Power-Efficient Approaches to Reliability

Niti Madan, Rajeev Balasubramonian

UUCS-05-010

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

December 2, 2005

Abstract

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 towards smaller transistor sizes and lower supply voltages. Fault detection and recovery can be achieved through redundancy. State-of-the-art implementations execute two copies of the same program as two threads, either on the same or on separate processor cores, and periodically check results. While 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 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. These strategies include dynamic frequency scaling, in-order execution, and parallelization of the trailing thread.

1 Introduction

Radiation-induced soft errors [19, 35] in computer systems are on the rise. High energy particles such as neutrons from cosmic rays and alpha particles in packaging material strike semiconductor devices and generate electron-hole pairs. This causes enough charge to be deposited that the state of a transistor can be switched, resulting in propagation of incorrect results for a brief period of time. The frequency of occurrence of soft errors is once every 1000 to 10,000 hours, which isn't a huge concern for modern microprocessors. However, these soft errors are expected to increase significantly as transistor density increases in future generations of microprocessors. Also, lower supply voltages cause reduced noise margins and increase susceptibility to noise-induced switching. A very small amount of charge is enough to exceed voltage threshold levels and switch a transistor's state. A recent study [28] shows that the soft-error rate per chip is projected to increase by nine orders of magnitude from 1992 to 2011. Even personal computing systems will eventually need fault-tolerant designs and will likely support recovery mechanisms.

Reliability can be provided at the circuit or process level. For comprehensive fault coverage, every circuit would have to be re-designed. 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 [2, 7, 18, 20, 23, 24, 25, 26, 29, 33, 34] have explored architecture-level solutions that can provide fault tolerance with modest performance and complexity overheads. In most solutions, an instruction is executed twice and results are compared to detect faults. To keep up with the projected soft-error rate for future microprocessors, we may have to further increase levels of redundancy. Already, power consumption is a major concern in modern microprocessors that dissipate more than 100W when executing common single-threaded desktop applications. 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. This paper represents one of the few efforts at improving the power-efficiency of reliability mechanisms.

In a processor that employs redundancy, the *"checker instruction"* can be made to flow through a similar pipeline as the *"primary instruction"* or through a specialized different pipeline. The former approach is well suited to a chip multi-processor (CMP) [22] or simultaneous multi-threaded processor (SMT) [32], where the processor is already designed to accommodate multiple threads. With minor design modifications, one of the threads can serve as the *checker thread* [7, 20, 25, 33]. In examples of the latter approach [2, 29], the checker thread flows through a heavily modified helper pipeline that has low complexity.

A case can be made for either of the two approaches above. By leveraging a CMP or SMT, the hardware overhead of redundancy is minimal. Dual-core and dual-threaded processors are already being commercially produced [8, 13, 30] and most major chip manufacturers have announced plans for more aggressive multi-threaded designs [1, 9, 10, 11, 12, 16, 17]. It is only natural that these designs be extended to provide reliability. Further, thread contexts can be dynamically employed for either checker or primary threads, allowing the operating system or application designer to choose between increased reliability or increased multi-threaded performance. The disadvantage of this approach is that every checker instruction now flows through a complex out-of-order pipeline, effectively incurring nearly a 100% increase in power consumption. The use of a modified helper pipeline for checking [2, 29] incurs a lower power overhead in the second approach above. 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. We believe that heterogeneous CMPs will eventually allow us to derive the best of the two approaches above.

As a starting point, consider the following redundant multi-threading architecture based on the Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) model proposed by Gomaa *et al.* [7]. The primary (leading) thread executes on an out-of-order core and the checker (trailing) thread executes on a different out-of-order core. 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 Figure 1). Each checker instruction flows through a complex out-of-order pipeline, resulting in enormous power overheads. In an effort to reduce this power overhead, 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. Since the primary

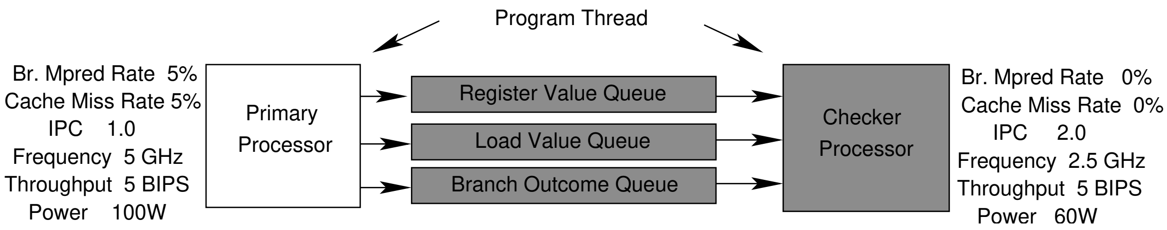

Figure 1: **An example of the effect of scaling the frequency of the checker core.**

and checker threads must have equal throughputs, we allow the checker thread to execute at a high IPC and scale its frequency down. A simple dynamic adaptation mechanism is very effective at selecting the appropriate frequency for the checker thread, typically resulting in significant power savings for a workload where one primary and one checker thread execute together on a 2-way CMP. We 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 multi-threaded workloads executing on a CMP of SMTs. Finally, we examine the potential of dynamic voltage scaling and of parallelization of the verification workload. The paper makes two major novel contributions: (i) exploiting information generated by the leading thread to improve trailing thread IPC, thereby enabling frequency and power reductions, (ii) a comprehensive analysis of the design space to identify promising approaches to power-efficient reliability.

The paper has been organized as follows. Section 2 describes the basic redundant multi-threading implementations that serve as baseline processor models in this study. Section 3 describes various power reduction strategies for the trailing thread. The proposed ideas are evaluated in Section 4 and we contrast our approach with related work in Section 5. Section 6 summarizes the conclusions of this study.

2 Baseline Reliable Processor Model

We have based our reliable chip-multiprocessor architecture on the model proposed in [7, 20]. The architecture consists of two communicating cores that execute copies of the same application for fault-detection. One of the cores (a.k.a leading core) executes ahead of the second core (a.k.a 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 (Figure 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 L1 data cache and there is no need for coherence operations between the L1 data caches of the leading and trailing cores. The communication of data between the cores is facilitated by the first-in-first-out 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, allowing it to have perfect branch prediction. The power saving in the trailing core by not accessing the L1D cache and 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 re-order buffer (ROB) size of the trailing core, it is guaranteed that a load instruction in the trailing core will always find its load value in the LVQ. This implementation uses *asymmetric commit* to hide inter-core communication latency – the leading core is allowed to commit instructions before checking. The leading core commits stores to a store buffer (StB) instead of 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. When external interrupts or exceptions are raised, the leading thread must wait for the trailing thread to catch up before servicing the interrupt.

The inter-core paths that communicate load values and register results require a high-bandwidth interconnect and can dissipate a non-trivial amount of power. Some of this overhead can be mitigated by employing narrow wires and a low-power repeater configuration, or by employing heuristics to identify a relevant subset of results that need to be checked [7]. For this initial study, we attempt no optimizations to inter-core communication so we can better understand the effect of our techniques on the basic redundancy model. As we will later show, it may be beneficial to send all register results rather than a subset of results.

In our single-thread model, we assume an implementation where each core on the chip multi-processor can only support a single thread. Our multi-thread model is based on the CRTR architecture proposed in [7], where each core is a dual-threaded SMT. In the CRTR architecture, the trailing thread of one application shares its core with the leading thread of a different application. 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. Similar 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 $TH1$, the trailing thread is not allowed to fetch and if the slack is greater than $TH2$, the leading thread is not allowed to fetch. In cases where both threads within an SMT core are allowed to fetch, the ICOUNT heuristic [31] is employed.

3 Managing Power Overheads

3.1 Power Reduction Strategies for the Trailing Core

In our proposed implementation, we assume that each core within the CMP is an SMT core. For a single-thread workload, we propose that the leading and trailing thread execute on neighboring cores. Since each core is an SMT, it is possible to execute the leading and trailing threads on a single core – this will avoid the overhead of inter-core 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 overhead of inter-core communication. For a multi-threaded workload, the CRTR implementation executes unrelated leading and trailing threads on a single SMT core. 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. However, 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 Power-efficient CRTR (P-CRTR). By changing the assignment of threads to cores, we are not increasing inter-core bandwidth requirements – the same amount of data as in CRTR is being communicated between two neighboring cores.

In a CMP fault-detection and recovery scheme, if the leading thread communicates its branch outcomes and load values to the trailing thread, then the trailing thread is capable of high throughputs. Thus, the slack between the two threads can eventually reduce. A reduced slack yields no benefit for overall performance or reliability. We exploit this property to reduce power overheads. The goal is to throttle the execution of the trailing thread so that a constant slack is maintained, and save power in the process. The CRTR implementation, on the other hand, attempted to increase multi-threaded performance by employing SMT cores within the CMP and executing the trailing thread with the leading thread of a different program and offering most resources to leading threads.

DFS. The first approach we consider to throttle trailing thread execution is Dynamic Frequency Scaling (DFS), a well-established technique that allows dynamic power to scale down linearly with clock frequency [6]. It is a low overhead technique with a frequency change only consuming a handful of (peak frequency) cycles. DFS does not impact leakage energy dissipated by the trailer.

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 out-of-order core. Hence, some enhancements need to be made to the in-order core. We propose using register value prediction (RVP). Along with the result of an instruction, the leading thread can also pass the input operands for that instruction to the trailing thread[1]. 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% accuracy, unless an earlier instruction's result has been influenced by a soft error.

[1]Optimizations to this basic mechanism will be subsequently discussed.

If an earlier instruction has been affected by a soft error, the trailing thread will detect and recover from it and the subsequent execution will be corrected. With a 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. It may be possible to even apply DFS to the in-order core to further reduce dynamic power. While previous studies [7] have attempted to reduce inter-core traffic, we believe it may be worthwhile to send additional information between cores because of the optimizations it enables at the trailer.

Workload Parallelization. Assuming that power is a quadratic function of performance 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 [23]. At regular intervals, the leading thread spawns a new trailing thread that verifies the results for a recently executed contiguous chunk of the program. If the trailing core employs RVP, it is not even necessary to copy initial state to the core before starting a verification thread.

3.2 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 thread. In essence, if the IPC of the leading thread is denoted by IPC_L, and the IPC of the trailing thread is IPC_T, 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, it means 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, it means 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, while the trailing thread will have to commit N more instructions before it enters that phase

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 $f_t -= 0.1f_{peak}$	ICOUNT : t_2 $f_t -= 0.1f_{peak}$	ICOUNT : t_2 $f_t += 0.1f_{peak}$	$l_1 : t_2$ $f_t = f_{peak}$
$TH0 < s_1 \leq TH1$	ICOUNT : t_1 $f_t -= 0.1f_{peak}$	ICOUNT : ICOUNT $f_t -= 0.1f_{peak}$	ICOUNT : ICOUNT $f_t = f_{peak}$	$l_1 : t_2$ $f_t = f_{peak}$
$TH1 < s_1 \leq TH2$	ICOUNT : t_1 $f_t += 0.1f_{peak}$	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 $f_t = f_{peak}$

Table 1: Fetch and frequency policies adopted for leading and trailing cores in P-CRTR. 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 1000 cycles.

itself. Once all 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, the faster the mechanism reacts to throughput variations. For our simulations, we assume a 10 cycle overhead for every dynamic frequency change. The time interval T is selected to be 1000 cycles so that the overhead of frequency scaling is marginal. To absorb throughput variations in the middle of an interval, a slack of 1000 is required. To ensure that the RVQ is half-full on average, we set $N - thresh$ to be 400 and $N + thresh$ to be 700. Frequency is reduced in steps that equal $f_L \times 0.1$.

3.3 Dynamic Frequency Scaling Algorithm for Multi-Threaded Workloads

If each trailer executes on a separate core, the single-thread DFS algorithm from the previous sub-section can be applied to tune the frequency of each trailing core. If two trailing threads execute on a single SMT core, 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 co-scheduled threads. Rather than always use 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 fine-grained control of each thread, we employ three slack thresholds. The action for each case is based on the following guidelines: (i) if slack for a trailer is less than $TH0$, there is no point fetching instructions for that trailer, (ii) if slack for a trailer is between $TH0$ and $TH1$ (the desireable range), the decision depends on the state of the other thread, (iii) if slack is between $TH1$ and $TH2$, we may need to quickly ramp up the frequency to its peak value in an effort to keep slack under control, (iv) if slack is greater than $TH2$, 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 leader's ROB size, and $TH1$ is picked so that the RVQ will be half-full, on average. The leading core always employs the ICOUNT fetch heuristic to select among 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 1000 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.

3.4 Analytical Power Estimates

We derive simple intuitive analytical models for the power consumed by each of the studied processor models. These models enable the interested reader to derive rough estimates of potential savings when using a different set of parameters. Our detailed simulation-based IPC and power results in Section 4 closely match the estimates from these simple analytical models.

For starters, consider leading and trailing threads executing on neighboring out-of-order 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 system is given by the following equation.

$$Baseline\_power = leakage_{leading} + dynamic_{leading} + leakage_{leading} +$$

$$dynamic_{leading}/wrongpath\_factor$$

When DFS is applied to the trailing core and eff_freq is its average operating frequency, assuming marginal stalls for the leading thread, the total power in the system is given by the following equation.

$$DFS\_ooo\_power = leakage_{leading} \ + \ dynamic_{leading} \ +$$

$$leakage_{leading} \ + \ dynamic_{leading} \times eff\_freq/wrongpath\_factor$$

If an in-order core with RVP is employed for the trailing thread, 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 out-of-order leading core.

$$DFS\_inorder\_power = leakage_{leading} \ + \ dynamic_{leading} \ + \ leakage_{leading}/lkg\_ratio +$$

$$dynamic_{leading} \times eff\_freq/(wrongpath\_factor \times dyn\_ratio) \ + \ RVP\_overhead$$

Finally, similar models can be constructed for CRTR and P-CRTR multi-threaded models. P_CRTR_ooo represents a model where both trailing threads execute on an SMT out-of-order 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.

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

$$Energy_{P\_CRTR\_ooo} = slowdown \times (2 \times leakage_{ooo} + dynamic_{ooo} \times (wrongpath\_factor +$$

$$wrongpath\_factor) \ + \ dynamic_{ooo} \times eff\_freq \times (1 + 1))$$

$$Energy_{P\_CRTR\_inorder} = slowdown \times (leakage_{ooo} \times (1 + 2/lkg\_ratio) +$$

$$dynamic_{ooo} \times (wrongpath\_factor + wrongpath\_factor) +$$

$$dynamic_{ooo} \times effective\_frequency \times (1 + 1)/dyn\_ratio) \ + \ 2 \times RVP\_overhead$$

Parameters such as $slowdown$, eff_freq, and $wrongpath\_factor$ have to be calculated through detailed simulations and are reported in Section 4.

3.5 Implementation Complexities

This sub-section provides a qualitative analysis of the complexity introduced by the mechanisms in this paper. Firstly, 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 frequently employ such buffers between different clock domains [27]. 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 threshold values.

In order to absorb throughput variations within an interval, we have employed a slack of 1000 instructions. A large slack value requires that we implement a large RVQ, LVQ, and StB. The RVQ and LVQ structures are RAMs and their access is off the critical path. They can be banked to reduce their power overheads. Every time the leading thread issues a load, an associative look-up of the StB is required in parallel with L1D cache access. The StB will typically contain of the order of 100 entries (for a slack of 1000) and its complexity should be comparable with that of the D-TLB. A set-associative StB can help lower its power overheads and still retain correctness (assuming store verifications are done in order).

An in-order core is likely to yield significant energy savings, but a non-trivial amount of power/complexity will have to be expended in implementing RVP. As an initial implementation, assume that the leader provides the input operands for every instruction along with the computed result in the RVQ. 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 multiplexor before the ALU selects these values instead of the values read from the register file (register file reads can even be disabled). If we wish to avoid increasing the inter-core bandwidth by a factor of three, the trailing core will have to maintain a speculative register file. As soon as an instruction enters the trailer's pipeline, its result is read from the RVQ and placed in the speculative register file. Thus, subsequent instructions always find their input operands in the speculative register file and are never stalled. We do not model the power consumed by these pipeline modifications in detail. We believe that the average power dissipation of the in-order core will not greatly increase because some power is saved by never accessing the L1D cache and branch predictor. In any case, we show results while making different assumptions for the power consumed by the in-order core. For the inter-core network, we pessimistically triple its power consumption when employing RVP.

4 Results

4.1 Methodology

We use a multi-threaded extension 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 2-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 90nm technology. Wattch's RAM array and wire models were used to compute power dissipated by the inter-core buffers (RVQ, LVQ, etc.) and the aggressive clock gating model (cc3) has been assumed throughout. While Wattch provides reliable relative power values for different out-of-order processor simulations, we felt it did not accurately capture the relative power values of an out-of-order and in-order core. Hence, the in-order power values generated by Wattch were scaled so that average power consumed by our benchmark set was in the ratio 1:7 or 1:2 for the in-order and out-of-order cores. This scaling factor is computed based on relative power consumptions for commercial implementations of Alpha processors [14]. Our in-order core is modeled loosely after the quad-issue Alpha EV5 that consumes half the power of the single-thread out-of-order Alpha EV6 and $1/7^{th}$ the power of the multi-thread out-of-order Alpha EV8. Since the ratio of power efficiency of in-order and out-of-order cores is a key parameter in this study, we also show results for other scaling factors. The baseline peak frequency for all cores is assumed to be the same (5 GHz). As an evaluation workload, we use the 8 integer and 8 floating point benchmark programs from the SPEC2k suite that are compatible with our simulator. The executables

Branch Predictor	Comb. of bimodal and 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)
L1 I-cache	32KB 2-way (per Core)
L1 D-cache	32KB 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

Table 2: Simplescalar Simulation Parameters

were generated with peak optimization flags. The programs were fast-forwarded for 2 billion instructions, executed for 1 million instructions to warm up various structures, and measurements were taken for the next 100 million instructions. To evaluate multi-threaded models, we have formed a 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 million instructions.

4.2 Single-Thread Results

We begin by examining the behavior of a single-threaded workload. Figure 2 shows the IPCs of various configurations, normalized with respect to the IPC of the leading thread executing on an out-of-order core. Such an IPC analysis gives us an estimate of the clock speed that the trailing core can execute at. Since the trailing thread receives load values

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
swim-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

Table 3: Benchmark pairs for the multi-threaded workload.

and branch outcomes from the leading thread, it never experiences cache misses or branch mispredictions. The first bar for each benchmark in Figure 2 shows the normalized IPC for such a trailing thread that executes on an out-of-order core with perfect cache and branch predictor. If the trailing thread is further augmented with register value prediction (the second bar), IPC improvement is only minor (for most benchmarks). The third bar shows normalized IPCs for an in-order core with perfect cache and 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 out-of-order core. The last bar augments the in-order core's perfect cache and branch predictor with register value prediction (RVP). The increase in IPC is significant, indicating that RVP would be an important feature within an in-order core to allow it to execute at low frequencies. The IPC of a core with perfect cache, branch predictor, and perfect RVP is limited only by functional unit availability and fetch bandwidth. The average normalized IPC is around 3, meaning that, on average, the frequency of a trailing thread can be scaled down by a factor of 3.

We then evaluate the dynamic frequency scaling heuristic on the single-thread programs with different forms of trailing cores. The heuristic was tuned to conservatively select high frequencies so that performance was not significantly degraded. The first bar in Figure 3 shows power consumed by a trailing out-of-order core (with perfect cache and branch predictor) that has been dynamically frequency scaled. Such a trailing redundant core imposes a power overhead that equals 50% of the power consumed by the leading core. Without the DFS heuristic, the trailing core would have imposed a power overhead of 86%. On average, the trailing core operates at a frequency that is 0.44 times the frequency of the leading thread. This detailed simulation result was verified by the analytical equations in the previous section. In our simulation model, $wrongpath\_factor$ is approximately 1.17 and leakage accounts for roughly 25% of the leading core's power dissipation. If plugged into the analytical model, we derive that the leading core consumes 100 units of power, the trailing core consumes 89 units of power, and the trailing core with DFS consumes 53 units of power, roughly matching our detailed simulation results. We can derive similar rough estimates by plugging in different assumptions for leakage and $wrongpath\_factor$. The net outcome of the above analysis is that low trailer frequencies selected by the DFS heuris-

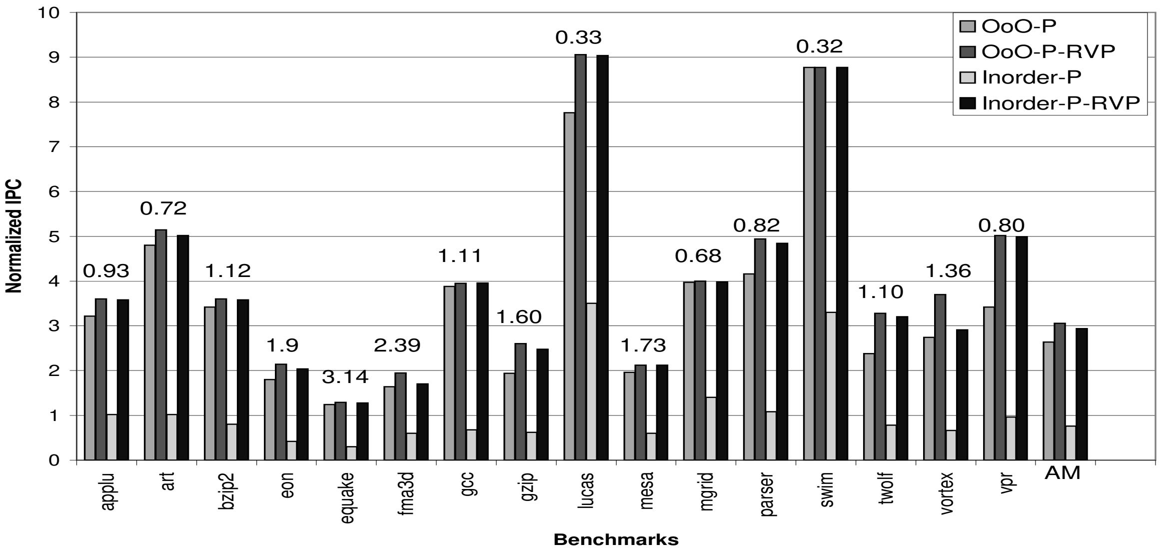

Figure 2: IPCs for different models, relative to the baseline out-of-order core. The absolute IPC for the baseline out-of-order core for each program is listed above each set of bars.

tic can reduce the trailer core's power by 40% and the overall processor power (leading and trailing core combined) by 19%.

Figure 3 also shows the power effect of employing an in-order trailing core. It is expected that future CMPs will be heterogeneous, providing a good mix of out-of-order and in-order cores [14]. 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 register value prediction. 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 4.86W (computations based on Wattch's RAM and wire models). We have further made the conservative assumption that the in-order core consumes as much as 55% of the power consumed by the single-threaded out-of-order core, on average (similar to the power ratio between the EV5 and EV6[2]). With these assumptions, we observe that the redundancy mechanism now consumes less than 30% of the power consumed by the leading thread. The frequency selected by the DFS heuristic for the in-order core was on average 0.42 times that of the leading core's frequency.

For all the above simulations, we assume an interval length of 1000 cycles, when making frequency decisions. Frequency changes were made for 70% of all intervals. If we as-

[2]Note that the in-order EV5 consumes only about 14% of the power consumed by the multi-threaded out-of-order EV8.

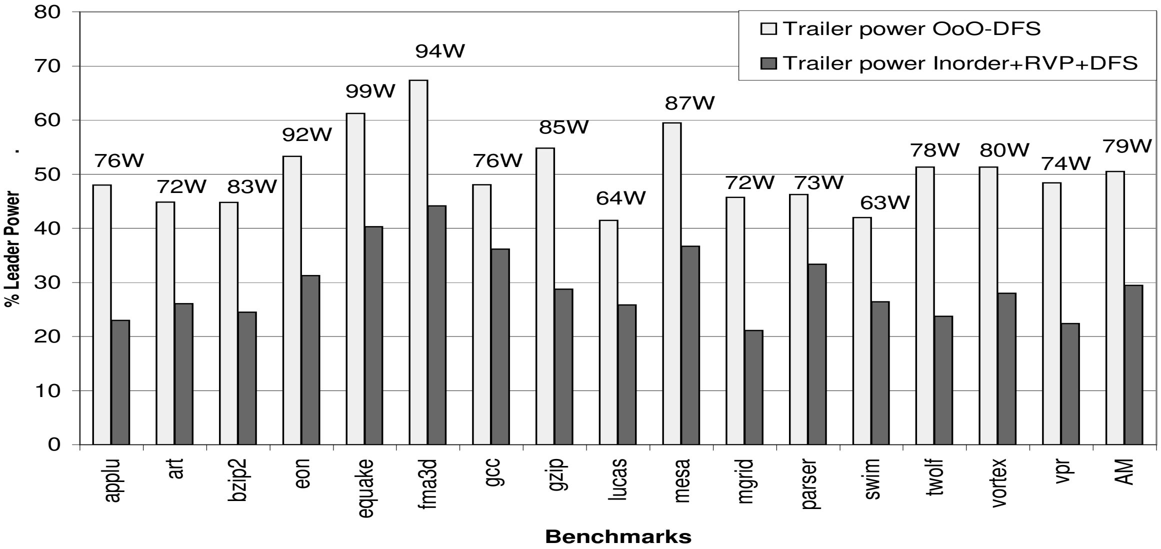

Figure 3: Power consumed by the trailing core as a function of the power consumed by the leading core. The number above each set of bars represents the absolute value of power dissipated by the leading core for each benchmark.

sume that a frequency change stalls the processor for 10 (peak frequency) cycles, the total overhead is only 0.7%. This number can be lowered by incorporating hysteresis within the algorithm, but this occasionally leads to an increase in slack and stalls for the leading thread. Hence, we instead chose to not include hysteresis and react quickly to variations in slack. The inter-core buffers were sized to accommodate a maximum slack of 1000 instructions.

Based on the above results, we make the following general conclusions. Executing the trailing thread on an out-of-order core has significant power overheads even if the trailing core's frequency is scaled. 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 a factor of two by employing register value prediction. This requires us to invest about 5W more power in transmitting additional data between cores (a pessimistic estimate), but allows us to scale the in-order core's frequency down by a factor of two. Hence, this is a worthwhile trade-off, assuming that the dynamic power consumed by the in-order core is at least 10W.

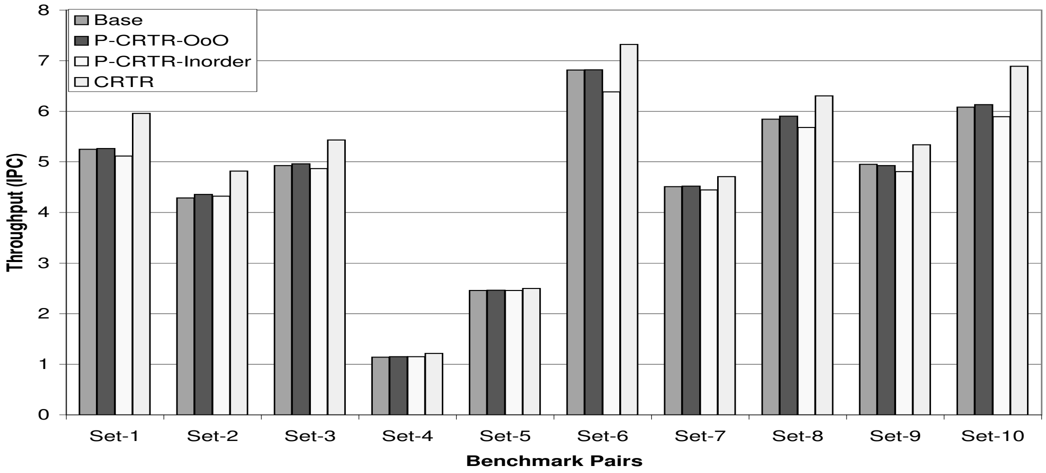

Figure 4: Total IPC throughput for multi-threaded workloads.

4.3 Multi-Thread Workloads

Next, we examine the most efficient way to execute a multi-threaded workload. As a baseline, we employ the CRTR model proposed by Gomaa *et al.* [7], where each out-of-order core executes a leading thread and an un-related trailing thread in SMT fashion. Within the power-efficient P-CRTR both leading threads execute on a single SMT out-of-order core and both trailing threads execute on a neighboring SMT out-of-order core that can be frequency scaled. The last bar in Figure 4 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 out-of-order core (no redundant threads are executed). It can be seen that the throughput of CRTR is about 8% better than a system where two leading threads execute on the same out-of-order core. This is because each leading thread in CRTR is co-scheduled with a trailing thread that does not execute wrong-path instructions and poses fewer conflicts for resources. Thus, any redundancy mechanism that schedules two leading threads together is bound to have a throughput at least 8% worse than CRTR. The second bar in Figure 4 shows IPCs for P-CRTR. 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 8% lower than CRTR, on average. The results of the earlier sub-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-inorder), where two leading threads execute on an SMT out-of-order core and the two trailing threads execute (by themselves) on two in-order cores with RVP and DFS. Again, the throughput of P-CRTR-inorder is similar to that of the baseline system, about 10% lower than CRTR, on average. Note, however, that P-CRTR-inorder is likely to have a lower area overhead than the other organizations in Figure 4 because the

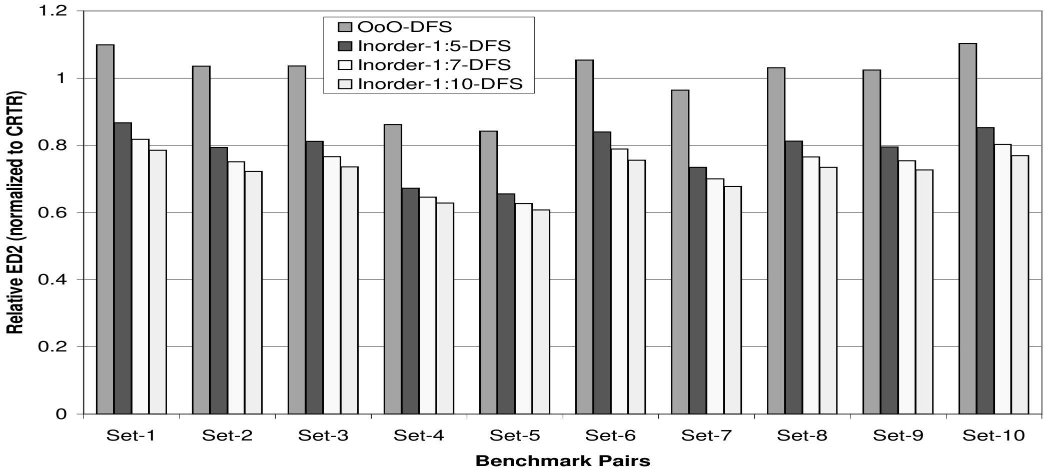

Figure 5: ED^2 for the entire system for various forms of trailers, normalized to the ED^2 of CRTR.

area occupied by two single-threaded in-order cores is less than the area occupied by one SMT out-of-order core [14]. We will now compare the power-efficiency of each of these approaches.

Figure 5 shows the $Energy \times Delay^2$ (ED^2) metric for different forms of P-CRTR, normalized to that for CRTR. Figures 4 and 5 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, where both trailers execute on an SMT out-of-order 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. The effective frequency for the trailing core is much higher (0.7 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, 5, 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 perfect cache and branch predictor, leading to low trailer frequencies and better overall ED^2 with P-CRTR. When scheduling redundant threads on a CMP of SMT out-of-order 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 addition of RVP to the trailing out-of-order core allowed minor reductions in effective trailer frequencies, yielding an overall ED^2 benefit of only 1% (not shown in figure).

The last three bars in Figure 5 represent P-CRTR-inorder models. As described earlier, the power consumed by the single-threaded in-order Alpha EV5 and the multi-threaded out-of-order Alpha EV8 is in the ratio 1:7. The third bar in Figure 5 represents such a model. The second and fourth bars represent systems where these ratios are 1:5 and 1:10, respectively. 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 average, P-CRTR-inorder (with power ratio similar to EV5 and EV8) improves ED^2 by 25% and power dissipation by 45%, relative to CRTR.

The net conclusion of these results is very similar to that in the previous sub-section. Leading threads executing on an out-of-order (either in single- or multi-threaded mode) can be verified on in-order cores. While more data has to be transmitted between cores, the power-efficiency of an in-order core more than compensates for the data transfer overhead.

4.4 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 (say), it may be possible to reduce the voltage by (say) 25% and observe dynamic power reduction within the trailer of 72%, instead of the 50% possible with just DFS. While DFS does not impact leakage power, DVFS can also reduce leakage as (to a first order) leakage is linearly proportional to supply voltage [5]. DVFS can be combined with body-biasing to further reduce leakage power [15]. 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, the frequency increase cannot happen until the voltage is increased, thereby causing stalls for leading threads. As observed earlier, a frequency change is made at the end of 70% of all 1000-cycle intervals. It is difficult to design a DFS mechanism that increases frequency only once every 100,000 cycles on 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. Figure 6 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% of

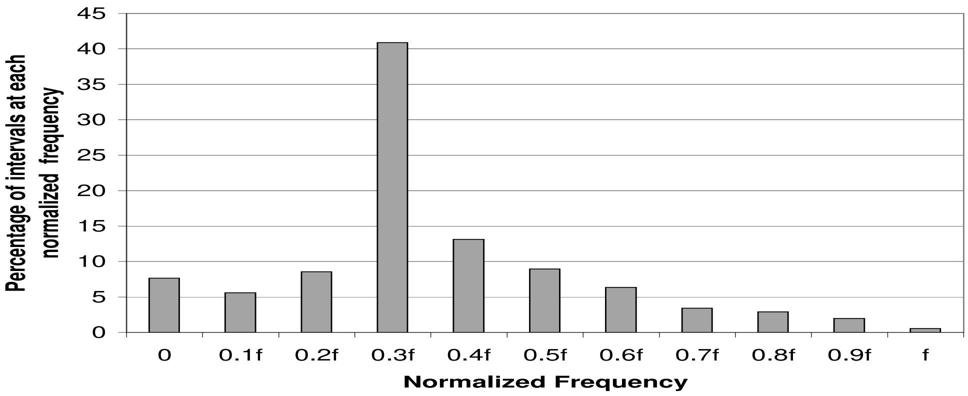

Figure 6: Histogram showing percentage of intervals at each normalized frequency.

all intervals and a frequency of $0.9\times$ peak frequency is exercised for 1.96% of all intervals. If we operate at a low voltage that can support a frequency of $0.9\times$ peak frequency, but not the peak frequency, we will be forced to increase voltage (and stall the leader for at least 10,000 cycles) for a maximum of 0.56% of all 1000-cycle intervals. This amounts to a performance overhead of up to 5.6%, which may be tolerable. However, employing a voltage that can support $0.8\times$ peak frequency, but not $0.9\times$ peak frequency, can lead to performance overheads of up to 25%. 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 would consume 14 units of power (ratio similar to the EV5 and EV8), of which about 10 units could be attributed to dynamic power. A DFS heuristic with an effective frequency of 0.5 would reduce the in-order core's dynamic power by 5 units. Thus, out of the total 109 units consumed by this system, 4 units can be attributed to in-order leakage and 5 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 [23]. 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 would equal 113 units (100 (leading core) + 8 (leakage on two in-order cores) + 2.5 (dynamic on first in-order core) + 2.5 (dynamic on second in-order core)). Thus par-

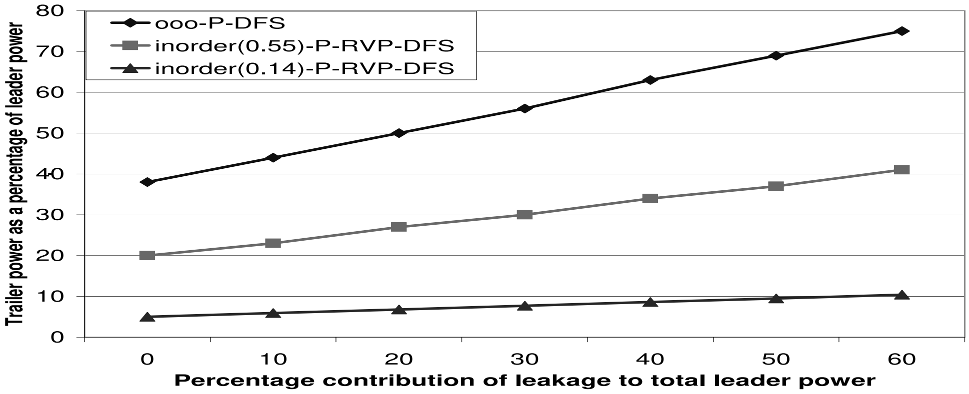

Figure 7: Trailer power as a function of contribution of leakage to the baseline processor.

allelization 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% (similar to that in the Xscale). Parallelization on two in-order cores yields a total power consumption of 108.8 units (100 (leading core) + 6 (leakage, linear function of supply voltage) + 1.4 (dynamic on first in-order core, quadratic function of supply voltage) + 1.4 (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, etc., can yield different quantitative numbers. For our simulation parameters, the analytical estimate above shows little power benefit from parallelizing the workload, even when we aggressively assume that voltage scaling has no overhead.

There are other problems associated with voltage scaling: (i) Lower voltages can increase a processor's susceptibility to faults. (ii) As voltage levels and the gap between supply and threshold voltages reduce, opportunities for voltage scaling may cease to exist. (iii) Parallelization has low scalability in voltage terms – parallelizing the workload across four cores allows frequency to be scaled down by a factor of four, but reductions in voltage become increasingly marginal.

4.5 Sensitivity Analysis

The previous section has provided analytical models that allow rough estimates of power to be calculated as various parameters are changed. Earlier graphs for the in-order core have already quantified results with different assumptions for the relative power between in-order and out-of-order cores. Another important parameter is leakage and Figure 7 shows power consumed by the trailer as a function of the contribution of leakage to the baseline out-of-order leading core. The three forms of trailers shown are an out-of-order core with the DFS heuristic, and in-order cores with RVP and DFS, that consume 0.55 times and 0.14 times the power of the out-of-order core. The overall conclusions of our study hold for all of these design points.

The effect of various parameters on IPC is harder to capture with analytical models and we report on some of our salient observations below. (i) The relative benefit of a perfect cache and branch predictor are significant for most processor models. For example, increasing the window size improves the ability of the baseline out-of-order core to tolerate cache misses, and likewise, also improves the ability of the core with the perfect cache/bpred to mine greater ILP. (ii) We have observed that for a multi-threaded workload, scheduling a leading and trailing thread on the same SMT core (as in CRTR) yields an 8% 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 because contention for resources becomes less of an issue. The gap is 5% when each core has four times the functional units listed in Table 2. (iii) 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 greatly 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. (iv) The slack between leading and trailing threads is closely related with interval size. If we examine slack every 1000 cycles to make a decision for trailer frequency, the slack must be about 1000 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 inter-core buffers getting full and stalling the leading thread. A large slack also allows more opportunities for frequency reduction, but incurs a non-trivial power overhead for the inter-core buffers.

5 Related Work

Many fault-tolerant architectures [2, 7, 20, 24, 25, 26, 33, 34] have been proposed over the last few years and our baseline models are based on previously proposed redundant multi-threading designs. AR-SMT [26] was the first design to use multi-threading for fault detection but was not optimized for performance. Mukherjee *et al.* later proposed fault detection using simultaneous multi-threading and chip-level redundant multi-threading [20, 25]. Vijaykumar *et al.* augmented the above techniques with recovery mechanisms [7, 33]. 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 inter-core interconnect. Our proposals, on the other hand, advocate transferring more data between threads to enable power optimizations at the trailer. Mukherjee *et al.* [21] characterize the architectural vulnerability factors (AVF) 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 [2] and SHREC [29], are inherently power-efficient because the helper pipelines they employ to execute redundant instructions are in-order-like. However, these designs require significant (potentially intrusive) modifications to a conventional microarchitecture and the helper pipeline cannot be used to execute primary threads. However, our results resonate with the ideas in these prior bodies of work and confirm that in-order pipelines provide the highest level of power efficiency. The DIVA implementation passes the inputs and outputs of every pipeline stage to the checker, resembling the execution of an in-order core with RVP.

A recent paper by Rashid *et al.* [23] represents one of the first efforts at explicitly reducing the power overhead of redundancy. As explained in Section 3, in their proposal, the leading thread is analyzed and decomposed into two parallel verification threads that execute on two other out-of-order 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: (i) In [23], 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 inter-core bandwidth, trailer core overheads can be reduced. (ii) [23] relies 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 dynamic frequency scaling and in-order execution. (iii) Our analytical results show that parallelization of the verification workload yields little benefit if we are already employing in-order cores augmented with RVP.

6 Conclusions and Future Work

In this paper, we have presented novel micro-architectural techniques for reducing the power overheads of redundantly threaded architectures. When executing a leading and trailing redundant thread, we take advantage of the fact that the leading thread prefetches data and resolves branch outcomes for the trailing thread. The results of the leading thread also allow the trailing core to implement a perfect register value predictor. All this information from the leading thread makes it possible for the trailing thread to achieve high IPC rates even with an in-order core. Dynamic frequency scaling further helps reduce the power consumption of the trailing thread. The major contributions of this paper are as follows:

- DFS heuristics 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.
- The proposal to invest in inter-core bandwidth to enable optimizations such as RVP, that further enable use of power-efficient in-order cores.
- Combining the above ideas to reduce the trailer core's power consumption to merely 10% 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.
- Analytical models that enable rough early estimates of different redundant multi-threading organizations.

As future work, we will study the thermal characteristics of redundant multi-threading 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.

References

[1] Advanced Micro Devices. AMD Demonstrates Dual Core Leadership, 2004. (`http://www.amd.com`).

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of MICRO-32*, November 1999.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of ISCA-27*, pages 83–94, June 2000.

[4] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[5] J. A. Butts and G. Sohi. A Static Power Model for Architects. In *Proceedings of MICRO-33*, December 2000.

[6] L. Clark. Circuit Design of XScale Microprocessors. In *Symposium on 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. In *Proceedings of ISCA-30*, June 2003.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001.

[9] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[10] P. Kongetira. A 32-Way Multithreaded SPARC Processor. In *Proceedings of Hot Chips 16*, 2004. (`http://www.hotchips.org/archives/`).

[11] K. Krewell. Intel's PC Roadmap Sees Double. *Microprocessor Report*, 18(5):41–43, May 2004.

[12] K. Krewell. Sun's Niagara Pours on the Cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[13] K. Krewell. UltraSPARC IV Mirrors Predecessor: Sun Builds Dualcore Chip in 130nm. *Microprocessor Report*, pages 1,5–6, Nov. 2003.

[14] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th International Symposium on Micro-Architecture*, December 2003.

[15] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors under Dynamic Workloads. In *Proceedings of ICCAD*, 2002.

[16] T. Maruyama. SPARC64 VI: Fujitsu's Next Generation Processor. In *Microprocessor Forum*, 2003.

[17] C. McNairy and R. Bhatia. Montecito - The Next Product in the Itanium Processor Family. In *Proceedings of Hot Chips 16*, 2004. (`http://www.hotchips.org/archives/`).

[18] A. Mendelson and N. Suri. Designing High-Performance and Reliable Superscalar Architectures: The Out-of-Order Reliable Superscalar O3RS Approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[19] S. Mukherjee, J. Emer, and S. Reinhardt. The soft-error problem: An architectural perspective. In *Proc. of 11th International Symposium on High Performance Computer Architecture HPCA*, 2005.

[20] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proceedings of ISCA-29*, May 2002.

[21] 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. In *Proceedings of MICRO-36*, December 2003.

[22] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*, October 1996.

[23] M. Rashid, E. Tan, M. Huang, and D. Albonesi. Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance. In *Proceedings of PACT-14*, 2005.

[24] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of MICRO-34*, December 2001.

[25] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of ISCA-27*, pages 25–36, June 2000.

[26] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[27] 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. In *Proceedings of HPCA-8*, pages 29–40, February 2002.

[28] 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. In *Proceedings of DSN*, June 2002.

[29] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of MICRO-37*, December 2004.

[30] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 System Microarchitecture. Technical report, Technical White Paper, IBM, October 2001.

[31] 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. In *Proceedings of ISCA-23*, May 1996.

[32] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA-22*, pages 392–403, June 1995.

[33] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of ISCA-29*, May 2002.

[34] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *Proceedings of DSN*, June 2004.

[35] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to reduce to soft-error rate in high performance microprocessors. In *Proc. of 31st Annual International Symposium on Computer Architecture ISCA*, 2004.