures," in *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, September 2001.

[16] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," in *Intel Technology Journal, 6(1)*, February 2002.

[17] A. Snavely and D. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading architecture," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[18] L. D. Davis, R. K. Belew, and L. B. Booker, "Bit-climbing, representational bias, and test suite design," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, (San Mateo, CA), pp. 18–23, Morgan Kaufmann, 1991.

[19] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The stanford hydra cmp," *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.

[20] L.A.Barroso, K.Gharachoroloo, R.McNamara, A. Nowatzyk, S.Qadeer, B.Sano, S.Smith, R.Stets, and B.Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," in *27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[21] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[22] R. Kumar, N. Jouppi, , and D. Tullsen, "Conjoined-core chip multiprocessing," in *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pp. 195–206, 2004.

[23] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *Proceedings of HPCA*, pp. 258–262, IEEE Computer Society, 2005.

[24] S. Baluja, "An Empirical Comparison of Seven Iterative and Evolutionary Function Optimization Heuristics," CMU-CS 95-193, Carnegie Mellon University, 1995.

[25] R. Razdan and M. Smith, "A highperformance microarchitecture with hardwareprogrammable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 172–180, IEEE Computer Society TC-MICRO and ACM SIGMICRO, 1994.

[26] S. Hauck, T. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 87–96, 1997.

[27] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135, 1996.

[28] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12–21, 1997.

[29] T. Miyamori and K. Olukotun, “A quantitative analysis of reconfigurable coprocessors for multimedia applications,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 2–11, 1998.

[30] C. R. Rupp, M. Landguth, T. Garverick, E. G. , H. Holt, J. M. Arnold, and M. Gokhale, “Holt and J. M. Arnold and M. Gokhale The NAPA adaptive processing architecture,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 28–37, 1998.

[31] S. Manne and A. K. D. Grunwald, “Pipeline Gating: Speculation Control for Energy Reduction,” in *Proceedings of International Symposium on Computer Architecture*, (Barcelona, Spain), 1998.

[32] A. Baniasadi and A. Moshovos, “Instruction flowbased front end throttling for Power-Aware High- Performance Processors,” in *Proceedings of International Symposium on Low Power Electronic Design*, 2001.

[33] D. H. Albonesi, “Dynamic IPC/Clock Rate Optimization,” in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, 1998.

[34] A. Snavely, D. Tullsen, and G. Voelker, “Symbiotic jobscheduling with priorities for a simultaneous multithreading processor,” in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002.

[35] K. Luo, J. Gummaraju, and M. Franklin, “Balancing throughput and fairness in smt processors,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, November 2001.

[36] D. Madon, E. Sanchez, and S. Monnier, “A Study of a Simultaneous Multithreaded Processor Implementation,” in *Proceedings of EuroPar ’99*, (Toulouse, France), pp. 716–726, Springer-Verlag, August 1999.

[37] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” CS TR 1342, University of Wisconsin-Madison, June 1997.

[38] G. K.Dorai and D. Yeung, “OTransparent Threads: Resource Allocation in SMT Processors for High Single-Thread Performance,” in *Proceedings of the 11th Annual International Conference on Parallel Architectures and Compilation Techniques*, (Charlottesville, VA), September 2002.

[39] G. K. Dorai, D. Yeung, and S. Choi, “Optimizing SMT processors for high single-thread performance,” *Journal of Instruction Level Parallelism*, pp. 1–35, April 2003.

[40] D. Kim and D. Yeung, “Design and evaluation of compiler algorithms for pre-execution,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[41] D. Kim and D. Yeung, “A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code,” *ACM Transactions on Computer Systems*, vol. 22, August 2004.

[42] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[43] T. Sherwood, S. Sair, and B. Calder, “Phase Tracking and Prediction,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 336–347, June 2003.

[44] S. E. Raasch and S. K. Reinhardt, “Applications of Thread Prioritization in SMT Processors,” in *Proceedings of the 1999 Multithreaded Execution, Architecture, and Compilation Workshop*, January 1999.

[45] G. K. Dorai, D. Yeung, and S. Choi, “Optimizing SMT processors for high single-thread performance,” in *Journal of Instruction-Level Parallelism Vol5*, pp. 1–35, April 2003.

[46] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, “On-the-fly garbage collection: An exercise in cooperation,” *Communications of the ACM*, vol. 21, pp. 966–975, Nov. 1978.