Exploiting Slack for Low Overhead Soft Error Reliability

Premkishore Shivakumar and Stephen W. Keckler
Department of Computer Sciences, The University of Texas at Austin

***Abstract*— Designing low overhead mechanisms for improving soft error reliability will be a key requirement at future technologies. The *slack* of an instruction is the number of cycles it can be delayed without extending the execution time. We make the observation that while the slack cycles do not affect execution time, they add to the total number of program cycles vulnerable to soft errors. In this paper, we show that exploiting slack to reduce this component of vulnerable cycles, has significant potential to improve soft error reliability. We explore two different mechanisms for exploiting slack, which reduce reduce soft error rate by 34-42%, at a performance overhead of only 6-10%. We also demonstrate that while redundant execution incurs high performance overhead, these techniques can efficiently adapt to the amount of slack available in different programs to achieve reliability improvement with minimum performance overhead.**

***Index Terms*—Soft error, slack, delay and observe, static scheduling slack**

I. Introduction

Soft errors due to neutron and alpha particle strikes are an increasing concern with exponentially rising transistor counts at future technologies. Historically, fault tolerance at the processor level has adopted an all-or-nothing approach, using techniques that are either restricted to parity, ECC, and redundant rows in caches, or employing worst-case design techniques such as redundant execution to achieve very high reliability at correspondingly high cost [1]. While prior studies indicate that just adding parity to major on-chip structures may be insufficient to counter latch and logic error scaling trends [2], redundant execution based mechanisms will be justified only for worst-case soft error rate scaling trends or for markets with very stringent reliability requirements, and will be over-designed in other cases.

Modern microprocessors employ several concurrency techniques such as speculative execution to improve performance. These techniques achieve higher performance by aggressively bringing future program state into the processor and mining them for available parallelism. As these techniques also increase the average number of cycles for which program state is resident on the processor, they increase the probability of the state being corrupted by a soft error. Mukherjee et al. [3] provided the insight that a structure is vulnerable to soft errors only during cycles when it contains state that can affect final program output, and introduced the metric *architecturally correct execution* cycles (ACE cycles) as a measure of reliabillity. A key observation is that while exploiting parallelism can significantly reduce execution time, fetching program state of the majority of program paths can be delayed by several cycles without extending the execution time. Fields et al. termed this latency tolerance exhibited by program state as *slack*, and demonstrated that there is a significant amount of slack in program execution [4]. In this paper, we show that there is significant potential for exploiting this abundant execution slack to reduce the amount of vulnerable processor state and improving reliability, and at the same time also minimize performance overhead. We explore two techniques for exploiting slack, a dynamic *delay and observe* approach, and a static technique using the compiler, achieving reliability improvement of 34-42% at only 6-10% performance overhead.

The rest of the paper is organized as follows. Section II introduces *slack* ACE cycles and discusses methods for estimating slack. Section III presents our methodology. Section IV evaluates our proposed techniques, and Section V discusses its implications. Finally, Section VI reviews some related work, and Section VII presents our conclusions.

II. Slack ACE Cycles

Slack can be broadly defined as the number of cycles an instruction or an event can be delayed without extending the execution time [4]. While the slack cycles do not contribute to the execution time of the program, the vulnerable processor state associated with that slack accumulates ACE cycles during this period, which we term *slack ACE cycles*. There are many scenarios where slack ACE cycles are accumulated in a conventional processor, providing opportunities for improving reliability. A simple example is when an instruction arrives ahead of the operand, and waits in the instruction window until it is ready to be executed, accumulating slack ACE cycles during that period. Techniques such as front-end throttling during periods of low utilization may be effective here in reducing these slack ACE cycles. A related scenario is when the data required by an instruction is present in the register file well before it is actually needed, adding slack ACE cycles until it is read. Compiler or hardware instruction scheduling techniques can delay register writes and advance register reads to compress the vulnerable window and reduce register file slack ACE cycles. Architectural constraints such as in-order commit of instructions from the ROB and stores from the store queue can also force younger instructions to accumulate slack ACE cycles while waiting for older instructions to commit.

There have been various techniques used to estimate slack:

a) Delay and observe: A simple approach that estimates instruction slack by delaying its execution by n cycles, and observing if the overall execution time is affected. Srinivasan et al. used this approach to compute the latency tolerance of program loads [5].

b) Static slack estimation: Compilers generally use static estimates of instruction slack to perform efficient instruction scheduling within region boundaries (e.g., hyperblock) to achieve higher performance ([6], [7]).

c) Profiling using a dependence-graph model: Fields et al. proposed an off-line profiling methodology based on

constructing a dependence-graph model of dynamic program execution and analyzing the graph to determine execution slack [4]. Muthler et al. estimated slack using this methodology and used it to improve performance [8].

d) Dynamic slack prediction: Fields et al. also implemented a dynamic hardware slack prediction mechanism to achieve processor energy reduction [4].

In this paper, we evaluate the first two techniques for reducing instruction slack, and thus the *slack* ACE cycles, to improve soft error reliability, while recognizing that the remaining two techniques are also equally applicable.

III. Methodology

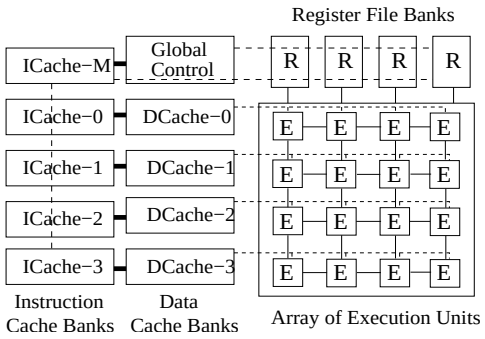

Fig. 1. TRIPS processor.

Figure 1 illustrates the organization of the TRIPS architecture we used for our analysis [9]. The TRIPS architecture contains a two-dimensional array of computation elements connected by a thin mesh operand routing network (OPN). Each ALU includes an integer unit, a floating point unit, an operand router (OPN), and instruction and operand buffers for storing instructions and their operands. The level-1 instruction and data cache units, along with the register files (RF) are distributed and placed along the periphery of the execution array. The processor also uses read (RQ) and write (WQ) queues, one per register file bank, for register forwarding, and one load-store queue (LSQ) bank per data cache bank for load and store data forwarding. A 4×4 execution array, with 64 instruction window entries at each ALU, thus represents an instruction window of 1024 instructions. The TRIPS architecture uses a block-atomic, static-placement dynamic-issue (SPDI) execution model. Blocks of 128 instructions are fetched and mapped for execution as one unit. The 1K instruction window thus allows eight such instruction blocks to be simulaneously mapped and speculatively executed on the processor. The compiler statically assigns each block instruction a particular instruction window entry; the instruction then dynamically issues based on when it becomes ready. Instruction blocks are deallocated en masse after the block register and store outputs have been committed.

We use the methodology proposed by Mukherjee et al. to estimate ACE cycles [3]. The methodology only counts cycles that contribute to correct program execution, and ignores all cycles including but not limited to program state that is idle, invalid, mis-speculated, predicated-false, dynamically dead, logically masked and performance-enhancing in nature [3]. We extend this methodology in two ways. First, we augment the framework to compute the slack ACE cycles for each structure in addition to the total ACE cycles. Second, we add support for computing the contribution from the individual static instruction blocks to the total dynamic ACE cycles, providing a reliability characterization of the program. Associating an ACE cycle with a particular block is straightforward in the TRIPS architecture, since instructions are statically mapped to the hardware [9]. We compute the ACE cycles for the intruction and operand buffers, register file, RQ, WQ, LSQ storage structures and the OPN communication buffers. The ACE cycles presented in the rest of the paper is the sum of the ACE cycles across all these structures. The execution cycles and ACE cycles are both computed using a cycle-accurate, validated execution driven simulator that models the TRIPS architecture and the distributed protocols in great detail [9]. We used 25 benchmarks from the EEMBC 2.0 suite for our results.

Slack ACE cycles	Instruction Buffers	Operand Buffers	LSQ Structures	Register Structures RF/RQ/WQ
89.6%	70.8%	11.6%	3.8%	13.8%

TABLE I
Distribution of Slack ACE Cycles.

IV. Exploiting Slack to Improve Soft Error Reliability

Table I shows the distribution of processor slack ACE cycles. Slack ACE cycles are shown to account for a significant fraction ($\sim$ 90%) of the total ACE cycles, due to the abundant slack available in programs [4]. The remaining table entries show how the slack ACE cycles are divided among the individual processor structures. The instruction buffers are the dominant component (70.8%), with the operand buffers and register file structures also contributing a significant fraction. We propose two schemes to reduce the slack ACE cycles, and quantify the reduction in the overall processor ACE cycles and the processor SER.

We build on the simple observation that execution of the current TRIPS instruction block effectively commences when its data dependences with older blocks in the program are satisfied through the data forwarding logic. In particular, we focus on the inter-block register dependences, delivered through the register forwarding logic from one of the other concurrently executing blocks. The schemes identify the appropriate block register input to trigger just-in-time instruction fetch of the block, holding the instructions until that time in ECC or parity protected instruction caches. The goal here is to overlap the time the register value takes to travel from the register file interface to the consumer instruction at the destination execution unit, with the time the instruction takes to travel from the instruction cache interface to the same execution unit. If the register input trigger is too early then there is not much reduction in the slack ACE cycles. On the other hand, if the register input trigger is too late it can significantly extend

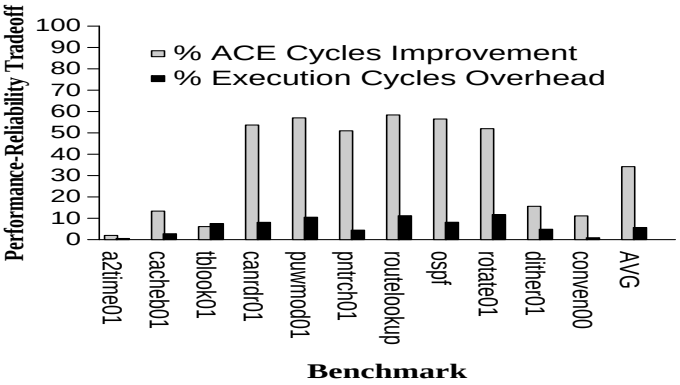

Fig. 2. Reliability-Performance Tradeoff with Delay and Observe.

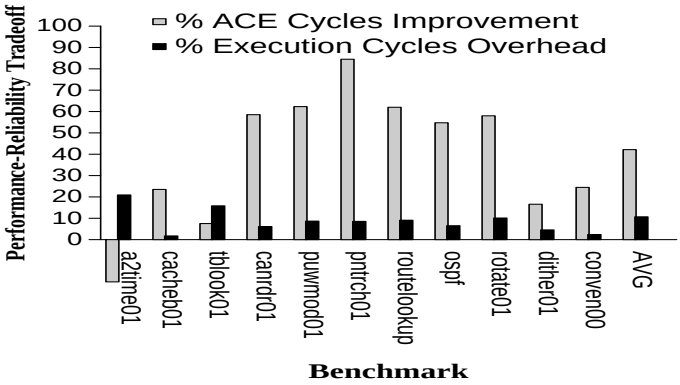

Fig. 3. Reliability-Performance Tradeoff using Static Slack.

execution time, and hence also lead to an overall increase in ACE cycles. We propose to use the slack associated with each register input to identify the optimal register input trigger to maximize reduction in ACE cycles, and minimize execution time overhead. We explore two of the methods described in Section II: 1) a dynamic *delay and observe* approach, and 2) a static compiler estimation of slack.

A. Delay and Observe

Basically, the *delay and observe* approach explores each register input one-by-one as the potential choice for the block instruction fetch trigger. For each choice it observes whether the execution time was delayed and exceeded a specified acceptable threshold, which we set at 15% in this paper. The following paragraph describes the detailed algorithm we implement, which basically uses gradient descent to arrive at the optimal setting.

The execution of a program is divided into multiple intervals. During each execution interval, application performance in instructions-per-cycle (IPC) and the ACE cycles accumulated are computed. The first interval is used to determine the baseline IPC, and ACE cycles for the program in that interval. In the next four intervals, we explore four different natural choices for the register input trigger: 1) the first arriving register input, 2) the second arriving register input, 3) the third arriving register input, and 4) the fourth arriving register input for that block. If a block has less than four register inputs, instruction fetch is triggered as soon as all its register inputs have arrived. For instance, a block with two register inputs really has only the first two choices for the register input trigger, with choices 3 and 4 being identical to 2. The optimal register input trigger maximizes the reduction in ACE cycles, while incurring a performance overhead that is within the acceptable threshold. All register input triggers are discarded and the baseline instruction fetch mechanism is used, if none of the register input triggers fall within the acceptable threshold, although there is still some performance overhead due to this sampling process. The program then continues execution with this optimal setting for the next 20 intervals, after which this entire process is repeated to account for changes in program behaviour. In this paper, we chose the interval size to be 50K cycles to react quickly to changes in program behaviour without letting the tuning mechanism cause a high overhead. Currently, we use the simulator to measure the ACE cycles in each interval, as described in Section III. Using hardware performance counters to measure ACE cycles was suggested in prior work [3], and we think it is a promising area for future work.

Figure 2 presents the results for a subset of the benchmarks along with the average. The technique provides significant improvements for many benchmarks, with an average reduction of 34% in ACE cycles at just 5.6% performance overhead. These results prove that there is substantial slack in these applications which can be exploited to improve reliability. Further, preliminary results for a subset of SPEC integer and floating point benchmarks showed a reliability improvement of 28% with just 2.4% performance overhead, and a reliability improvement of 5.5% with 3.7% overhead respectively.

B. Compiler Based Static Slack Estimation

The TRIPS compiler performs instruction scheduling for each instruction block. The goal of instruction scheduling is to minimize the completion time of the block by exploiting instruction level parallelism, minimizing static routing latency between dependent instructions, and minimizing dynamic latencies such as contention whenever possible. The algorithm uses heuristics to model contention on the ALUs and OPN links, and provide lookahead for scheduling dependence chains. While the scheduling region is limited to the block, the algorithm also accounts for inter-block (global) critical paths (such as loop-carried dependences) to model the block register input arrival times, which is critical in estimating the slack of register inputs [7].

At the end of scheduling the block, the same algorithm is used to compute the critical path length through each block register input. The register input with the longest critical path length has no slack, and if used to trigger block fetch will most likely extend the execution time. As described earlier, the goal is to overlap the time the register value takes from the register file interface to the consumer instruction, with the time the instruction takes from the instruction cache interface. The compiler approximates this by annotating each block with the register input that has an estimated arrival time just smaller than that of the register input with the longest critical

path length. The hardware looks for this block register input annotation to trigger instruction fetch for the block.

Figure 3 presents the results for a subset of the benchmarks along with the average across all of of them. The technique provides an average improvement of 42% in ACE cycles, improving over the *delay and observe* approach. Although the performance overhead is double of the *delay and observe* technique, it is still low at 11%.

V. Discussion

These techniques demonstrate that substantial savings in ACE cycles of about 35-40% can be obtained at only 6-10% overhead if the processor intelligently exploits slack. While redundant execution provides even higher reliability, near 100%, it also incurs a significant performance overhead ($\sim$ 35%) [1]. The overhead of redundant execution is proportional to processor utilization, since almost every program operation is redundantly performed. Gomaa et al. proposed selective redundant execution during periods of low processor utilization to significantly reduce this performance overhead [10].

In this section, we show that techniques based on exploiting slack can adapt to the amount of execution slack available, and efficiently extract reliability improvements even during periods of high utilization. We use the infrastructure described in Section III to identify the instruction block that contributes the maximum to overall dynamic program ACE cycles. We compare the reliability improvement and the execution overhead of redundantly executing only this block, with exploiting the compiler determined static scheduling slack of the register input trigger for this block. Table II presents the results for the top block of two benchmarks. $pntrch01$ (pointer chasing) is an application with low inherent parallelism, and $t\_run\_test\$14$ is the top block which contributes 20.5% to the overall ACE cycles. While redundant execution improves reliability by 51%, it incurs 28.5% performance overhead. Exploiting slack, on the other hand, achieves only 18% reduction in ACE cycles but incurs negligible performance overhead of 1.1%. The comparison is even more stark with $conven00$, which has high parallelism and hence lesser execution slack. Redundant execution of $convolEncode\$10$ incurs almost 65% overhead to provide a 12.8% increase in reliability. On the other hand, exploting slack adapts to the lesser slack available and provides 9.6% reduction in ACE cycles with 0.6% overhead.

VI. Related Work

The work that comes closest to this is the concept of triggers and actions proposed by Weaver et al. [2]. They reduced the slack ACE cycles accumulated on an L2 miss by flushing the instruction window, which incurred minimal performance overhead in an in-order architecture. While this technique manages the originally available slack, the techniques we propose aim to reduce slack by spreading the computation appropriately. Muthler et al. used profiling to estimate instruction slack and used it to improve performance [8]. Fields et al. implemented dynamic slack prediction to reduce processor energy consumption [4]. Karnik et al. showed that almost 70% of the hardware logic paths are non-critical, and exploited this circuit level slack for explicit capacitance insertion to improve soft error rate [11]. In this paper, we demonstrate that exploiting microarchitectural slack can substantially improve soft error reliability.

Technique	Application Block	Percentage Contribution	ACE Cycles Reduction	Execution Overhead
Selective Redundant Execution	pntrch01 $t\_run\_test\$14$	20.5%	51%	28.5%
	conven00 $convolEncode\$10$	24.6%	12.8%	64.3%
Exploiting Scheduling Slack	pntrch01 $t\_run\_test\$14$	20.5%	18%	1.1%
	conven00 $convolEncode\$10$	24.6%	9.6%	0.6%

TABLE II
Comparison of Techniques

VII. Conclusions and Future Work

We show that there is substantial potential for exploiting slack to improve soft error reliability. We demonstrate that a significant amount of ACE cycles ($\sim$ 90%) are accumulated due to slack. We explore two techniques, a dynamic *delay and observe* approach, and a static technique using the compiler, achieving reliability improvement of 34-42% at only 6-10% performance overhead. Further, these techniques efficiently adapt to the available slack in programs to provide lower overhead than redundant execution. In the future, we intend to explore the potential of both dynamic hardware slack prediction similar to [4], and using profiling [8] to compute slack and using it to improve soft error reliability.

References

[1] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives." in *ISCA*, 2002, pp. 99–110.

[2] C. Weaver, J. S. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor." in *ISCA*, 2004, pp. 264–275.

[3] S. S. Mukherjee, C. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor." in *MICRO*, 2003, pp. 29–42.

[4] B. A. Fields, R. Bodík, and M. D. Hill, "Slack: Maximizing performance under technological constraints." in *ISCA*, 2002, pp. 47–58.

[5] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *MICRO*, 1998, pp. 148–159.

[6] S. S. Muchnick and P. B. Gibbons, "Efficient instruction scheduling for a pipelined architecture," *SIGPLAN Not.*, vol. 39, no. 4, pp. 167–174, 2004.

[7] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, "A spatial path scheduling algorithm for edge architectures," in *ASPLOS XII*, 2006, pp. 129–140.

[8] G. Muthler, D. Crowe, S. Patel, and S. Lumetta, "Instruction fetch deferral using static slack," in *MICRO*, 2002, pp. 51–61.

[9] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *ISCA*, June 2003, pp. 422–433.

[10] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection." *IEEE Micro*, vol. 26, no. 1, pp. 92–99, 2006.

[11] T. Karnik, S. Vangal, V. Veeramachaneni, P. Hazucha, V. Erraguntla, and S. Borkar, "Selective Node Engineering for Chip-Level Soft Error Rate Improvement," *VLSI Circuits Symposium*, June 2002.