Exploiting Locality to Improve Circuit-level Timing Speculation

Jing Xin, *Northwestern University* and Russ Joseph, *Northwestern University*

Abstract—Circuit-level timing speculation has been proposed as a technique to reduce dependence on design margins, eliminating power and performance overheads. Recent work has proposed microarchitectural methods to dynamically detect and recover from timing errors in processor logic. This work has not evaluated or exploited the disparity of error rates at the level of static instructions. In this paper, we demonstrate pronounced locality in error rates at the level of static instructions. We propose timing error prediction to dynamically anticipate timing errors at the instruction-level and reduce the costly recovery penalty. This allows us to achieve $\mathbf{43.6\%}$ power savings when compared to a baseline policy and incurs only $\mathbf{6.9\%}$ performance penalty.

Index Terms—Error locality, execution unit, timing speculation, low-power design, reliability.

1 Introduction

Recent work has shown that circuit-level timing speculation can provide *better than worst-case designs* that reduce safety margins by introducing non-zero error rates [3]. Timing speculation allows the processor to function at voltage, frequency, and thermal operating points which would not guarantee traditional signal setup time constraints for all logic paths. Under this paradigm, timing errors occasionally appear in the pipeline. They are dynamically detected and the processor initiates a recovery mechanism to prevent the errors from affecting architecturally visible state. These techniques can be used to either dynamically lower the operating voltage to reduce power consumption or raise the operating frequency to improve instruction throughput. Recent work has sought to reshape error rates through dynamic management of the processor [11] or design time optimizations of logic [4].

However, previous work has not thoroughly explored the impact that instruction sequence and program values can have on timing error rate. Some studies suggest that locality related to control and data flow properties of the program may have substantial impact on hardware faults which we believe also extend specifically to timing errors. Li et al showed that trace based analysis can successfully diagnose permanent hardware faults in modern pipelines [7]. This hints that there is a strong correlation between instructions/data and detected errors. We believe that this information can be used to reason about the likelihood of a timing error appearing when the same instruction sequence reappears. In a broader context, earlier studies have shown strong locality patterns in data [9] and demonstrated energy savings when narrow width operands are detected [2].

In this work, we explore error locality in static instructions whose data usage patterns sensitize delay fault paths (*timing error locality*) and describe how this locality can be utilized to reduce the effective timing errors (errors which require full recovery). Our initial work evaluates very simple mechanisms that predict instructions likely to produce a timing error in the pipeline and make preparations that reduce the recovery cost. In our evaluation, this decreased recovery cost would allow a processor implementing dynamic voltage scaling (DVS) and Razor based error recovery to avoid the full cost of pipeline flush and replay. This effectively lowers the error rate and allows the processor to operate at more aggressively scaled voltages and hence consume significantly less power. Our approach is enabled largely by fast, value sensitive delay fault models for processor execution units.

• *Manuscript submitted: 17-Sep-2009. Manuscript accepted: 08-Oct-2009. Final manuscript received: 15-Oct-2009.*

2 Timing error locality in pipelines

In low-power pipelines, the penalty of timing error recovery imposes a limit on voltage scaling and may limit some applications of timing speculation. Even in designs where cost of error recovery is relatively low (e.g. less than ten cycles), this could significantly restrict the optimal error rate and demand that the processor utilize a less aggressive operating voltage with a consequently diminished power saving opportunity. In this work, we examine timing error locality and explore ways to lower error recovery costs, allowing us to further the use of timing speculation for energy efficiency.

We examine the relationship between error rate and static instructions by evaluating three benchmarks (`basicmath`, `fft` and `gsm`) from the MiBench embedded benchmark suite [6] on a single issue in-order pipeline equipped with Razor-like error recovery. Our error model captures path delay faults by simulating gate-level transitions along critical paths. In this study we focus on high-activity execution units including the single-cycle integer ALU and pipelined integer multiplier; we plan to extend our models to the entire pipeline in future work.

On average about 80% of the static instructions are reused and they account for over 99.5% of the dynamic executed instructions. Figure 1 shows the distribution of error rate across reused static instructions for different operating conditions. We classify instruction error rates into three groups, *error free*, *low error rate* (less than or equal to 20%), and *high error rate* (greater than 20%).

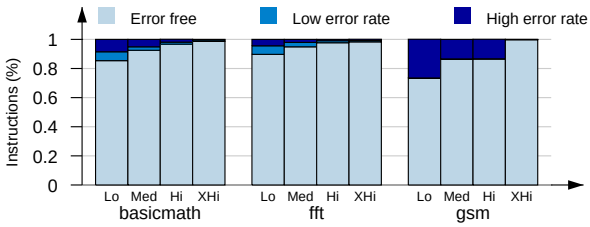


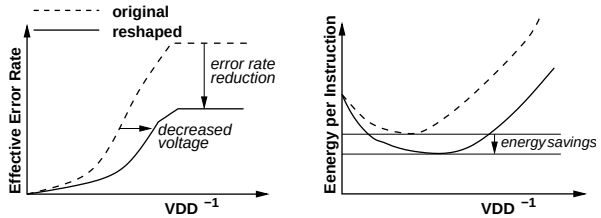
Fig. 1. Distribution of error rate across instructions

We evaluate this workload at four different operating conditions: `Lo`, `Med`, `Hi` and `XHi` VDD which increase circuit latencies by 40%, 30%, 20% and 10% respectively. As the operating voltage decreases, the proportion of instructions with *high error rate* increases as well. In general *error free* and *high error rate* instructions constitute the majority of the population, which gives us an opportunity to distinguish them and hence to make preparations to reduce the recovery cost. Different applications have varying distributions of these three kinds of instructions. This illustrates the potential of online techniques to adapt to an application.

3 Error padding to reduce error recovery time

By aggressively scaling the supply voltage, we increase the number of unique logic paths that are timing critical and subsequently increase the error rate. This helps timing speculation to decrease the energy per instruction (EPI) and move it towards a minimal value reached at the critical voltage. Decreasing the supply voltage beyond this point causes the error rate to sharply rise and consequently increases the effort the pipeline spends in error recovery, which has a negative impact on EPI. Figure 2 conceptually shows how error prediction and padding could reshape the effective error rate (EER) and EPI.

By efficiently predicting and padding timing errors, the operating voltage can be further decreased, and the effective error rate for a fixed voltage is reduced as shown in Figure 2(a). As Figure 2(b) shows, this lowers the optimal operating voltage and improves energy savings.



(a) EER (b) EPI

Fig. 2. Reshaped effective error rate and energy per instruction

In this work we offer a novel approach to reshape the effective error rate curve by predicting and isolating timing errors using locality information and pipeline scheduling.

3.1 Error prediction

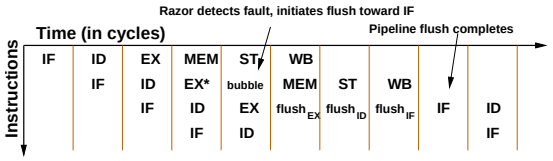
We propose and evaluate a simple dynamic timing error prediction strategy which relies on pre-decode bits in the I-Cache. We augment the I-Cache by adding two bits of pre-decode information per instruction. We refer to these state bits as *criticality counts*. When an instruction miss occurs, main memory (or L2 Cache) will transfer the required cache block to the cache structure and the criticality counts for instructions in the block are reset to zero. On each I-Cache hit, we fetch the criticality count along with the instruction. During instruction decode, we flag an instruction as being critical or non-critical by comparing its criticality count to a criticality threshold. Instructions with criticality counts equal to or greater than this threshold are marked critical. In general, critical tags are incremented in the presence of a timing error for that instruction and decremented if the instruction does not produce a timing error. We explored various policies for updating criticality counts and evaluated many thresholds. We found that saturating counters with an asymmetric update policy ($\Delta\ up\ =\ 3$ and $\Delta\ down\ =\ 1$) and a criticality threshold of two worked well. The asymmetric update policy allows us to respond quickly to changes in working set or data flow patterns. We choose to implement criticality counts as pre-decode bits in the I-cache rather than a dedicated structure. This design decision reflects the notion that we can piggyback use of address decoding structures and tag resolution already present in the I-cache for read accesses. We add a write port that allows us to update only the criticality counts during writeback. We use CACTI [12] to quantify the additional area and energy. We estimate 14.2% increase in the I-cache area and 1.4% increase in energy for the processor. We also note that count update can be done off the critical path and hence would not impact cycle time. Our approach does not preclude the use of a separate structure to cache criticality predictions if this offered better performance or power benefits.

We make one additional optimization which helps to reduce false positive error predictions due to stale criticality counts. After a long interval of non-use, instructions which were once critical often become non-critical on subsequent execution. Note that our asymmetric update policy identifies this decreased error rate much slower than it identifies increased error rate. In terms of criticality predictions, this lead to a large number of false positives. We found that periodically clearing the criticality counts every one million instructions served as an effective remedy. We believe that this approach may eventually evolve to more sophisticated mechanisms. We plan to investigate more adaptive and flexible schemes in our future work.

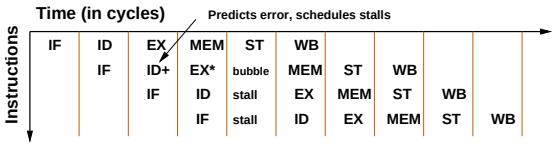
3.2 Error padding mechanism

By identifying instructions that are likely to produce timing errors, we can perform a technique which we call *error padding*. For instructions marked as critical, we force an extra cycle of computation for each cycle of the nominal instruction execution and force subsequent instructions to stall. The additional cycles are used to enable error correction via late arriving signals captured in Razor flip-flops. By scheduling stall cycles for subsequent instructions, we avoid collisions in the pipeline

due to the recovery and ensure that dependent instructions receive the correct, forwarded values produced by the critical instruction. At the circuit-level this does not actually prevent timing errors from occurring; it merely reduces the number of timing errors that need full microarchitectural recovery. Figure 3(a) shows Razor error recovery in a six stage pipeline. Note that once an error is detected, a pipeline flush is initiated and instructions are refetched to complete recovery. Under error padding, we can avoid the full cost of recovery (as shown in Figure 3(b)) by anticipating timing errors at the ID stage.



(a) Pipeline Timing for a failure in the EX stage of the pipeline (Razor) [3].



(b) Pipeline Timing for a predicted failure in Error Padding

Fig. 3. Pipeline diagrams for baseline error recovery and the proposed error padding. The presence of '+' and '*' denote error prediction and occurrence of an error respectively.

Note that once the instruction is marked as critical, we incur the full cost of error padding regardless of whether the instruction generates a timing error or not. Consequently, there is a cost for aggressively marking instructions which have low error rates as critical. On the other hand, if the instruction is not marked critical, instruction execution and any necessary error detection/recovery work as they would in a standard Razor pipeline. Via asymmetric update, this instruction will be flagged as critical on its next appearance in the pipeline.

4 Evaluation

Our evaluation focuses on the power and performance impact that error padding has on a high-performance, low-power embedded processor. We extend the `M5` full system simulator ([1]) to model a single issue in-order Alpha core. Our processor adopts Razor error detection and recovery to apply circuit-level timing speculation. We model timing errors by integrating a fast, home-grown gate-level simulator into the `M5` processor model. We developed gate-level Verilog models for two key logic structures in the execution backend: a Kogge-Stone adder and a pipelined Wallace tree multiplier. On a per cycle basis, we drive the circuit simulator with program data values, allowing us to model the coupling between program control/data flow patterns and timing errors in the execution units. We choose to focus on logic dominated structures in the processor back-end and do not directly address logic in the processor front-end, because they have impact on performance due to recovery time. In essence, the counterflow pipeline recovery used in Razor designs is sensitive to the pipeline depth of timing errors. Front-end timing errors would result in fewer instructions being squashed and would incur far lower recovery penalties. Also we do not include timing errors in SRAM structures because there is a growing body of recent work which has explored tolerance to variation in SRAM structures [10], [8]. These techniques are all orthogonal to and could be applied in combination with error padding mechanism. Our experimental results consider the power overheads in reading and maintaining criticality counts.

We select three integer benchmarks from the SPEC CPU2000 suite and six benchmarks from MiBench [6]. All applications are compiled for the Alpha architecture using the `gcc` toolchain using full optimizations. We use representative execution windows for the MiBench and SPEC benchmarks.

4.1 Predictor Accuracy

We first evaluate the efficiency of our simple timing error predictor under different operating voltages. We utilize the predictor estimation terminology in [5] and focus on the sensitivity (the percentage of timing errors which are correctly identified) in Figure 4 under four different operating voltages as described in Section 2. At lower operating voltages, propagation latencies increase, more paths become critical, and the error rate spikes. For `Lo` VDD, our predictor can identify about 80% of the timing errors on average which suggests that error padding could effectively eliminate the recovery penalty. As VDD increases, timing errors become less frequent and are more difficult to identify. Moreover, if an instruction is falsely identified as critical, an additional padding penalty is incurred. This could reduce the benefits of error prediction and padding.

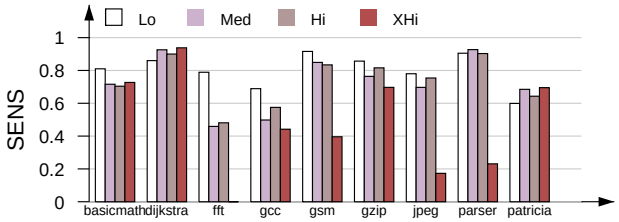


Fig. 4. Comparison of Sensitivity

4.2 Performance and power impact

To evaluate the performance of error prediction and padding mechanism, we selected three different operating modes, `Lo` VDD, `Hi` VDD (as described in Section 2), and a dynamically controlled voltage. The fixed voltages illustrate behavior at the extremes of the operating conditions and error rates. We designed the adaptive voltage scaling mechanism to minimize power consumption while limiting performance penalty to 10% relative to an unoptimized baseline. Our mechanism periodically evaluates the error recovery time relative to the elapsed cycle count once every $1K$ timing errors. We periodically reset the operating voltage to avoid local minima. In this paper, we choose a baseline voltage which does not

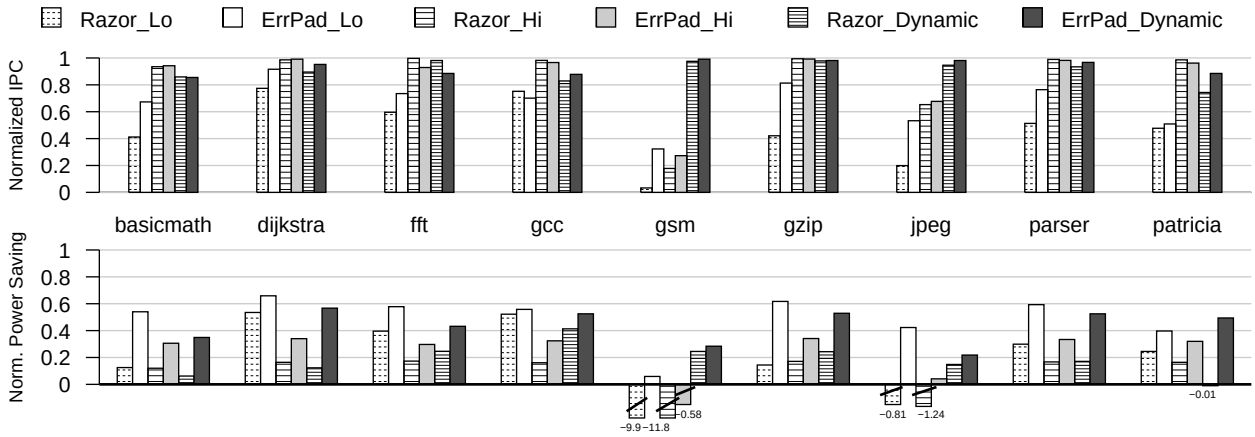


Fig. 5. Comparing performance (top) and power savings (bottom)

include temperature or process variation margins, so the ability to scale the voltage comes directly from timing slack due to under sensitized critical paths.

We compare the performance of our mechanism with a baseline Razor-like design in Figure 5(top). All results are normalized to a baseline design which operates at a fixed maximum voltage and has no timing errors at all. On average our mechanism has 20.0% higher normalized IPC than the Razor-like design under `Lo` voltage, and 9.5% higher IPC under `Hi` voltage. Under the low voltage, the applications reveal different error tolerances and have large differences in performance. For example, `dijkstra` has about 20% performance loss for `Lo` voltage, but `gsm` has over 90% performance loss. This is obviously not a tolerable performance level, but it serves as a useful illustration of extreme behavior. With the dynamic control mechanism, we can limit the average performance loss to 10%. In general, we can achieve significant improvement in performance penalty through our technique.

We compare the normalized power savings for `Lo` VDD, `Hi` VDD and `Dynamic` control in Figure 5 (bottom). For fixed voltage, error prediction and padding can gain more power savings by scaling beyond the critical voltage and masking the predicted timing errors. This is especially true for `Lo` voltage while `Hi` voltage does not provide much potential for scaling. There are two particularly interesting benchmarks, `gsm` and `jpeg`, which have negative power savings, due to a large number of timing errors. In this case, our predictor still helps by masking the errors and helps eliminate negative power savings. Dynamic control limits the performance loss by adapting voltage periodically so we can avoid the overwhelming performance penalty and power overhead for `gsm` and `jpeg` with fixed operating voltages. In this case, instead of scaling to 77% of nominal voltage on average with Razor, we can scale to 58% of nominal voltage with average power saving of 43.6% and 6.9% performance loss compared to the baseline architecture. This corresponds to 26.3% additional power savings compared to Razor and 2.7% less performance penalty.

5 Conclusion

In this work, we proposed *error prediction* and *error padding* to reshape the effective error rate curve by predicting and isolating timing errors using locality information and pipeline scheduling. By anticipating timing errors we can avoid the full cost of timing error recovery. With our dynamic control mechanism, we can scale to the 58% of nominal operating voltage with average power saving of 43.6% and 6.9% performance loss.

References

[1] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using m5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2003.

[2] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, Jan. 1999.

[3] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *The 36th International Symposium on Microarchitecture (MICRO-36)*, 2003.

[4] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. BlueShift: Designing Processors for Timing Speculation from the Ground Up. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009*, pages 213–224, 2009.

[5] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 122–31, June 1998.

[6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th annual Workshop on Workload Characterization*, 2001.

[7] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008*, 2008.

[8] X. Liang, R. Canal, G. Wei, and D. Brooks. Process variation tolerant 3T1D-based cache architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[9] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996.

[10] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou. Yield-aware cache architectures. In *roceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, December 2006.

[11] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. EVAL: Utilizing processors with variation-induced timing errors. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture-Volume*, 2008.

[12] P. Shivakumar and N. P. Jouppi. CACTI 3.0:An Integrated Cache Timing, Power, and Area Model. In *WRL Research Report 2001/2, Compaq Western Research Laboratory*, 2001.