# An Adaptive Resource Partitioning Algorithm in SMT Processors

Huaping Wang, Israel Koren and C. Mani Krishna
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
E-mail:{hwang,koren,krishna}@ecs.umass.edu

## Abstract

*Simultaneous Multithreading (SMT) increases processor throughput by allowing the parallel execution of several threads. However, fully sharing processor resources may cause resource monopolization by a single thread or other misallocation, resulting in overall performance degradation. Static resource partitioning techniques have been suggested, but are not as effective as dynamically controlling the resource usage of each thread since program phases are not fixed all the time.*

*In this paper, we propose an Adaptive Resource Partitioning Algorithm (ARPA) that dynamically assign resources to each thread according to thread behavior changes. ARPA analyzes the resource usage efficiency of each thread in a time period and assigns more resources to threads which can use them in a more efficient way. The purpose of ARPA is to improve the efficiency of resource utilization, thereby improving overall instruction throughput. Our simulation results on a large set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy, ICOUNT by 55.8% considering overall instruction throughput, achieving 42.3% more improvement than static resource allocation policy. It also generates 7.3% more gains than the current best dynamic resource allocation technique, Hill-climbing. Considering the fairness accorded to each thread, ARPA achieves 14.0% improvement over ICOUNT and attains 7.2% and 5.9% more improvement than Static and Hill-climbing over ICOUNT respectively.*

## 1 Introduction

Simultaneous Multithreading (SMT) is an increasingly popular technique for improving overall instruction throughput by effectively countering the impact of both long memory latencies and limited available parallelism within a single thread [3, 4, 5, 16]. Through processor resource sharing, SMT takes advantage not only of the existing instruction level parallelism (ILP) within each thread but also thread level parallelism (TLP) among them. In an SMT model, all the processor resources can be shared among threads except some architecture state related resources which are separated to maintain the correct state of each logical processor.

Traditionally, a fetch policy [18] decides which threads enter the pipeline to share available resources. Threads compete for resource access and there are no individual restrictions on the resource usage of each thread. Unfortunately, threads with outstanding L2 data cache misses often run slowly as they must wait for these misses to be served. Such threads can occupy a disproportionately large share of overall system resources, and slow down other threads [17].

Statically partitioning resources to each thread has been suggested as one way of preventing a thread from clogging resources [12]. However, such techniques are limited by the fact that different threads have differing requirements, and that these can vary with time. Recently, techniques have been published do dynamically partition resources [7, 8]. Such techniques can lead to significant improvements in performance.

Resource partitioning approaches [7, 8, 12] mainly focus on some critical resources which significantly impact performance if clogged by some threads. Commonly, they apply the same partitioning principles to all the resources to be partitioned. [12] studies the effect of partitioning the instruction queue or the reorder buffer. DCRA [7] separately partitions queue and register entries using the same sharing model. Threads exceeding the specified bound are prevented from entering the pipeline. Hill-climbing [8] partitions integer rename registers among the threads, assuming that integer issue queue (IQ) and reorder buffer (ROB) will be roughly proportionately partitioned. It does not directly control the floating point IQ and the corresponding renaming registers.

In this paper, we present a new Adaptive Resource Partitioning Algorithm (ARPA) which concentrates on partitioning the following shared queue structures: instruction fetch queue (IFQ), IQ and ROB. We do not partition renaming registers since partitioning ROB can efficiently control the sharing of registers. Doing so, however, would be quite easy. We assume a shared ROB as in [7, 8] (if it is not shared, we will constrain the usage by each thread). We do not constrain the usage of individual queues. Instead, we impose an upper bound on the sum of IFQ and ROB assigned to each thread. The total number of instructions, in any thread, occupying these queues should not exceed this bound. The IQ is partitioned proportionately. Since a thread's usage of different hardware resources are dependent on each other, partitioning one type of resources will indirectly control the usage of the other resources.

The purpose of ARPA is to prevent resource underutilization and make each resource unit used efficiently, thus improving overall instruction throughput. ARPA analyzes the resource usage efficiency of each thread and assigns more resources to threads which can use them in a more efficient way. Our simulation results on a large set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy, ICOUNT by 55.8% considering overall instruction throughput, achieving 42.3% more improvement than static resource allocation policy. It also generates 7.3% more gains than the current best dynamic resource allocation technique, Hill-climbing. Considering the fairness accorded to each thread, ARPA achieves 14.0% improvement over ICOUNT and attains 7.2% and 5.9% more improvement than Static and Hill-climbing over ICOUNT respectively.

The rest of this paper is organized as follows. In the next section, we describe some related work. In Section 3 we present our adaptive resource partitioning algorithm and describe its implementation. Our evaluation methodology is presented in Section 4 followed by numerical results in Section 5. Finally, we present a summary in Section 6.

## 2 Related work

Prior related work can be categorized into three groups: fully flexible resource distribution [6, 9, 10, 18], static resource allocation [11, 12] and partly flexible dynamic resource partitioning [7, 8].

Tullsen *et al.* [16] have been pioneers in SMT architecture research and in [18] they exploit several fetch policies which determine how threads are selectively fetched to share a common pool of resources. RR is their simplest policy; it fetches instructions from all threads in Round Robin order, disregarding the resource usage of each thread. ICOUNT is a policy which dynamically biases toward threads that will use processor resources most efficiently, thereby improving processor throughput. It outperforms RR and is easy to implement. However, ICOUNT cannot prevent some threads with a high L2 miss rate from being allocated an excessive share of pipeline resources.

STALL and FLUSH [17] are two techniques built on top of ICOUNT to ameliorate this problem. STALL prevents a thread with a pending L2 miss from entering the pipeline. FLUSH, an extension of STALL, flushes all instructions from such a thread: this obviously has an energy overhead. FLUSH++ [6] combines FLUSH and STALL.

Data Gating (DG) [9] stalls threads when the number of L1 data misses exceeds a given threshold. Predictive Data Gating (PDG) [9] prevents a thread from fetching instructions as soon as a cache miss is predicted. Both techniques build upon ICOUNT to prevent resource hogging.

Static resource partitioning [11, 12] evenly splits critical resources among all threads, thus preventing resource monopolization by a single thread. However, this method lacks flexibility and can cause resources to remain idle when one thread has no need for them, even if other threads could benefit from additional resources.

DCRA [7] is a partly dynamic resource sharing algorithm. Each thread is assigned a resource usage bound and these bounds are changed dynamically. The bound is higher for threads with more L1 Data cache misses. However, DCRA does not work well on applications with high data cache miss rates and extremely low baseline performance. Allocating more resources to such threads improves their performance by very little and comes at the expense of decreased performance of other resources-starved threads.

Hill-climbing [8] uses performance feedback to direct the partitioning. This learning-based algorithm starts from equal partitioning, then moves an equal amount of resources from all the other threads to a "trial" thread. Hill-climbing appears to be the best resource partitioning technique currently available.

Like DCRA [7] and Hill-climbing [8], our algorithm, ARPA, also partitions resources dynamically. However, ARPA's analysis of program behavior results in a more effective use of resources.

## 3 ARPA: Adaptive Resource Partitioning Algorithm

### 3.1 Framework

Figure 1 shows a high-level flow chart of how ARPA works. We divide the whole program execution into fixed-sized epochs (measured in processor cycles) and start with equally partitioned resources among the threads. After each epoch, we analyze the current resource usage to identify whether the threads have used their allocated resources efficiently in this epoch. Our resource partitioning decision is driven by these analyses. The analysis and partitioning process will be repeated every epoch until the end of the program.

### 3.2 Resource Utilization Analysis

In an SMT processor, the overall instruction throughput is not only determined by the set of threads running simultaneously, but is also significantly affected by the sharing scheme among threads. The objective of ARPA is to increase the performance of processors by improving the resource usage efficiency of their resources. In the next several sections, we will describe how we analyze the resource usage efficiency and use the analysis to drive the partitioning.

#### 3.2.1 Metric of Usage Efficiency

As shown in Figure 1, the whole program execution is divided into thousands of equal intervals (epochs). Our utilization analysis is carried out interval by interval.

We use Committed Instructions Per Resource Entry (CIPRE) to represent the usage efficiency of processor

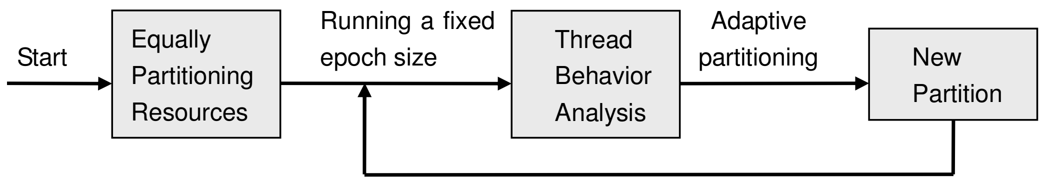


**Figure 1. A simple description of the ARPA algorithm**

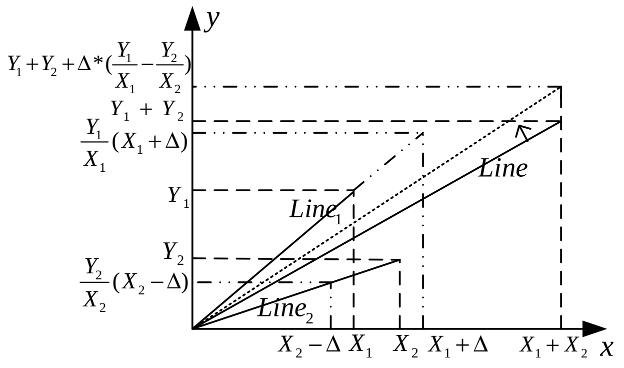


**Figure 2. An example to illustrate the CIPRE changes after a specific epoch, $n + 1$**

resources in each interval. It is important to note that CIPRE can express two different characteristics: (a) the usage efficiency of all processor resources and (b) the usage efficiency of the resources which are allocated to the specific thread. We focus on the former.

Note that a thread with a higher CIPRE does not necessarily have a higher IPC. For example, given threads A and B running simultaneously with 50 and 20 queue entries, respectively, suppose thread A commits 2000 instructions and thread B commits 1000 instructions during an epoch of length 1000 cycles. Therefore, the IPC of thread A is 2 and that of thread B is 1; the CIPRE of thread A is $\frac{2000}{50} = 40$ while the CIPRE of thread B is $\frac{1000}{20} = 50$. Because the CIPRE of B is greater than that of A, we say that thread B is more productive in this epoch. Resources allocated to B contribute more per unit to system performance than resources allocated to A although the IPC of A is greater than that of B. Therefore, giving more resources to the higher-CIPRE thread does not necessarily mean giving the high-IPC thread more resources.

### 3.2.2 Partitioning Process

ARPA follows an adaptive resource partitioning strategy. It adjusts the number of resources allocated to threads at each epoch based on the CIPRE metric. Threads with a greater CIPRE value will take some resources from threads with a lower CIPRE in every epoch until the CIPREs of the two threads are close to each other. That is to say, all threads can use their allocated resources with approximately equal efficiency, thus improving the usage efficiency of all processor resources. At the same time, resource starvation will be avoided by giving each thread a minimum number of resources.

We use a two-thread example shown in Figure 2 to illustrate this process. Figure 2 shows the change in the CIPRE value when a program completes epoch $n + 1$. $X_1$ and $X_2$ are the number of resource entries allocated to threads 1 and 2, respectively. $Y_1$ and $Y_2$ are the number of committed instructions of threads 1 and 2 during epoch $n$, respectively. $\Delta$ is the number of resource entries that a thread can transfer to another thread in any one epoch.

The CIPREs of threads 1 and 2, and the CIPRE of all processor resources at the end of epoch $n$ are shown below in (1), (2) and (3), respectively; we assume $CIPRE_1 > CIPRE_2$ in this example.

$$CIPRE_1 = \frac{Y_1}{X_1} \tag{1}$$

$$CIPRE_2 = \frac{Y_2}{X_2} \tag{2}$$

$$CIPRE_{Epoch\_n} = \frac{Y_1 + Y_2}{X_1 + X_2} \tag{3}$$

It is easy to show that

$$CIPRE_2 < CIPRE_{Epoch\_n} < CIPRE_1 \tag{4}$$

Since thread 1 achieves a more efficient usage of the allocated resources in epoch $n$, ARPA will transfer to it $\Delta$ resources from thread 2 in the next epoch. That is to say, thread 1 will be assigned $X_1 + \Delta$ resource entries and thread 2 will be restricted to $X_2 - \Delta$ resource entries in epoch $n + 1$. If both threads 1 and 2 still use their allocated resources with the same efficiency as in epoch $n$, the CIPRE of the total resources in epoch $n + 1$ will be:

$$CIPRE_{Epoch\_n+1} = \frac{Y_1 + Y_2 + \Delta * (\frac{Y_1}{X_1} - \frac{Y_2}{X_2})}{X_1 + X_2} \tag{5}$$

Compared with the no-adjustment case which has the same CIPRE as in (3), after epoch $n+1$, the CIPRE of all resources is increased by

$$I = \frac{\Delta * (\frac{Y_1}{X_1} - \frac{Y_2}{X_2})}{X_1 + X_2} \tag{6}$$

Whenever $CIPRE_1$ is greater than $CIPRE_2$, resources continue to be transferred from thread 2 to thread 1 in subsequent epochs as long as each thread has at least its specified minimum allocation. The CIPRE keeps increasing and getting ever closer to $CIPRE_1$ (but will not exceed $CIPRE_1$). That is to say, $Line$ is getting closer to $Line_1$ as shown in Figure 2.

As thread 1 obtains more resources, its resource usage efficiency, *i.e.*, $CIPRE_1$, will tend to decrease. At the same time, $CIPRE_2$ will increase gradually (as the number of resources allocated to a thread reduces, the usage efficiency of the remaining resources will increase). In one situation, $CIPRE_1$, $CIPRE_2$ and $CIPRE$ will be getting closer and closer ($Line_1$, $Line_2$ and $Line$ in Figure 2 will nearly overlap). Both threads can use the allocated resources at the same efficiency and the CIPRE of all resources reaches its optimal value. Another situation is that the CIPRE value of thread 1 may still be greater than that of thread 2 even when thread 1 takes all the resources it can from thread 2. As mentioned previously, ARPA assures each thread a certain minimum number of resources to avoid resource starvation or under-utilization of threads in such situation.

ARPA improves the resource usage efficiency of all resources by allocating more resources to the high-CIPRE thread but still avoids resource starvation. A more detailed analysis is presented in Section 5.

## 3.3 Partitioning Algorithm

Figure 3 presents the pseudocode of ARPA. At the end of an epoch, the *ComInsts* function computes the CIPRE value of each thread: this is the number of committed instructions divided by the total number of IFQ and ROB entries allocated to the thread in the current epoch. We then compare the CIPRE of each thread and select the thread with the greatest CIPRE as the reference thread (if the CIPREs of the two threads are equal, we do not move resources between them in the next epoch). The reference thread can take `STEP` entries of IFQ and ROB from every other thread. IQ entries are also proportionately moved.

## 3.4 Implementation of ARPA

Figure 4 shows how to implement ARPA. The top layer in Figure 4 is the baseline SMT processor structure used in our study. We do not modify this part.

The middle layer lists the counters and comparators we add to the processor for each thread, which will be used for resource partitioning using ARPA. We need one *In-flight_Instructions_Counter* per thread to monitor the current usage of queue entries by each thread. The counter will be incremented as instructions are fetched and decremented as instructions are committed. The *Committed_Instructions_Counters* are used to count the committed instructions for each thread in the current epoch. A *Committed_Instructions_Counter* will be reset to zero at the start of each epoch while an *In-flight_Instructions_Counter* will not be reset during the execution of a thread. We use one comparator per thread to determine if the current resource usage of the thread has already exceeded its specified bound; if so,

```
#define EPOCH        Fixed for the entire execution
#define Num          Number of running threads
#define ComInsts(x)  Compute CIPRE[x]
#define max(A,n)     Get the index of the maximum
                     value in the array A[0:n]
#define STEP         Number of queue entries moved
                     at each comparison

For every EPOCH cycles{
   //compute CIPRE of each thread.
   for(tid = 0; tid < Num; tid ++){
       CIPRE[tid] = ComInsts(tid);
   }
   // select the reference thread
   Ref_tid = max(CIPRE, Num);
   // assign resources
   for(tid = 0; tid < num; tid ++){
       Partition[Ref_tid] += STEP;
       Partition[tid] -= STEP;
   }
}
```

**Figure 3. ARPA pseudo-code**

a throttling signal will be generated to throttle further fetching for this thread.

The bottom layer is the implementation of the algorithm. At the end of each epoch, we run the resource allocation algorithm. The resources upper bound assigned to each thread is saved in its Partition Register. At the start of the program, this assignment is set to be equal for every thread. In every epoch the Partition Registers will be read and the CIPRE computed for each thread. Based on this value, a new partition will be generated and these new partition values will be updated in the Partition Registers. As was done in [8], we suggest to implement this in software. At the end of each epoch, an interrupt signal can be sent to one of the application threads, using its hardware context to execute the partitioning algorithm. The overhead of running the algorithm is considered in this paper in the same way as in [8].

## 4 Evaluation Methodology

## 4.1 Configuration

Our simulator is based on Simplescalar [2] for the Alpha AXP instruction set with Wattch [1] power extensions. We modified SimpleScalar to support simultaneous multithreaded processors. Moreover, we have decoupled the centralized Register Update Unit (RUU) structure adapted by SimpleScalar and have separate issue queue, reorder buffer and physical registers. Our baseline processor configuration is shown in Table 1. Other detailed features are based on the SMT architecture of Tullsen et al. [18].

Out simulator adds support for the dynamic partitioning of the fetch queue and reorder buffer. We keep counters for the number of In-flight instructions (which are the instructions in the fetch queue and reorder buffer) per thread, allowing a thread to fetch instructions as long as its In-flight instructions have not

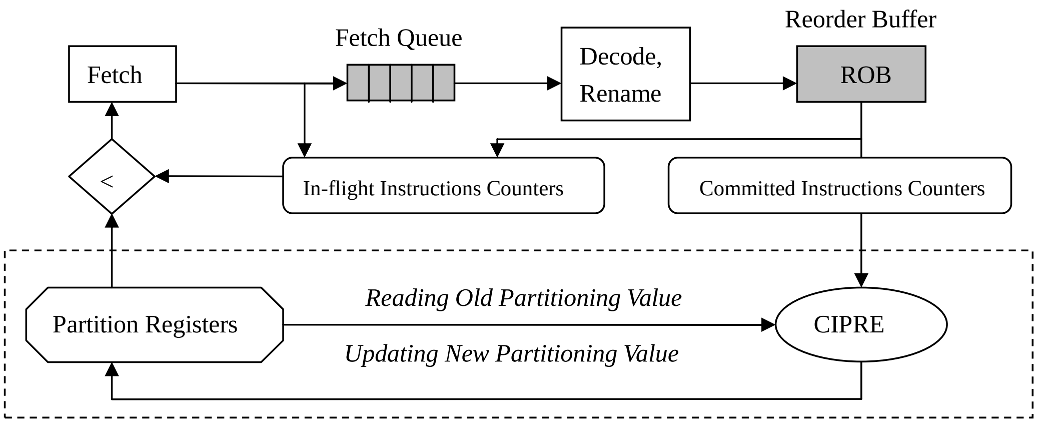

Fetch
Fetch Queue
Decode, Rename
Reorder Buffer
ROB
<
In-flight Instructions Counters
Committed Instructions Counters
Partition Registers
Reading Old Partitioning Value
Updating New Partitioning Value
CIPRE

**Figure 4. An implementation of ARPA**

**Table 1. Baseline parameters**

| Parameter | Value |
|---|---|
| IF,ID,IS Width | 8-way |
| Queue size | 32 IFQ, 80 IQ, 64 LSQ |
| Functional Units | 6 Int, 4 FP, 4 ld/st<br>2 Int Mul/Div, 2 FP Mul/Div |
| Physical Registers | 256 Int, 256 FP |
| Reorder Buffer size | 256 entries |
| BTB | 2048 entries, 4-way associative |
| Branch Prediction | 4K entries gshare,<br>10-bit global history |
| L1 D-cache | 128KB, 4-way, writeback |
| L1 I-cache | 128KB, 4-way, writeback |
| Combined L2 cache | 1MB, 4-way associative |
| L2 Cache hit time | 20 cycles |
| Main memory hit time | 300 cycles |

**Table 2. Twenty two SPEC CPU2000 benchmarks used in this study.**

| App | # skipped (in millions) | Type | App | # skipped (in millions) | Type |
|---|---|---|---|---|---|
| mcf | 4000 | MEM | gcc | 1000 | ILP |
| lucas | 2000 | MEM | wupwise | 2500 | ILP |
| applu | 500 | MEM | vortex | 0.5 | ILP |
| equake | 3400 | MEM | gap | 65 | ILP |
| twolf | 400 | MEM | mesa | 250 | ILP |
| vpr | 1150 | MEM | perlbmk | 500 | ILP |
| art | 2900 | MEM | gzip | 40 | ILP |
| swim | 250 | MEM | crafty | 10 | ILP |
| parser | 250 | MEM | bzip2 | 200 | ILP |
| ammp | 2600 | MEM | eon | 3 | ILP |
| apsi | 30 | ILP | fma3d | 3000 | ILP |

exceeded its assigned limit. The counter for In-flight instructions is similar to that in [18] for implementing the ICOUNT fetch policy. When the number of In-flight instructions exceeds the assigned bound, we apply fetch throttling [19, 20] to this thread until it releases some of its entries or is allocated more resources. The issue queue will be partitioned proportionally with these two queue structures.

We use the ICOUNT fetch policy to fetch instrucitons. Other parameters are set as shown in Table 1.

## 4.2 Workloads

Table 2 lists the benchmarks used in our simulations. All benchmarks are taken from the SPEC2000 suite and use the reference data sets. We use the pre-compiled alpha binaries from C. Weaver source:(www.simplescalar.com); these binaries are built with the highest level of compiler optimization. From these 22 benchmarks, we created multiprogrammed workloads following the methodology proposed in [7, 8, 17]. SPEC benchmarks are first categorized into memory-bound and computation-bound programs (represented by MEM and ILP, respectively, in Table 2). Based on the MEM or ILP character of different benchmarks, we create our multiprogrammed workloads with 2-benchmark and 4-benchmark combinations as shown in Table 3. All the workloads are labeled to indicate the character and number of threads, as well as a number to distinguish one workload from another. MIX workloads select half of their threads from ILP and the other half from MEM. We select simulation regions of different benchmarks based on [13] as shown in Table 2. We simulate 100 million instructions for each benchmark in the workload.

## 4.3 Metrics

Measuring the performance of a single thread is simple, but for multithreaded workloads, things become more complicated. We need to consider not only the overall throughput of the processor but also the fairness accorded to each thread running on the processor. Several performance metrics have been proposed to measure SMT performance in the past years: however, no one measure has emerged as a standard. In our paper, we therefore show the throughput and fairness results quantified by each of three metrics to give a comprehensive comparison of different algorithms. The metrics are explained as follows.

**Table 3. Benchmark combinations based on cache behavior of threads.**

| Name | Combinations | Name | Combinations | Name | Combinations |
|---|---|---|---|---|---|
| MEM.2.1 | applu, ammp | MIX.2.1 | applu, vortex | ILP.2.1 | apsi, eon |
| MEM.2.2 | art, mcf | MIX.2.2 | art, gzip | ILP.2.2 | fma3d, gcc |
| MEM.2.3 | swim, twolf | MIX.2.3 | wupwise, twolf | ILP.2.3 | gzip, vortex |
| MEM.2.4 | mcf, twolf | MIX.2.4 | lucas, crafty | ILP.2.4 | gzip, bzip2 |
| MEM.2.5 | art, vpr | MIX.2.5 | mcf, eon | ILP.2.5 | wupwise, gcc |
| MEM.2.6 | art, twolf | MIX.2.6 | twolf, apsi | ILP.2.6 | fma3d, mesa |
| MEM.2.7 | swim, mcf | MIX.2.7 | equake, bzip2 | ILP.2.7 | apsi, gcc |
| MEM.4.1 | ammp, applu, art, mcf | MIX.4.1 | ammp, applu, apsi, eon | ILP.4.1 | apsi, eon, fma3d, gcc |
| MEM.4.2 | art, mcf, swim, twolf | MIX.4.2 | art, mcf, fma3d, gcc | ILP.4.2 | apsi, eon, gzip, vortex |
| MEM.4.3 | ammp, applu, swim, twolf | MIX.4.3 | swim, twolf, gzip, vortex | ILP.4.3 | fma3d, gcc, gzip, vortex |
| MEM.4.4 | mcf, twolf, vpr, parser | MIX.4.4 | gzip, twolf, bzip2, mcf | ILP.4.4 | gzip, bzip2, eon, gcc |
| MEM.4.5 | art, twolf, equake, mcf | MIX.4.5 | mcf, mesa, lucas, gzip | ILP.4.5 | mesa, gzip, fma3d, bzip2 |
| MEM.4.6 | equake, parser, mcf, lucas | MIX.4.6 | art, gap, twolf, crafty | ILP.4.6 | crafty, fma3d, apsi, vortex |
| MEM.4.7 | art, mcf, vpr, swim | MIX.4.7 | swim, fma3d, vpr, bzip2 | ILP.4.7 | apsi, gap, wupwise, perlbmk |

$$Avg\_IPC = \frac{\sum IPC_i}{T} \tag{7}$$

Avg_IPC metric only quantifies the overall throughput improvement and doesn't taken fairness into consideration. Therefore, the problem of this metric is that it may boost IPC by starring some threads.

$$Avg\_Single\_Weighted\_IPC = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{T} \tag{8}$$

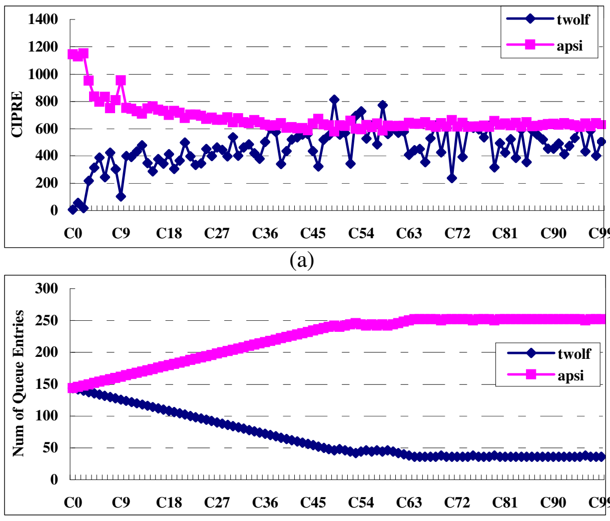
Avg_Single_Weighted_IPC metric weights IPC on each thread's behalf to its nature IPC if run alone and reflects the fairness accorded to each thread. However, the drawback of this metric is it does not give any importance to the overall throughput and may bias against the thread with very low IPC. For example, consider thread A with single thread IPC 3.0 and thread B with single thread IPC 0.1 running simultaneously, with thread A achieving 1.5 IPC and thread B 0.09 IPC with the ICOUNT fetch policy and Static Partitioning achieves 2.1 IPC and 0.06 IPC respectively. The Avg_Single_Weighted_IPC of Static Partitioning is 7.1% slower than that of ICOUNT although the overall throughput of Static Partitioning is much better.

$$Avg\_Baseline\_Weighted\_IPC = \frac{\sum_{threads} \frac{IPC_{new}}{IPC_{baseline}}}{T} \tag{9}$$

Avg_Baseline_Weighted_IPC weights IPC of each thread with IPC of corresponding thread in the baseline scheme or the reference scheme. It reflects the change in IPC of each thread for the optimized scheme compared to the baseline scheme. Regardless of how each thread would run in single thread mode, Avg_Baseline_Weighted_IPC benefits from any thread running faster.

## 5 Results and Analysis

We first illustrate the adaptive nature of ARPA through an example. Then, we compare ARPA improvement with other schemes using three different metrics introduced in Section 4.3 across the 42 workloads.



(b)

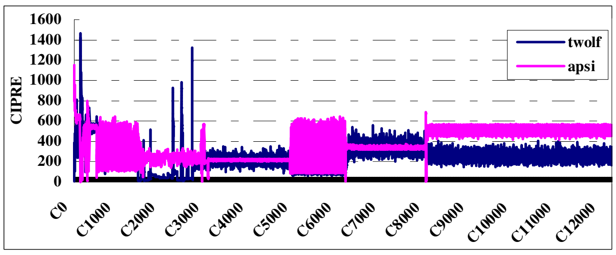
**Figure 5. An example illustrating the adaptive nature of ARPA for epochs 0 to 99.**

Finally, we provide a sensitivity analysis of ARPA to the STEP, EPOCH and the size of queue entries.

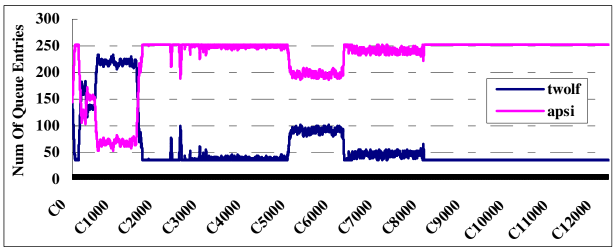
### 5.1 Adaptive Process

Figure 5 illustrates the adaptive nature of resource partitioning by ARPA. Threads *twolf*, a memory-bound program and *apsi*, a computation-bound program are running simultaneously. Figure 5(a) displays the CIPRE changes of these two threads for the epochs 0 to 99 when using ARPA, while Figure 5(b) shows the resulting partitioning of queue entries between the two threads for epochs 0 to 99.

In the first epoch, we equally partition resources to *twolf* and *apsi*, as indicated in the Figure 5(b). The CIPRE of *twolf* is higher than that of *apsi* in this epoch, and in epoch 2, *twolf* can take $\Delta$ queue entries from *apsi*. We can see that the CIPRE of *twolf* is bigger than that of *apsi* until epoch 49. Therefore, *twolf* will take $\Delta$ queue entries from *apsi* at each epoch until epoch 49. The allocated number of queue entries of *twolf* increases



(a)



(b)

**Figure 6. An example illustrating the adaptive nature of ARPA.**

linearly while the allocated number of queue entries of *apsi* decreases linearly during this time period. Now the CIPREs of the two threads have become close to each other in epoch 49. In other words, the two threads are using their allocated resources with similar efficiency. Between epoch 49 and 63, *twolf* takes back and forth some number of queue entries from *apsi*. At epoch 64, the number of queue entries of *apsi* reaches its minimum. Although the CIPREs of *twolf* beat those of *apsi* most of time after epoch 64, *twolf* cannot take resources from *apsi* in order to prevent resource under-utilization. The number of queue entries become stable for each thread and the resources allocated to each thread will remain in this setting if no big phase changes occur.

Figure 5 provides a detail of the tuning process over a short time period (100 epochs). In order to understand the resource adaptation process during a long program execution time, we also display in Figure 6(a) and Figure 6(b), respectively the CIPRE changes and the corresponding resource partitions of these two threads for the whole program execution which lasts 12206 epochs.

There are different stable resource allocation phases during the execution of the workload combination *twolf* and *apsi* as indicated in Figure 6. The first stable phase comes after the resource tuning process as shown in Figure 5 and lasts short. The second stable phase is also short compared to the following five phases; during this phase, the number of resources allocated to each thread are close to each other. In the third tuning process phase, CIPREs of *twolf* beat that of *apsi* in most epochs, which allows *twolf* to own more resources to improve resources usage efficiency. The similar tuning process happens at the start of the next three stable phases. In the final stable phase, although the CIPREs of *twolf* are bigger than those of *apsi* all the time, resource allocations are fixed since the number of queue entries of *apsi* reaches low-bound limitation. As we can see, a static resource partitioning can not satisfy these varied program phases. ARPA tunes resource based on the contribution of those resources to performance and grasp program phase changes, thus improves performance.

## 5.2 ARPA Improvement

Figure 7 compares the *Avg_Single_Weighted_IPC* of different schemes across the 42 workloads listed in Table 3. The schemes include ICOUNT, Static, Hill-climbing, ARPA. The Epoch size we used in these experiments is 32K cycles and STEP size is 2 queue entries. We allow each thread to keep at least quarter number of equally partitioned queue entries to avoid resource starvation.

From Figure 7, we see that ARPA outperforms ICOUNT and static partitioning significantly in MEM and MIX workloads. For some workloads like MEM.2.5, MIX.4.2, the improvement of ARPA over ICOUNT and Static is more than 50%. The ICOUNT policy gives priority to threads which move faster through the pipeline, *i.e.*, threads which have an efficient resource usage. However, ICOUNT cannot constrain threads from clogging resources, resulting in poor performance when this happens. Because the memory-bound threads in MEM and MIX workloads more readily clog resources than do computation-bound threads in ILP workloads, we can see from Figure 7 that the improvement in MEM and MIX workloads is much greater than that in ILP workloads in both 2-thread workloads and 4-thread workloads. Static partitioning can prevent resource monopolization by a single thread. This characteristic benefits especially to the resource tight situation since the possibility of resource monopolization increases when the number of resources reduce. From the Figure 7 we can see, Static achieves more improvement over ICOUNT in 4-thread workloads than in 2-thread workloads. However, it does not consider program phase changes and the needs of individual threads. As a result, the performance improvement of ARPA over Static is considerable.

Hill-climbing [8] is the best-performing algorithm from the literature. From Figure 7, we can see that Hill-climbing outperforms ICOUNT and Static significantly in MIX workloads. Using the *Avg_Single_Weighted_IPC* metric, Hill-climbing achieves a 8.1% improvement over ICOUNT, close to the results published in [8], increasing our confidence in the precision of our implementation of this algorithm. However, because Hill-climbing does not analyze the behavior of individual threads and makes its decisions based only on periodic trials, there is scope for further improvement. Figure 7 shows that ARPA outperforms Hill-climbing in all but 8 of the 42 workloads.

Figure 8 shows the *Avg_IPC* improvement and *Avg_Single_Weighted_IPC* improvement of different schemes over ICOUNT respectively. The figure averages the MEM, MIX, ILP workloads separately in both 2-thread and 4-thread workloads.

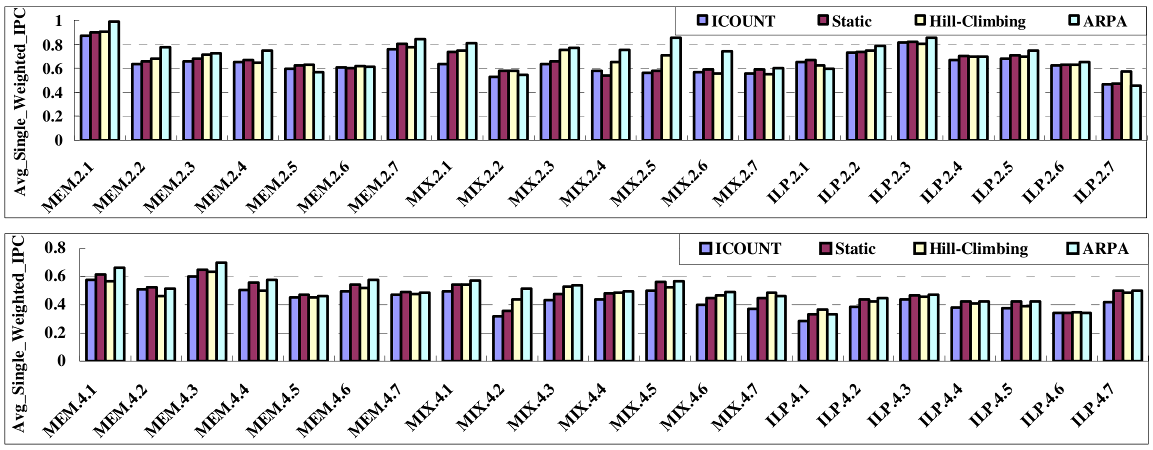


**Figure 7. Avg_Single_Weighted_IPC of different schemes in 42 workloads.**

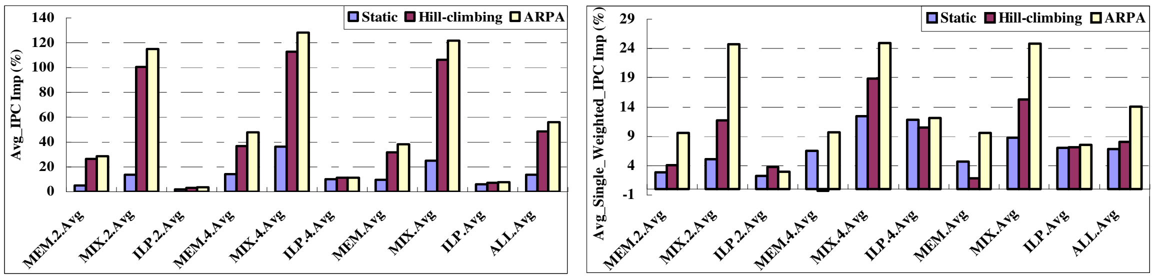


**Figure 8. Avg_IPC improvement of different schemes over ICOUNT**

The *Avg_IPC* improvement of Static, Hill-climbing and ARPA over ICOUNT is much better than the *Avg_Single_Weighted_IPC* improvement for the MEM and MIX groups. For example, Hill-climbing achieves 100.3% *Avg_IPC* improvement over ICOUNT, but using the *Avg_Single_Weighted_IPC* metric, it only gets 11.7% improvement for MIX.2.Avg. However, for ILP groups, the improvement of the two metrics are close to each other. This is caused by the metric characteristics as explained in Section 4.3. Static, Hill-Climbing and ARPA control the resource utilization by clogging threads to improve the overall throughput. Since the clogging thread which takes more time on using clogged resources usually has a low single-thread IPC (more aggressive clogging, less IPC), a small absolute IPC reduction will result great weighted IPC reduction. That is to say, the weighted IPC improvement of one thread can not make up for the weighted IPC loss of another thread for the *Avg_Single_Weighted_IPC* metric although the overall throughput increases greatly.

The *Avg_IPC* improvements of 2-thread workloads are less than that of 4-thread workloads for all three schemes. It is obvious that with the same number of resources, 4-thread workloads result in greater resource clogging than 2-thread workloads. The improvement of MIX workloads is much greater than that of MEM and ILP workloads. By preventing the resource clogging of memory-bound threads in MIX workloads, the optimized scheme can greatly improve the performance of the computation-bound threads, thus improving the overall throughput. The optimized scheme is especially beneficial in clogging aggressive and resource-tight situations.

ARPA performs much better than Static and Hill-climbing in MEM and MIX workloads but shows no significant advantages for ILP workloads compared to these two schemes. The reason is that computation-bound threads require fewer resources to exploit ILP, Static and Hill-climbing have provided optimization to great extent.

Figure 8 shows that with *Avg_IPC* metric, Static, Hill-Climbing and ARPA achieve 13.5%, 48.5% and 55.8% improvement over ICOUNT, respectively. With *Avg_Single_Weighted_IPC* metric, they also achieve 6.8%, 8.1% and 14.0% improvement over ICOUNT. ARPA achieves 7.3% and 5.9% more improvement than the current best-performing algorithm, Hill-climbing, as expressed by the *Avg_IPC* metric and *Avg_Single_Weighted_IPC* metric respectively.

As mentioned on Section 4.3, the drawback of *Avg_Single_Weighted_IPC* is that it does not give any importance to the overall throughput and may bias against threads with very low IPC. However, regardless of how each thread would run in single thread mode, *Avg_Baseline_Weighted_IPC* reflects the improvement of optimized scheme over the baseline scheme. Figure 9 shows the *Avg_Baseline_Weighted_IPC* speedup of Static, Hill-climbing and ARPA over the ICOUNT baseline. ARPA achieves a better *Avg_Baseline_Weighted_IPC* speedup than Static and Hill-climbing for almost all workloads in MEM and MIX groups; the improvement in ILP group is not significant

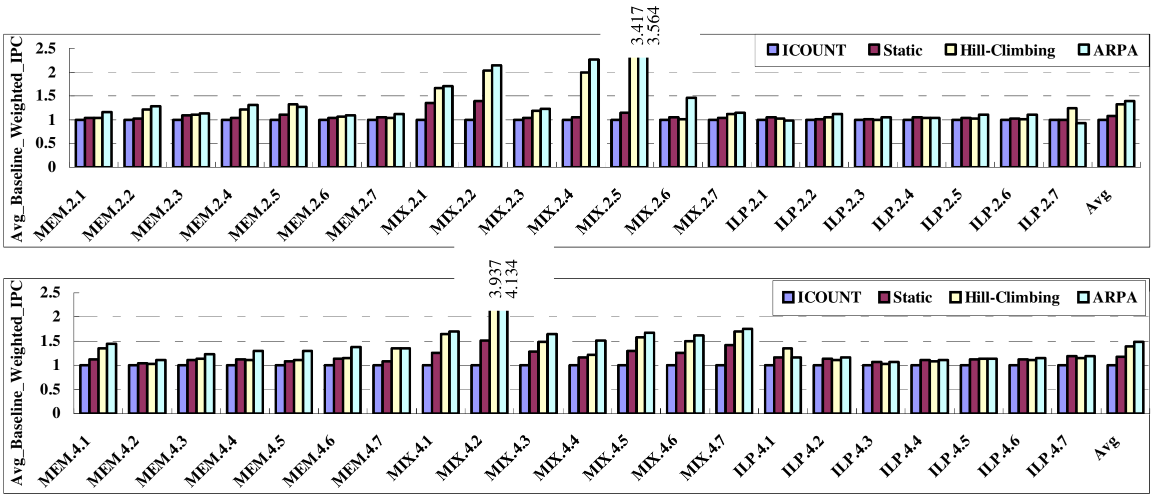
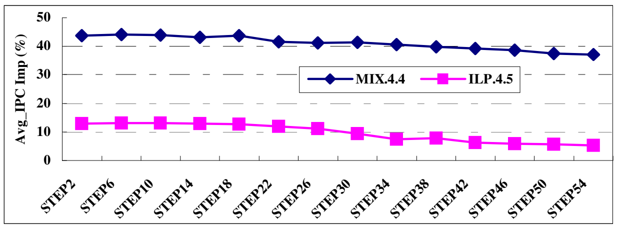
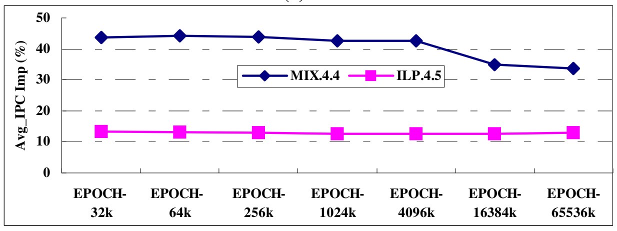


**Figure 9. Avg_Baseline_Weighted_IPC of different schemes in 42 workloads.**



(a)



(b)

**Figure 10. Avg_IPC improvement of ARPA over ICOUNT as EPOCH and STEP sizes change.**

compared with that in the MEM and MIX groups. Static, Hill-climbing and ARPA respectively achieve 7.8%, 32.6% and 39.1% improvement in 2-thread workloads, and 18.2%, 39.4% and 48.0% improvement in 4-thread workloads. ARPA outperforms both Static and Hill-climbing. Compared to Static, it achieves 30.6% more *Avg_Baseline_Weighted_IPC* improvement and compared to Hill-climbing, it achieves 7.6% more *Avg_Baseline_Weighted_IPC* improvement.

## 5.3 Sensitivity Analysis

In this section, we study the impact of the EPOCH and STEP sizes for ARPA. Then, we compare the ARPA with other schemes when the amount of resources changes.

### 5.3.1 Epoch and Step Size

Figure 10(a) and Figure 10(b) show the average IPC improvement of ARPA over ICOUNT as the STEP size changes from 2 to 54 with EPOCH size fixed to 256K, and as the EPOCH size changes from 32K to 65536K with STEP size fixed to 2. We focus on two representative 4-thread workloads from the 42 workloads to show the sensitivity of ARPA to STEP and EPOCH sizes.

From Figure 10 (a) we can see that as the STEP size increases from 2 to 18, the performance improvement of ARPA over ICOUNT is roughly the same. It drops as we continue to increase the STEP size beyond STEP 18. With a small STEP size, ARPA may take a slightly long tuning time (if EPOCH size is not too big) to the optimal partitioning, which has no big effect on performance. However, if STEP size is big, partitioning may miss some optimal level, thus causing performance loss.

From Figure 10 (b) we can see that as the EPOCH size is increased from 32K to 65536K cycles, MIX.4.4 and ILP.4.5 exhibit a different IPC improvement tendency: the performance improvement of MIX.4.4 has significant reduction as EPOCH size is increased beyond 4096k, while for ILP.4.5, the performance improvement is roughly constant for all tested EPOCH sizes. The reason is that ILP.4.5 performs well when resources are equally partitioned; since ARPA starts with equal resource partitioning, there is no significant performance decrease even with great EPOCH sizes for ILP.4.5. However, equally partitioned resources is not ideal for MIX.4.4. ARPA cannot grasp the thread behavior changes accurately when using a big EPOCH size and this causes performance loss.

### 5.3.2 Queue Entries

We will now examine the impact of the size of LSQ, IQ and ROB on performance for different schemes.

Figure 11 shows the average IPC of ARPA versus ICOUNT, Static and Hill-climbing when LSQ, IQ and ROB increase from (32, 40, 128) to (96, 120, 384).

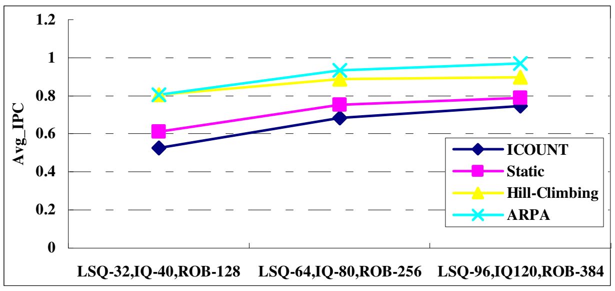


**Figure 11. Avg_IPC of different schemes as the number of queue entries changes.**

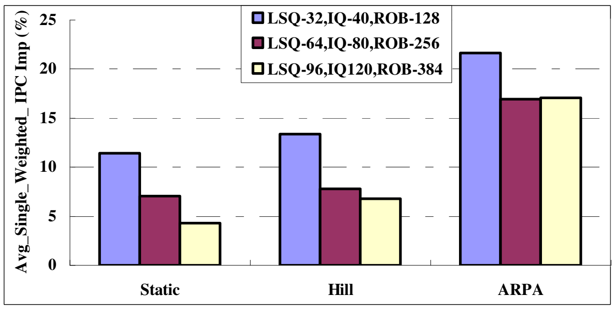


**Figure 12. Avg_Single_Weighted_IPC improvement of different schemes over ICOUNT as the number of queue entries changes.**

From Figure 11 we can see that average IPC improves when the size of queue entries increases from (32, 40, 128) to (96, 120, 384) for each scheme. However, the degree of improvement declines when the size of queue entries is increased from (64,80,256) to (96,120,384) because of the ILP limitation of workloads. As the number of queue entries is increased to (96,120,384), the average IPCs of Static and ICOUNT draw close to each other since an increased number of resources reduces the resource clogging of ICOUNT and the benefit of resource sharing of ICOUNT over resource partitioning of Static exhibits.

ARPA and Hill-climbing performs much better than ICOUNT and Static in all resource sizes since the partly flexible dynamic resource partitioning can combine the resource sharing benefit of ICOUNT and resource partitioning benefit of Static, thus improve performance significantly.

Average IPCs of ARPA and Hill-climbing at (32, 40 and 128) are close to each other, while as the size of queue entries increases, the degree of performance improvement of ARPA is bigger than that of Hill-climbing. The reason is that Hill-climbing is a learning-based algorithm and the increased number of resources will increase the search space, thereby increasing the learning time and the chances being trapped on local-maxima.

Figure 12 compares the average single_execution weighted IPC improvement of different schemes over ICOUNT as LSQ, IQ and ROB increase from (32, 40, 128) to (96, 120, 384). From Figure 12, we can see that as the number of resources increases, the improvements over ICOUNT decrease for almost all the schemes. It is obvious that the increased number of resources mitigates the resource clogging and decrease the optimization space. The performance improvement of ARPA over ICOUNT is biggest among all the schemes in any number of queue sizes, achieving 21.7%, 16.9% and 17.1%, respectively, while Hill-climbing only achieves 13.4%, 7.8% and 6.8% improvement.

## 6 Conclusion

This paper proposes an Adaptive Resource Partitioning Algorithm (ARPA) for SMT processors. The algorithm identifies the resource usage efficiency of each thread using the CIPRE metric and gives more resources to threads which can use them in a more efficient way. The efficient usage of processor resources greatly improves the overall instruction throughput. Our experimental results show that ARPA improves *Avg_IPC* by 55.8% over ICOUNT, while Static only achieves 13.5% improvement over ICOUNT. Compared with the currently best-performing algorithm Hill-climbing, ARPA also achieves 7.3% more *Avg_IPC* improvement. Allocating more resources to threads which can use them more efficiently does not always mean giving more resources to threads with a higher IPC. In fact, ARPA is an adaptive process that allows threads to share resources more fairly and efficiently. With the *Avg_Single_Weighted_IPC* metric, ARPA achieves 14.0% improvement over ICOUNT and attains 7.2% and 5.9% more improvement than Static and Hill-climbing over ICOUNT respectively. With the *Avg_Baseline_Weighted_IPC* metric, ARPA gains 43.6% improvement over ICOUNT, achieving 30.6% and 7.6% more improvement than Static and Hill-climbing.

## Acknowledgment

The authors would like to thank Dr. James Donald of Princeton University for help on the modification of the SMT simulator. This work was supported in part by NSF undergrants EIA-0102696 and CCR-0234363.

## References

[1] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A Framework for Architectural-level Power Analysis and Optimizations,” *Proc. 27th Ann. Int’l Symp. Computer Architecture*, pp. 83-94, June 2000.

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

[3] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, “An Elementary Processor Architecture with Simultaneous

Instruction Issuing from Multiple Threads," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 136-145, May 1992.

[4] W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," *Proc. First Int'l Symp. High Performance Computer Architecture*, pp. 49-58, June 1995.

[5] 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*, vol.17, no.5, pp. 12-19, Sept. 1997.

[6] F. J. Cazorla, E. Fernández, A. Ramírez, and M. Valero, "Improving Memory Latency Aware Fetch Policies for SMT Processors," *Proc. Fifth Int'l Symp. High Performance Computing*, pp. 70-85, Oct. 2003.

[7] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fern'andez, "Dynamically Controlled Resource Allocation in SMT Processors," *Proc. 37th Int'l Symp. Microarchitecture*, pp. 171-182, Dec. 2004.

[8] S. Choi and D. Yeung, "Learning-Based SMT Processor Resource Distribution via Hill-Climbing," *Proc. 33rd Ann. Int'l Symp. Computer Architecture*, pp. 239-251, 2006.

[9] A. El-Moursy and D. H. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," *Proc. 9th Int'l Symp. High Performance Computer Architecture*, pp. 31-40, Feb. 2003.

[10] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughout and Fairness in SMT Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software*, pp. 164-171, Nov. 2001.

[11] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol.6, no.1, pp. 4-15, Feb. 2002.

[12] S. E. Raasch and S. K. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," *Proc. 12th Int'l Conf. Parallel Architecture and Compilation Techniques*, pp. 15-26, Sept. 2003.

[13] S. Sair and M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suite," Technical Report, IBM T.J. Watson Research Center, 2000.

[14] J. J. Sharkey, D. Balkan, and D. Ponomarev, "Adaptive Reorder Buffers for SMT processors," *Proc. 15th Int'l Conf. Parallel Architecture and Compilation Techniques*, pp. 244-253, Sept. 2006.

[15] A. Snavely, D. M. Tullsen, and G. M. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," *Proc. Int'l Conf. Measurement and Modelling of Computer Systems*, pp. 66-76, June 2002.

[16] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pp. 392-403, June 1995.

[17] D. M. Tullsen and J. A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," *Proc. 34th Int'l Symp. Microarchitecture*, pp. 318-327, Dec. 2001.

[18] 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," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, pp. 191-202, May 1996.

[19] H. Wang, Y. Guo, I. Koren, and C. M. Krishna, "Compiler-Based Adaptive Fetch Throttling for Energy Efficiency," *Proc. Int'l Symp. Performance Analysis of Systems and Software*, pp. 112-119, Mar. 2006.

[20] S. Lee and J. Gaudiot, "Throttling-Based Resource Management in High Performance Multithreaded Architectures." *IEEE Trans. Computers*, vol.55, no.9, pp. 1142-1152, Sept. 2006.