Power-Efficient Approaches to Redundant Multithreading

Niti Madan, *Student Member*, *IEEE*, and Rajeev Balasubramonian, *Member*, *IEEE*

Abstract—Noise and radiation-induced soft errors (transient faults) in computer systems have increased significantly over the last few years and are expected to increase even more as we move toward smaller transistor sizes and lower supply voltages. Fault detection and recovery can be achieved through redundancy. The emergence of chip multiprocessors (CMPs) makes it possible to execute redundant threads on a chip and provide relatively low-cost reliability. State-of-the-art implementations execute two copies of the same program as two threads (redundant multithreading), either on the same or on separate processor cores in a CMP, and periodically check results. Although this solution has favorable performance and reliability properties, every redundant instruction flows through a high-frequency complex out-of-order pipeline, thereby incurring a high power consumption penalty. This paper proposes mechanisms that attempt to provide reliability at a modest power and complexity cost. When executing a redundant thread, the trailing thread benefits from the information produced by the leading thread. We take advantage of this property and comprehensively study different strategies to reduce the power overhead of the trailing core in a CMP. These strategies include dynamic frequency scaling, in-order execution, and parallelization of the trailing thread.

Index Terms—Reliability, power, transient faults, soft errors, redundant multithreading (RMT), heterogeneous chip multiprocessors, dynamic frequency scaling.

1 Introduction

A recent study [24] shows that the soft-error rate [16] per chip is projected to increase by nine orders of magnitude from 1992 to 2011. This is attributed to growing transistor densities and lower supply voltages that increase susceptibility to radiation and noise. Such soft errors or transient faults do not permanently damage the device but can temporarily alter the state, leading to the generation of incorrect program outputs.

Fault tolerance can be provided at the circuit or process level. For comprehensive fault coverage, every circuit would have to be redesigned. This not only increases design complexity, but also has the potential to lengthen critical paths and reduce clock frequencies. For this reason, many recent studies [1], [7], [17], [19], [21], [25], [27] have explored architecture-level solutions that can provide fault tolerance with modest performance and complexity overheads. In most solutions, generally referred to as redundant multithreading (RMT), an instruction is executed twice and results are compared to detect faults. Most studies on reliability have paid little attention to power overheads in spite of the fact that future microprocessors will have to balance three major metrics: performance, power, and reliability. A recent paper by Gomaa and Vijaykumar [8] opportunistically employs redundancy, thereby deriving a desirable point on the performance-reliability curve. Because redundancy is occasionally turned off, this approach also indirectly reduces power overheads. In this paper, we focus on maintaining a constant level of error coverage and explore different strategies to improve the power efficiency of reliability mechanisms (while occasionally compromising marginal amounts of performance).

In a processor that employs redundancy, the "*checker instruction*" can be made to flow through a similar pipeline as the "*primary instruction*."[1] This approach is well suited to a chip multiprocessor (CMP) or simultaneous multithreaded processors (SMTs), where the processor is already designed to accommodate multiple threads. With minor design modifications, one of the thread contexts can be made to execute the checker thread [7], [17], [21], [27]. Furthermore, thread contexts can be dynamically employed for either checker or primary threads, allowing the operating system or application designer to choose between increased reliability and increased multithreaded performance. However, this approach has significant power overheads as each checker instruction now flows through a complex out-of-order (OoO) pipeline. In an alternative approach, the checker thread can flow through a heavily modified helper pipeline that has low complexity [1], [25]. Even though the area overhead is modest, the area occupied by this helper pipeline is not available for use by primary threads even if reliability is not a primary concern for the application. As we shall show in this paper, heterogeneous CMPs can allow us to derive the better of the two approaches above.

As a starting point, we consider the following RMT architecture based on the Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) model proposed by Gomaa et al. [7]. The primary thread executes on an OoO

1. The main program thread is referred to as the *primary* or *leading thread*. The redundant thread is referred to as the *checker* or *trailing thread*. Correspondingly, these threads execute on *primary/leading cores* or *checker/trailing cores*.

• *The authors are with the School of Computing, University of Utah, 50 S. Central Campus Drive, Rm. 3190, Salt Lake City, UT 84112. E-mail: {niti, rajeev}@cs.utah.edu.*

Manuscript received 1 Sept. 2006; revised 28 Jan. 2007; accepted 23 Feb. 2007; published online 28 Mar. 2007.
Recommended for acceptance by R. Iyer and D.M. Tullsen.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0270-0906.
Digital Object Identifier no. 10.1109/TPDS.2007.1090.

1045-9219/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

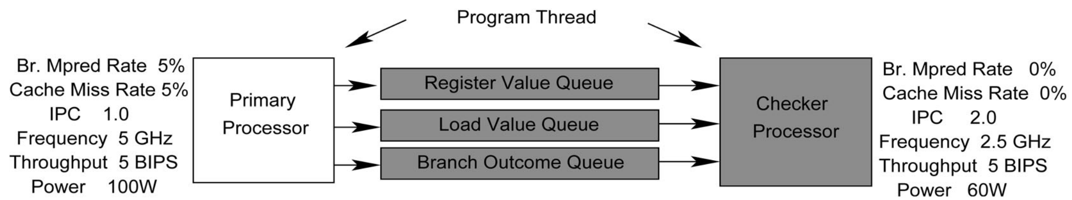

Fig. 1. An example of the effect of scaling the checker core's frequency. In this example, by operating at half the peak frequency, the checker's dynamic power is reduced from 80 to 40 W. Leakage power is not affected.

core and the checker thread executes on a different OoO core within a CMP. Branch outcomes, load values, and register results produced by the primary thread are fed to its checker thread in the neighboring core, so it can detect and recover from faults (as shown in Fig. 1). In an effort to reduce the power overhead of CRTR, we make the following observations: The checker thread experiences no branch mispredictions or cache misses because of the values fed to it by its primary thread. The checker thread is therefore capable of a much higher instruction-per-cycle (IPC) throughput rate than its primary thread. This allows us to operate the checker core in a low-power mode while still matching the leading thread's throughput. Fig. 1 shows how the checker core's frequency can be scaled down in order to reduce dynamic power. We also explore the potential of using an in-order pipeline for the checker core and show that some form of value prediction is required to enable it to match the throughput of the primary thread. We also extend our evaluation to multithreaded workloads executing on a CMP of SMTs. Finally, we examine the potential of dynamic voltage scaling and of parallelization of the verification workload. Some of the conclusions of this work resonate well with prior research such as the proposal of Austin to employ in-order checker pipelines that are fed with leader-generated inputs [1]. On the other hand, some of our conclusions argue against the voltage scaling approach proposed by Rashid et al. [19]. The major contributions of this paper are listed as follows:

- dynamic frequency scaling (DFS) techniques that match throughputs of leading and trailing threads and that are able to execute the trailing core at an effective frequency as low as 0.42 times peak frequency,
- techniques that increase intercore traffic to enable the use of simple and power-efficient trailer cores,
- a combination of the above ideas that helps reduce the overhead of redundancy to merely 10 percent of the leader core's power consumption,
- an exhaustive design space exploration, including the effects of parallelizing the verification workload and employing voltage scaling,
- quantifying the power performance trade-off when scheduling the redundant threads of a multi-threaded workload, and
- analytical models that enable rough early estimates of different RMT organizations.

The paper has been organized as follows: Section 2 outlines the relationship of this work with prior art. Section 3 describes the RMT implementations that serve as baseline processor models in this study. Section 4 describes various power reduction strategies for the trailing thread. The proposed ideas are evaluated in Section 5 and we summarize the conclusions of this study in Section 6.

2 Related Work

Many fault-tolerant architectures [1], [7], [17], [20], [21], [22], [27], [28] have been proposed over the last few years and our baseline models are based on previously proposed RMT designs. Most of this prior work has leveraged information produced by the leading thread, but the focus has been on the performance-reliability trade-off, with few explicit proposals for power-efficiency. Active-Stream/redundant Stream SMT (AR-SMT) [22] was the first design to use multithreading for fault detection. AR-SMT proposed sending all register values to the trailing thread to boost its performance. In our work, we exploit register values to enable in-order execution of the trailing thread for power efficiency. Mukherjee et al. later proposed fault detection by using simultaneous multithreading and chip-level RMTs [17], [21]. Vijaykumar et al. augmented the above techniques with recovery mechanisms [7], [27]. Most of these research efforts have been targeted at improving thread-level throughput and have not been optimized for power efficiency. Gomaa et al. [7] discuss techniques such as Death and Dependence-Based Checking Elision (DDBCE) to reduce the bandwidth requirements (and, hence, power overheads) of the intercore interconnect. Our proposals, on the other hand, advocate transferring more data between threads to enable power optimizations at the trailer. Mukherjee et al. [18] characterize the architectural vulnerability factors (AVFs) of various processor structures. Power overheads of redundancy can be controlled by only targeting those processor structures that have a high AVF.

Some designs, such as DIVA [1] and SHared REsource Checker (SHREC) [25], are inherently power-efficient because the helper pipelines that they employ to execute redundant instructions are in-order-like. DIVA has two in-order pipelines: *Checkcomm*, which checks all memory values, and *Checkcomp*, which checks all computations. These helper pipelines are fed with input values generated by the primary pipeline. However, these designs require (potentially intrusive) modifications to the pipeline of the conventional primary microarchitecture and the helper pipelines cannot be used to execute primary threads. We extend this concept by executing the redundant thread on a general-purpose in-order core augmented with register value prediction (RVP) and by limiting the data that has to be extracted from the primary pipeline. Even though

DIVA was among the first RMT proposals, recent research in this area has moved away from that concept and focused more on the exploitation of heavily threaded hardware. Our conclusions show that the DIVA concepts are worth revisiting and are complexity effective in a processor with heterogeneous in-order and OoO cores.

A recent paper by Rashid et al. [19] represents one of the first efforts at explicitly reducing the power overhead of redundancy. In their proposal, the leading thread is analyzed and decomposed into two parallel verification threads that execute on two other OoO cores. Parallelization and the prefetch effect of the leading thread allow the redundant cores to operate at half the peak frequency and lower supply voltage and still match the leading thread's throughput. Our approach differs from that work in several ways: 1) In [19], redundant loads retrieve data from caches, leading to complex operations to maintain data coherence between multiple threads. In our implementation, redundant instructions receive load results and even input operands from the leading thread. We claim that, by investing in intercore bandwidth, trailer core overheads can be reduced. 2) Rashid et al. [19] rely on voltage scaling and body biasing to benefit from parallelization. We have explicitly steered away from such techniques because of the associated overheads. Unlike their work, we leverage DFS and in-order execution. 3) Our analytical results show that parallelization of the verification workload yields little benefit if we are already employing in-order cores augmented with RVP.

Some papers have looked at reducing resource and instruction redundancy for reducing performance and power overheads in RMT techniques without reducing the fault coverage. Kumar and Aggarwal [12] apply register reuse and narrow-width operand register sharing techniques to reduce the performance and power overheads in the register file. In another recent paper by the same authors [13], many instructions are classified as *self-checking*, such as those that have a *zero* operand. The results produced by these instructions will often be equal to their nonzero operands and these instructions need not be redundantly executed by the trailing thread. These techniques indirectly reduce the power overheads by executing fewer instructions for verification. Other approaches that reduce the power overheads indirectly are RMT techniques that reduce the fault coverage such as opportunistic RMT [8]. In that work, in addition to exploiting instruction reuse, redundant threads execute only when the primary thread is stalled on level-2 (L2) cache misses. This paper explores techniques that can reduce power and area overheads while maintaining a constant level of error coverage. These techniques are often orthogonal to other techniques that, for example, trade off reliability for better performance or power.

3 Baseline Reliable Processor Model

We have based our reliable chip-multiprocessor architecture on the model proposed by Gomaa et al. [7] and Mukherjee et al. [17]. The architecture consists of two communicating cores that execute copies of the same application for fault detection. One of the cores (the leading core) executes ahead of the second core (the trailing core) by a certain amount of slack. The leading core communicates its committed register results to the trailing core for comparison of values to detect faults (Fig. 1). Load values are also passed to the trailing core, so it can avoid reading values from memory that may have been recently updated by other devices. Thus, the trailing thread never accesses its level-1 (L1) data cache and there is no need for coherence operations between the L1 data caches of the leading and trailing cores. This implementation uses *asymmetric commit* to hide intercore communication latency: The leading core is allowed to commit instructions before checking. The leading core commits stores to a store buffer (StB) instead of to memory. The trailing core commits instructions only after checking for errors. This ensures that the trailing core's state can be used for a recovery operation if an error occurs. The trailing core communicates its store values to the leading core's StB and the StB commits stores to memory after checking.

The communication of data between the cores is facilitated by the first-in, first-out (FIFO) Register Value Queue (RVQ) and Load Value Queue (LVQ). As a performance optimization, the leading core also communicates its branch outcomes to the trailing core (through a branch outcome queue (BOQ)), allowing it to have perfect branch prediction. The power saved in the trailing core by not accessing the L1D cache and the branch predictor is somewhat offset by the power consumption of the RVQ and LVQ. If the slack between the two cores is at least as large as the reorder buffer (ROB) size of the trailing core, then it is guaranteed that a load instruction in the trailing core will always find its load value in the LVQ. When external interrupts or exceptions are raised, the leading thread must wait for the trailing thread to catch up before servicing the interrupt.

The assumed fault model is exactly the same as in [7] and [17]. The following conditions are required in order to detect and recover from a single fault:

- The data cache, LVQ, and buses that carry load values must be error-correcting code (ECC) protected as the trailing thread directly uses these load values.
- When an error is detected, the register file state of the trailing thread is used to initiate recovery. The trailing thread's register file must be ECC protected to ensure that values do not get corrupted once they have been checked and written into the trailer's register file.

Other structures in each core (including the RVQ) need not have ECC or other forms of protection as disagreements will be detected during the checking process. The BOQ need not be protected as long as its values are only treated as branch prediction hints and confirmed by the trailing pipeline. Similarly to the baseline model in [7] and [17], we assume that the trailer's register file is not ECC protected. Hence, a single fault in the trailer's register file can only be detected.[2] All other faults can be detected and recovered from. The proposed mechanisms in this paper preserve this basic fault coverage.

In our single-thread model, we assume an implementation where each core on the CMP can only support a single thread. Our multithread model is based on the CRTR architecture [7], where each core is a dual-threaded SMT. In

2. If no ECC is provided within the register file, then Triple Modular Redundancy will be required to detect and recover from a single fault.

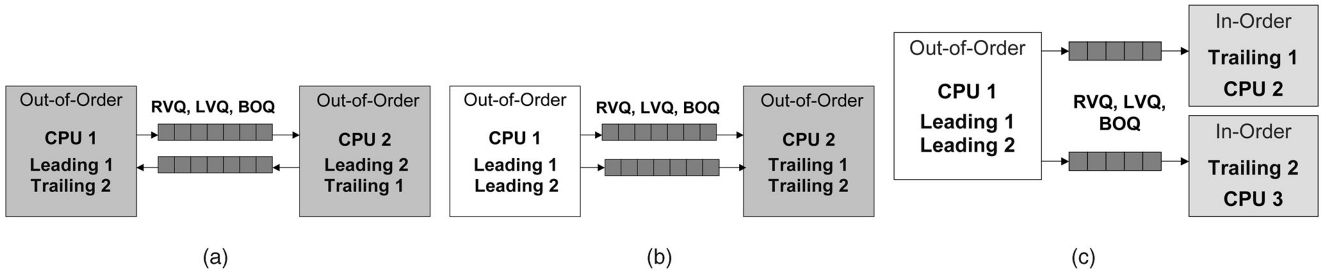

Fig. 2. Power-efficient chip-level RMT design space. (a) CRTR. (b) P-CRTR. (c) P-CRTR-in order.

the CRTR architecture, the trailing thread of one application shares its core with the leading thread of a different application (shown in Fig. 2a). We require that the slack for each application remain between two thresholds: $TH1$ and $TH2$. The lower threshold $TH1$ is set to the ROB size available to the trailing thread so that load results can be found in the LVQ. The higher threshold $TH2$ is set to the size of the RVQ minus the ROB size for the leading thread so that all completing instructions in the leading thread are guaranteed to find an empty slot when writing results into the RVQ. Similarly to the fetch policy in [7], the slack values determine which threads are allowed to fetch within each SMT core. If the slack for an application is less than $TH1$, then the trailing thread is not allowed to fetch and, if the slack is greater than $TH2$, then the leading thread is not allowed to fetch. In cases where both threads within an SMT core are allowed to fetch, the ICOUNT heuristic [26] is employed.

4 Managing Power Overheads

4.1 Power Reduction Strategies for the Trailing Core

For a single-thread workload, we propose that the leading and trailing thread execute on neighboring cores. If each core has SMT capability, then it is possible to execute the leading and trailing threads on a single core and this avoids the overhead of intercore communication. However, as we will show later, the power savings possible by executing the trailer on a neighboring core are likely to offset the power overheads of intercore communication.

For a multithreaded workload, the CRTR implementation executes unrelated leading and trailing threads on a single SMT core (Fig. 2a). Since the trailing thread never executes wrong-path instructions and never accesses the data cache, the leading thread that executes in tandem is likely to experience little contention, thereby yielding high throughputs. Applying a power-saving strategy to a trailing thread in this setting will slow the leading thread that executes on that same core. Hence, to enable power optimizations, we propose executing two leading threads on the same SMT core and the corresponding trailing threads on neighboring cores, referred to as P-CRTR (Fig. 2b). By changing the assignment of threads to cores, we are not increasing intercore bandwidth requirements: The same amount of data as in CRTR is being communicated between cores. Since the leading threads compete for resources within a single core in P-CRTR, a performance penalty is incurred.

An RMT system attempts to maintain a roughly constant slack between leading and trailing threads. Since the trailing thread benefits from perfect caching and branch prediction, it tends to catch up with the leading thread. This provides the opportunity to throttle the execution of the trailer in a manner that lowers its power consumption and maintains the roughly constant slack.

4.1.1 Dynamic Frequency Scaling (DFS)

The first approach that we consider to throttle trailing thread execution is DFS, a well-established technique that allows dynamic power to scale down linearly with clock frequency [15]. It is a low-overhead technique: In Intel's Montecito, a frequency change can be effected in a single cycle [15]. In most systems, DFS does not result in a dynamic energy reduction as execution time is also linearly increased. As we show in our results, the application of DFS to the trailing core of an RMT system has a minimal impact on execution time. Hence, in this particular case, DFS results in a reduction in dynamic power *and* dynamic energy. DFS does not impact leakage power (and leakage energy) dissipated by the trailer.

A competitive alternative to DFS is run and stall, where the trailing thread operates at peak frequency for a while and then shuts off its clock for a while. We expect that the fraction of stall time in run and stall will equal the average frequency reduction in a DFS-based mechanism. Run and stall will also not impact leakage unless voltage is turned off during the stall (voltage changes are known to have much higher delay overheads). Given the similarity with DFS, we do not further evaluate run and stall in this paper.

4.1.2 In-Order Execution

When throttling the trailing core, we may see greater power savings by executing the trailing thread on an in-order core. A short and simple pipeline can have a significant impact on both dynamic and leakage power. Unfortunately, for many program phases, an in-order core, even with a perfect D-cache and branch predictor, cannot match the throughput of the leading OoO core. Hence, some enhancements need to be made to the in-order core. We considered the effect of increasing fetch bandwidth and functional units in a simple in-order core, but that did not help match the leading core's throughput. In fact, for our simulation parameters, even with a perfect cache and branch predictor, doubling the fetch bandwidth and the arithmetic logic units (ALUs) of

the in-order core resulted in only about a 1 percent performance improvement. This is not surprising because data dependency is the biggest bottleneck that limits instruction level parallelism (ILP) in an in-order core. As soon as a pair of dependent instructions is encountered, the dependent instruction (and every instruction after it) is forced to stall at least until the next cycle, regardless of the fetch and ALU bandwidth.

We therefore propose the use of RVP. Along with the result of an instruction, the leading thread can also pass the input operands for that instruction to the trailing thread. Instructions in the trailing core can now read their input operands from the RVQ instead of from the register file. Such a register value predictor has a 100 percent accuracy in the absence of soft errors. With perfect RVP, instructions in the trailer are never stalled for data dependences and ILP is constrained only by a lack of functional units or instruction fetch bandwidth. The value predictions must be confirmed by the trailing core or else an error in the leading core may go undetected. When an instruction in the trailing core is committed, the trailing register file is read to confirm that the input operands match those in the RVQ. The error coverage is exactly as before: A single soft error in the trailer's register file can be detected, whereas a single soft error elsewhere can be detected and recovered from. Although RVP entails some changes to the commit pipeline in the trailer, it allows us to leverage the benefits of an in-order core.

It may be possible to even apply DFS to the in-order core to further reduce dynamic power. Contrary to previous studies [7] that attempt to reduce intercore traffic, we believe that it may be worthwhile to send additional information between cores because of the power optimizations that it enables at the trailer.

The use of an in-order core has another major advantage (not quantified in this paper). Since an in-order core has less speculative state, it requires fewer rename registers. It may be possible to include ECC within the in-order core's small register file and still meet cycle time constraints. As discussed in Section 3, a trailing core with an ECC-protected register file has higher error recovery coverage.

4.1.3 Workload Parallelization

Assuming that power is a quadratic function of performance [9] and that a workload can be perfectly parallelized, it is more power efficient to execute the workload in parallel across N low-performance cores than on a single high-performance core. The trailing thread is an example of such a highly parallel workload [19]. At regular intervals, the leading thread spawns a new trailing thread that verifies the results for a recently executed contiguous chunk of the program. At the start of the interval, the initial register state must be copied into the trailing core. In the next section, we show that such an approach yields little benefit.

4.2 Dynamic Frequency Scaling Algorithm for a Single Thread

The power-efficient trailing cores rely on a DFS mechanism to match their throughputs to that of the leading core. For a single-thread workload, the goal of the DFS mechanism is to select a frequency for the trailing thread so that a constant slack is maintained between the leading and trailing threads. In essence, if the IPC of the leading thread is denoted by IPC_L and the IPC of the trailing thread is IPC_T, then we can maintain equal throughputs and constant slack by setting the trailing core's frequency f_T to $f_L \times IPC_L/IPC_T$, where f_L is the leading core's frequency. The same effect can be achieved with a simple heuristic that examines the size of the buffer that feeds results from the leading to the trailing thread (for example, the RVQ).

Initially, the trailing core is stalled until the leading core commits N instructions. At this point, an RVQ (that does not filter out register values) will have N entries. The trailing core then starts executing. The RVQ is checked after a period of every T cycles to determine the throughput difference between the two cores. If the RVQ has $N - thresh$ entries, then it means that the trailing thread is starting to catch up with the leading thread. At this point, the frequency of the trailing thread is lowered one step at a time. If the RVQ has $N + thresh$ entries, then it means that the leading thread is starting to pull away and the frequency of the trailing thread must be increased. We observed that increasing the frequency in steps causes the slack to increase drastically as the leading thread continues to extend its lead over subsequent intervals. Note that the leading thread has just entered a high IPC phase of the program, whereas the trailing thread will have to commit N more instructions before it enters that phase itself. Once all of the queues are full, the leading thread is forced to stall. To minimize this occurrence, the frequency of the trailing thread is immediately increased to the leading thread's peak frequency if the RVQ has $N + thresh$ entries.

The smaller the value of T is, the faster the mechanism reacts to throughput variations. For our simulations, we conservatively assume a 10-cycle overhead for every dynamic frequency change. The time interval T is selected to be 1,000 cycles so that the overhead of frequency scaling is marginal. To absorb throughput variations in the middle of an interval, a slack of 1,000 is required. To ensure that the RVQ is half full on the average, we set $N - thresh$ to be 400 and $N + thresh$ to be 700. Frequency is reduced in steps that equal $f_L \times 0.1$.

4.3 Dynamic Frequency Scaling Algorithm for Multithreaded Workloads

If each trailer executes on a separate core (as in Fig. 2c), then the single-thread DFS algorithm in Section 4.2 can be applied to tune the frequency of each trailing core. If two trailing threads execute on a single SMT core (as in Fig. 2b), then the DFS algorithm will have to consider the slack for both threads in determining the core frequency. We employ fetch throttling strategies to accommodate the potentially conflicting needs of coscheduled threads. Rather than always using ICOUNT as the fetch policy for the trailing core, it helps to periodically throttle fetch for the thread that has a lower IPC_L/IPC_T ratio and give a higher priority to the other trailing thread. This further boosts the IPC value for the other trailing thread, allowing additional frequency reductions.

The detailed algorithm for invoking fetch throttling and frequency scaling is formalized in Table 1. To allow a fine-grained control of each thread, we employ three slack

TABLE 1
Fetch and Frequency Policies Adopted for Leading and Trailing Cores in P-CRTR

l_1 = leading thread for application 1; l_2 = leading thread for application 2
t_1 = trailing thread for application 1; t_2 = trailing thread for application 2
s_1 = slack for application 1; s_2 = slack for application 2
l_1 and l_2 execute on core 1; t_1 and t_2 execute on core 2
The fetch and frequency policy attempt to maintain a slack between $TH0$ and $TH1$.
In our simulations, $TH0$ is set to 80, $TH1$ is set to 700, $TH2$ is set to 1000.

Slack conditions	$s_2 \leq TH0$	$TH0 < s_2 \leq TH1$	$TH1 < s_2 \leq TH2$	$TH2 < s_2$
$s_1 \leq TH0$	ICOUNT : stall f_t −= $0.1 f_{peak}$	ICOUNT : t_2 f_t −= $0.1 f_{peak}$	ICOUNT : t_2 f_t += $0.1 f_{peak}$	$l_1 : t_2$ $f_t = f_{peak}$
$TH0 < s_1 \leq TH1$	ICOUNT : t_1 f_t −= $0.1 f_{peak}$	ICOUNT : ICOUNT f_t −= $0.1 f_{peak}$	ICOUNT : ICOUNT $f_t = f_{peak}$	$l_1 : t_2$ $f_t = f_{peak}$
$TH1 < s_1 \leq TH2$	ICOUNT : t_1 f_t += $0.1 f_{peak}$	ICOUNT : ICOUNT $f_t = f_{peak}$	ICOUNT : ICOUNT $f_t = f_{peak}$	$l_1 : t_2$ $f_t = f_{peak}$
$TH2 < s_1$	$l_2 : t_1$ $f_t = f_{peak}$	$l_2 : t_1$ $f_t = f_{peak}$	$l_2 : t_1$ $f_t = f_{peak}$	stall : ICOUNT $f_t = f_{peak}$

For each entry in the grid, the first two terms indicate which thread is fetched for core 1 and core 2, and the last term indicates the frequency selected for the trailing core. Fetch policies are adjusted every cycle and frequencies can be adjusted at intervals of 1,000 cycles.

thresholds. The action for each case is based on the following guidelines:

1. If the slack for a trailer is less than $TH0$, then there is no point fetching instructions for that trailer.
2. If the slack for a trailer is between $TH0$ and $TH1$ (the desirable range), then the decision depends on the state of the other thread.
3. If the slack is between $TH1$ and $TH2$, then we may need to quickly ramp up the frequency to its peak value in an effort to keep slack under control.
4. If the slack is greater than $TH2$, then we can stop fetching instructions for the leader.

Table 1 describes the action taken for every combination of slack values for both threads. As before, $TH0$ is a function of the trailing thread's ROB size, $TH2$ is a function of the sizes of the RVQ and the leader's ROB size, and $TH1$ is picked so that the RVQ will be half full on the average. The leading core always employs the ICOUNT fetch heuristic to select between the two leading threads unless one of the slacks is greater than $TH2$. Fetch throttling is a low-overhead process and can be invoked on a per-cycle basis. Slack values are evaluated every cycle and any changes to the fetch policy are instantly implemented. Changes in frequency are attempted only every 1,000 cycles to limit overheads. The above algorithm has been designed to react quickly to changes so as to minimize stalls for leading threads. This leads to frequency changes in almost every interval unless the peak or lowest frequency is being employed. Incorporating some hysteresis in the algorithm reduces the frequency change overhead, but introduces additional stalls for leading threads.

4.4 Analytical Model

To better articulate the factors that play a role in the overall power consumption, we derive simple analytical power models for the proposed RMT systems. These models also allow an interested reader to tweak parameters (contribution of leakage, ratio of in-order to OoO power, and so forth) and generate rough estimates of power overheads without detailed simulations. The analytical equations were derived after studying the detailed simulation results described in the next section. We found that, when various parameters were changed, our detailed simulation results were within 4 percent of the analytical estimates.

For starters, consider leading and trailing threads executing on neighboring OoO cores in a CMP. Assuming that the leader has $wrongpath_factor$ times the activity in the trailer (because of executing instructions along the wrong path), the total power in the baseline RMT system is given by

$$Baseline_power = leakage_{leading} + dynamic_{leading} + leakage_{leading} + dynamic_{leading}/wrongpath_factor.$$

When DFS is applied to the trailing core, and eff_freq is its average operating frequency (normalized to the peak frequency), assuming marginal stalls for the leading thread, the total power in the system is given by

$$DFS_ooo_power = leakage_{leading} + dynamic_{leading} + leakage_{leading} + dynamic_{leading} \times eff_freq/wrongpath_factor.$$

If scaling the frequency by a factor eff_freq allows us to scale the voltage by a factor $eff_freq \times v_factor$ (in practice, v_factor is greater than 1), then the trailing core's power is

$$DVFS_ooo_power = leakage_{leading} + dynamic_{leading} + leakage_{leading} \times eff_freq \times v_factor + dynamic_{leading} \times eff_freq^3 \times v_factor^2/wrongpath_factor.$$

If an in-order core with RVP is employed for the trailing thread, then the following equation is applicable, assuming that the in-order core consumes lkg_ratio times less leakage and dyn_ratio times less dynamic power than the OoO leading core:

$$DFS_inorder_power = leakage_{leading} + dynamic_{leading} + leakage_{leading}/lkg_ratio + dynamic_{leading} \times eff_freq/(wrongpath_factor \times dyn_ratio) + RVP_overhead. \tag{1}$$

We now consider the effect of parallelizing the verification workload across N in-order trailing cores. Assuming that we only employ DFS for each in-order core, trailing thread power is given by

$$DFS_inorder_WP_power = leakage_{leading} + dynamic_{leading} + N \times leakage_{leading}/lkg_ratio + N \times dynamic_{leading} \times eff_freq/(wrongpath_factor \times dyn_ratio) + RVP_overhead.$$

Note that the dynamic power remains the same as in (1). eff_freq goes down by a factor N, but that amount is now expended at N different cores. In other words, the same amount of work is being done in either case. The leakage power increases because leakage is a function of the number of transistors being employed. Parallelization has a benefit only if we are also scaling voltage. Power is then expressed as follows:

$$DVFS_inorder_WP_power = v_factor \times leakage_{leading}/lkg_ratio + N \times v_factor^2 \times dynamic_{leading} \times eff_freq/(wrongpath_factor \times dyn_ratio \times N^2) + RVP_overhead.$$

Finally, similar models can be constructed for CRTR and P-CRTR multithreaded models. P_CRTR_ooo represents a model where both trailing threads execute on an SMT OoO core, $P_CRTR_inorder$ represents a model where each trailing thread executes on its individual in-order core, and $slowdown$ represents the throughput slowdown when executing leading threads together instead of with trailing threads. $RVP_overhead$ includes additional power consumed within the RVQ to enable RVP:

$$Energy_{CRTR} = 2 \times (leakage_{ooo} + dynamic_{ooo} \times (1 + wrongpath_factor))$$

$$Energy_{P_CRTR_ooo} = slowdown \times (2 \times leakage_{ooo} + dynamic_{ooo} \times (wrongpath_factor + wrongpath_factor) + dynamic_{ooo} \times eff_freq \times (1+1))$$

$$Energy_{P_CRTR_inorder} = slowdown \times (leakage_{ooo} \times (1 + 2/lkg_ratio) + dynamic_{ooo} \times (wrongpath_factor + wrongpath_factor) + 2 \times RVP_overhead + dynamic_{ooo} \times effective_frequency \times (1+1)/dyn_ratio).$$

Parameters such as $slowdown$, eff_freq, and $wrongpath_factor$ have to be calculated through detailed simulations. For example, for our simulation parameters, $wrongpath_factor$ was 1.17, $slowdown$ for P_CRTR_ooo was 9.4 percent, and eff_freq for the single-thread model was 0.44.

4.5 Implementation Complexities

This section provides an analysis of the complexity introduced by the mechanisms in this paper. First, to enable DFS, each core must have a clock divider to independently control its operating frequency. The clocks may or may not be derived from a single source. The buffers between the two cores must be designed to allow variable frequencies for input and output. Multiple clock domain processors employ such buffers between different clock domains [23]. While changing the frequency of the trailing core, we will make the conservative assumption that the leading and trailing cores are both stalled until the change has stabilized. The dynamic frequency selection mechanism can be easily implemented with a comparator and a few counters that track the size of the RVQ, the current frequency, the interval, and the threshold values.

A large slack is required to absorb throughput variations within an interval, also requiring that we implement a large RVQ, LVQ, BOQ, and StB. To accommodate a slack of 1,000 instructions, we implement a 600-entry RVQ, a 200-entry BOQ, a 400-entry LVQ, and a 200-entry StB per trailing thread. All queues and buffers are modeled as FIFO queues. Their access is off the critical path, but can incur nontrivial power overheads. Values are written to and read from these queues in sequential order, allowing them to be implemented with single-read and write ports (each row has as many entries as the fetch/commit width). The peak power of the largest structure, the RVQ, was computed to be 0.89 W (with Wattch's power model [3]). It must be noted that low-overhead DFS (such as the single-cycle overhead in Montecito) enables a smaller interval and, therefore, a smaller slack, RVQ, LVQ, BOQ, and StB. Hence, our assumptions for intercore power overheads are pessimistic.

Although an in-order core can yield significant power savings, additional power is expended in implementing RVP. In one possible implementation, each entry of the RVQ now contains the instruction's source operands in addition to the result. This increases the RVQ's peak power consumption from 0.89 W to 2.59 W. As an instruction flows through the in-order pipeline, it reads the corresponding input operands from the RVQ and control signals are set so that the multiplexer before the ALU selects these values as inputs. The pipeline is modified so that the register file is read at the time of commit and not before the execute stage. Our simulations estimate that RVP incurs an additional average power cost of 2.54 W for intercore transfers. This estimate is admittedly simplistic, as it does not take the cost of pipeline modifications into account.

5 Results

5.1 Methodology

We use a multithreaded version of Simplescalar 3.0 [4] for the Alpha AXP ISA for our simulations. The simulator has been extended to implement a CMP architecture, where each core is a two-way SMT processor. Table 2 shows relevant simulation parameters. The Wattch [3] power model has been extended to model power consumption for the CMP architecture at a 90 nm technology at 5 GHz and 1.1 V supply voltage. The aggressive clock gating

TABLE 2
Simplescalar Simulation Parameters

Branch Predictor	Comb. bimodal/2-level (per core)	Bimodal Predictor Size	16384
Level 1 Predictor	16384 entries, history 12	Level 2 Predictor	16384 entries
BTB	16384 sets, 2-way	Branch Mpred Latency	12 cycles
Instruction Fetch Queue	32 (per Core)	Fetch width/speed	4/2 (per Core)
Dispatch/Commit Width	4 (per Core)	IssueQ size	40 (Int) 30 (FP) (per Core)
Reorder Buffer Size	80 (per Thread)	LSQ size	100 (per Core)
Integer ALUs/mult	4/2 (per Core)	FP ALUs/mult	1/1 (per Core)
(Single thread) L1 I-cache	32KB 2-way (per Core)	L1 D-cache	32KB 2-way, 2-cyc (per Core)
(Multi-thread) L1 I-cache	128KB 2-way (per Core)	L1 D-cache	128KB 2-way, 2-cyc (per Core)
L2 unified cache	2MB 8-way, 20 cycles (per Core)	Frequency	5 GHz
I and D TLB	256 entries, 8KB page size	Memory Latency	300 cycles for the first chunk

model (cc3) has been assumed throughout. Wattch's RAM array and wire models were used to compute the power dissipated by the intercore buffers (RVQ, LVQ, and so forth). The RVQ was modeled as a single-read and single-write ported structure with 150 total rows, each row accommodating results for four instructions (32 bytes of data), resulting in a peak power dissipation of 0.89 W. With RVP included, the size of a row expands to 96 bytes and the peak power dissipation to 2.59 W. To model the intercore bus, power-optimized wires [2] were employed. The distance between two cores was assumed to be equivalent to the width of a single core (4.4 mm, which is obtained from [11] and scaled to a 90-nm process). Assuming a bus width of 35 bytes (without RVP), we computed a peak power consumption of 0.67 W for the intercore bus per trailing thread. With RVP, the interconnect power consumption is 2 W per thread.

Although Wattch provides reliable relative power values for different OoO processor configurations, we felt that it did not accurately capture the relative power values for OoO and in-order cores. An in-order core's power efficiency is derived from its simpler control paths and Wattch primarily models the data paths within the processor. Hence, the in-order power values obtained from Wattch were multiplied by a scaling factor to bring these numbers more in tune with empirical industrial data. The scaling factor was chosen so that the average power for in-order and OoO cores was in the ratio 1:7 or 1:2. This ratio is based on relative power consumptions for commercial implementations of Alpha processors (after scaling for technology) [10]. Our in-order core is loosely modeled after the quad-issue Alpha EV5 that consumes half the power of the single-thread OoO Alpha EV6 and $1/7$th the power of the multithread OoO Alpha EV8. The EV8 is perhaps more aggressive than OoO cores of the future (such as Intel's Core) and, similarly, the EV5 may be more aggressive than the typical future in-order core (such as Sun's Niagara). Hence, the above ratios are only intended to serve as illustrative design points from a single family of processors. The analytical model can be used to estimate power overheads for other power ratios. The baseline peak frequency for all cores is assumed to be 5 GHz.

As an evaluation workload, we use the eight integer and eight floating-point benchmark programs from the SPEC2k suite that are compatible with our simulator. The executables were generated with peak optimization flags. The programs were fast-forwarded for 2,000,000,000 instructions, executed for 1,000,000 instructions to warm up various structures, and measurements were taken for the next 100,000,000 instructions. To evaluate multithreaded models, we formed a multiprogrammed benchmark set consisting of 10 different pairs of programs. Programs were paired to generate a good mix of high-IPC, low-IPC, FP, and Integer workloads. Table 3 shows our benchmark pairs. Multithreaded workloads are executed until the first thread commits 100,000,000 instructions.

5.2 Single-Thread Results

We begin by examining the behavior of a single-threaded workload. Fig. 3 shows the IPCs of various configurations, normalized with respect to the IPC of the leading thread executing on an OoO core. Such an IPC analysis gives us an estimate of the clock speed at which the trailing core can execute. Since the trailing thread receives load values and branch outcomes from the leading thread, it never experiences cache misses or branch mispredictions. The

TABLE 3
Benchmark Pairs for the Multithreaded Workload

Benchmark Set	Set #	IPC Pairing	Benchmark Set	Set #	IPC Pairing
bzip-vortex	1	Int/Int/Low/Low	vpr-gzip	2	Int/Int/High/Low
eon-vpr	3	Int/Int/High/High	swim-lucas	4	FP/FP/Low/Low
art-applu	5	FP/FP/Low/High	mesa-equake	6	FP/FP/High/High
gzip-mgrid	7	Int/FP/Low/Low	bzip-fma3d	8	Int/FP/Low/High
eon-art	9	Int/FP/High/Low	twolf-equake	10	Int/FP/High/High

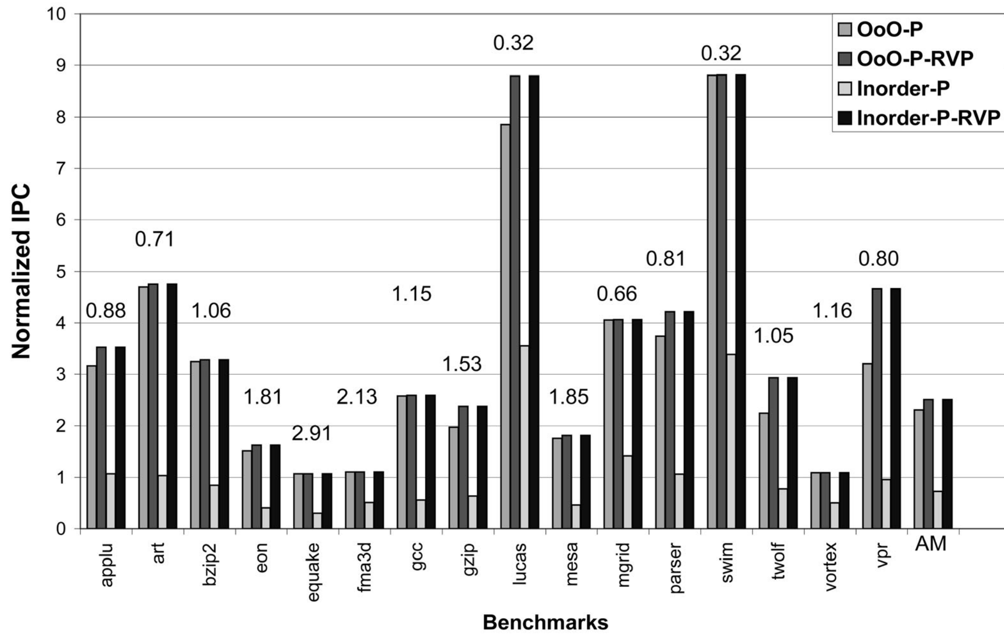

Fig. 3. IPCs for different models, relative to the baseline OoO core. The absolute IPC for the baseline OoO core for each program is listed above each set of bars.

first bar for each benchmark in Fig. 3 shows the normalized IPC for such a trailing thread that executes on an OoO core with a perfect cache and a branch predictor. If the trailing thread is further augmented with RVP (the second bar), then the IPC improvement is only minor (for most benchmarks). The third bar shows normalized IPCs for an in-order core with a perfect cache and a branch predictor. It can be seen that, for many programs, the normalized IPC is less than 1. This means that an in-order trailing core cannot match the leading thread's throughput unless it operates at a clock speed much higher than that of the OoO core. The last bar augments the in-order core's perfect cache and branch predictor with RVP. The increase in IPC is significant, indicating that RVP will be an important feature within an in-order core to allow it to execute at low frequencies. The IPC of a core with a perfect cache, a branch predictor, and a perfect RVP is limited only by functional unit availability and fetch bandwidth. The average normalized IPC is around 3, meaning that, on the average, the frequency of a trailing thread can be scaled down by a factor of 3.

We then evaluate the DFS heuristic on the single-thread programs with different forms of trailing cores. The heuristic is tuned to conservatively select high frequencies so that the leading thread is rarely stalled because of a full RVQ. The performance loss, relative to a baseline core with no redundancy, is therefore negligible. The second bar in Fig. 4 shows power consumed by a trailing OoO core (with a perfect cache and a branch predictor) that has been dynamically frequency scaled. Such a trailing redundant core imposes a power overhead that equals 53 percent of the power consumed by the leading core. Without the DFS heuristic, the trailing core would have imposed a power overhead of 90 percent (first bar in Fig. 4). On the average, the trailing core operates at a frequency that is 0.44 times the frequency of the leading thread. Note that DFS does not impact leakage power dissipation. The net outcome of the above analysis is that low trailer frequencies selected by the DFS heuristic can reduce the trailer core's power by 42 percent and the overall processor power (leading and trailing cores combined) by 22 percent.

Fig. 4 also shows the power effect of employing an in-order trailing core. Future CMPs will likely be heterogeneous, providing a good mix of OoO and in-order cores [10]. As seen previously, a perfect cache and branch predictor is not enough to allow the in-order core's IPC to match the leading core's IPC. Hence, the system has been augmented with RVP. We have pessimistically assumed that the buffers between the two cores now carry two additional 64-bit values per instruction, leading to an additional average power dissipation of 2.54 W. With the in-order to OoO power ratio of 1:2, we observe that the redundancy mechanism now consumes less than 26 percent of the power consumed by the leading thread. The frequency selected by the DFS heuristic for the in-order core is on the average 0.42 times that of the leading core's frequency. For the in-order core to OoO power ratio of 1:7, the power consumed by the trailing thread is 8.5 percent of the leading thread.

For all the above simulations, we assume an interval length of 1,000 cycles when making frequency decisions. Frequency changes were made for 70 percent of all intervals. If we assume that a frequency change stalls the processor for 10 (peak frequency) cycles, then the total overhead is only 0.7 percent. Our frequency change overhead is very conservative when compared to the recent implementation of DFS in Intel's Montecito core, where a frequency change is effected in a single cycle. The frequency change overhead can also be reduced by incorporating hysteresis within the algorithm, but this occasionally leads to increased slack and stalls for the leading thread. Given the low 0.7 percent performance overhead, we chose to not

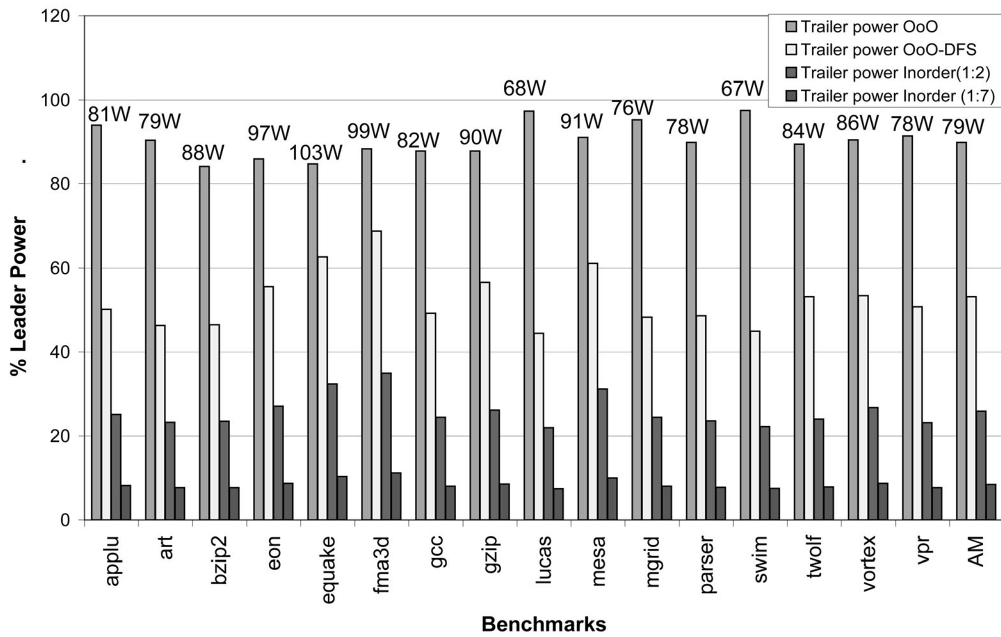

Fig. 4. Power consumed by the trailing core as a function of the power consumed by the leading core. The number above each set of bars represents the absolute value of power dissipated by the leading core for each benchmark.

include hysteresis and instead react quickly to variations in the slack.

Based on the above results, we make the following general conclusions: Executing the trailing thread on an OoO core has significant power overheads, even if the trailing core's frequency is scaled (partially because DFS does not impact leakage). An in-order core has much lower power overheads but poor IPC characteristics, requiring that it operate at a clock speed higher than the leading core. The IPC of the in-order core can be boosted by employing RVP. This requires us to invest about 2.54 W more power in transmitting additional data between cores (a pessimistic estimate), but this allows us to operate the in-order core at a frequency that is less than half the leading core's peak frequency. Hence, this is a worthwhile trade-off, assuming that the dynamic power consumed by the in-order core is at least 5 W.

5.3 Multithread Workloads

Next, we examine the most efficient way to execute a multithreaded workload. As a baseline, we employ the CRTR model proposed by Gomaa et al. [7], where each OoO core executes a leading thread and an unrelated trailing thread in SMT fashion. Within the power-efficient P-CRTR-OoO, both leading threads execute on a single SMT OoO core, and both trailing threads execute on a neighboring SMT OoO core that can be frequency scaled. The last bar in Fig. 5 shows the total leading thread throughput for CRTR for each set of program pairs defined in Table 3. The first bar shows the total leading thread throughput in a baseline system where both leading threads execute on a single SMT OoO core (no redundant threads are executed). It can be seen that the throughput of CRTR is about 9.7 percent better than a system where two leading threads execute on the same OoO core. This is because each leading thread in CRTR is coscheduled with a trailing thread that does not execute wrong-path instructions and poses fewer conflicts for resources (ALUs, branch predictor, data cache, and so forth). The second bar in Fig. 5 shows IPCs for P-CRTR-OoO. The DFS heuristic selects frequencies such that the leading core is rarely stalled and throughputs are very similar to that of the baseline system, about 9.4 percent lower than CRTR on the average. The results of the earlier section indicate that an in-order core augmented with RVP is likely to entail a lower power overhead. Hence, we also evaluate a system (P-CRTR-in order) where two leading threads execute on an SMT OoO core and the two trailing threads execute (by themselves) on two in-order cores with RVP and DFS. Again, the throughput of P-CRTR-in order is similar to that of the baseline system, about 12 percent lower than CRTR on the average. Note, however, that P-CRTR-in-order is likely to have a lower area overhead than the other organizations in Fig. 5 because the area occupied by two single-threaded in-order cores is less than the area occupied by one SMT OoO core [10]. The performance penalty for P-CRTR can be primarily

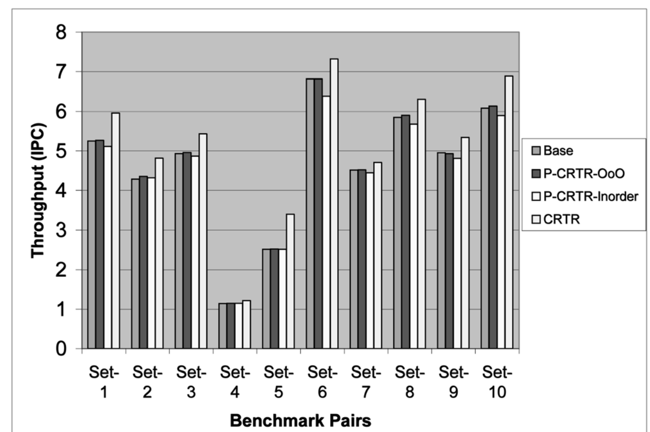

Fig. 5. Total IPC throughput for multithreaded workloads.

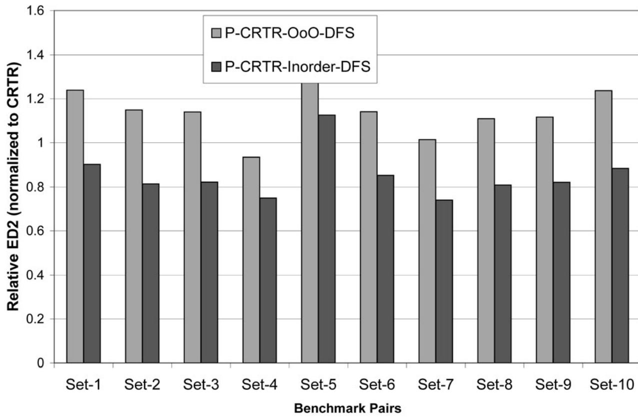

Fig. 6. ED^2 for the entire system for various forms of trailers, normalized to the ED^2 of CRTR.

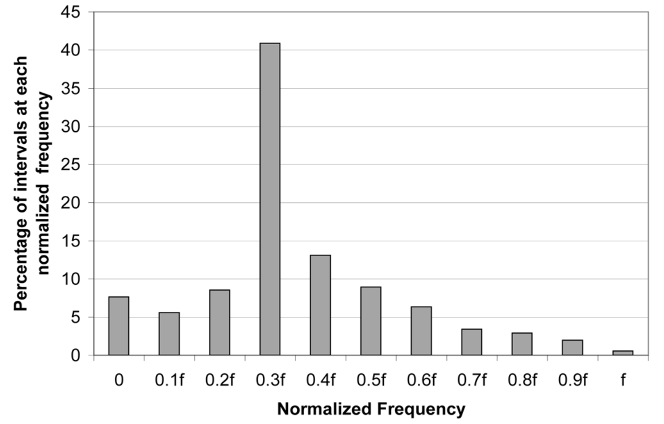

Fig. 7. Histogram showing the percentage of intervals at each normalized frequency.

attributed to higher ALU, cache, and branch predictor contention. For example, the average L1 cache miss rate for the leading threads in P-CRTR was 6.5 percent higher than that in CRTR.

Fig. 6 shows the $Energy \times Delay^2$ (ED^2) metric for different forms of P-CRTR, normalized to that for CRTR. Figs. 5 and 6 provide the data necessary to allow readers to compute other metrics in the $E - D$ space. The first bar shows ED^2 for P-CRTR-OoO, where both trailers execute on an SMT OoO core. The DFS heuristic scales frequencies for the trailing core, allowing it to consume less total power than CRTR. However, CRTR has a throughput advantage that allows it to have a better (lower) ED^2 than P-CRTR for many programs. On the average, the ED^2 of P-CRTR is 17 percent more than CRTR. Although CRTR imposes an average power overhead of 97 percent for redundancy, P-CRTR imposes an overhead of 75 percent. The effective frequency for the trailing core is much higher (0.77 times peak frequency) than that seen for the single-thread workloads because the selected frequency has to be high enough to allow both threads to match leading thread throughputs. Some workloads (sets 4 and 7) are able to lower their trailer frequencies enough to yield lower ED^2 than CRTR. This was also observed for other workloads that had a similar combination of Int/FP/IPC. In general, workloads composed of low IPC programs (those with high branch mispredict rates and cache miss rates) are likely to see a higher benefit from a perfect cache and a branch predictor, leading to low trailer frequencies and better overall ED^2 with P-CRTR. When scheduling redundant threads on a CMP of SMT OoO cores, the operating system can optimize ED^2 by taking program behavior into account and accordingly adopting a schedule similar to CRTR or P-CRTR.

The second bar in Fig. 6 represents the P-CRTR-in-order model. We assume that the in-order to OoO power consumption ratio is 1:7 for this graph. By executing the trailing thread on a frequency-scaled in-order core with perfect cache, branch predictor, and RVP, significant power reductions are observed (enough to offset the additional power overhead of data transfers between cores). On the average, P-CRTR-in-order improves ED^2 by 15 percent, relative to CRTR. The average power overhead of redundancy is 20 percent. Benchmark set 5 has 12 percent higher ED^2 when compared to CRTR due to a 25 percent performance loss. We observed that, by coscheduling the leading threads on the same core, branch predictor conflicts increased significantly for this benchmark pair. The total power overhead associated with RVP for the multithreaded workloads is 5.64 W.

The net conclusion is similar to that in the previous section. Leading threads executing on an OoO core (in single or multithreaded mode) can be verified efficiently on in-order cores. Although more data has to be transmitted between cores, the power efficiency of an in-order core compensates for the data transfer overhead.

5.4 Potential for Voltage Scaling

Frequency scaling is a low-overhead technique that trades off performance and power and allows us to reduce the dynamic power consumed by the trailing thread. One of the most effective techniques to reduce power for a minor performance penalty is dynamic voltage and frequency scaling (DVFS). If our heuristic determines that the trailer can operate at a frequency that is half the peak frequency (for example), then it may be possible to reduce the voltage by (say) 25 percent and observe dynamic power reduction within the trailer of 72 percent instead of the 50 percent possible with just DFS. Although DFS does not impact leakage power, DVFS can also reduce leakage (to a first-order) as leakage is linearly proportional to supply voltage [5]. DVFS can be combined with body biasing to further reduce leakage power [14]. However, these techniques require voltage changes that can consume a large number of cycles of the order of $50\mu s$ [6]. Even if voltage is modified only in small steps, each voltage change will require tens of thousands of cycles. If an increase in frequency is warranted, then the frequency increase cannot happen until the voltage is increased, thereby causing stalls for the leading threads. As observed earlier, a frequency change is made at the end of the 70 percent of all 1,000-cycle intervals. It is difficult to design a DFS mechanism that increases frequency only once every 100,000 cycles on the average and poses minimal stalls for the leading thread. Therefore, it is unlikely that the overhead of dynamic voltage scaling will be tolerable.

We, however, observed that there may be the potential to effect some degree of conservative voltage scaling. Fig. 7 shows a histogram of the percentage of intervals spent at each frequency by the in-order trailer with RVP. Peak frequency is exercised for only 0.56 percent of all intervals.

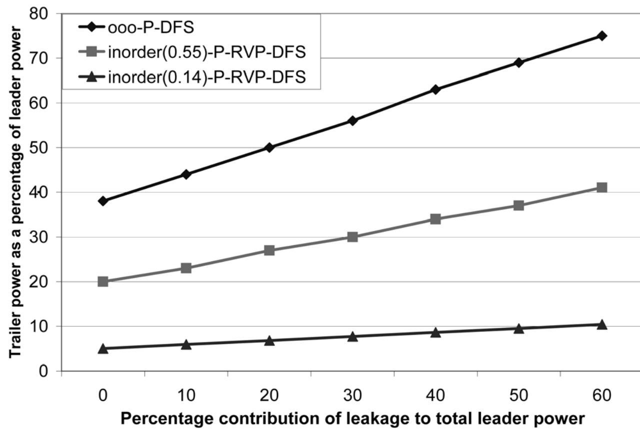

Fig. 8. Trailer power as a function of contribution of leakage to the baseline processor.

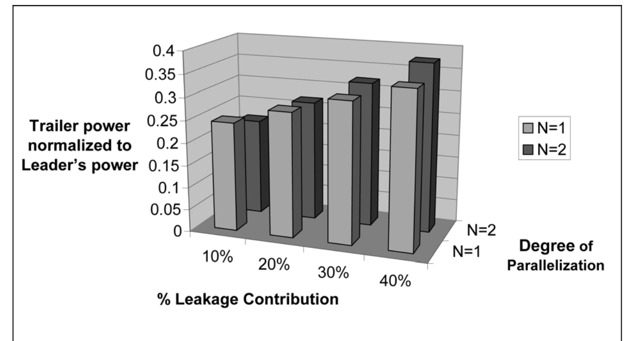

Fig. 9. Power overhead of the trailing core, relative to the leading core, with and without parallelizing the verification workload.

If we operate at a low voltage that can support a frequency of 0.9 × peak frequency but not the peak frequency, then we will be forced to increase the voltage (and stall the leader for at least 10,000 cycles) for a maximum of 0.56 percent of all 1,000-cycle intervals. This amounts to a performance overhead of up to 5.6 percent, which may be tolerable. The corresponding power benefit may be marginal, especially considering the small amount of power consumed within each in-order core, as illustrated below.

Consider the following example scenario when the trailer is executed on a frequency-scaled in-order core augmented with RVP. If 100 units of power are consumed within the leader, an in-order trailer will consume 14 units of power (ratio similar to the EV5 and EV8), of which about 10 units can be attributed to dynamic power. A DFS heuristic with an effective frequency of 0.5 will reduce the in-order core's dynamic power by five units. Thus, out of the total 109 units consumed by this system, four units can be attributed to in-order leakage and five units to in-order dynamic power. Any additional optimizations to this system must take note of the fact that the margin for improvement is very small.

Based on the simple analysis above, we also examine the potential benefits of parallelizing the verification workload [19]. With RVP, the trailing thread has a very high degree of ILP as every instruction is independent. Instead of executing a single trailing thread on an in-order core, the trailing thread can be decomposed into (say) two threads and made to execute in parallel on two in-order cores. When the workload is parallelized by a factor of 2, the effective frequency can be lowered by a factor of 2. Hence, the power consumed by this system will equal 113 units (100 for the leading core + 8 for leakage on two in-order cores + 2.5 for dynamic on first in-order core + 2.5 for dynamic on second in-order core). Thus, parallelization with DFS does not reduce dynamic power, but increases leakage power and is therefore not worthwhile. For parallelization to be effective, DFS has to be combined with a technique such as DVFS or body biasing. Assume that an effective frequency of 0.5 can be combined with a voltage reduction of 25 percent (similar to that in the Xscale). Parallelization on two in-order cores yields a total power consumption of 108.8 units (100 for leading core + 6 for leakage, which is a linear function of supply voltage, + 1.4 for dynamic on first in-order core, which is a quadratic function of supply voltage, + 1.4 for dynamic on second in-order core). The reduction in dynamic power is almost entirely negated by the increase in leakage power. Clearly, different assumptions on voltage and frequency scaling factors, leakage, in-order power consumption, and so forth can yield different quantitative numbers. In the next section, we use our analytical model to show that, for most reasonable parameters, workload parallelization yields little power benefit, even when we aggressively assume that voltage scaling has no overhead.

There are other problems associated with voltage scaling: 1) Lower voltages can increase a processor's susceptibility to faults, 2) as voltage levels and the gap between the supply and threshold voltages reduce, opportunities for voltage scaling may cease to exist, and 3) parallelization has low scalability in voltage terms: Parallelizing the workload across four cores allows frequency to be scaled down by a factor of 4, but reductions in voltage become increasingly marginal.

5.5 Sensitivity Analysis

As an example application of the analytical model, we present power overheads as a function of the contribution of leakage to the baseline OoO leading core (Fig. 8). The three forms of trailers shown are an OoO core with the DFS heuristic and in-order cores with RVP and DFS, which consume 0.55 times and 0.14 times the power of the OoO core, respectively. The overall conclusions of our study hold for all of these design points.

We discussed workload parallelization in an earlier section and reconfirm our observations with the help of analytical models for various design points. Fig. 9 shows the effect of workload parallelization and leakage contribution on the trailing core's power, where the trailing thread executes on an in-order processor enabled with DVFS. Based on detailed simulation results, we assume $wrongpath_factor = 1.17$, $eff_freq = 0.44$, $v_factor = 1.25$, and lkg_ratio and $dyn_ratio = 1.8$. For $\mathrm{N} = 2$, as the contribution of leakage power increases, the workload parallelization yields marginal improvement over the base case $(\mathrm{N} = 1)$. Note that the base case has a single DFS-enabled in-order core and does not employ voltage scaling. Even for low leakage contribution, the power reduction with the workload parallelization is only 2.5 percent.

The effect of various parameters on IPC is harder to capture with analytical models and we report on some of our salient observations:

1. The relative benefits of a perfect cache and a branch predictor are significant for most processor models. For example, increasing the window size improves the ability of the baseline OoO core to tolerate cache misses and likewise improves the ability of the core with the perfect cache/branch predictor to mine greater ILP.
2. We have observed that, for a multithreaded workload, scheduling leading and trailing threads on the same SMT core (as in CRTR) yields a 9.4 percent throughput improvement over a model where the leading threads are scheduled on the same SMT core. As the total number of functional units is increased, this performance gap reduces to 6.3 percent because contention for resources becomes less of an issue.
3. For an in-order core with RVP, the only limitation to IPC is the number of functional units available and the fetch bandwidth. The effective frequency of the in-order core can be reduced by increasing the number of functional units and, hence, the IPC. Likewise, the ability to fetch from multiple basic blocks in the same cycle has a much greater impact on the IPC of the in-order core with RVP than on other cores.
4. The slack between leading and trailing threads is closely related with interval size. If we examine the slack every 1,000 cycles to make a decision for trailer frequency, then the slack must be about 1,000 in order to absorb a significant IPC change of 1.0 in the leading thread in the next interval. A smaller slack can lead to the intercore buffers getting full and stalling the leading thread. A large slack allows more opportunities for frequency reduction, but incurs a nontrivial power overhead for the intercore buffers.

6 Conclusions and Future Work

In this paper, we have presented novel microarchitectural techniques for reducing the power overheads of RMT. When executing leading and trailing redundant threads, we take advantage of the fact that the leading thread prefetches data and resolves branches for the trailing thread. The results of the leading thread also allow the trailing core to implement a perfect register value predictor. All of the information from the leading thread makes it possible for the trailing thread to achieve high IPC rates even with an in-order core, thereby justifying the cost of high intercore traffic. DFS further helps reduce the power consumption of the trailing thread. Our results indicate that workload parallelization and voltage scaling hold little promise. We quantify the power performance trade-off when scheduling the redundant threads of a multithreaded workload and derive analytical models to capture the insight from our detailed simulations. None of the mechanisms proposed in this paper compromises the error coverage of the baseline system.

As our future work, we will study the thermal characteristics of RMT implementations. As the degree of redundancy increases, the contribution of redundant threads to total system power also increases. In such a setting, it may be worth studying how the power consumed by in-order cores can be further reduced.

Acknowledgments

This work was supported in part by US National Science Foundation (NSF) grant CCF-0430063 and by an NSF CAREER award.

References

[1] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO-32),* Nov. 1999.
[2] R. Balasubramonian, N. Muralimanohar, K. Ramani, and V. Venkatachalapathy, "Microarchitectural Wire Management for Performance and Power in Partitioned Architectures," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA-11),* Feb. 2005.
[3] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture (ISCA-27),* pp. 83-94, June 2000.
[4] D. Burger and T. Austin, "The Simplescalar Toolset, Version 2.0," Technical Report TR-97-1342, Univ. of Wisconsin-Madison, June 1997.
[5] J.A. Butts and G. Sohi, "A Static Power Model for Architects," *Proc. 33rd Ann. Int'l Symp. Microarchitecture (MICRO-33),* Dec. 2000.
[6] L. Clark, "Circuit Design of XScale Microprocessors," *Proc. Symp. VLSI Circuits (Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits),* June 2001.
[7] M. Gomaa, C. Scarbrough, and T. Vijaykumar, "Transient-Fault Recovery for Chip Multiprocessors," *Proc. 30th Int'l Symp. Computer Architecture (ISCA-30),* June 2003.
[8] M. Gomaa and T.N. Vijaykumar, "Opportunistic Transient Fault Detection," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA-32),* June 2005.
[9] E. Grochowski, R. Ronen, J. Shen, and H. Wang, "Best of Both Latency and Throughput," *Proc. 22nd Int'l Conf. Computer Design (ICCD-22),* Oct. 2004.
[10] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. 36th Ann. Int'l Symp. Microarchitecture (MICRO-36),* Dec. 2003.
[11] R. Kumar, V. Zyuban, and D. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA-32),* June 2005.
[12] S. Kumar and A. Aggarwal, "Reduced Resource Redundancy for Concurrent Error Detection Techniques in High Performance Microprocessors," *Proc. 12th Int'l Symp. High-Performance Computer Architecture (HPCA-12),* Feb. 2006.
[13] S. Kumar and A. Aggarwal, "Self-Checking Instructions—Reducing Instruction Redundancy for Concurrent Error Detection," *Proc. 15th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT '06),* Sept. 2006.
[14] S. Martin, K. Flautner, T. Mudge, and D. Blaauw, "Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors under Dynamic Workloads," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '02),* 2002.
[15] C. McNairy and R. Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," *IEEE Micro,* vol. 25, no. 2, Mar./Apr. 2005.
[16] S. Mukherjee, J. Emer, and S. Reinhardt, "The Soft-Error Problem: An Architectural Perspective," *Proc. 11th Int'l Symp. High Performance Computer Architecture (HPCA-11),* 2005.
[17] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed Design and Implementation of Redundant Multithreading Alternatives," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* May 2002.
[18] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. 36th Ann. Int'l Symp. Microarchitecture (MICRO-36),* Dec. 2003.

[19] M. Rashid, E. Tan, M. Huang, and D. Albonesi, "Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance," *Proc. 14th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT '05),* 2005.

[20] J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," *Proc. 34th Ann. Int'l Symp. Microarchitecture (MICRO-34),* Dec. 2001.

[21] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. 27th Int'l Symp. Computer Architecture (ISCA-27),* pp. 25-36, June 2000.

[22] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. 29th Int'l Symp. Fault-Tolerant Computing (FTCS '99),* June 1999.

[23] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, "Energy Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA-8),* pp. 29-40, Feb. 2002.

[24] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* June 2002.

[25] J. Smolens, J. Kim, J. Hoe, and B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures," *Proc. 37th Ann. Int'l Symp. Microarchitecture (MICRO-37),* Dec. 2004.

[26] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA-23),* May 1996.

[27] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery via Simultaneous Multithreading," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* May 2002.

[28] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '04),* June 2004.

Niti Madan received the BE degree in electrical engineering from the University of Delhi in 2001 and the MS degree in computer science from the University of Utah in 2004. She is pursuing the PhD in computer science at the University of Utah. Her research focuses on reliability-aware and power-efficient architectures and multicore and multithreaded architectures. She is a student member of the IEEE and the ACM.

Rajeev Balasubramonian received the BTech degree in computer science and engineering at the Indian Institute of Technology, Bombay, in 1994 and the MS and PhD degrees in computer science from the University of Rochester in 2000 and 2003, respectively. He is an assistant professor in the School of Computing at the University of Utah. His research focuses on the design of high-performance microprocessors that can efficiently tolerate long on-chip wire delays, high-power densities, and frequent soft errors. He received a US National Science Foundation Faculty Early Career Development (CAREER) award in 2006 and holds three patents. He is a member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.