Utilization-Based Resource Partitioning for Power-Performance Efficiency in SMT Processors

Huaping Wang, Israel Koren, *Fellow, IEEE,* and C. Mani Krishna, *Fellow, IEEE,*

Abstract—Simultaneous multithreading (SMT) increases processor throughput by allowing parallel execution of several threads. However, fully sharing processor resources may cause resource monopolization by a single thread or other misallocations, resulting in overall performance degradation. Static resource partitioning techniques have been suggested, but are not as effective as dynamic ones since program behavior does change over the course of its execution.

In this paper, we propose an Adaptive Resource Partitioning Algorithm (ARPA) that dynamically assigns resources to threads according to changes in thread behavior. ARPA analyzes the resource usage efficiency of each thread in a given time period and assigns more resources to threads which can use them more efficiently. Its purpose is to improve the efficiency of resource utilization, thereby improving overall instruction throughput. Our simulation results on a set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy ICOUNT by 55.8% with regard to overall instruction throughput and achieves a 33.8% improvement over Static Partitioning. It also outperforms the current best dynamic resource allocation technique, Hill-climbing, by 5.7%. Considering fairness accorded to each thread, ARPA attains 43.6%, 18.5% and 9.2% improvements over ICOUNT, Static Partitioning and Hill-climbing, respectively, using a common fairness metric.

We also explore the energy efficiency of dynamically controlling the number of powered-on reorder buffer entries for ARPA. Compared with ARPA, our energy-aware resource partitioning algorithm achieves 10.6% energy savings, while the performance loss is negligible.

Index Terms—Simultaneous multithreading, resource partitioning, power-performance efficiency

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 [9], [12], [21], [26]. 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 threads. In an SMT processor, all the resources can be shared among threads, except for some resources related to the architectural state which are separated to maintain the correct state of each logical processor.

Traditionally, a fetch policy [23] determines 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 threads. Unfortunately, some threads may occupy a disproportionately large share of system resources, and slow down others. Statically partitioning resources among threads has been suggested as a way to prevent a single thread from clogging resources [18]. 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 proposed to dynamically partition resources [7], [8]. Such techniques can lead to significant improvements in performance.

Resource partitioning approaches [7], [8], [18] mainly focus on certain critical resources which significantly impact performance if clogged by some threads. Commonly, they apply the same partitioning principles to all the resources. [18] studies the effect of partitioning the instruction queue (IQ) or the reorder buffer (ROB). DCRA [7] separately partitions queue and register entries using the same sharing model. Hill-climbing [8] partitions the integer rename registers among the threads, assuming that the integer IQ and ROB will be proportionately partitioned.

In this paper, we present a new Adaptive Resource Partitioning Algorithm (ARPA) [25] which concentrates on partitioning the following shared queue structures: instruction fetch queue (IFQ), IQ and ROB. We do not partition the renaming registers since partitioning ROB can effectively control the sharing of registers. Doing so, however, would be quite easy. The physical implementation of ROB can either be one ROB per thread [20], [23] or a single ROB shared by multiple threads [7], [8], [18]. We assume a shared ROB structure for consistency with previous resource partitioning schemes [7], [8], [18]. Moreover, the shared ROB implementation allows a more flexible ROB usage by the threads, resulting in greater performance benefits. However, our partitioning

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

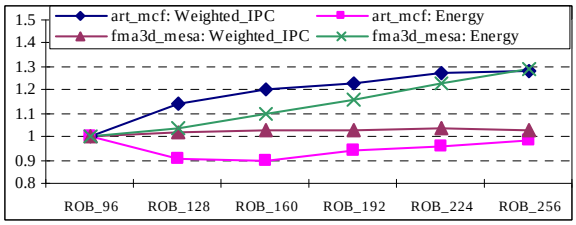

Fig. 1. The normalized Weighted_IPC and Energy as a function of ROB size for two workloads

algorithm can be applied to the divided ROB structure as well. If the ROB is physically divided, we will constrain the ROB usage by each thread. The difference is that for a private ROB structure, the ROB usage of each thread cannot exceed its allocated ROB subdivision, while for a shared ROB structure, each thread can use more than its equally partitioned share. We do not constrain the usage of individual queues. Instead, we impose an upper bound on the sum of IFQ and ROB entries 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 resource will indirectly control the usage of the other resources.

The goal of ARPA is to use resources more 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 more efficiently while still avoiding resource starvation of any thread. Our simulation results on a set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy ICOUNT, Static Partitioning and the current best dynamic resource allocation technique, Hill-climbing, by 43.6%, 18.5% and 9.2%, respectively, using a common fairness metric.

Architecture adaptation [27] has been effective in saving energy in single-thread processors by adaptively activating and deactivating hardware resources in accordance with the changes in the application's behavior [2], [5], [16]. In this paper, we study the benefits of applying architecture adaptation to SMT processors. To the best of our knowledge, this is the first attempt to explore architecture adaptation in SMTs for saving energy.

Compared to the single-thread case, resource requirements of multi-threaded workloads running on an SMT processor have a higher variability. Figure 1 illustrates the significant variations in resource requirements of both memory- and computation-bound workloads. *art_mcf* consists of the two memory-bound benchmarks; *fma3d_mesa* consists of the two computation-bound benchmarks. When the ROB size increases from 96 to 256 entries, the *Weighted-IPC* (see equation 7 below) of *art_mcf* improves by almost 30%, while there is almost no performance improvement for *fma3d_mesa*. However, the energy consumption increases linearly with the ROB size for *fma3_mesa*. Considering both energy and performance, a 96-entry ROB is ideal for *fma3d_mesa*, while a 256 entry configuration is better for *art_mcf*. Therefore, architecture adaptation may benefit SMT processors. We therefore, incorporate architecture adaptation into ARPA to adaptively control the number of powered-on resources and partition resources among threads targeting both performance and energy. Our experimental results show that with architecture adaptation, ARPA achieves 10.6% energy savings, while the resulting performance loss is negligible.

The rest of this paper is organized as follows. In the next section, we describe 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. Section 6 concludes the paper with a summary of its contributions.

2 Related Work

Prior resource partitioning related work can be categorized into three groups: fully flexible resource distribution [6], [10], [14], [23], static resource allocation [15], [18] and dynamic resource partitioning [7], [8].

Tullsen *et al.* [23] have presented several fetch policies that 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 a Round Robin order, disregarding the resource usage of each thread. ICOUNT is a policy that dynamically biases toward threads which 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 resources. STALL and FLUSH [22] are two techniques built on top of ICOUNT to ameliorate this problem. STALL prevents a thread with pending L2 misses 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. Instead of preventing all threads with long-latency loads from entering the pipeline, the Memory-Level Parallelism (MLP)-aware fetch policy [30] assigns as many resources as needed to MLP-intensive threads in order to fully exploit the high level of MLP and only prevents threads with isolated long-latency loads from being allocated more resources.

Static resource partitioning [15], [18] 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 dynamic resource sharing algorithm. Threads are assigned resource usage bounds and these are changed dynamically. The bound is higher for threads with more L1 Data cache misses. However,

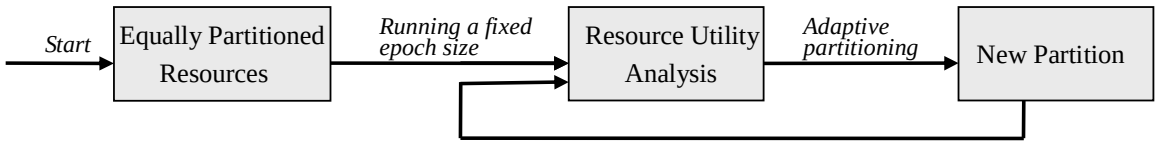

Fig. 2. A high-level description of the ARPA algorithm

DCRA does not work well for 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 resource-starved threads. Hill-climbing [8] uses performance feedback to direct the partitioning. This learning-based algorithm starts with equal partitioning, then moves an equal amount of resources from all the other threads to a "trial" thread, where each one of the existing threads serves as a "trial" tread at its turn. Hill-climbing appears to be the best resource partitioning technique currently available. Like DCRA [7] and Hill-climbing [8], our algorithm, ARPA [25], partitions resources dynamically. However, ARPA's analysis of program behavior results in a more effective use of resources.

Architecture adaptation [27] techniques which have been proposed for single-thread processors to save energy are usually applied to power-hungry hardware components such as IQ, Load-Store Queue (LSQ), ROB, register file and caches. Energy savings are achieved at the cost of a small performance degradation. We combine in this paper architecture adaptation with resource partitioning, targeting both performance and energy.

In [5], [11], the authors examine the power saving potential of an adaptive IQ structure. They divide the IQ into separate equal-sized chunks, allocated/deallocated dynamically according to the prevailing ILP. Power savings are achieved by turning off unused chunks. Abella *et al.* [1] propose a power-efficient adaptive resizing scheme for the IQ and register file. Their scheme is based on monitoring how much time instructions spend in both the IQ and ROB and limits their occupancy based on these statistics. Ponomarev *et al.* [16] present a mechanism to dynamically and independently resize the IQ, ROB and LSQ. Downsizing is driven by directly using sampled estimates of their individual occupancies. Upsizing is more aggressive using the relative rate of blocked dispatches to limit the performance penalty.

In [20], the authors propose an adaptive ROB scheme for SMT processors. Allocation/Deallocation of the ROB entries to a thread is based on whether the thread is in a commit-bound or an issue-bound phase. The purpose of ROB adaptation is to prevent threads from clogging shared resources thus improving performance. In contrast, our architecture adaptation of ARPA aims to reduce the energy consumption in addition to improving the performance.

3 ARPA: Adaptive Resource Partitioning Algorithm

Existing dynamic resource allocation techniques consider indirect feedback (e.g., L1 data cache misses [7]; pressure on shared resources [20], etc) to optimize performance or are based on periodic "trials" to select the best partitioning [8]. Instead of using indirect feedbacks [7], [20] or exhaustive "trials" [8], ARPA allocates resources based on the resource usage efficiency of each thread. By assigning more resources to threads which can use them more efficiently, ARPA can not only improve the overall resource usage efficiency, but also ameliorate different kinds of clogging, thereby improving the overall instruction throughput.

3.1 Framework

Figure 2 shows a high-level flowchart of ARPA. 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, the system analyzes the current resource usage to determine whether the threads have used their allocated resources efficiently in this epoch. Its resource partitioning decision is driven by these analyses. The analysis and partitioning process is repeated every epoch until the end of the program.

3.2 Resource Utilization Analysis

Our resource utilization analysis is carried out at the end of each epoch and is based on the following metric.

3.2.1 Metric of Usage Efficiency

We use Committed Instructions Per Resource Entry (CIPRE) to represent the usage efficiency of processor resources in each epoch. CIPREs in successive epochs are similar to each other if there are no significant program phase changes. The CIPRE metric can express two different characteristics: (a) the usage efficiency of all processor resources and (b) the usage efficiency of only the resources that are allocated to a specific thread.

Note that a thread with a higher CIPRE does not necessarily have a higher IPC. Consider, for example, two threads A and B that run simultaneously with 50 and 20 queue entries, respectively, and suppose that 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

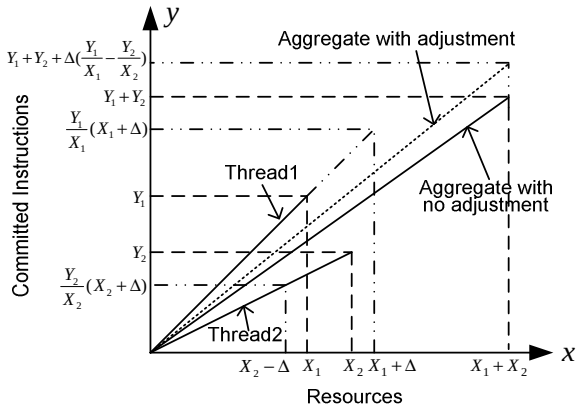

Fig. 3. An example illustrating the CIPRE changes after epoch $n+1$

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 allocating the high-IPC thread more resources.

3.2.2 The Partitioning Process

ARPA follows an adaptive partitioning strategy and adjusts the number of resources allocated to threads based on their CIPRE metric. In every epoch a thread with a greater CIPRE value will take some resources from a thread with a lower CIPRE until the CIPREs of the two threads are close to each other. Through adaptive resource tuning, all threads will use their allocated resources with approximately equal efficiency, thus improving the usage efficiency of all resources. However, there is a possibility that threads with lower CIPRE values will lose most of their resources but their CIPREs will still be lower than that of the most efficient thread, resulting in resource starvation. ARPA avoids this situation by assigning each thread a minimum number of resources no matter what its CIPRE is.

In order to explain ARPA better, we use a two-thread example to illustrate the tuning process. Figure 3 shows the change in the CIPRE value when a program completes epoch $n+1$. X_1 and X_2 are the numbers of resource entries allocated to threads 1 and 2, respectively, during epoch n. Y_1 and Y_2 are the numbers of committed instructions of threads 1 and 2, respectively, during epoch n. Δ 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 as follows.

$$CIPRE_i = \frac{Y_i}{X_i}, \quad i = 1, 2 \tag{1}$$

$$CIPRE^{(n)}_{overall} = \frac{Y_1 + Y_2}{X_1 + X_2} \tag{2}$$

Assuming that $CIPRE_1 > CIPRE_2$ in this example, it is easy to show that

$$CIPRE_2 < CIPRE^{(n)}_{overall} < CIPRE_1 \tag{3}$$

Since thread 1 achieves a more efficient usage of the allocated resources in epoch n, ARPA will transfer to it Δ resources from thread 2 in the next epoch. That is to say, thread 1 will be assigned $X_1 + \Delta$ resource entries while thread 2 will be restricted to $X_2 - \Delta$ entries in epoch $n+1$. If both threads 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:

$$CIPRE^{(n+1)}_{overall} = \frac{Y_1 + Y_2 + \Delta \cdot (\frac{Y_1}{X_1} - \frac{Y_2}{X_2})}{X_1 + X_2} \tag{4}$$

Compared with the no-adjustment case which has the same CIPRE as in (2) after epoch $n+1$, the CIPRE of all resources is increased by

$$CIPRE^{(n+1)}_{overall} - \frac{Y_1 + Y_2}{X_1 + X_2} = \frac{\Delta \cdot (\frac{Y_1}{X_1} - \frac{Y_2}{X_2})}{X_1 + X_2} \tag{5}$$

As long as $CIPRE_1$ is greater than $CIPRE_2$, resources continue to be transferred from thread 2 to thread 1 in subsequent epochs subject to the constraint that each thread has at least its specified minimum allocation. The overall CIPRE keeps increasing and getting ever closer to $CIPRE_1$ (but will never exceed $CIPRE_1$). That is to say, the line for *Aggregate with adjustment* is getting ever closer to the line for $Thread_1$ in Figure 3.

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 ever closer (the lines for $Thread_1$, $Thread_2$ and *Aggregate with adjustment* in Figure 3 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 is still greater than that of thread 2 even when thread 1 already took all the resources it could from thread 2. As mentioned previously, ARPA assures each thread a certain minimum number of resources to avoid resource starvation or under-utilization.

Although the objective of ARPA is to improve the efficiency of resource utilization, thereby improving overall instruction throughput, our experimental results (in Section 5) show that ARPA also provides a good balance between throughput and fairness.

3.3 Architecture Adaptation in ARPA

The CIPRE metric used by ARPA to drive the resource partitioning is a "relative" concept (compared to other threads sharing the resources). It is possible that the absolute resource usage is very low. In such cases, those

additional resources do not contribute to performance and can be powered off in order to save energy. In this section, we incorporate architecture adaptation [27] into ARPA to adaptively control the number of powered-on resources and partition resources among threads targeting both performance and energy.

We focus on the dynamic instruction scheduling logic (IQ, LSQ, ROB and renaming registers) to perform architecture adaptation, since these are the most power-hungry components, consuming between them about 55% of the total power dissipation [28]. In some designs like Pentium III, the ROB is integrated with the physical registers to support register renaming. The ROB's power consumption can be more than 27% of the total power [17]. Adaptation techniques can be applied either to only one resource such as IQ [5] or ROB [17], or to multiple resources simultaneously [1], [16] to reduce energy consumption. We attempt to control the number of powered-on ROB entries which act as the renaming registers as well. By adaptively tuning the number of ROB entries, instructions can be fetched as needed. This not only reduces the resource competition among threads but also achieves significant energy savings in units like IQ logic by avoiding unnecessary wake-up and selection operations.

3.3.1 Multi-banked Structure

We assume a shared ROB structure for consistency with previous resource partitioning schemes [7], [8], [18] although our scheme can be applied to the divided ROB structure as well. The assumed shared ROB implementation is a single buffer with multiple pairs of head and tail pointers, one pair for each thread. Instructions for each thread are dispatched to the free ROB entries after that thread's tail pointer in program order and are committed starting from its head pointer, thus maintaining logical correctness. The occupied ROB entries are marked with thread IDs which are needed for the commit logic to distinguish among the different threads. The commit logic checks the entries after each thread's head pointer to identify whether to commit the instructions or not. Although the commit complexity of a shared-ROB implementation is higher than that of a divided ROB, it allows a more flexible ROB usage by the threads, resulting in greater performance benefits.

In order to dynamically control the number of powered-on ROB entries to save energy, a large ROB is partitioned into independent banks. Each bank has its own precharger, sense amplifiers and input/output drivers and can be powered on or off to save energy according to changes in program behavior. To power off a bank, its bypass switch is turned on and the power supply to the bank is disabled; to power on a bank, the bypass switch is turned off and the power supply is enabled. Note that a bank can be powered off only after all the instructions residing in it have been committed. We define the power-off latency of a bank as the time duration from receiving the deallocation signal to when the power supply is disabled. Once a bank has been deallocated, instructions cannot be dispatched to it anymore. A detailed hardware implementation is described in [16].

Banks can be selected for deallocation in two ways. The approaches in [16], [20] deallocate banks from the highest to the lowest index sequentially. Such an in-order policy may have high power-off latencies. We can also deallocate banks according to their expected power-off latencies. It is obvious that an empty bank has the shortest power-off latency while a bank with tail pointers has the longest latency since it must wait until the tail moves out of the bank and all the instructions in it have been committed. Here, we deallocate banks according to their expected power-off latencies. Our first choice is to power off empty banks. If none can be found, we pick a bank with only head pointers. If neither category is available, we power off a bank with no pointers. If none of these is available, we choose banks with tail pointers. On average, it costs tens of cycles to power-off a bank after receiving the deallocation signal. We have in our experiments taken this overhead into consideration.

3.3.2 ROB Adaptation

The architecture adaptation in ARPA consists of upsizing and downsizing the available ROB space as outlined below.

Upsizing: During each epoch, we count the number of full ROB occupancy cycles. If the ROB is fully occupied in most cycles, it means that threads may face a shortage of ROB entries and we then turn on a powered-off bank and allocate additional entries to the thread with the highest CIPRE since this thread can use the resources more efficiently.

Downsizing: When the number of full ROB occupancy cycles is low in an epoch, it does not necessarily mean that resources are redundant and should be downsized. Some threads may still need the available resources to improve performance. We coordinate with other hints to decide whether to downsize or only repartition resources.

Although ARPA's goal is to assign more resources to the thread with the highest CIPRE, it is possible that this thread already holds enough resources, and additional resources will not be used as efficiently and can instead be powered off to save energy. To determine this we check whether it is rare for the in-flight instructions (the instructions in the IFQ and ROB) of this thread to use up all the allocated entries. If this is the case, we will not allocate additional resources to this thread even if its CIPRE value is the highest among all the threads. These resources will instead be powered off to save energy.

3.4 The Algorithm

Figure 4 presents the pseudocode of ARPA. The code in the shaded box has been added for architecture adaptation. We can see that the architecture adaptation is

```
#define num              Number of running threads
#define ComInsts(x)      Compute CIPRE[x]
#define max(A,n)         Get index of the max value in array A[0:n]
#define Δ                Number of queue entries moved at each epoch
#define NumOffBank       Number of powered-off banks
#define Bank_size        Number of ROB entries per bank
For every epoch cycle {
  for(tid = 0; tid < num; tid ++)      /* compute CIPRE of each thread */
    CIPRE[tid] = ComInsts(tid);
  Ref_tid = max(CIPRE, num);           /* select reference thread */
  if(full_ROB >= Threshold1 &&  NumOffBank >=1 ){
    NumOffBank --;                     /* power on a ROB bank */
    Partition[Ref_tid] += Bank_size;}
  else{                                /* power off a ROB bank */
    if(full_ROB < Threshold1 && full_in_flight[Ref_tid] <= Threshold2){
      for(tid = 0; tid < num; tid ++)
        if(tid != Ref_tid)
          Partition[tid] -= Bank_size/(num-1);
      NumOffBank ++;}
    else{                              /* resource reallocation */
      for(tid = 0; tid < num; tid ++){
        Partition[Ref_tid] += Δ;
        Partition[tid] -= Δ; }}}}
```

Fig. 4. Pseudo-code of ARPA with architecture adaptation

independent of the basic ARPA algorithm and can be easily integrated into ARPA in order to save energy.

The *ComInsts* function in Figure 4 computes the CIPRE value of each thread at the end of an epoch: 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 largest CIPRE as the reference thread (if the CIPREs of two threads are equal, we randomly select one of them).

Whether architecture adaptation or only resource repartitioning (basic ARPA) is performed, is determined by two metrics: *full_ROB*, which is the number of cycles when the ROB is fully occupied during an epoch, and *full_in_flight* for the reference thread, which is the number of cycles when the in-flight instructions of the reference thread fully occupy all the allocated entries during an epoch. If *full_ROB* is equal to or greater than Threshold1 and there are still powered-off banks, upsizing is performed. The new ROB entries will be allocated to the reference thread since it can use additional resources most efficiently. If *full_ROB* is less than Threshold1 and *full_in_flight* of the reference thread is less than or equal to Threshold2, the algorithm turns off the ROB entries which would be allocated to the reference thread. We set the total number of entries taken from other threads to be equal to the bank size (For a 2-thread workload, a number of Bank_size entries from the non reference thread will be powered off. For a 4-thread workload, integer part of Bank_size/(num-1) resource entries are taken from each non reference thread and the still remaining entries are randomly taken from any non reference thread). In other cases, the basic ARPA will be performed: the reference thread can take Δ entries of IFQ and ROB from every other thread. IQ entries are also proportionately reassigned.

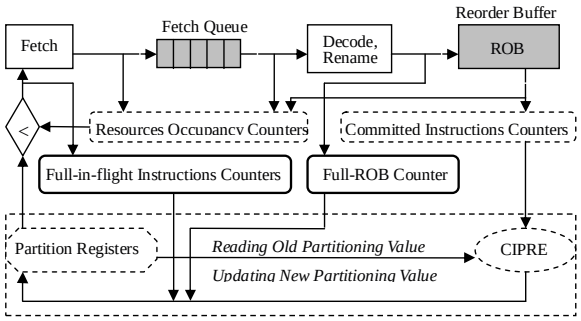

Fig. 5. An implementation of ARPA with architecture adaptation

3.5 Implementation of ARPA

Figure 5 shows the required processor modifications for implementing ARPA with architecture adaptation. The top layer in Figure 5 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: these will be used for resource partitioning and architecture adaptation. The counters needed for resource partitioning are marked with dashed-line boxes. We need *Resources_Occupancy_Counters* per thread to monitor the in-flight instructions (overall IFQ and ROB occupancy) and IQ occupancy for each thread. To monitor the in-flight instructions of a thread, the counter will be incremented as instructions are fetched and decremented as instructions are committed. To monitor the IQ usage, the counter will be increased as instructions are dispatched into IQ and decremented as instructions leave the IQ. 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 *Resources_Occupancy_Counter* will not be reset during the execution of a thread. We use two comparators per thread to determine if the current resource usages of the thread have already exceeded its specified bound; if so, a throttling signal will be generated to throttle further fetching for this thread.

The counters in the solid-line boxes have been added for architecture adaptation. We need a *Full-ROB_Counter* to monitor the full ROB usage rate in an epoch. This counter will be incremented by one whenever the dispatch stage notices that the ROB is full in that cycle. The *Full-in-flight_Instructions_Counters* are used to count the number of full occupancy of in-flight instructions for each thread. If the comparator shows that the number of in-flight instructions of the thread exceeds its bound, the *Full-in-flight_Instructions_Counter* of this thread is incremented by one.

The bottom layer is the implementation of the algorithm. At the end of each epoch, we execute the architecture adaptation or allocation steps. The resources upper bound assigned to each thread is saved in its Partition Register. At the start of the program, this bound

TABLE 1
Baseline parameters

Parameter	Value
IF,ID,IS Width	8-way
Queue size	32 IFQ, 80 IQ, 64(128) LSQ
Functional Units	6 Int, 4 FP, 4 ld/st 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, 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
22 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

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 the Partition Registers updated. 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 taken into account in this paper in the same way as was done in [8].

4 Evaluation Methodology

4.1 Configuration

Our simulator is based on Simplescalar [4] for the Alpha AXP instruction set with Wattch [3] power extensions. We modified SimpleScalar to support SMT processors. Moreover, we have decoupled the centralized Register Update Unit structure adopted by SimpleScalar and have separate IQ, ROB and physical registers. Our baseline processor configuration is shown in Table 1. We assume 64 LSQ entries in the baseline architecture. In order to demonstrate the efficiency of ARPA's ROB adaptation, we also increase the number of LSQ entries to 128 to ensure that the ROB entries are not a redundant resource compared to others. A resource sensitivity analysis is presented in Section 5.4.2. Other detailed features are based on the SMT architecture of Tullsen *et al.* [23]. Wattch is used to measure the energy consumption and has been retuned for state-of-the-art technology scaling parameters; we use a 45nm, 4GHz, 1.2v process.

Our simulator adds support for dynamic partitioning of the IFQ and ROB. We keep counters for the number of in-flight instructions per thread, allowing a thread to fetch instructions as long as its in-flight instructions have not exceeded its assigned limit. The counter for in-flight instructions is similar to that in [23] for implementing the ICOUNT fetch policy. When the number of in-flight instructions exceeds the assigned bound, we apply fetch throttling [13], [24] to this thread until it releases some of its entries or is allocated more resources. The IQ is partitioned proportionately.

In order to support architecture adaptation, we maintain a counter for the number of cycles during which the ROB is full and another for the number of cycles when in-flight instructions fully occupy the allocated resources per thread. We modified Wattch to calculate the energy consumption of this adaptive ROB structure. We use the ICOUNT fetch policy to fetch instructions. 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 produced by 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], [22]. 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 created our multiprogrammed workloads with 2-thread and 4-thread combinations as shown in Table 3. All the workloads are labelled to indicate the character and number of threads, as well as an index to distinguish one workload from another. MIX workloads select half of their threads from ILP and the other half from MEM. We selected simulation regions of different benchmarks following the approach proposed in [19] as shown in Table 2 and stopped simulations after running 400 million instructions. This simulation methodology is widely used by other researchers [7], [8].

4.3 SMT Performance 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

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

instruction throughput of the processor but also the fairness accorded to each thread running on the processor. Several metrics have been proposed to measure SMT performance [8], [14], [22], [29]. In order to provide a comprehensive comparison of the different algorithms, we show in our paper the throughput and fairness results quantified by each of the following four metrics.

$$Avg\_IPC = \frac{\sum IPC_i}{T} \tag{6}$$

The *Avg_IPC* metric [8], [14] only quantifies the overall throughput and does not take fairness into consideration. Therefore, using this metric may boost the overall IPC by starving some threads.

$$Single\_WIPC = \frac{\sum \frac{IPC_i}{SingleIPC_i}}{T} \tag{7}$$

The *Single_WIPC* metric [8], [14] weighs the IPC of each thread when running on an SMT with respect to its IPC if run alone, and reflects the fairness accorded to each thread. The drawback of this metric is that it does not assign any importance to the overall throughput and may bias against a thread with very low IPC. For example, consider thread A with single thread IPC of 3.0 and thread B with single thread IPC of 0.1 running simultaneously, with thread A achieving IPC=1.5 and thread B IPC=0.09 with the ICOUNT fetch policy, while Static Partitioning achieves IPC=2.1 and IPC=0.06, respectively. The *Single_WIPC* of Static Partitioning is lower by 7.1% than that of ICOUNT although the overall throughput of Static Partitioning is much better.

$$HMean\_WIPC = \frac{T}{\sum \frac{SingleIPC_i}{IPC_i}} \tag{8}$$

The *HMean_WIPC* metric [8], [14] is an efficient complement to the *Single_WIPC* metric. It takes both overall throughput and fairness into consideration.

$$Baseline\_WIPC = \frac{\sum \frac{IPC_{new,i}}{IPC_{baseline,i}}}{T} \tag{9}$$

The *Baseline_WIPC* metric [22] weighs the IPC of each thread with respect to its IPC in the baseline or 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 a single thread mode, *Baseline_WIPC* 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 with other schemes using the above four metrics across the 42 workloads; we also show the energy and performance benefits of integrating architecture adaptation into ARPA. Finally, we provide a sensitivity analysis of ARPA to its main parameters and the number of queue entries.

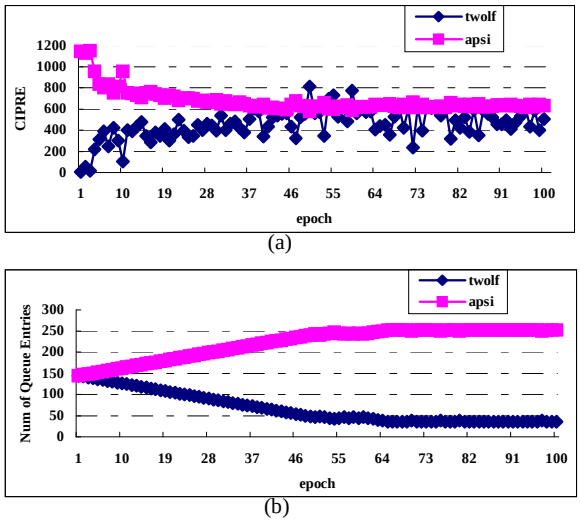

Fig. 6. An example illustrating the adaptive nature of ARPA for epochs 1 to 100.

5.1 Adaptivity of ARPA

Figure 6 illustrates the adaptive nature of the resource partitioning by the basic ARPA (without architecture

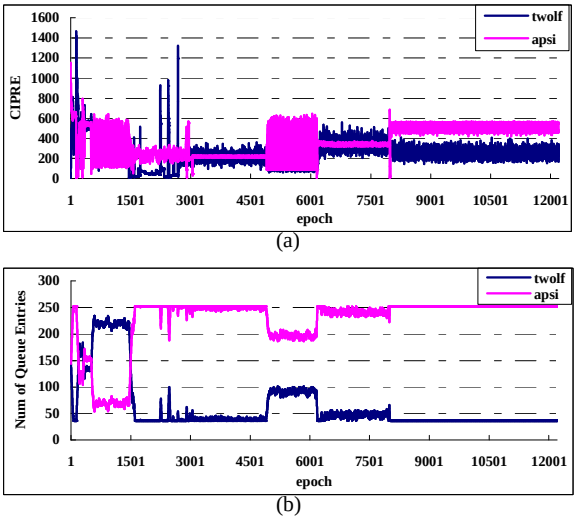

Fig. 7. An example illustrating the adaptive nature of ARPA.

adaptation). *twolf*, a memory-bound thread, and *apsi*, a computation-bound thread, are running simultaneously. Figure 6(a) displays the CIPRE changes for these two threads for epochs 1 to 100 when using ARPA, while Figure 6(b) shows the resulting partitioning of queue entries between the two threads.

In the first epoch, we equally partition resources to *twolf* and *apsi*, as indicated in Figure 6(b). The CIPRE of *apsi* is higher than that of *twolf* in this epoch, and consequently in epoch 2, *apsi* takes $\Delta = 2$ queue entries from *twolf*. We can see that the CIPRE of *apsi* is larger than that of *twolf* until epoch 49. Therefore, *apsi* takes 2 queue entries from *twolf* at each epoch until epoch 49. The allocated number of queue entries of *apsi* increases linearly while the allocated number of queue entries of *twolf* 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, a small number of queue entries move back and forth between the two threads. At epoch 64, the number of queue entries allocated to *twolf* reaches its minimum of 36. Although the CIPREs of *apsi* are higher than those of *twolf* most of the time after epoch 64, *apsi* cannot take additional resources from *twolf* in order to prevent resource starvation. The number of queue entries becomes stable for each thread and the resources allocated to each thread will remain in this setting if no program phase changes occur.

Figure 6 illustrates 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 show in Figures 7(a) and 7(b), respectively, the CIPRE changes and the corresponding resource allocations of these two threads for the entire program execution that lasts 12206 epochs.

The transitions from one stable phase to the next only take a few tens of epochs and there are different stable resource allocation phases during the execution of the workload combination of *twolf* and *apsi* as indicated in Figure 7. The first stable phase comes after the resource tuning process shown in Figure 6, and is short. The second stable phase is also short compared to the following five phases; during this phase, the numbers of resources allocated to each thread are close to each other. In the third tuning process phase, the CIPREs of *twolf* are higher than those of *apsi* in most epochs, allowing *twolf* to own more resources to improve resource usage efficiency. Similar tuning processes happen at the start of the next three stable phases. In the final stable phase, although the CIPRE of *apsi* is larger than that of *twolf* all the time, resource allocations are fixed since the number of queue entries of *twolf* has reached the low-bound limit. Clearly, a static resource partitioning can not satisfy these varied program phases. ARPA retunes the resource allocation whenever program phase changes occur.

5.2 Performance Results

Figure 8 compares the *Single_WIPC* of ARPA without architecture adaptation to those of other schemes across the 42 workloads listed in Table 3. ICOUNT [23] is a traditional fetch policy and is therefore suitable as a baseline scheme for comparison. Static Partitioning [18] assigns resources equally to each thread for the entire program execution. We compare ARPA with it to see the benefits of an adaptive scheme over a non-adaptive one. Hill-climbing [8] has been so far the best resource partitioning scheme achieving significant improvements over previous schemes like STALL/FLUSH++ [6], [22]. To avoid a cluttered figure, we only choose ICOUNT, Static Partitioning and Hill-climbing for comparison. The epoch size we used in these experiments is 32K cycles and the Δ size is 2 queue entries. ARPA is not sensitive to the size of the epoch and Δ as long as they are not too large. The number of LSQ entries we used is 64. We allow each thread to keep at least a quarter of the equally partitioned queue entries to avoid resource starvation. (We ran experiments varying the minimum number of resource entries that each thread should keep and concluded that 25% of the equally partitioned queue entries is a good choice).

From Figure 8 we see that ARPA outperforms ICOUNT and Static Partitioning significantly in MEM and MIX workloads. For some workloads like MIX.2.5 and MIX.4.2, the improvement of ARPA over ICOUNT and Static Partitioning 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 8 that the improvement in MEM and MIX workloads is much

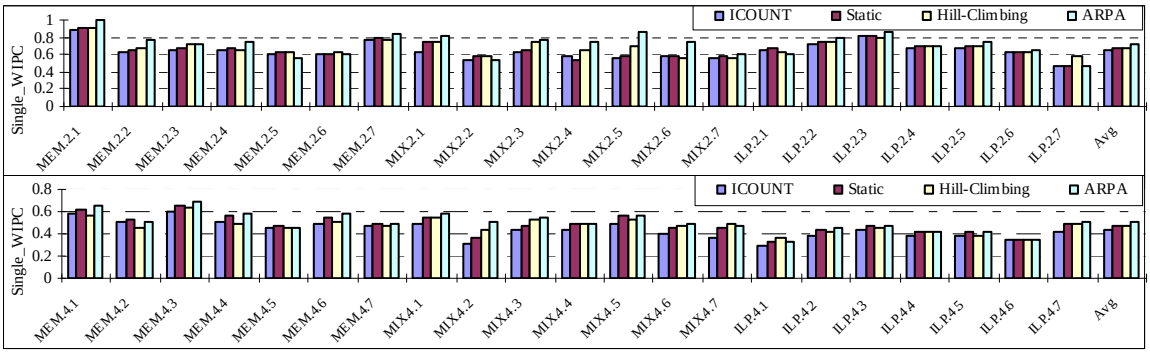

Fig. 8. *Single_WIPC* of different schemes for 42 workloads.

greater than that in ILP workloads for both 2-thread and 4-thread workloads. Static Partitioning can prevent resource monopolization by a single thread. This characteristic benefits especially resource-tight situations, since the possibility of resource monopolization increases when the number of resources reduces. From Figure 8 we can see that Static Partitioning achieves higher improvement over ICOUNT in 4-thread workloads than in 2-thread workloads. However, Static Partitioning does not consider program phase changes and the needs of individual threads. As a result, the performance improvement of ARPA over Static Partitioning is considerable.

Hill-climbing [8] is the best-performing published partitioning algorithm. From Figure 8, we can see that Hill-climbing significantly outperforms ICOUNT and Static Partitioning in MIX workloads. Using the *Single_WIPC* metric, Hill-climbing achieves an 8.1% improvement over ICOUNT (close to the results published in [8]). However, Hill-climbing does not analyze the behavior of individual threads and makes its decisions based only on periodic trials. As with any periodic search procedure, Hill-climbing not only takes considerable learning time, especially if the number of shared threads is big, but also may get stuck in local maxima if multiple peaks exist. ARPA assesses the resource usage efficiency of each thread and directly shifts resources to the right thread, thus achieving better performance compared to Hill-climbing. As shown in Figure 8, ARPA outperforms Hill-climbing in all but 8 of the 42 workloads. Out of these eight workloads, ILP.2.7 (consisting of the *gcc* and *apsi* threads) exhibits the highest benefit of Hill-climbing vs. ARPA. In this workload, both threads have a high ILP but *apsi* needs a much larger number of resources (than *gcc*) to fully exploit its ILP and has relatively low CIPRE values compared to *gcc* most of the time. As a result, ARPA prefers to allocate more resources to *gcc*, being unaware of the potentially high CIPRE values that *apsi* can achieve once it accumulates a large number of resources. In contrast, Hill-climbing does not analyze the individual threads and randomly attempts different allocations of resources managing to try out an allocation with a much higher number of entries assigned to *apsi* rather than *gcc* and achieves a better overall performance.

With the *Single_WIPC* metric, Static Partitioning, Hill-Climbing and ARPA achieve 6.8%, 8.1% and 14.0% improvements over ICOUNT. ARPA achieves a 5.7% improvement over Hill-climbing.

Figure 9 shows the *Avg_IPC* and *HMean_WIPC* improvements of different schemes over ICOUNT across 42 workloads. The figure shows the averages for the MEM, MIX, ILP workloads separately in both 2-thread and 4-thread workloads.

Avg_IPC quantifies overall instruction throughput and *HMean_WIPC* takes both instruction throughput and fairness into consideration. Static Partitioning equally partitions resources among threads and makes threads share resources in a fair manner. As we can see from Figure 9, the performance improvements of Static Partitioning over ICOUNT measured by *Avg_IPC* metric and *HMean_WIPC* metric are close with each other, achieving 13.5% and 15.6% improvement over ICOUNT, respectively. However, the drawback of Static Partitioning is that it does not consider program phase changes and lacks flexibility. Compared to Static Partitioning, Hill-climbing and ARPA adjust resource partitioning according to program behavior changes. However, their resource sharing is not as fair as that of Static Partitioning. Therefore, the improvement in terms of the *HMean_WIPC* metric (over Static Partitioning) is less than that in terms of the *Avg_IPC* metric.

From Figure 9, we can see that Hill-climbing and ARPA achieve 48.5% and 55.8% improvements over ICOUNT using the *Avg_IPC* metric, while considering the *HMean_WIPC* metric, they achieve 20.8% and 29.4% improvement over ICOUNT, respectively.

The *Single_WIPC* metric ignores the overall throughput and may bias against threads with very low IPC. In contrast, *Baseline_WIPC* reflects the improvement in IPC of each thread for an optimized resource partitioning scheme over a baseline scheme and it does not depend on how well would each thread run in the single thread mode. Figure 10 shows the *Baseline_WIPC* speedup of Static Partitioning, Hill-climbing and ARPA over the ICOUNT baseline. ARPA achieves a better *Baseline_WIPC* speedup than Static Partitioning and Hill-

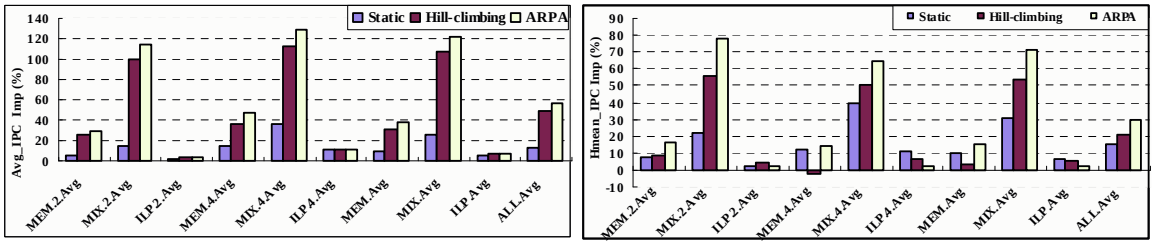

Fig. 9. *Avg_IPC* and *HMean_WIPC* improvement of different schemes over ICOUNT across 42 workloads

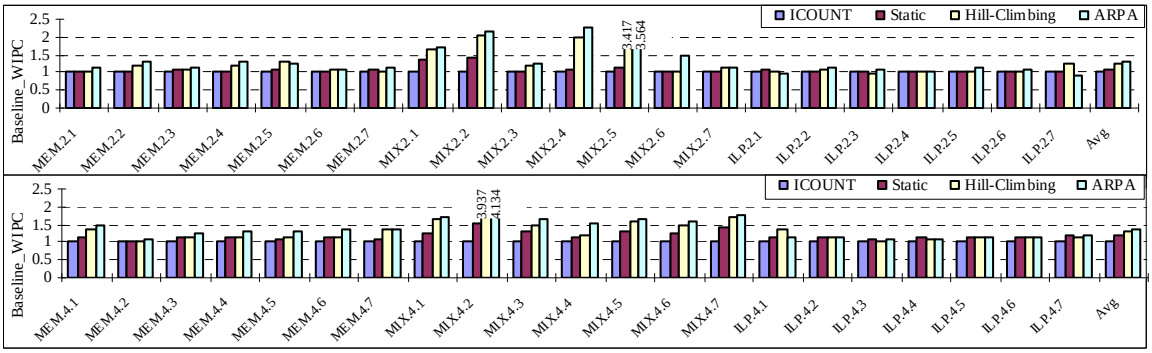

Fig. 10. *Baseline_WIPC* of different schemes for 42 workloads.

climbing for almost all workloads in the MEM and MIX groups but the improvement in the ILP group is not as significant. Static Partitioning, Hill-climbing and ARPA achieve 7.8%, 32.6% and 39.1% improvements in the 2-thread workloads, and 18.2%, 39.4% and 48.0% improvements in the 4-thread workloads, respectively. Compared to Static Partitioning, ARPA achieves a 18.5% improvement in *Baseline_WIPC* while compared to Hill-climbing, it achieves a 9.2% improvement in *Baseline_WIPC*.

5.3 Architecture Adaptation Results

Architecture adaptation can save chip-wide energy. It is especially important as energy consumption becomes a pivotal design issue. In this section we compare the performance and energy of ARPA with architecture adaptation (ARPA_AA) with other schemes (e.g., AdapROB [20], ARPA_ROB and ARPA (without architecture adaptation)). AdapROB [20] deallocates ROB entries from a thread when it is in the issue-bound phase and allocates ROB entries to a thread when it is in the commit-bound phase. The ROB adaptation of AdapROB aims to prevent threads from clogging shared datapath resources and targets only performance. In order to verify that the number of powered-off ROB banks of ARPA_AA is not built upon overdesigned ROB entries and determine the advantage of a dynamic scheme over a static one, we also present the baseline results, ARPA_ROB, where the number of available ROB banks is set to be equal to the average number of ROB banks that are powered on by ARPA_AA (shown in Figure 11). ARPA_ROB is the same as ARPA, except that it uses the average number of powered-on ROB banks of ARPA_AA. We randomly select 30 workloads listed in Table 3; five from each group.

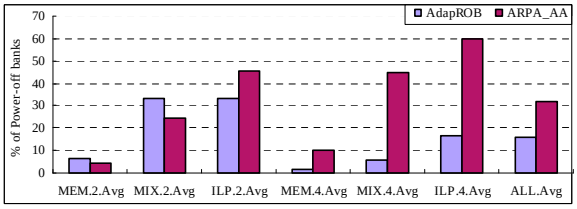

Fig. 11. The percentage of powered-off banks for AdapROB and ARPA_AA

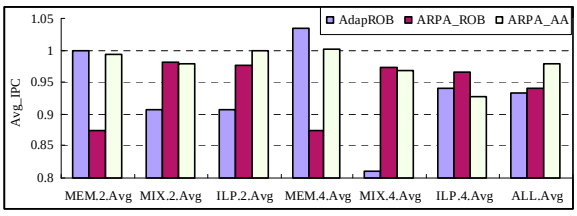

Fig. 12. *Avg_IPC* of different schemes normalized with respect to ARPA.

Both ARPA and ARPA_AA are insensitive to the epoch size. Since a small epoch size allows a more detailed analysis of Threshold1 and Threshold2 (vary between 0 and epoch size) with small granularity, we use an epoch size of 8K cycles for both ARPA and ARPA_AA. The bank_size used in ARPA_AA and AdapROB is 8. There are in all 32 banks since the total number of ROB entries we used in our experiments is 256. The number of LSQ entries we used is 128 in these experiments. We set both Threshold1 and Threshold2 to 3K. A detailed sensitivity analysis is presented in Section 5.4.

5.3.1 Performance Evaluation

We first show the average percentage of powered-off banks of AdapROB and ARPA_AA for the MEM, MIX and ILP workloads separately for the 2-thread and 4-thread workloads in Figure 11. The percentage of powered-off banks of ARPA_AA is highest for the ILP groups and lowest for the MEM groups for both 2-thread and 4-thread workloads since memory-bound workloads need more ROB entries to exploit instruction level parallelism. The percentage of powered-off banks in 4-thread workloads is higher than that in 2-thread workloads since 4-thread workloads have higher thread level parallelism and need fewer ROB entries to exploit instruction level parallelism. ARPA_AA manages to power off, on average, 8 banks for 2-thread workloads and 13 banks for 4-thread workloads. AdapROB powers off fewer ROB banks than ARPA_AA since the purpose of allocation/deallocation of ROB banks is to prevent threads from clogging shared datapath resources and does not target energy.

Figure 12 compares the *Avg_IPC* of different schemes across the 30 workloads. It shows the averages for the MEM, MIX, ILP workloads separately in both 2-thread and 4-thread workloads. We normalized results with respect to ARPA for all the schemes. The *Avg_IPC* of AdapROB is about 6.4% lower than that of ARPA on average. AdapROB deallocates ROB entries of a thread which has a high need for IQ entries to reduce the pressure on the IQ and allocates ROB entries to the thread which has a low need for IQ entries. However, it is possible that allocating more ROB entries to a thread with a high need for IQ entries results in better performance improvements than allocating more ROB entries to a thread with a low need for IQ entries. Differently from AdapROB, ARPA directly takes the committed instructions of each thread into account and assigns more resources to a thread which uses resources in a more efficient way. The performance improvement of ARPA over AdapROB is significant for the benchmarks in the 4-thread MIX group.

ARPA_ROB uses on average 24 ROB banks for 2-thread workloads and 19 banks for 4-thread workloads. The performance reduction of ARPA_ROB over ARPA in MEM groups is significant – a 13% degradation for both 2- and 4-thread MEM groups. Although ARPA_AA uses on average fewer ROB banks than ARPA_ROB in ILP groups, it achieves similar performance to that of ARPA_ROB. A fixed size of pre-allocated resources to individual threads is often non-optimal for power and performance purposes since resource requirements of multi-threaded workloads running on SMT processors vary significantly.

The *Avg_IPC* of ARPA_AA is very close to that of ARPA in most workloads. Compared to ARPA, the performance loss across 30 workloads averages only 2.2%. However, the energy savings of ARPA_AA compared with ARPA is very significant as we will see next.

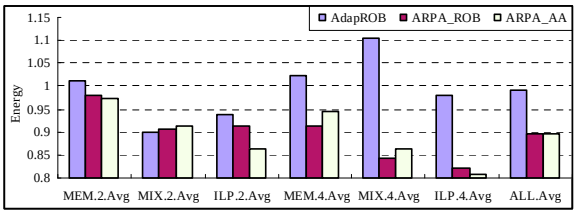

Fig. 13. Energy consumption of different schemes normalized with respect to ARPA

5.3.2 Energy Savings

Figure 13 compares the average energy consumption of different schemes for the MEM, MIX and ILP workloads separately for the 2-thread and 4-thread workloads. All results are normalized with respect to ARPA. As we can see, although AdapROB powers off deallocated ROB banks as does ARPA_AA, the average energy savings over ARPA are small. Its energy consumption is even 10% higher than that of ARPA for the 4-thread MIX group because of the significant performance degradation compared to ARPA. AdapROB does not aim to power off the maximum number of ROB entries to save energy and ROB adaptation is only for improving performance, which is different from ARPA_AA.

ARPA_ROB achieves significant energy savings over ARPA by using fewer ROB entries. The energy savings in MIX and ILP groups are much higher than that in MEM groups since performance degradation in MIX and ILP is not as significant as that in MEM groups when using a small number of ROB entries. On average, the energy savings of ARPA_ROB over ARPA are very close to that of ARPA_AA over ARPA.

By controlling the number of powered-on ROB entries, ARPA_AA fetches instructions just as needed. This not only saves the static and dynamic power associated with active ROB entries but also the dynamic power of units like IQ logic with additional wake-up and selection operations. ARPA_AA achieves significant energy savings over ARPA. Compared to ARPA, it achieves 8.4% and 12.8% energy savings for 2-thread and 4-thread workloads, respectively, while keeping the performance loss very low.

5.4 Sensitivity Analysis

In this section, we study the impact of the main parameters of our algorithm, namely, bank size, Threshold1 and Threshold2. Then, we compare the performance of different schemes when the amount of resources change. To simplify our experiments, we randomly selected 2 workloads from each group in Table 3, forming a set of 12 workloads for the sensitivity analysis.

5.4.1 Bank Size and Thresholds

Figure 14 shows the energy savings of ARPA_AA over ARPA as the bank size changes from 8 to 64 entries for the 12 randomly selected workloads. As we increase the bank size from 8 to 64, the energy savings of ARPA_AA

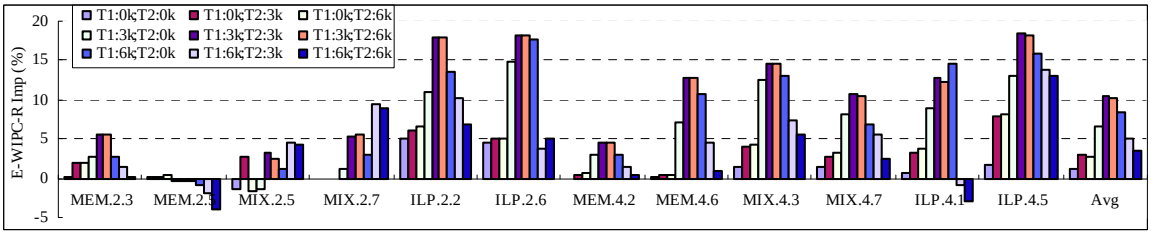

Fig. 15. *Energy-to-WIPC-Ratio (E-WIPC-R)* improvement of ARPA_AA over ARPA for different values of Threshold1 and Threshold2 for 12 out of the 42 workloads.

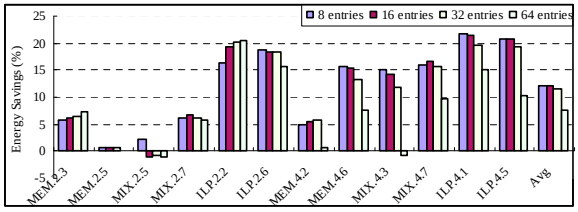

Fig. 14. Energy Savings of ARPA_AA over ARPA as the bank size changes for 12 out of 42 workloads.

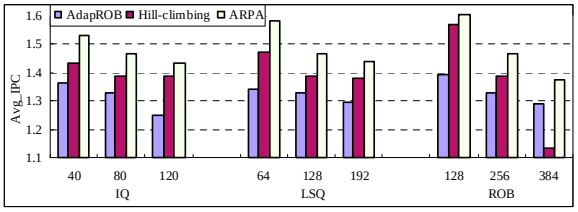

Fig. 16. *Avg_IPC* of different schemes over ICOUNT as the size of the queues changes for 12 out of 42 workloads

over ARPA decrease for most benchmarks. It is obvious that we can do a more fine-grained adaptation with a smaller bank size, thereby following program behavior more closely. Moreover, the energy consumption of ARPA_AA increases significantly when the bank size is 64. It is even higher than that of ARPA for MIX.2.5 and MIX.4.3, since a big bank size will cause greater performance loss, thus increasing the energy consumption. As the bank size decreases to a very low value, the area needed to implement the banked ROB components and the hardware implementation complexity of multi-banked structure increase significantly. Since banks of size 8, 16 and 32 do not differ greatly in energy consumption, it would be better to select a size of 32.

As we have seen in Section 3.4, ARPA_AA uses the number of full ROB occupancy cycles during an epoch (compared to Threshold1) and the number of cycles when the in-flight instructions of the reference thread fully occupy the allocated queue entries in an epoch (compared to Threshold2) to decide whether an architecture adaptation or resource partitioning is performed. We now examine how these two thresholds impact the *Single_WIPC* and energy consumption of ARPA_AA. Combining *Single_WIPC* and energy consumption, we use *Energy-to-WIPC-Ratio (E-WIPC-R)* as a metric.

Figure 15 shows the *E-WIPC-R* improvement of ARPA_AA over ARPA as both Threshold1 (T1) and Threshold2 (T2) increase from 0K to 6K, with a step size of 3K. As both thresholds increase, the architecture adaptation becomes more aggressive. As we can see, when T1 increases from 0K to 3K, the *E-WIPC-R* improvements of ARPA_AA increase significantly for most T2 values. However, as T1 increases from 3K to 6K, T2 plays a significant role and for most benchmarks, the *E-WIPC-R* improvement decreases significantly. Therefore, setting T1 to 6K is too aggressive. Among all the tested settings, ARPA_AA achieves the best *E-WIPC-R* improvements for T1=T2=3K.

We also tested ARPA_AA for both T1 and T2 between 2K to 4K, with a step size of 1K. The difference is small for the tested T1 and T2 values. This means that a small deviation of T1 and T2 from their optimal values has little effect on the behavior of ARPA_AA. This is a desirable property for our algorithm.

5.4.2 Queue Entries

We now examine the impact of the size of IQ, LSQ and ROB on the performance of different schemes. Figure 16 shows the average performance improvement of AdapROB, Hill-climbing and ARPA over ICOUNT as we change the number of LSQ entries from 40 to 80, and 120, the number of IQ entries from 64 to 128, and 192 and the number of ROB entries from 128 to 256, and 384. The size of the other two queues is kept unchanged as we change the number of one queue structure.

We can see that as the number of IQ, LSQ and ROB entries increases, the improvements over ICOUNT decrease for all the schemes since the increased number of resources mitigates the resource clogging problem of ICOUNT. ARPA achieves better performance improvements over ICOUNT than that of all the other schemes for any queue sizes. ARPA analyzes resource usage efficiency of each thread directly and assigns more resources to the thread which can use them more efficiently. In contrast, Hill-climbing partitions resources based on periodic "trials" to select the best partitioning and AdapROB allocates/deallocates ROB entries of a thread according to whether the allocation/deallocation adds/reduces pressure on IQ. When the number of IQ entries increases to 120, the performance improvement of AdapROB over ICOUNT decreases more significantly than those of the other two schemes. The reason is that as the number of IQ entries increases beyond a certain point, the competition among IQ entries reduces, thus

weakening the advantage of AdapROB. Hill-climbing is sensitive to the changes of ROB size and its performance improvements over ICOUNT decrease significantly (compared to that of ARPA) as the ROB size changes from 128 to 384. As the number of ROB entries increases, the search space of Hill-climbing increases accordingly, increasing the time spent in non-optimal partitions.

6 Conclusion

This paper has presented an Adaptive Resource Partitioning Algorithm (ARPA) for SMT processors. This algorithm identifies the resource usage efficiency of each thread using the CIPRE metric and allocates more resources to threads which can use them more efficiently. The more 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 Partitioning only achieves a 13.5% improvement over ICOUNT. Compared with the currently best-performing algorithm Hill-climbing, ARPA achieves a 5.7% *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 respect to the *Single_WIPC* metric, ARPA achieves a 14.0% improvement over ICOUNT and attains 6.8% and 5.7% improvements over Static Partitioning and Hill-climbing, respectively. With respect to the *Baseline_WIPC* metric, ARPA achieves 43.6%, 18.5% and 9.2% improvements over ICOUNT, Static Partitioning and Hill-climbing, respectively. With respect to *HMean_WIPC* metric, ARPA's improvement is 29.4%, 11.0%, and 7.2%, respectively.

We also explored the benefits of incorporating architecture adaptation into ARPA to adaptively control the number of powered-on resources and partition resources among threads for energy savings. Our experimental results show that ARPA with architecture adaptation can achieve 10.6% energy savings, while the performance loss is negligible.

Acknowledgments

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

References

[1] J. Abella and A. González, "Power-Aware Adaptive Issue Queue and Register File," *Proc. 10th Int'l Conf. High Performance*, pp. 34-43, Dec. 2003.

[2] D. H. Albonesi *et al.*, "Dynamically Tuning Processor Resources with Adaptive Processing," *IEEE Computer*, vol. 36, no. 12, pp. 49-58, Dec. 2003.

[3] 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.

[4] 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.

[5] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook and D. H. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," *Lecture Notes in Computer Science*, vol. 2008, pp. 25-39, Jan. 2001.

[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. Fernández, "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, June 2006.

[9] 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.

[10] 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.

[11] D. Folegnani and A. González, "Energy-effective Issue Logic," *Proc. 28th Int'l Symp. Computer Architecture*, pp. 230-239, Jun. 2001.

[12] 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.

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

[14] 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.

[15] D. T. Marr *et al.*, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, pp. 4-15, Feb. 2002.

[16] D. Ponomarev, G. Kucuk and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," *Proc. 34th Int'l Symp. Microarchitecture*, pp. 90-101, Dec. 2001.

[17] D. Ponomarev, G. Kucuk and K. Ghose, "Energy-Efficient Design of the Reorder Buffer," *Lecture Notes in Computer Science*, vol. 2451, pp. 289-299, 2002.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] D. M. Tullsen *et al.*, "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.

[24] 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.

[25] H. Wang, I. Koren and C. M. Krishna, "An Adaptive Resource Partitioning Algorithm for SMT Processors," *Proc. 17th Int'l Conf. Parallel Architecture and Compilation Techniques*, pp.230-239, Oct. 2008.

[26] W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," *Proc. First Int'l Symp. High Performance Computer Architecture*, pp. 49-58, June 1995.
[27] Z. Zhu and X. Zhang, "Look-Ahead Architecture Adaptation to Reduce Processor Power Consumption," *IEEE Micro*, vol. 25, no. 4, pp. 10-19, 2005.
[28] K. Wilcox and S. Manne, "Alpha Processors: A History of Power Issues and a Look to the Future," *Cool-Chips Tutorial*, Nov. 1999.
[29] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no.4, pp. 42-53, 2008.
[30] S. Eyerman and L. Eeckhout, "Memory-level parallelism aware fetch policies for simultaneous multithreading processors,"*ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no.1, pp. 1-33, 2009.

PLACE PHOTO HERE

Huaping Wang received the BS and MS degrees in 1999 and 2002, respectively, both from the Department of Information and Electronic Engineering of Zhejiang University, Hangzhou, China. In 2004, she joined the Architecture and Real-Time Systems (ARTS) lab in Department of Electrical and Computer Engineering, University of Massachusetts, Amherst as a PhD student and received the Ph.D. degree in 2010. Now she is a senior software engineer in Marvell Semiconductor Inc. Her current research interests include pre-silicon power performance analysis, embedded computing systems.

PLACE PHOTO HERE

Israel Koren (M'76 - SM'87 - F'91) is currently a Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He has been a consultant to numerous companies including IBM, Analog Devices, Intel, AMD and National Semiconductors. His research interests include Fault-Tolerant systems, Computer Architecture, VLSI yield and reliability, Secure Cryptographic systems, and Computer Arithmetic. He publishes extensively and has over 200 publications in refereed journals and conferences. He is an Associate Editor of the VLSI Design Journal, and the IEEE Computer Architecture Letters. He served as General Chair, Program Chair and Program Committee member for numerous conferences. He is the author of the textbook "Computer Arithmetic Algorithms," 2nd Edition, A.K. Peters, 2002, a co-author of "Fault Tolerant Systems," Morgan-Kaufman, 2007, and an editor/co-author of "Defect and Fault-Tolerance in VLSI Systems," Plenum, 1989.

PLACE PHOTO HERE

C. Mani Krishna received his PhD from the University of Michigan in 1984. Since then, he has been on the faculty of the department of Electrical and Computer Engineering at the University of Massachusetts. His research interests include real-time systems, fault-tolerant computing, performance evaluation, distributed computing, and power-aware systems. He has coauthored texts on real-time systems and fault-tolerant computing.