University of Maryland Technical Report CS-TR-4723 and UMIACS-TR-2005-30

# Hill-Climbing SMT Processor Resource Scheduler

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

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

May 18, 2005

**Abstract**

Multiple threads in SMT processor share resources to increase resource utilization and improve overall performance. At the same time, they compete against each other for the shared resources, causing resource monopolization or underutilization. Therefore, resource scheduling mechanism is important because it determines the throughput as well as fairness of the simultaneously running threads in SMT processor. To achieve optimal SMT performance, all the earlier mechanisms schedule resources based on a couple of indicators, such as cache miss count, predecoded instruction count, or resource demand/occupancy. Those indicators trigger scheduling actions, expecting improved performance. However, since combinations of indicators can not cover all possible cases of threads behavior, earlier mechanisms may face a situation where the expected positive correlation between the indicators and the performance becomes weak, losing the chance of performance improvement.

In this paper, we developed a novel methodology using hill-climbing or gradient descent algorithm to find the optimal resource share of simultaneously running threads. We carefully vary the resource share of multiple threads toward the direction which improves the SMT performance. Instead of monitoring the behavior of indicators, we use the past resource share and SMT performance change history to determine the resource share shift direction for the future. Simulation result shows that our hill-climbing resource scheduler outperforms FLUSH by 5.0% and DCRA by 1.4%, on average, using the weighted IPC as a metric.

**Keywords:** SMT processor, hill-climbing resource scheduler, locality of performance

## 1 Introduction

The multiple threads in SMT processor share resources to improve overall performance [1, 2]. But, at the same time, they compete against each other for shared resources. Without proper resource scheduling, resource may be held by a thread, which does not fully utilize it, losing the throughput. In addition, some threads may monopolize resources and other threads may be discriminated, losing the fairness. Another important reason for SMT resource scheduling is that each thread has different resource requirement, and the amount of performance degradation, when the given resource is less than the thread's demand, varies significantly. Therefore, the resource scheduler should give more resources to a thread that degrades the performance the most without them.

In SMT processor, complex competitions and interactions happen among multiple threads for the shared resources, requiring carefully designed resource scheduler.

### 1.1 Hill-climbing resource scheduler

The basic idea of hill-climbing or gradient descent algorithm is to always head toward the best neighboring state, which is better than the current one. By moving this way, we eventually reach the peak of the hill (or the best state). If the hill-defining function is derivative, the gradient vector guides the best movement direction.

We applied the hill-climbing algorithm to SMT processor resource scheduling and developed a novel *hill-climbing resource scheduler*. Our goal here is to achieve the best performance of SMT processor by finding the optimal hardware resource share among multiple threads. Figure 1 shows an example of SMT processor performance for all combinations of RUU resource share among three threads. The arrow mark indicates the optimal resource share, which all resource schedulers are looking for. As Figure 1 shows, the shape of the performance graph looks like hill. This motivates us applying hill-climbing algorithm to finding the optimal resource share of multiple threads.

Please note that each data point in Figure 1 requires an SMT simulation and it is not possible to figure out the whole shape of the hill at run time. In hill climbing resource scheduler, however, we just need to know the shape of the neighboring positions to make decision on future movement. More specifically, our hill-climbing resource scheduler carefully varies the resource share of multiple threads toward the direction which improves the SMT performance. We use the past resource share and SMT performance change history to determine the resource share shift direction for the future.

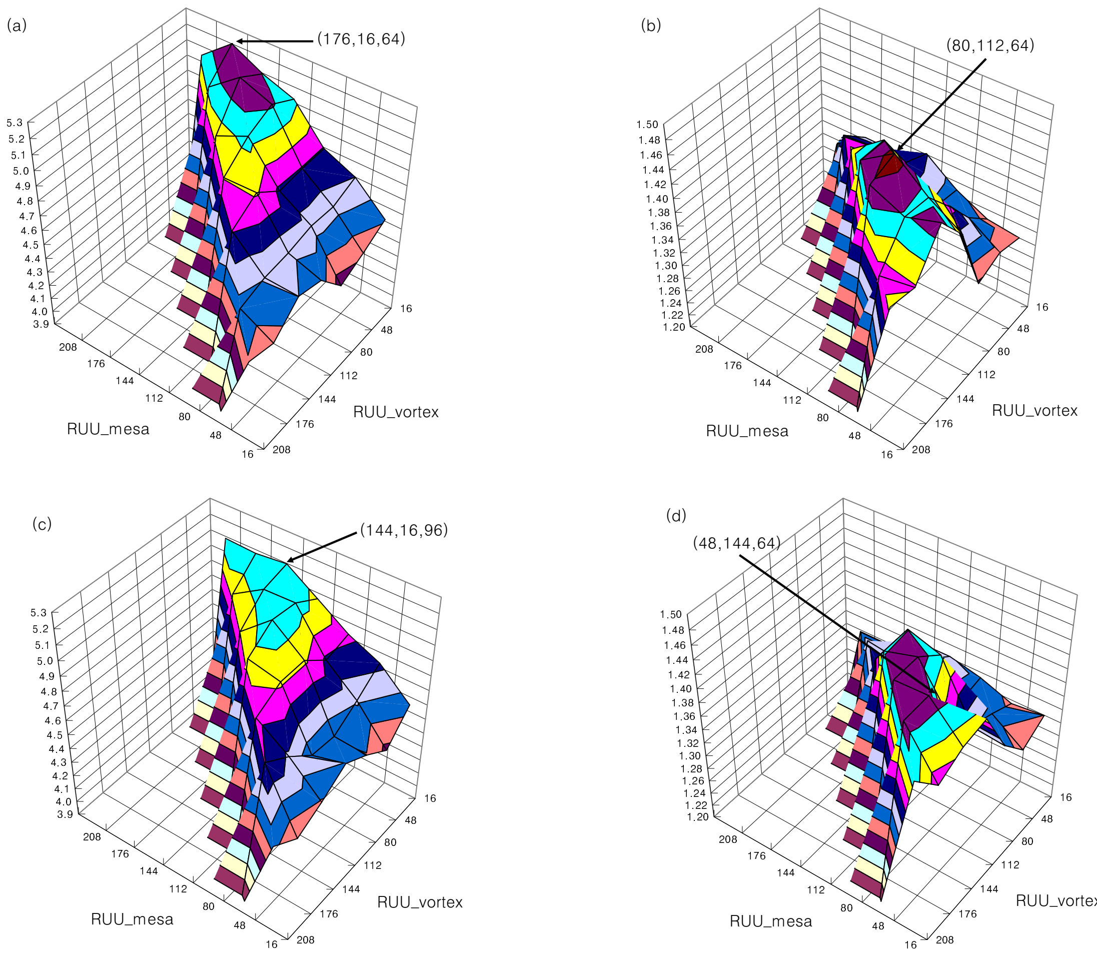

Figure 1: The performance of SMT processor when mesa, vortex, and fma3d run simultaneously. (a) and (c) show the sum of IPC at 1600K and 1760K processor cycle, respectively. (b) and (d) show the sum of weighted IPC at 1600K and 1760K processor cycle, respectively. In each graph, the arrow indicates the peak of the hill. Since the total number of RUU is fixed, the number of RUU allowed to the last application, fma3d, is dependent on those of the other two applications. So, the number of RUU for fma3d is not shown in the graph.

## 1.2 Related Work

Earlier SMT resource schedulers control fetch speed or key resource allocation at run time to maintain proper resource share among multiple threads. Their resource scheduling actions are triggered or guided by indicators, such as cache miss count, pre-decoded instruction count, and resource demand/occupancy.

ICOUNT [2] and FPG [3] favor fast threads, which cause fewer cache misses or fewer branch mis-prediction, by giving fetch priority to threads that consume smaller amount of pre-decode stage resources or shows high branch prediction accuracy. Therefore, ICOUNT and FPG improve throughput by increasing the IPC of the fast threads. However, pipeline may be clogged by a thread with long latency operations, thus reducing the throughput.

STALL [4] minimizes the pipeline clog by stalling fetch of a thread when long latency operation is detected. However, at the time this mechanism detects a long latency operation, it may be too late and a thread may hold many resources until the long latency operation is serviced.

FLUSH [4] corrects the late detection of the pipeline clog by flushing instructions of a thread with long latency operation. This forces the victim thread to release the resources immediately. However, flushing requires more fetch bandwidth and power consumption.

STALL-FLUSH [4] minimizes the number flushed instructions by, first, stalling a thread when long latency operations are detected, and second, flushing the thread when resource is exhausted. However, both FLUSH and STALL-FLUSH prevent exploiting memory parallelism because they tend to allow only one outstanding L2 cache miss at a time per thread.

DCRA [5] favors memory-bound threads by giving pre-computed large share of resources to threads with outstanding L1 cache miss. Therefore, memory-bound threads enjoy memory parallelism. In addition, compute-bound threads are allowed to use resources from memory-bound threads when they are not used, thus increasing the resource utilization. However, the resource share computing formula may give more resources than is necessary to memory-bound threads, which does not have any memory parallelism, causing inefficiency. In other words, classifying threads into two groups (with or without L1 cache miss) may not represent the variety of the thread characteristics.

The difference between earlier SMT resource schedulers and hill-climbing resource scheduler can be viewed in five aspects.

- All the earlier mechanisms rely on indicators to schedule resources. On the contrary, our mechanism uses the past performance. Figure 2 visualizes this difference. Our experiment shows that hill-climbing resource scheduler outperforms earlier mechanisms because of the following two reasons. First, as we discussed earlier in this section, the combinations of indicators can not cover all possible cases of threads behavior. Therefore, earlier mechanisms may face a situation where the expected positive correlation between the indicators and the performance becomes weak, losing the chance of performance improvement. Second, the performance of the future is likely to be similar to that of the past, which we call *locality of performance* and detail in Section 2.3.

- The static resource sharing models [6, 7, 8] have fixed resource partitions to ensure fair resource share among multiple threads at the cost of decreased resource utilization. However, resource schedulers [2, 3, 4, 5] dynamically control the resource share at run time based on the indicators. Hill-climbing resource scheduler lies between static resource sharing model and earlier resource schedulers in terms of resource share update frequency. In hill-climbing resource scheduler, the resource share is updated every pre-determined amount of processor cycle, which we call *epoch* and detail in Section 2.2.

- The goal of all the SMT resource schedulers is to reach the peak of the hill, which is defined as an SMT performance, by finding the optimal resource share. We are the first focusing the shape of the hill and searching for the peak. Our approach is a natural way of handling the resource scheduling problem.

- Depending on our performance goal, the performance-defining (or hill-defining) function can be either one of the average IPC, average weighted IPC [2], or harmonic mean of weighted IPC [9]. In either case, our hill-climbing algorithm searches optimal resource share that maximizes the function value.

- Since hill-climbing algorithm is a generic methodology for solving optimization problems, we can extend our hill-climbing resource scheduler to solve similar problems, like cache partitioning or bandwidth partitioning.

The rest of the paper is organized as follows. Section 2 describes our hill-climbing resource scheduler. Section 3 explains the experimental methodology. Section 4 presents the performance of the hill-climbing resource scheduler. Finally, in Section 5, we conclude our research.

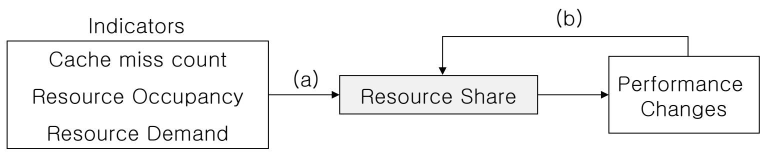

Figure 2: The way of maintaining proper resource share among multiple threads by (a) earlier resource schedulers and (b) hill-climbing resource scheduler.

## 2 Hill-climbing resource scheduler

We identified RUU as the most important shared resource because the contention against RUU tends to exist longer than any other resource contentions. For example, when a thread has a data cache miss, the cache miss load and all the subsequent instructions belonging to the same thread can not release RUU entries until the cache miss is serviced. Compared to RUU, pipeline bandwidth and functional unit does not cause hold-and-wait situation, decreasing the needs for scheduling. Therefore, in this research, we focus on scheduling RUU entries. Shared fetch queue is implicitly taken care of by our scheduler because we prioritize or stall fetching threads to control RUU occupancy. It is important to note that our hill-climbing resource scheduler is generic methodology and can be applied to any shared resources, if they need scheduling.

### 2.1 Issues on hill-climbing resource scheduler

There are several issues on hill-climbing resource scheduler, which are generic to all hill-climbing algorithm implementations or specific to our mechanism. The listed issues below are addressed in the Section 2.2 and Section 2.3 when we describe the implementation of hill-climbing resource scheduler.

**Hill shape change**: The shape of the hill keeps changing as the characteristics of threads change at run time. Figure 1 visualizes the hill shape changes. The hills in (a) and (b) are changed to (c) and (d), respectively, after 160K cycles. As these figures show, the shape and the position of the peak of the hill is shifting over time. Therefore, we should keep searching for the moving target.

**Gradient computation**: Ideally, we need to move toward the gradient vector. However, the performance function, which represents our hill, is not well formed formula nor derivative. The only way of figuring out shape of the hill is to pick sample positions and evaluate their performance one by one. Out of those sample performance, we can compute movement direction.

**Local maxima**: Generally, local maxima is a big issue in hill-climbing algorithm and we need a mechanism to escape from the local maxima. However, it is not a concern in our case, because the "hill shape change" effectively acts as a perturbation (or shaking the hill) to help escaping from the local maxima. In addition, the performance of each individual thread is monotone non-decreasing as the amount of allowed resource to the thread is increasing. Therefore, the SMT performance, as a function of resource partition size of all running threads, generally has small number of local maxima, if any.

**Searching speed**: Generally, the searching speed is important for hill-climbing algorithm because of the large search space, many local maxima, and significant amount of iterations to reach within the acceptable error range. In our case, the search space is small, there are fewer amount of local maxima, and it takes few iterations to reach the goal. However, there are three reasons why we need fast searching speed. First, as the shape of the hill is changing, the speed of searching the peak of the hill should be faster than that of the hill shape change. Second, sampling based gradient computation makes things worse because sampling the performance takes several thousands of processor cycles, delaying the time to find a direction for the next movement. Third, during the time we are on the way to the peak of the hill, threads are using less optimal resource share, thus losing the performance of the SMT processor.

## 2.2 Hill-climbing resource scheduler

We use hill-climbing algorithm to implement our hill-climbing resource scheduler. There are several terms that we defined to describe the hill-climbing resource scheduler.

**Epoch**: Execution time is divided into equal sized epochs, which we set to be 32K processor cycle. Between epochs, we choose the next sample position, and during each epoch, we evaluate the performance of the sample position.

**Performance evaluation function**: The performance evaluation function defines the hill that our resource scheduler climbs. Depending on the performance goal, the performance evaluation function can be either one of (1) average IPC, (2) average weighted IPC, (3) harmonic mean of weighted IPC, or (4) IPC of a high priority thread. Once a function is chosen, our algorithm will search for the optimal resource schedule that maximizes the function value.

**Advantaged thread**: Between epochs, one thread is elected to be the advantaged thread. Each thread takes turns in round robin fashion and becomes the advantaged thread. The sample position for the subsequent epoch is chosen toward giving more resources to advantaged thread.

**Delta**: The amount of additional resources given to the advantaged thread is determined by $delta_i$

of the advantaged thread $i$. If the sample movement toward giving more resources to thread$_i$ improves the SMT performance, we increment $delta_i$ by 2. Otherwise, we set $delta_i$ to 1.

**Resource partition**: Between epochs, the sample position is computed by incrementing the resource partition for the advantaged thread by $(delta_{adv} \times (N-1))$, and decrementing the resource partitions for the non-advantaged threads by $delta_{adv}$, where thread $adv$ represents the advantaged thread and $N$ represents the total number of running threads. After an epoch, we check if the advantaged thread fully exploits its given chance by evaluating the performance during the previous epoch. If the SMT performance of the previous epoch becomes better than that of the past epochs, we accept the sample position and consider it as the current position. Otherwise, we discard the sample position.

**Performance of the past epochs**: We have to decide whether the performance of the previous epoch is improved over that of the past epochs. For this purpose, we maintain the performance of the past epochs, which is

$$\mathsf{past\_perf} = \mathsf{cur\_perf} \times p + \mathsf{past\_perf} \times (1-p) \tag{1}$$

As the formula implies, the variable past_perf is updated partly $(p)$ by the most recent performance and partly $(1-p)$ by the old performance.

Figure 3 describes hill-climbing resource scheduler in pseudo code. The hardware unit for resource scheduler is invoked every Epoch_Size cycles to execute the routine (line 6). There are two shaded boxes in Figure 3. Box (a) evaluates the performance of the previous epoch. Box (b) sets up the new partition for the next epoch. In box (a), if the performance of the previous epoch is improved (line 8), we accept the partition used for the previous epoch and increment the delta[adv_thread] (line 10). Otherwise, we roll-back the partition (line 12-15) and decrement the delta[adv_thread] (line 16). In box (b), we elect a new advantaged thread (line 20) and give advantage by increasing the resource partition for the advantaged thread (line 21-24).

Figure 4 shows an example hill-climbing resource scheduling of the RUU resource among three threads running on SMT processor. In this figure, $(p_0, p_1, p_2)$ represents the coordinate in the search space, where $p_i$ is the amount of RUU entries allowed to thread$_i$. Because of the constraint, $p0+p1+p2 = 256$ $(= total\_RUU\_size)$, the search space is two dimensional. The vector $d_i$ indicates the movement direction when thread$_i$ is advantaged thread. In this example, $d_0 = (2, -1, -1)$, $d_1 = (-1, 2, -1)$, and $d_2 = (-1, -1, 2)$. Initially our search begins at the center of the space, which is (85, 85, 86). A sequence of vectors, labeled as **a, b, c, d, e, f**, and **g**, shows an example of

```
#define Epoch_Size       32k
#define N                Total number of running threads
#define Upper_Bound      The upper bound of delta. We used 9
#define eval_perf(X)     Evaluate the over all performance of SMT during the epoch X.
#define decay(X, Y)      Decay and compute the past performance. We used (X * 0.25 + Y * 0.75)

For every Epoch_Size cycles {
    perf = eval_perf(epoch_id);                         // evaluate the performance of the previous epoch

    if (perf > past_perf) {                             // previous epoch performs better than the past epochs
        if (delta[adv_thread] < Upper_Bound)            // accept the partition and increase delta
            delta[adv_thread] += 2;
    } else {
        partition[adv_thread] -= delta[adv_thread] * (N - 1);   // rollback to the previous partition
        for (i = 0 ; i < N ; i++)
            if (i != adv_thread)
                partition[i] += delta[adv_thread];
        delta[adv_thread] = 1;                          // decrease delta
    }

    past_perf = decay(perf, past_perf);                 // set the past performance              (a)

    epoch_id++;                                         // update epoch_id
    adv_thread = epoch_id % N;                          // elect a new advantaged thread

    partition[adv_thread] += delta[adv_thread] * (N - 1);   // try new partition for the new advantaged thread
    for (i = 0 ; i < N ; i++)
        if (i != adv_thread)
            partition[i] -= delta[adv_thread];                                                   (b)
}
```

Figure 3: Hill-climbing resource scheduler implementation in pseudo code. The shaded box (a) evaluates the performance of the previous epoch and (b) sets up the new partition for the next epoch.

resource scheduling movement trail.

## 2.3 Design choices

There are several design choices that we made for the hill-climbing resource scheduler implementation.

**Movement direction**: The general hill-climbing algorithm needs gradient. However, we can not compute gradient because the shape of the hill is represented not as a well formed function, but as performance outcome. So, we try a sample position around the current one for an epoch and measure the performance. Based on this trial, we can take the sample movement as a starting position of the next trial or discard the sample movement. There are two alternative implementations in choosing the movement direction: breadth first search (the direction to try is chosen in round robin fashion) and depth first search (pursue one direction until there is no performance improvement). Based on our experiment, breadth first search performs better than depth first search, because the latter exhibits longer searching path. So, the algorithm presented in Figure 3 uses breadth first

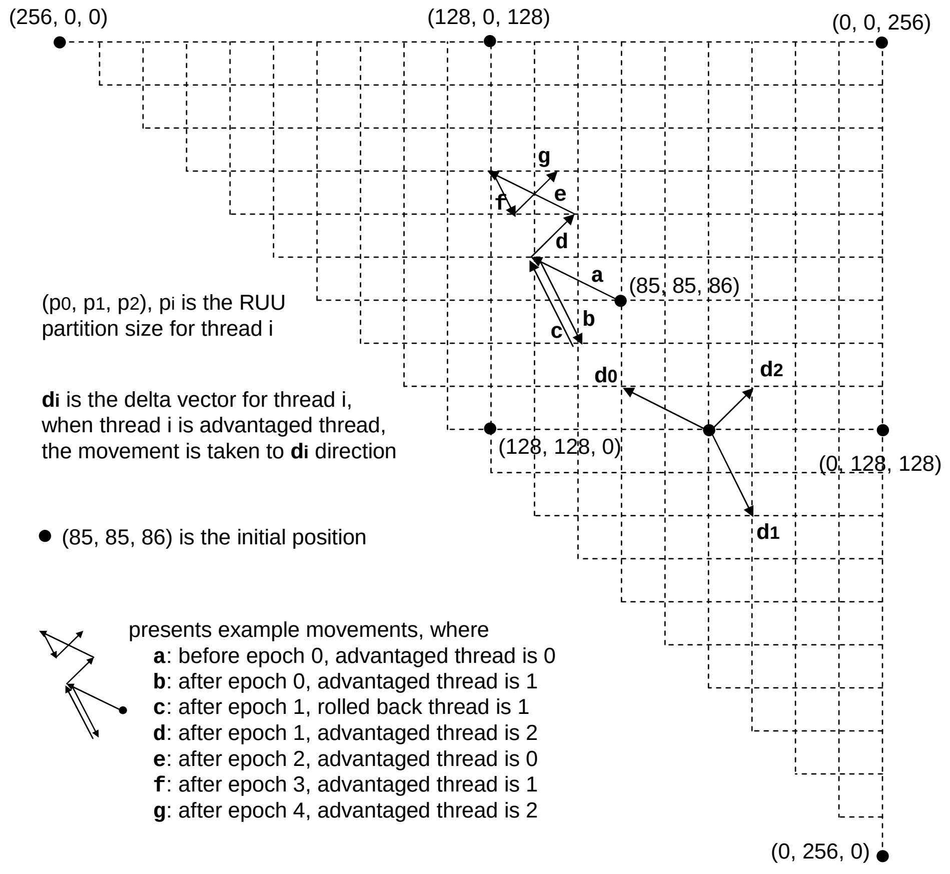
(256, 0, 0)
(128, 0, 128)
(0, 0, 256)
g
e
f
d
a
(85, 85, 86)
b
c
(p0, p1, p2), pi is the RUU partition size for thread i
d0
d2
di is the delta vector for thread i, when thread i is advantaged thread, the movement is taken to di direction
(128, 128, 0)
(0, 128, 128)
d1
(85, 85, 86) is the initial position
presents example movements, where
a: before epoch 0, advantaged thread is 0
b: after epoch 0, advantaged thread is 1
c: after epoch 1, rolled back thread is 1
d: after epoch 1, advantaged thread is 2
e: after epoch 2, advantaged thread is 0
f: after epoch 3, advantaged thread is 1
g: after epoch 4, advantaged thread is 2
(0, 256, 0)

Figure 4: The search space for hill-climbing resource scheduler and example movement. We schedule RUU resource for three threads.

search by making advantaged thread in round robin fashion.

**Adaptive delta**: As the delta becomes larger, the searching speed becomes faster, but there is more chance of passing over the optimal position. In addition, once we reached an optimal position, large delta will degrade the performance because it tries a position which is far away from the optimal one for an epoch and then returns to the optimal position again and again. Therefore, we chose to have per thread adaptive delta. If the movement toward favoring a thread$_i$ increases the performance, we increment the $delta_i$ to accelerate the movement. Otherwise, we set the $delta_i$ to 1.

**Decaying the performance of the past epochs**: To decide whether the performance of the previous epoch is improved or not, we need to maintain the performance of the past epochs as a reference. One way of maintaining the past performance is keeping the maximum performance that we found so far, which allows only the movement that outperforms any of the previous one. However, the shape of the hill keeps changing and the maximum performance of present (or future) may be less than the maximum performance that we found so far. Therefore, we need to decay the performance of the past epochs. Our choice is using the most recent performance as well as the old performance as shown in Equation 1. This equation implies a moving average of the performance along the past trail, giving the largest weight to the most recent position. The empirically chosen value of the weight for the most recent position ($p$ in Equation 1) is 0.25.

**Maintaining the locality of performance**: As we mentioned, the shape of the hill keeps changing and we are searching for the moving target. In this environment, we use the performance of the earlier history to schedule resources for the future. To make our resource scheduler working, we need to assume the *locality of performance* behavior, which we defined as *if the performance of a position* **P** *is an optimal one, the optimal position in near future will be close to* **P**. In other words, locality of performance assumes that the shape of the hill changes slowly. With this assumption, the optimal resource share, which our resource scheduler found so far, can be the best starting point of the searching for the future resource share. Two factors affect the locality behavior, the epoch size and the application characteristics.

- Epoch size: As the size of the epoch becomes smaller, the shape of the hill changes faster because the shape is volatile and affected by any small events, losing the locality of performance behavior. As the size of the epoch becomes larger, the hill shape changes slowly, but it does not represent the real shape of the hill due to the summary error. Therefore, we want to make the epoch size as small as possible. But, it is bound to the locality of performance behavior. We tried several epoch size and 32K-cycle has the best performance.

- Application characteristics: Our observation shows that the hill shape change frequency depends on the application characteristics. When the application characteristics change slowly, our hill-climbing resource scheduler works fine. When the application characteristics change fast, however, our method can not catch up with the hill shape changes. As a result, our search ignores high frequency hill shape changes and follows more coarse grained shape changes, thus losing the chance of finer grained, or more accurate, resource scheduling. We will experiment the effect of the application characteristics change on our scheduling in Section 4.4.

## 2.4 Hill-climbing resource scheduler implementation

Figure 5 shows the block diagram of hardware modifications to implement hill-climbing resource scheduler. As the figure shows, hill-climbing resource scheduler uses the number of committed

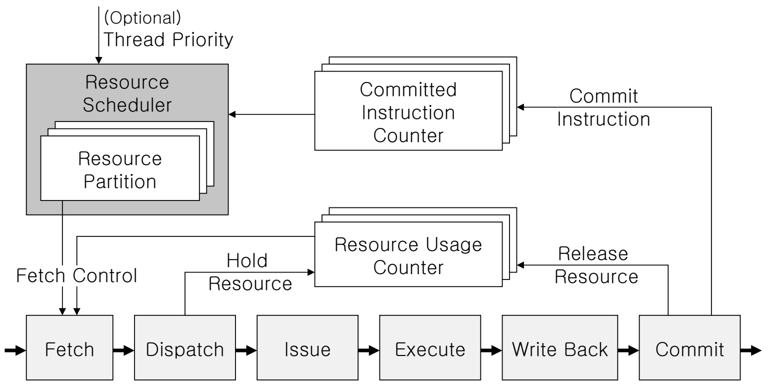

Figure 5: The block diagram of hill-climbing resource scheduler implementation in hardware.

instructions to compute the performance of the previous epoch. Depending on the definition of the performance, the resource scheduler optionally needs externally given thread priority to compute performance. Fetch unit compares the resource partition computed by the resource scheduler and resource usage counter to prioritize (or stall) thread fetching.

We need three counters per thread: committed instruction counter, which is embedded in most of the modern processor, resource usage counter, which is updated every time resource is held and released by a thread, and resource partition register, which is updated every epoch by the resource scheduler and referenced every cycle by the fetch unit.

In the implementation of Figure 3, the most expensive control logic is the performance evaluation part (line 7 - 8), which needs to compute either one of sum of IPC, sum of weighted IPC, or harmonic mean of weighted IPC[1]. However, since our hill-climbing resource scheduler is invoked infrequently (*i.e.* every epoch), the latency of the control logic is not critical, thus simplifying the logic design.

Alternatively, the algorithm in Figure 3 can be implemented in software and invoked every epoch by the timer interrupt. It is a feasible approach because the execution time of the routine in Figure 3 is very small compared to the epoch size. In software implementation, the required additional hardware budget becomes minimized at the cost of run-time context switch overhead and scheduling delay. In this paper, we assume hardware implementation of hill-climbing resource scheduler.

[1]We'd better minimize the inverse of the harmonic mean, instead of maximizing the harmonic mean

| Processor Parameters | |
|---|---|
| Fetch/Issue/Commit width | 8 / 8 / 8 |
| RUU / LSQ / IFQ size | 256 / 128 / 32 |
| Functional unit | 8-Int Add, 4-Int Mul/Div, 8-Memory port, 8-FP Add, 4-FP-Mul/Div |
| **Branch Predictor Parameters** | |
| Branch Predictor | Hybrid gshare/Bimodal |
| gshare / Bimodal Predictor Size | 8192 / 2048 |
| Meta Table Size | 8192 |
| BTB Size | 2048, 4 way |
| RAS Size | 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 | 1Mkbyte, 64byte block size, 4 way, 20 cycle latency |
| Mem config | 200 cycle first chunk, 6 cycle inter chunk |

Table 1: SMT simulator settings.

## 3 Methodology

To evaluate the performance of hill-climbing resource scheduler, we developed detailed event driven SMT simulator based on SimSSMT used in [10], which is derived from SimpleScalar [11]. Table 1 lists our SMT processor settings.

The hill-climbing resource scheduler computes the amount of RUU entries allowed to each thread. When a thread consumes all of its given amount of RUU, we stall fetching the thread. In addition, we give fetch priority to a thread with largest difference between the allowed amount and the current consumption of RUU resource.

To choose the representative simulation window, we ran SimPoint [12] up to 16G instructions. And then, we picked the earliest representative region out of 16G-instruction range. Table 2 shows the number of instructions that we skipped before the performance simulation.

Table 2 lists 22 SPEC{int, fp}2000 applications used in our study. We used pre-compiled alpha binary from Chris Weaver[2], which is compiled with highest optimization level. We simulated applications using reference input set. Table 3 shows the simulation workload. We referenced two earlier researches, [4] and [5], to choose and group applications. As the table shows, we classified

[2]His SPEC2000 alpha binary is available at SimpleScalar web page.

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

Table 2: SPEC2000 applications included in the experiment.

the workload into 6 groups based on the L2 cache miss frequency (ILP, MIX, and MEM) and the number of simultaneously running threads (2 and 4). After skipping instructions from all thread, we simulated 500M instructions.

## 4 Evaluating hill-climbing resource scheduler

### 4.1 Definition of performance

We evaluate the performance of SMT resource schedulers in terms of three metrics: average IPC, average weighted IPC [2], and harmonic mean of weighted IPC [9]. They are defined in Equation 2-4, where $IPC_i$ is the IPC of thread$_i$ in SMT environment, $SingleIPC_i$ is the IPC of stand-alone execution of thread$_i$, and $N$ is the number of simultaneously running threads.

$$\text{Average\_IPC} = \frac{\sum IPC_i}{N} \tag{2}$$

$$\text{Average\_Weighted\_IPC} = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{N} \tag{3}$$

$$\text{Harmonic\_Mean} = \frac{N}{\sum \frac{SingleIPC_i}{IPC_i}} \tag{4}$$

All three metrics have their own meaning and usage: average IPC shows the throughput improvement, average weighted IPC shows the execution time reduction, and harmonic mean shows the fairness improvement. Since people may want to choose one of the above three metrics based on their demand, we show the result of our experiment in terms of the three metrics.

| ILP2 | MEM2 | MIX2 |
|---|---|---|
| apsi eon | applu ammp | applu vortex |
| fma3d gcc | art mcf | art gzip |
| gzip vortex | swim twolf | wupwise twolf |
| gzip bzip2 | mcf twolf | lucas crafty |
| wupwise gcc | art vpr | mcf eon |
| fma3d mesa | art twolf | twolf apsi |
| apsi gcc | swim mcf | equake bzip2 |

| ILP2 | MEM2 | MIX2 |
|---|---|---|
| apsi eon fma3d gcc | ammp applu art mcf | ammp applu apsi eon |
| apsi eon gzip vortex | art mcf swim twolf | art mcf fma3d gcc |
| fma3d gcc gzip vortex | ammp applu swim twolf | swim twolf gzip vortex |
| gzip bzip2 eon gcc | mcf twolf vpr parser | gzip twolf bzip2 mcf |
| mesa gzip fma3d bzip2 | art twolf equake mcf | mcf mesa lucas gzip |
| crafty fma3d apsi vortex | equake parser mcf lucas | art gap twolf crafty |
| apsi gap wupwise perlbmk | art mcf vpr swim | swim fma3d vpr bzip2 |

Table 3: SMT simulation workload classification.

## 4.2 Resource schedulers

As we mentioned, based on the choice of performance evaluation function, hill-climbing resource scheduler finds the optimal resource partition that maximizes the function value. We used three different performance evaluation functions to see how our resource schedulers faithfully pursue their own performance goals. The hill-climbing resource schedulers labeled HILL-Thru, HILL-wIPC, and HILL-hMean use Equation 2, 3, and 4, as their performance evaluation function, respectively. For example, in HILL-hMean hill-climbing resource scheduler, after we try a sample movement, we compute the harmonic mean of the previous epoch's IPC of all threads. If it is greater than the harmonic mean of past epochs, we accept the sample movement. Otherwise, we roll-back the sample movement. To implement the performance evaluation function of HILL-wIPC and HILL-hMean, we took $SingleIPC_i$ from the external input. We compare our three hill-climbing resource schedulers with some of the earlier resource schedulers: ICOUNT [2], FLUSH [4], and DCRA [5].

## 4.3 Performance of hill-climbing resource scheduler

Figure 6 compares the performance of resource schedulers in terms of average IPC, as defined in Equation 2. HILL-Thru achieves the best performance and outperforms FLUSH by 6.3% and DCRA by 3.6% on average. In addition, HILL-wIPC and HILL-hMean have as good performance

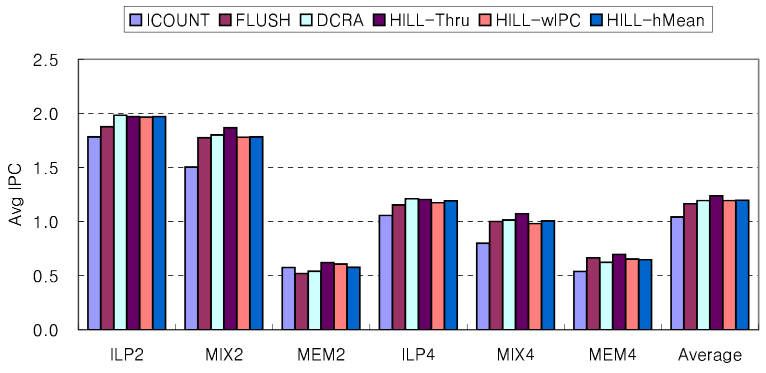

Figure 6: Average IPC of resource schedulers.

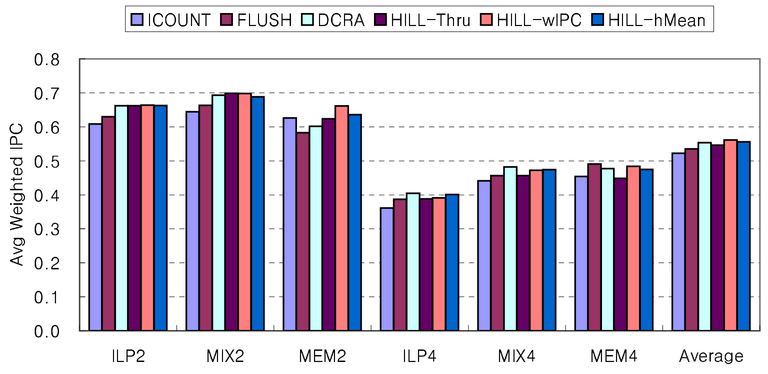

Figure 7: Average weighted IPC resource schedulers.

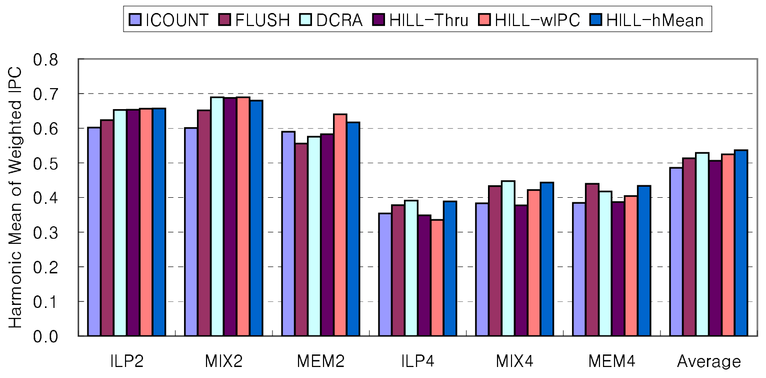

Figure 8: Harmonic mean of weighted IPC of resource schedulers.

as DCRA. Figure 7 compares the performance of resource schedulers in terms of average weighted IPC, as defined in Equation 3. As we expected, HILL-wIPC achieves the best performance and outperforms FLUSH by 5% and DCRA by 1.4% on average. Figure 8 compares the performance of resource schedulers in terms of harmonic mean, as defined in Equation 4. As we expected, HILL-hMean achieves the best performance and outperforms FLUSH by 4.5% and DCRA by 1.4% on average.

Among three types of workload groups, ILP, MIX, and MEM, hill-climbing resource scheduler has the advantage in MEM group. There are two reasons. First, hill-climbing resource scheduler can schedule resources for memory bound applications with or without memory parallelism properly, which we will explain in more detail in Section 4.5. Second, our epoch granularity resource scheduling is effective, because real resource occupancy can not catch up with finer grained resource schedule changes, if we do not allow flushing to correct the resource occupancy by a thread with data cache misses. This is true especially for MEM group applications, which, on the other hand, means that our epoch granularity scheduling is too coarse grained for ILP group applications.

As Figure 6, 7, and 8 show, DCRA performs fairly well in all three metrics. However, our three hill-climbing resource schedulers achieve the best performance in terms of the metric that each hill-climbing resource scheduler is maximizing. Considering that a single mechanism can not achieve the best performance on all three metrics, we have the flexibility of choosing a performance function for the hill-climbing resource scheduler based on the metric that we count the most.

## 4.4 Application characteristics change

The shape of the hill keeps changing. This is a unique characteristics in our hill-climbing problem because all the earlier hill-climbing based problem solvers assume fixed hill shape. Therefore, we analyzed how our resource scheduling is changing as the hill shape changes.

The source of hill shape change is the run time thread characteristics change and interactions between the simultaneously running threads. Figure 9 and Figure 10 show the correlation between the application characteristics change and the resource partition changes by the hill-climbing resource scheduler. Please note that we do not show the hill shape changes in the figures. Instead, we can guess the hill shape will change as the application characteristics change. The characteristics of the application include L1 data cache, L2 unified cache, and branch predictor miss count per epoch. In Figure 9, (a) and (b) show the application characteristics change of applu and vortex, respectively. (c) shows the RUU partition changes by the hill-climbing resource scheduler. As

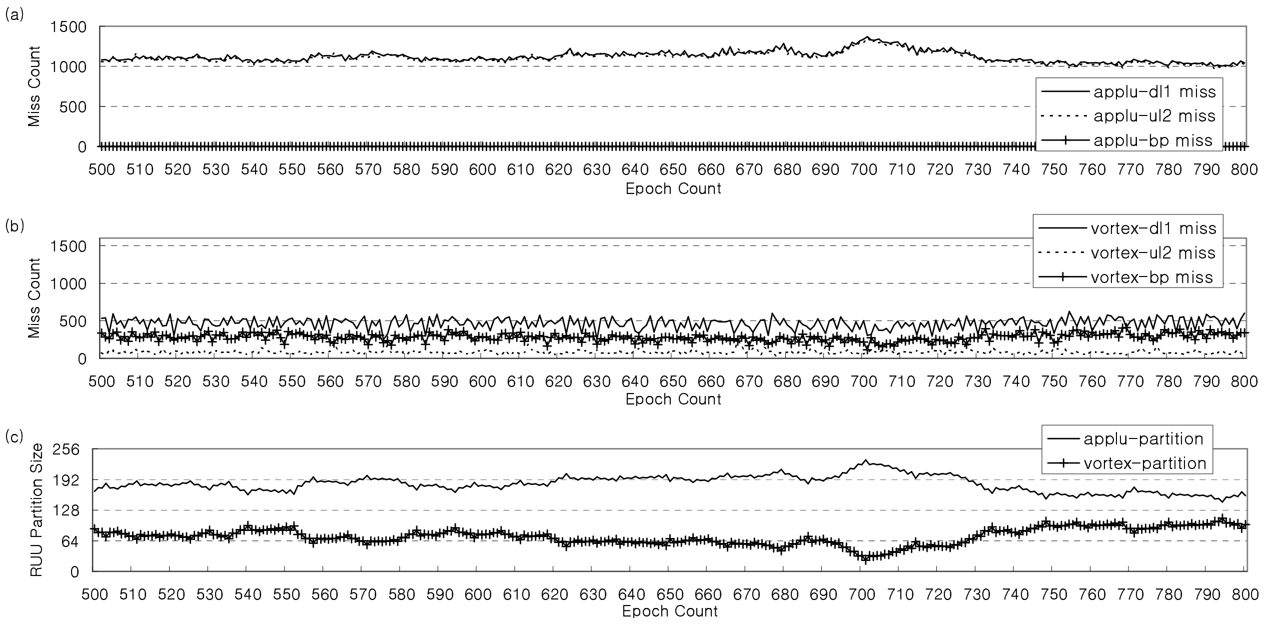

Figure 9: The correlation between application behavior and RUU partition of applu-vortex execution. (a) and (b) show application behavior of applu and vortex, respectively. (c) shows the RUU partition changes by hill-climbing resource scheduler.

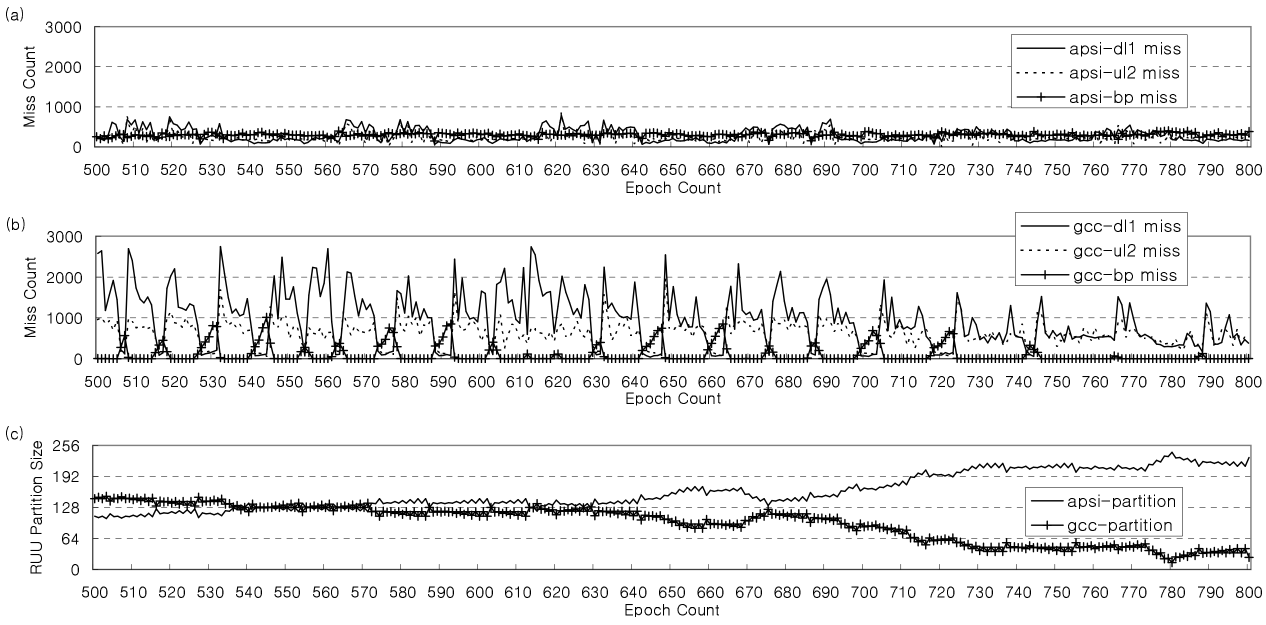

Figure 10: The correlation between application behavior and RUU partition of apsi-gcc execution. (a) and (b) show application behavior of apsi and gcc, respectively. (c) shows the RUU partition changes by hill-climbing resource scheduler.

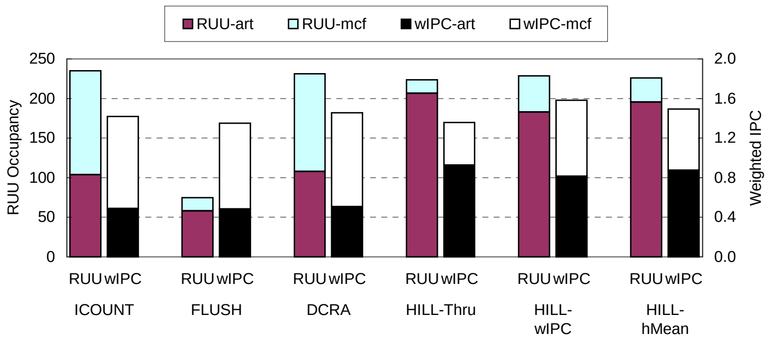

Figure 11: Average RUU occupancy and sum of weighted IPC of art-mcf execution. Segments in the bar show individual RUU occupancy and weighted IPC of two applications.

the graph shows, the RUU partition changes in the same way as the L1 data cache miss count of applu. Therefore, we can guess that our hill-climbing resource scheduler successfully follows the ever moving peak of the hill.

In Figure 10, (a) and (b) show the application characteristics of apsi and gcc, respectively. (c) shows the RUU partition changes. As the figure shows, hill-climbing resource scheduler ignores the frequent application characteristics changes, especially L1 data cache miss count of gcc changes. Instead, the resource scheduler follows the more coarse grained behavior changes. Therefore, we can guess that the shape of the hill is changing very fast and our hill-climbing resource scheduler can not catch up with the fast moving peak of the hill, losing the chance of better resource scheduling.

## 4.5 Handling memory parallelism

Figure 11 best highlights the difference between DCRA and hill-climbing resource scheduler. In this figure, each resource scheduler has two bars. The first bar shows the average RUU occupancy and the second bar shows the sum of weighted IPC. For this graph, we ran art and mcf on SMT simulator. Two segments in each bar represent the contribution of two applications. Both art and mcf are memory bound applications. In art, there are significant amount of memory parallelism because the array element accessed in each loop iteration is independent[3]. So, as we give more RUU entries to art, art exploits more memory parallelism by overlapping multiple cache misses. In mcf, however, all memory accesses are serialized via pointer chain[4] and giving more RUU entries does not help. As Figure 11 shows, DCRA treats two applications in the same way because both of them are

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

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

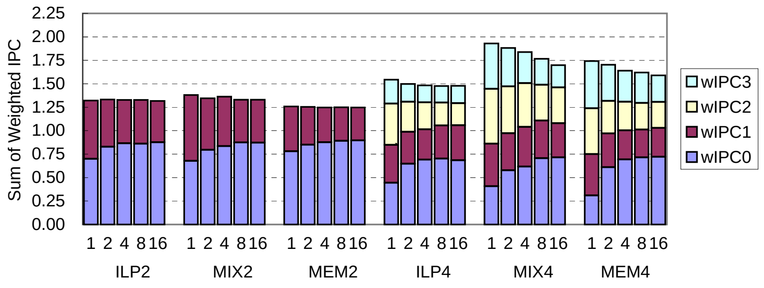

Figure 12: Weighted IPC of individual thread as we give priority to thread0 by the factor of 1, 2, 4, 8, and 16.

memory bound, allocating same amount of RUU entries to two applications. However, hill-climbing resource scheduler gives more RUU entries to art because hill-climbing resource scheduler figures out that giving more resources to art, rather than mcf, improves the performance.

## 4.6 Prioritizing threads using hill-climbing resource scheduler

Enforcing priority among multiple threads in SMT processor was studied by [13, 14]. We experimented the possibility of prioritizing threads by modifying the performance evaluation function of hill-climbing resource scheduler. Equation 5 shows the modified performance evaluation function for prioritizing threads, where $P_i$ is the externally given priority of thread$_i$.

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

Figure 12 shows our results when $P_0$ is set to be 1, 2, 4, 8, and 16, while $P_i$ ($i \neq 0$) is 1. With large value of $P_0$, small $IPC_0$ increase makes bigger improvement of the sum of prioritized IPC. Thus, the hill-climbing resource scheduler tends to improve $IPC_0$ to maximize sum of prioritized IPC. As a result, thread$_0$ gets the priority. In Figure 12, each segment in the bar represents the weighted IPC ($i.e.$ $\frac{IPC_i}{SingleIPC_i}$) of individual thread, making the height of the stack the sum of weighted IPC. The $P_0$ value of each stack is shown in x-axis. With two thread execution, the performance of thread$_0$ reaches 89.5% (MEM2-16) of the single threaded execution, degrading the performance as much as 4.7% (MIX2-16 compared to MIX2-1). With 4 thread execution, the performance of the thread$_0$ reaches 72.1% (MEM4-16) of the single threaded execution, degrading the performance as much as 12% (MIX4-16 compared to MIX4-1).

## 5 Conclusion

The goal of all the SMT resource schedulers is to reach the maximum SMT performance by finding the optimal resource share. We are the first applying the hill-climbing algorithm to achieve the resource scheduling goal. Since hill-climbing algorithm is a generic methodology for solving optimization problems, we can extend our hill-climbing resource scheduler to solve similar problems.

In this paper, we presented a novel hill-climbing resource scheduler, which carefully vary the resource share of multiple threads toward the direction which improves the SMT performance. Instead of monitoring the behavior of indicators, we use the past resource share and SMT performance change history to determine the resource share shift direction for the future. Due to the locality of performance behavior, the resource scheduling, guided by the past performance history, shows performance improvement over the earlier resource schedulers.

Depending on our performance goal, the performance evaluation function for hill-climbing resource scheduler can be either one of the average IPC, average weighted IPC [2], or harmonic mean of weighted IPC [9]. In either case, hill-climbing resource scheduler searches optimal resource share that maximizes the performance evaluation function value. The simulation result shows, our hill-climbing resource schedulers achieve the best performance in terms of the metric that each of them is maximizing. For example, a hill-climbing resource scheduler, which schedules resources toward maximizing average weighted IPC, outperforms FLUSH by 5.0% and DCRA by 1.4%, on average, using the weighted IPC as a metric. Considering that a single mechanism can not achieve the best performance on all three metrics, we can choose a performance evaluation function for the hill-climbing resource scheduler based on the metric that we count the most.

## References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton, “Hyperthreading technology architecture and microarchitecture,” in *Intel Technology Journal, 6(1)*, February 2002.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.