Identifying and Predicting Timing-Critical Instructions to Boost Timing Speculation

Jing Xin and Russ Joseph
Department of EECS
Northwestern University
jingxin2011@u.northwestern.edu rjoseph@eecs.northwestern.edu

ABSTRACT

Circuit-level timing speculation has been proposed as a technique to reduce dependence on design margins and eliminating power/performance overheads. Recent work has proposed microarchitectural methods to dynamically detect and recover from timing errors in processor logic. To a large extent existing work has relied on statistical error models and has not evaluated potential disparity of error rates at the level of static instructions. In this paper, we analyze gate-level hardware models for an execution pipeline and demonstrate pronounced locality in instruction-level error rates due to value locality and data dependences. We propose timing error prediction to dynamically anticipate timing errors at the instruction-level and error padding techniques to avoid the full recovery cost of timing errors. We show that with simple prediction strategies our mechanism can reduce 80% of the performance penalty incurred by error recovery on average. This allows us to alleviate some limitations of timing speculation and improves energy-efficiency by 21% when compared to baseline timing speculation techniques using the same dynamic adaptive tuning mechanism.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: Processor Architectures; B.8 [**Hardware**]: Performance and Reliability

General Terms

Design, performance

Keywords

Pipeline, timing speculation, power management

1 Introduction

The traditional CMOS design paradigm applies a worst-case design methodology assuming that all circuit paths must obey timing constraints. Conventional optimizations speed up critical paths through gate sizing and v_t selection to improve frequency while sacrificing power. Often optimizations turn a blind eye to path activity because under the design paradigm all critical paths are of equal importance – any one of them could produce a timing violation. Consequently, the impact on the power budget for accelerating an infrequently used long path might be the same as a commonly used critical path. Furthermore, designers also introduce guardbands to overcome circuit-level sensitivities to environmental temperature, supply voltage noise, and process variation, which also add a non-negligible energy overhead to the system.

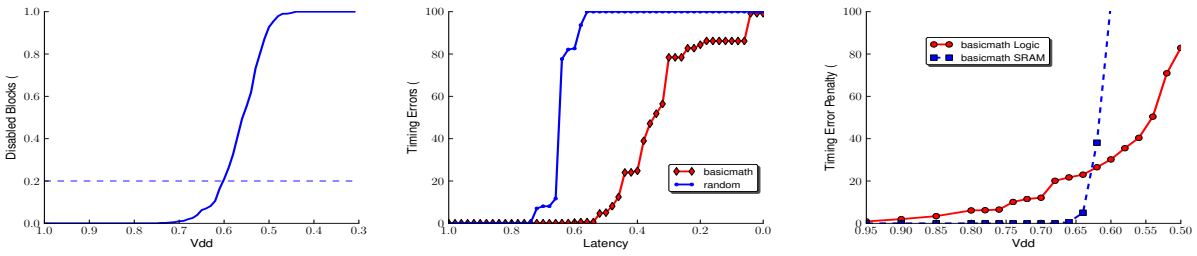
In recent years, researchers have shown a growing interest in circuit-level timing speculation [7] as a means to garner more performance from and boost energy-efficiency of the system. Under a *better-than-worst-case (BTWC)* design paradigm, designers reduce safety margins and relax set-up time constraints with the expectation that timing errors will occasionally appear in the system [3,7]. The designer includes detection and recovery mechanisms so that errors can be dynamically flagged. The processor initiates a recovery mechanism to prevent the errors from affecting architecturally visible state. Execution is allowed to resume once recovery is complete. 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 [35] or design time optimizations of logic [13, 21].

Much of existing work assumes timing errors are strong functions of voltage/thermal conditions but independent of dataflow within the program or true gate-level transitions within the pipeline logic. We believe this to be a missed opportunity because dynamic instruction sequence and program values can have a dramatic impact on timing error rate. Recent work supports this notion. Li *et al.* [23] showed that trace based analysis can successfully diagnose permanent hardware stuck at faults within a pipeline, hinting at important reproducibility at the instruction level which we believe would also be true with timing faults. More recently, Hoang *et al.* [10] demonstrated that some very simple code transformations can have significant impact on reducing timing error rates. In related work, Sartori and Kumar showed that various compiler optimizations can impact error rates [36]. While these efforts show a connection between static code sequences and illustrate that code generation can have an impact on circuit-level timing issues, they do not directly examine dynamic timing error patterns nor suggest how hardware might identify links between timing errors and dynamic code sequences.

In this work, we advocate a closer examination of how in-

This work was supported by NSF CAREER CCF-0644332 and NSF CSR-0720820.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil



(a) SRAM (b) Logic (c) SRAM v.s. Logic

Figure 1: Timing Speculation in SRAM and Logic

dividual instruction sequences stimulate timing paths in execution logic. We perform gate-level simulations to explore *timing error locality* in static instructions whose data usage patterns sensitize delay fault paths. We then evaluate simple mechanisms that predict instructions likely to produce timing errors in the pipeline and make preparations to reduce the recovery cost. This allows the processor to operate at more aggressively scaled voltages and hence consume significantly less power. This paper makes the following primary contributions:

- We introduce the concept of *timing error locality* in which static instructions exhibit consistent error generation patterns over a period of program execution.
- We explore a wide range of timing error prediction strategies to efficiently identify timing-critical instructions (instructions that tend to have timing errors) based on timing error locality.
- We propose an *error padding* mechanism to isolate timing errors from the rest of the processor pipeline. This reduces the number of timing errors that require full pipeline error recovery.

Our results show that, error prediction based on timing error locality can effectively identify more than 80% of the timing-critical instructions. By combining prediction with an *error padding* mechanism, our techniques improve power efficiency by 21% compared to baseline timing speculation using the same adaptive tuning mechanism.

This paper is organized as follows: Section 2 discusses challenges in timing speculative architectures which motivate this work, and Section 3 introduces timing error locality in execution pipelines. Section 4 describes dynamic timing error prediction and error padding mechanisms, and Section 5 discusses three adaptive voltage tuning algorithms to maximize energy efficiency for our error padding system. Section 6 describes our simulation setup, and Section 7 evaluates the efficiency of our predictors as well as tuning algorithms. Section 8 describes our work with relation to other research, and finally Section 9 concludes.

2 Background and Motivation

While timing speculation has been shown to be an effective means to improve energy-efficiency or performance in the face of manufacturing and environmental variation, there are still a number of obstacles that may prevent it from achieving its full potential. For memory dominated structures, the SRAM array lookups have significantly more regularity and allow for dynamic partitioning and resizing which may allow them to improve performance in an energy-efficient manner by sacrificing capacity [30]. There has been a significant amount of work in adapting SRAM structures for use in both low-voltage modes and severe parameter variation [25, 30, 31]. In contrast, logic dominated structures in the processor backend, most notably execution units, would be difficult to accelerate without significant impact to the power budget. While there has been some recent work in logic optimization for timing speculative designs [1, 13], speeding up critical logic paths often applies aggressive gate sizing and v_t assignment. This could incur significant energy costs in complex execution units which contain a large number of critical paths. Furthermore, many of the long critical paths in execution logic cannot be avoided through standard manipulations like superpipelining because they are essential to important performance loops [6].

Figure 1 shows timing speculation for a representative SRAM structure (cache) and logic structure (execution unit). Figure 1(a) shows how cache blocks can be disabled to permit voltage scaling for a 32KB cache with 64B block size. We can scale to 60% nominal voltage with about 20% blocks disabled. Figure 1(b) shows how the timing error rates increase when we reduce clock latency. We studied execution unit while running `basicmath`(MiBench [16]) and a random sequence of inputs. Because timing error rates are heavily dependent on input sequences, different applications may exercise critical paths to varying degrees. This figure shows real applications provide opportunities for further scaling. Figure 1(c) compares the effects of applying dynamic voltage scaling (DVS) to different component groups in a timing speculative processor design running `basicmath`. The timing error penalty is calculated by comparing the performance with an oracle processor with no timing errors at all. The figure shows that the error recovery penalty of logic dominates when VDD is kept near to the nominal value. It is only at very aggressive scaling points that SRAM timing errors overtake logic. While the logic curves feature have non-zero error values just below the nominal VDD, they have rather gentle slopes and may permit voltage scaling over a rather wide range. The slopes of these curves are very consistent with validated analytic timing error models and empirical data [7, 40]. There are some clear implications for timing speculative designs. First, logic timing errors are likely to be encountered shortly after the supply voltage is scaled, even though they may not be initially prohibitive. Second, once SRAM timing errors begin to appear, they quickly become prominent and will likely prevent further voltage scaling. We focus on alleviating the impact of logic timing errors because they are likely to be the first obstacle under aggressive voltage scaling.

3 Timing Error Locality in Execution Pipelines

In this section, we examine locality properties of timing error rates by simulating execution logic at the gate-level. We specifically evaluate timing error rates for individual static

System	Logic	Benchmark	Latency (%)	Error Rate (%)	Static Insts	Num Critical Insts For			
						50%	90%	99%	100% errors
Alpha	ALU	basicmath	75.1	9.8	533	11	33	49	66
		dijkstra	63.9	9.4	154	6	12	15	20
	Shiftbr	basicmath	83.5	9.9	371	4	17	26	29
		dijkstra	81.9	9.4	163	8	16	22	24
OpenRISC	ALU	basicmath	68.1	9.2	4571	2	4	56	104
		dijkstra	68.1	9.8	2808	1	2	16	51
	LSU	basicmath	56.2	8.5	3901	54	247	644	765
		dijkstra	57.3	8.6	2199	34	149	550	563

Table 1: Analysis of Critical Instructions for Alpha and OpenRISC

instructions and show that most timing errors have temporal locality during dynamic execution. As we will show, we can exploit this locality to hide timing errors.

3.1 Error Rates for Static Instructions

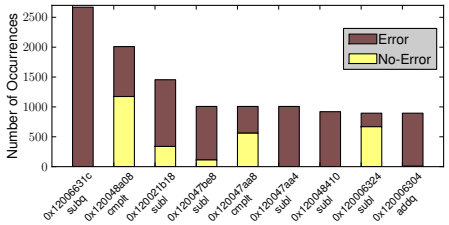
The degree to which timing criticality occurs depends primarily on sequencing of opcodes and operands. Timing faults in combinational logic are well known to be a function of an initialization vector which determines starting values for internal nodes and a sensitizing vector which causes the revealing transitions. Within the execution logic, timing errors manifest when the instruction opcodes and operands produce such a set of critical input vectors. Control flow, dependence structure, and value locality can all influence the sequence of opcodes/operands and have an impact on error rate. Control flow and data dependencies determine the sequencing of operations given to the hardware, while the degree of value locality determines which set of operands are used by the instructions. Previous studies have shown that many static instructions use a relatively narrow set of values [28]. The degree of overlap that these values have with timing critical vectors significantly impacts the error rate. If an instruction's frequently used values belong to the set of critical input vectors, then this instruction is likely to have high error rates.

To understand the relationship between applications and the timing faults they generate, we conduct a trace-driven study of timing error rates using detailed gate-level Verilog simulation of functional units as they execute instruction sequences from SPEC CPU [41] and MiBench [16] applications. We designed integer execution units (ALU and shifter-branch unit) for the Alpha ISA in Verilog, synthesized the design to target a 45nm standard cell library [18], and performed place and route to calculate interconnect delay. To explore timing error patterns across different platforms, we also studied execution units, namely the ALU and LSU-Effective-Address-Generator for the OpenRISC 1200 microprocessor [22]. We cross-compile the benchmarks for those two systems and picked representative instruction sequences to drive the gate-level models and produce timing error traces.

Table 1 summarizes the timing error behavior for two MiBench applications (`basicmath` and `dijkstra`). We model the impact of dynamically scaling the supply voltage to an operating point at which approximately 10% of the dynamic instructions produce timing violations in both benchmarks (we do not consider SRAM structures for this study). The table shows the effective execution latency under this operating point normalized to the clock period. The table also shows how many static instructions are needed to account for 50%, 90%, 99% and 100% of the dynamic timing errors. We can clearly see that a small number of static instructions are responsible for the overwhelming majority of timing errors. For instance, for the Alpha ALU, half of all the timing errors can be attributed to 11 and 6 static instructions respectively for `basicmath` and `dijkstra`. This is a common phenomena which we have observed in a wide range of benchmarks. We term this small set of problematic instructions *timing-critical instructions* in following discussion. For the ALU, tens of instructions would commonly be responsible for over 90% of the timing errors. For the LSU-Effective-Address-Generator in OpenRISC, the number of critical instructions is larger because the effective address generator hardware is smaller and simpler than a full ALU. We need to scale to significantly lower operating points to achieve the target 10% error rate. At this point, many paths begin to fail.

To take a closer look at the timing-critical instructions and their behavior, we analyzed `basicmath` in detail in Figure 2. Figure 2(a) shows the nine most frequently executed static instructions with non-zero timing errors rates. Among the critical instructions 4/9 of them have almost 100% error rate. A common characteristic of these four instructions is that they have good value locality and their values frequently exercise critical paths. In other cases where an instruction has poor value locality, it may generate timing errors more sporadically as the operands intersect and diverge from the the critical timing vectors. Figure 2(b) provides a code snippet with 2 timing-critical instructions labeled E and F. Figure 2(c) shows the behavior of these two selected instructions. These instructions appear within the same basic block of the `printf_fp` function and perform basic arithmetic and comparison. Although they have identical execution counts, they have very different error rates. Figure 2(c) details the minimum required computational latencies for these instruction over execution sequences 200 to 400. We observe that instruction `E` tends to exercise a set of operands that happen to stress the critical paths more than instruction `F`. The structure of `E`'s critical path is such that it is highly independent of the circuit state generated by the previous instruction. It therefore exhibits a predictable and fairly constant latency. For Instruction `F`, the stability of timing error rate is highly dependent on the operating point (DVS scaling). When the supply voltage allows an operational latency around 70% of nominal (somewhere between the extremes) this instruction will alternate between producing and not producing a timing error. Notice that when viewed from a coarse enough granularity (i.e. sufficiently large windows), this instruction will still have a stable average error rate.

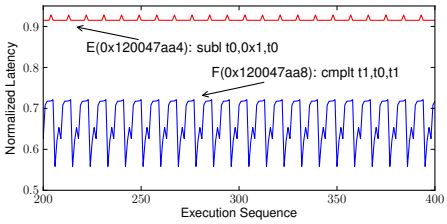
Our analysis revealed stability and predictability of error rates over time. Specifically, over relatively small time scales, error rates for static instructions exhibit pronounced locality. Instructions which are not in the timing critical set, have a zero or near zero error rate and hence these error rates are trivially stable. However, instructions which have moderate to high error rates or generate irregular timing error patterns also have relatively stable error rates. This is due to averaging over many samples.



(a) Error Rates of Timing-Critical Insts

A	120047a94:	lda s0,-24(s0)
B	120047a98:	ldl t0,16(s0)
C	120047a9c:	addq s3,t0,t0
D	120047aa0:	ldl t1,96(fp)
E	120047aa4:	subl t0,0x1,t0
F	120047aa8:	cmplt t1,t0,t1
G	120047aac:	bne <printf_fp+0xf2c>

(b) Code Snippet with 2 Critical Insts



(c) Latency for 2 Selected Insts

Figure 2: Analysis of Timing-Critical Insts for `basicmath`

3.2 Parameter Variation and Its Affect on Locality

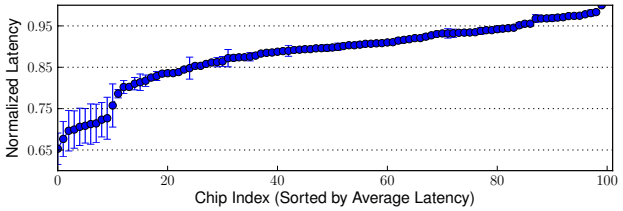
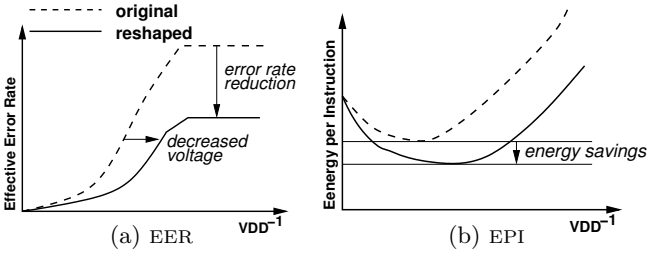


Figure 3: Latency Distribution for 100 Chips for the Selected Instruction (`0x12006631C`) in `basicmath`

Parameter variation alters the error rate curves and membership in the criticality set, but in general have little impact on the size of the set. Timing-critical sets should be consistently small relative to the instruction working set for a wide range of applications over several different generations of our execution hardware model whenever total error rates are small.

To examine the effect of parameter variation, we generate 100 instances of execution units using a random and systematic parameter variation model [8]. Then we run the same `basicmath` simulation and record the latency of one most frequently executed critical instruction (PC=`0x12006631C`). We show the mean and standard deviation for all 100 samples in Figure 3. We sort samples by mean error rates. Parameter variation may affect path lengths, causing one near critical path to become a true critical path in one chip, or speeding it up so that it is less critical in another chip. However, this will not change the timing locality and predictability of timing errors for most hardware instances.

4 Dynamically Predicting Timing Errors



(a) EER (b) EPI

Figure 4: Reshaped Effective Error Rate and Energy per Instruction

By aggressively scaling the supply voltage, the number of timing critical logic paths increases. Scaling below the critical voltage causes the error rate to spike and would lead to massive slowdown because the aggregate recovery time is too large to tolerate. Figure 4 conceptually shows the trend of *effective error rate* (EER, errors which need to be recovered) and *energy per instruction* (EPI) according to the decrease of VDD (dashed lines).

We propose a novel approach to reshape the EER and EPI curve by efficiently identifying instructions that are likely to incur timing errors and isolating those errors through prediction and pipeline scheduling. Figure 4(a) shows our technique can either operate at a lower voltage for a specified EER or decrease the EER for a specified operating VDD; Figure 4(b) shows how our technique can aid timing speculation by further improving the EPI. This is achieved through both reducing the recovery penalty and operating at lower VDD.

In this section, we describe the mechanism of error prediction and *error padding*. To explore the potential of identifying instructions that tend to incur timing errors (items in the critical instruction set), we examined a series of prediction strategies. We then show how we can efficiently mask the potential timing errors through pipeline scheduling.

4.1 Prediction Strategies

We adopt some frequently-used and easy-to-implement prediction strategies to detect likely timing errors [9, 27, 29, 38]. These predictors apply the principle of temporal locality for timing errors. Instructions which have produced errors in the recent past are likely to produce errors in the future.

- **Last Value (Last)** A table of one-bit values indexed by the low order address bits of the PC. When a timing error occurs, the corresponding bit in the table is set. Otherwise the table entry is cleared.
- **Miss Distance Counter (MDC)** A table of PC indexed, resetting, zero-saturating counters. If an instruction commits without generating a timing error, the counter is decremented. For instructions that commit an error, the counter is reset to all ones. A timing error is predicted any time the counter contains a non-zero value. We evaluate two, three, and four bits for the counters in our study (`MDC2`, `MDC3` and `MDC4`).
- **Saturating Counter (Sat)** This a standard bimodal predictor as typically used in branch prediction, we evaluate only two-bit counters for this strategy.
- **Most Recent Entry (MRE)** A table of recently executed critical instructions are maintained in a cache-like structure tagged by their PC addresses. Each tag is associated with a two bit biased saturating counter. Non-zero values indicate timing critical status. We apply a Least Recently Used (LRU) replacement policy when the number of critical instructions (working set size) is greater than the table size.

4.2 Hardware Implementation and Cost

The `Last`, `MDC` and `Sat` can be implemented with non-tagged entries. They can be readily adapted into a directional branch predictor with some potential for shared hardware resources (e.g. the decoder). However, `MRE` is a tagged buffer

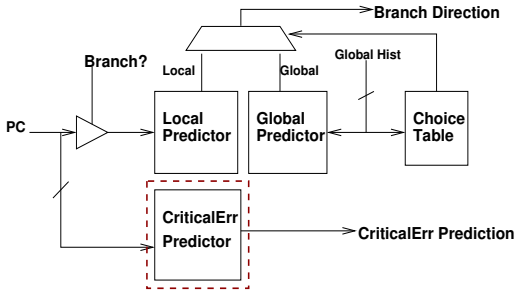


Figure 5: Critical Instruction Predictor Integrated into Branch Predictor

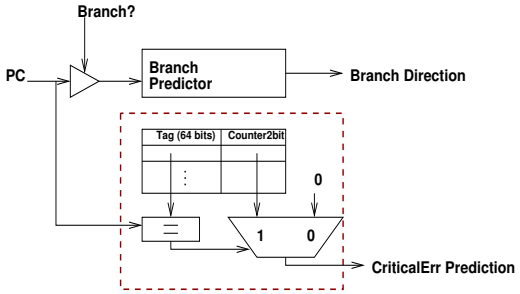


Figure 6: Tagged Critical Instruction Predictor

structure and would have to be implemented as a stand-alone structure. We will compare the performance for a wide range of sizes for these predictor strategies in Section 7.1. All of the prediction strategies involve SRAM structures to hold instruction error rate history. We refer to these SRAM structures as *timing criticality counter tables* in following sections.

Figure 5 shows a generic non-tagged predictor integrated with a standard two-level branch predictor. Since we can share the decoding logic with local branch predictor, the hardware cost of this scheme is limited to the bits used for storing the counters (i.e. $2048 \times 2 \simeq 4K$ bits for 2048-entry 2-bit counters) and modifications to the output path (e.g. sense amps). The effective hardware cost of this prediction mechanism would be on par with the local history table of the branch predictor in our baseline design. A previous study on the energy-efficiency of branch predictors [32] shows that the hardware should increase the total processor power by less than 2%. We include this overhead in our final results.

Figure 6 shows the design of a stand-alone `MRE` predictor. It is not constrained by the structure of existing components in the processor but cannot reuse any hardware. To support `MRE`, we need to store the PC addresses for the critical instructions in the predictor. The hardware cost for a 32-entry predictor is approximately $2K$ bits plus the logic needed for decoding, tag comparison, and replacement. The use of PC tags adds to the overall hardware costs and hence limits the practical size of the predictor.

4.3 Masking Pipeline Timing Errors

After identifying instructions that are likely to produce timing errors, we can perform a technique which we call *error padding*. Figure 7 shows timing error recovery in a eight stage pipeline. 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 8) by anticipating timing errors at the `ID` stage. We assume that all the instruction executions can finish if we extend a cycle. This is realistic since the supply voltage is never scaled aggressively enough to make this untrue. Since the prediction and update are not on the critical path, we assume there are no direct adverse performance effects due to predictor access.

Figure 9 demonstrates how the error prediction and *error padding* can be integrated into an eight stage pipeline. The prediction is made during the instruction decode (`ID`) stage. If the instruction is flagged as critical, the scheduling logic expects a one-cycle stall during execution. We do not anticipate that this would add much complexity to pipeline scheduling and control logic since we are essentially adding hazard detection similar to existing mechanisms used to implement load-use interlock. When this instruction advances

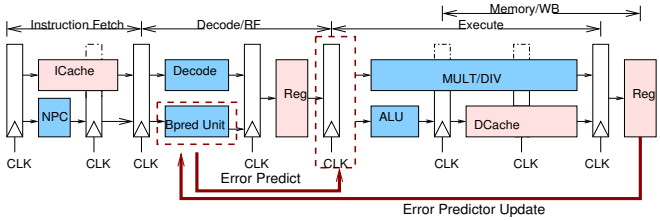


Figure 9: Simplified Logical Pipeline Structure Used to Demonstrate Error Padding

to an execution stage, the upstream pipeline is stalled for a cycle regardless of whether there is a true timing error or not. In the event that there is a timing error, the stall limits the recovery penalty since it effectively prevents any errant values from being forwarded to upstream instructions. If no timing error occurs then the performance is impacted by the unnecessary single cycle stall, but there are no other effects.

During the writeback (`WB`) stage, the instructions that have (or could have) generated timing error during their execution are noted by the conventional timing speculation logic. The writeback logic updates the timing criticality predictor during the retirement of the instruction.

4.4 Predictor Performance Evaluation Metrics

For our critical instruction predictor, we incur the full cost of *error padding* once a instruction is marked as critical 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 (*false positive penalty*). 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-like pipeline. The recovery cost associated with misdiagnosing critical instructions as non-critical is the *false negative penalty*.

To evaluate the potential benefit of the timing error predictors, we adopt language and metrics used in evaluating binary classifiers and diagnostic tests. Confidence estimators based on the concept of binary classification have been applied in branch speculation control [14]. Here we examine classes of prediction schemes that identify instructions which are likely to produce timing errors. Relevant metrics for this type of study include:

- **Sensitivity**: number of correctly predicted timing errors relative to the number of dynamic instructions identified as critical.
- **Specificity**: number of correctly predicted non faulting dynamic instructions relative to the number of dynamic instructions which do not produce timing errors.

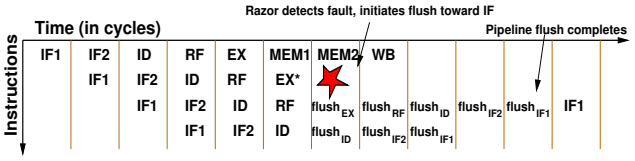


Figure 7: Pipeline Timing for Error Recovery (Razor [7])

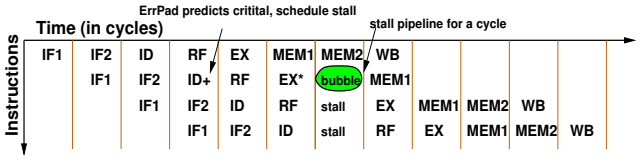


Figure 8: Pipeline Timing for a Correctly Predicted Timing Error

- **Positive Predictive Value (PPV)**: number of correctly predicted timing errors relative to the number of critical predictions.

- **Negative Predictive Value (NPV)**: number of correctly predicted non faulting dynamic instructions relative to the number of non critical predictions.

Predictors with high *Sensitivity* will detect most timing errors; predictors with high *Specificity* will correctly identify most non-critical dynamic instructions; predictors with high PPV will have trust-able critical predictions; predictors with high NPV will have trust-able non-critical predictions. Since most of the instructions should be non-critical under the error rate ranges relevant for our study, *Specificity* and NPV are not as useful; their values could be high even if we do not predict any timing errors. In following sections, we will focus on *Sensitivity* and PPV to evaluate the performance and efficiency of predictors.

5 Dynamic Voltage Tuning and Optimization

The frequency of timing errors depends partially on program phase and consequently, the optimal operating voltage changes during program execution. To maximize the benefits of timing speculation, dynamic adaption mechanisms must be used to actively tune the voltage during program phase changes. Previous work has studied dynamic voltage scaling (DVS) and dynamic voltage frequency scaling (DVFS) on different tasks/applications [11] or under process variation [15, 26, 35]. Our goal for dynamic tuning is to find the optimal operating voltage with low power consumption while sacrificing minimum performance penalty. This is made more challenging by error padding since the optimal operating voltage will depend on many factors including the raw error rate and prediction accuracy. Instead of targeting some small, fixed error rate [7], we are trying to find the operating point with minimum "energy-delay". We examined three basic tuning mechanisms:

Hill Climbing [33] is a general mathematical optimization technique to search for some minimum (or maximum) target. The controller starts from a relatively high VDD and scales down. At the end of each interval it compares the current energy-delay with that of previous interval, and decides to scale up, down or retain its VDD. This mechanism is easy to implement and reacts fast. It is however, important to correctly tune the basic parameters (control step, interval size, etc.) because the controller could easily fall into oscillation, or adapt too slowly to reach dynamically changing optimal levels.

Sampling divides the execution to "sampling" and "running" periods. The controller sets some predefined voltages during the "sampling" period, and selects the one with the best energy-delay as operating voltage for the "running" period. This method can be combined with dynamic phase detection to automatically quit the running period and redo sampling when there is a phase change. Sampling is easy to implement as well, but exploration during the sampling period may lower the efficiency of the system.

Fuzzy Controller [35] approaches have been used to perform dynamic optimization of complex systems. The controller does some offline training to learn rules about optimal voltages for different applications, and relies on the rules to make choices. The training and rule generation should be performed periodically. This mechanism is more complex to implement and requires support from system software. In addition, the performance of the fuzzy controller is highly dependent on the quality of the training algorithm.

We discuss more implementation details for the above mechanisms in Section 6. We compare the performance for a wide range of sizes for these predictor strategies in Section 7.3.

6 Experimental Setup

Processor Parameters	
Frequency	1.0 GHz
Feature Size	45nm
Nominal VDD	1.0v 10% guardband
Pipeline Issue Width Functional Units	8 stages 2 inst, in-order 2 IntALU units 1 Shift/Branch unit 1 Ld/St Port 1 IntMult/Div
IntALU Shifter/Branch	2857 gates 2160 gates
Branch Predictor	Tournament 2-bit saturating counter 2048-entry local history 8192-entry global history 8192-entry choice table
L1 Inst Cache	32KB 8-way 64B blocks
L1 Data Cache	32KB 8-way 64B blocks

Table 2: Processor Parameters

Dynamic Tuning Mechanism	
error recovery penalty	8 cycles
error padding cost	1 cycle
Vdd Regulator Delay	0.5μm
Predictor Power Penalty	2%
Tuning granularity	2%
Lowerbound VDD	60%
Performance Tolerance Limit	10%
Control interval	100K insts
Fast Forward	100 million insts
Short Simulation	10 million insts
Long Simulation	100 million insts
Hill Climbing	threshold 1%
Sampling	10 sample intervals 90 running intervals
Fuzzy Controller	14 rules training 1200 intervals

Table 3: Simulation Parameters

Our evaluation focuses on the impact that timing error

prediction and error padding have on a high-performance, low-power embedded processor. We extend the `M5` system simulator [5] to model an eight stage in-order Alpha core comparable to recent Intel Atom designs [17]. Table 2 lists the microarchitectural configuration of the baseline processor and simulation configurations.

We model timing errors in the execution pipeline by constructing a detailed gate-level model of logic in integer functional units. Specifically, we create Verilog models for Alpha ALU and Shifter/Branch Unit execution hardware. The models are complete enough to support virtually all user-mode integer instructions with the noted exclusion of the Alpha's MVI SIMD instructions [39]. We synthesize the execution hardware using `Synopsys Design Compiler [42]` and target the `Nangate` 45nm standard cell library [18]. To account for the circuit level impact of interconnect, we preform place and route using `Timberwolf`([37]) and then extract wire delays. We annotate the ALU and Shifter/Branch designs with the wire and gate loading delays. After constructing this detailed gate-level model, we also implement a gate-level timing simulator in C++ and embed it into the `M5` simulator [5]. On a per cycle basis, we drive the gate-level simulator with program data values, allowing us to model the coupling between program control/data flow patterns and timing errors in the execution units. Finally, we model delay due to dynamic voltage scaling by applying the Alpha Power Law [34].

Structure	Type	Repair Method
I-Cache	Memory	Resize
BranchPred	Memory	Razor
RegFile	Memory	Razor
IntALU	Logic	Razor
ShiftBr	Logic	Razor
Mult/Div	Logic	Razor
D-Cache	Memory	Resize

Table 4: Structures Modeled for Simulation

Our processor adopts Razor error detection and recovery to apply circuit-level timing speculation [7]. The penalty to recover from a timing error in an execution unit is eight cycles. To model timing errors for SRAM structures, we apply the parameter variation models in [20]. Due to overheads we do not use Razor to recover from timing errors in large SRAM structures. We instead implement resizing methods to trade off the capacity for energy savings for I-Cache and D-Cache [30]. We apply simple block-level resizing where blocks are selectively mapped out in the I-Cache and D-Cache. For smaller SRAM structures like the register file and the branch predictor, we assume use of Razor [7] to tolerate timing errors. Table 4 shows the representative structures we modeled for the pipeline and their corresponding timing speculation methods. We call the processor implemented with Razor error-resilience and cache resizing *Razor-Resize* mechanism. This will be used as a baseline comparison for error prediction and *error padding*.

We choose to focus on logic dominated structures in the processor back-end and do not directly address logic in the processor front-end for two reasons. First, in power efficient processors with low-issue widths and in-order execution, a large number of critical paths are found in the execution hardware [13]. Second, counterflow pipeline recovery used in Razor-like designs is sensitive to the pipeline depth of timing errors. Front-end timing errors would result in fewer instructions being squashed and would incur lower recovery penalties on average.

For the dynamic tuning mechanisms introduced in Section 5, we assume hardware implementation cost is negligible. We model a voltage regulator which has a VDD step size of 20mV per 0.5μs. The parameters of the optimization mechanisms are shown in Table 3. For *Hill Climbing*, we do a voltage adjustment of one step up or down every interval depending on the change in energy-delay ($\Delta_{Energy-Delay}$); for *Sampling*, we use 10 intervals to do the sampling (start from middle value of VDD, and either increase voltage for 10 steps or decrease voltage for 10 steps depending on the performance penalty for current interval. Sampling will also stop if the performance penalty reaches the predefined limit), and the controller will choose the VDD with the best energy-delay for the next 90 intervals. For *Fuzzy Controller*, we run 1200 intervals to do the training and then formulate 14 rules to predict the voltage according to its cycle count and timing error stats. We assume the training is performed offline and will not affect performance.

Benchmark	Description
SPEC CPU2000 CINT	
gcc	C programming language compiler
gzip	Compression
mcf	Combinatorial optimization
parser	Word processing
MiBench Embedded	
basicmath	Simple mathematical calculations
bitcount	Bit counting functions
blowfish	A keyed, symmetric block cipher
dijkstra	Dijkstra's algorithm
FFT	Fast Fourier Transform
CRC32	Cyclic Redundancy Check
patricia	Trie implementation
susan	Smallest Univalue Segment Assimilating Nucleus

Table 5: Simulated Applications and Their Descriptions

We select four integer benchmarks from the SPEC CPU2000 suite [41] and eight benchmarks from MiBench [16]. The description of the benchmarks are shown in Table 5. All applications are compiled for the Alpha architecture using the `gcc` tool-chain with high optimization levels. We believe that these benchmarks are a good reflection of the workload for a general purpose low-power processor.

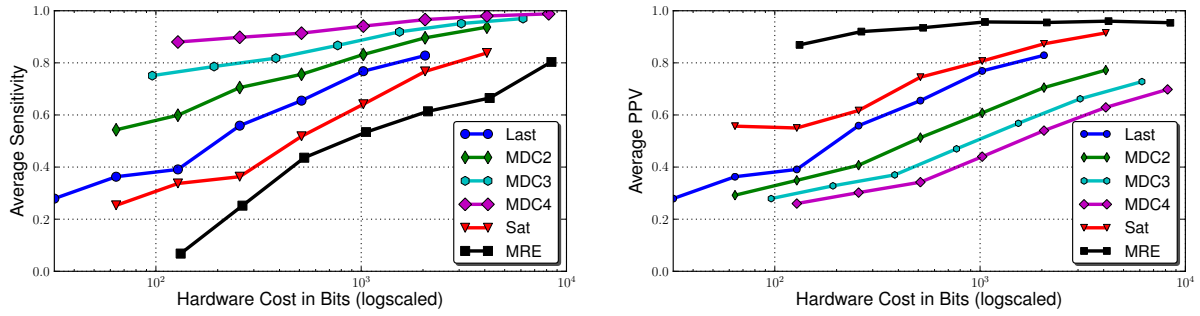
7 Results

In this section, we will provide detailed comparison of the predictor strategies introduced in Section 4.1, followed by the efficiency evaluation of error prediction and error padding. We compare the three widely used dynamic adaption algorithms discussed in Section 5. We also examine the impact of parameter variation on this approach.

7.1 Design Space for Timing Error Predictors

To understand how basic design parameters for timing error predictors affect prediction accuracy, we explore `Last`, `MDC2`, `MDC3`, `MDC4` and `Sat` predictors with size ranging from 32 to 2048 entries increasing by powers of two. Since the `MRE` predictor needs to maintain PC addresses tags and hence has higher per entry overhead, we evaluate sizes from 2 to 128 entries.

We collected instruction traces for each benchmark and chose an operating point with ≈10% error rate to ensure that



(a) *Sensitivity* (b) *Positive Predictive Value (PPV)*

Figure 10: Performance Comparison of Predictor Strategies as Function of Hardware Cost

the timing error rate is significant. For this experiment, we perform gate-level timing simulation on just the execution units. Figure 10 shows the predictor accuracy represented by average *Sensitivity* and Positive Predictive Value (*PPV*) across all the benchmarks. To get a reasonable comparison of predictor performance versus hardware cost, we plot the performance versus hardware implementation cost in bits (log-scaled).

From the figure we can see that not surprisingly, as the size of the predictor increases, the *Sensitivity* and *PPV* of each predictor type increases. However, after the predictor sizes reach some point (e.g., $2K$ bits), the prediction accuracy begins to saturate. From Figure 10(a) we can see for the same hardware cost, `MDC4` gives best accuracy in *Sensitivity*. This is because `MDC4` is the most cautious predictor, it will predict an instruction to be timing-critical if it produced a timing error in any of its 15 most recent executions. The tradeoff of this pessimistic prediction mechanism is the increased number of *false positives*. As shown in Figure 10(b), `MDC4` has the lowest *PPV* accuracy across all the predictors.

For all of the untagged predictors, the predictor table is indexed by lower portion of the PC address. There will be conflicts if two PC addresses fall into the same table index. This is especially pertinent if one of them is always critical and the other is not; the predictor may perform poorly on both of them. For the tagged predictor (`MRE`), we can safely avoid aliasing conflicts. Due to the limited table size it cannot maintain all the critical instructions in the working set, so `MRE` has the lowest *Sensitivity* as shown in Figure 10(a).

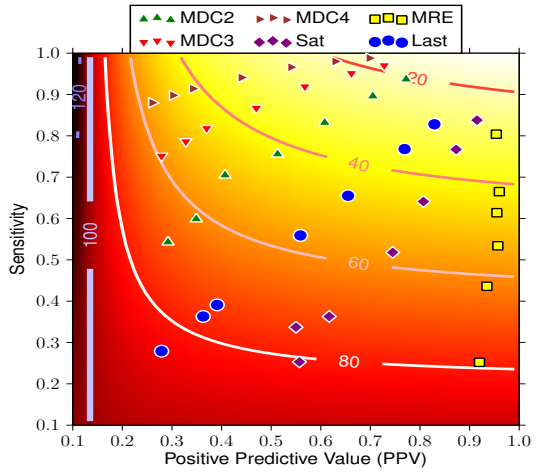


Figure 11: Penalty Analysis for Different Predictors. The lines show the performance penalty of our techniques compared to the performance penalty of Razor-like recovery.

Since *false positive* prediction penalty (an extra cycle added for *error padding*) is less than the *false negative* prediction penalty (pipeline flush involved for recovery), it is important to know the overall performance for those prediction strategies. Recall that in our implementation, the penalty of pipeline flush is eight cycles and the penalty of error padding is one cycle. We combine *Sensitivity* and *PPV* of all predictor instances and project their timing error penalty (in CPI) compared to standard Razor-like recovery in Figure 11. We assume a normalized CPI of one. In the figure, the lines show the timing error penalties (including both error recovery and error padding) compared to baseline Razor-like error recovery. The ideal case is the upper-right corner where we have 100% accuracy on *Sensitivity* and *PPV*, and thus each true timing error produces one cycle of stall. This effectively reduces the performance penalty to 12.5% of a baseline Razor design. The line of 100% shows where the penalty of error padding is equivalent to the penalty of the baseline Razor design. If the *PPV* accuracy keeps reducing, the penalty for error padding could be larger than standard Razor-like error recovery.

From the figure we can see a large number of the predictors have 40% or less performance penalty compared to the penalty of the baseline timing speculative design. `MDC4` gives best performance (with less than 20% penalty) for the largest size. This is because the penalty of *false positive* and *false negative* is asymmetric. The penalty of failing to identify a timing error is much larger than the penalty of misidentifying a non-critical instruction as critical (eight cycles versus one). Note that since `MRE` ensures high *PPV* accuracy, it stays in a good position with performance penalty less than the baseline.

Based on this study, we selected two candidate designs for detailed simulation: an 2048-entry `MDC2` predictor combined with branch predictor and an 32-entry `MRE` stand-alone predictor.

7.2 Fixed Voltage Study

To evaluate the efficiency of error predictors in applications independent of the voltage tuning, we set three fixed operating voltage conditions. We choose VDD to be 70%, 80% and 90% of nominal voltage (without margin), which we believe covers a relatively large voltage scaling range. The baseline comparison is a *Razor-Resize* system discussed in Section 6.

Figure 12 shows the *Sensitivity* of all the benchmarks (and average value in last column) we evaluated. For 70% VDD, `MDC2` can correctly identify 86.5% of the timing errors on average, while `MRE` can detect 78.2% of them. But for 90% VDD, `MRE` can identify 88.8% of the timing errors while `MDC2` can only detect *71.8%*. `MRE` is limited by the tagged predictor size.

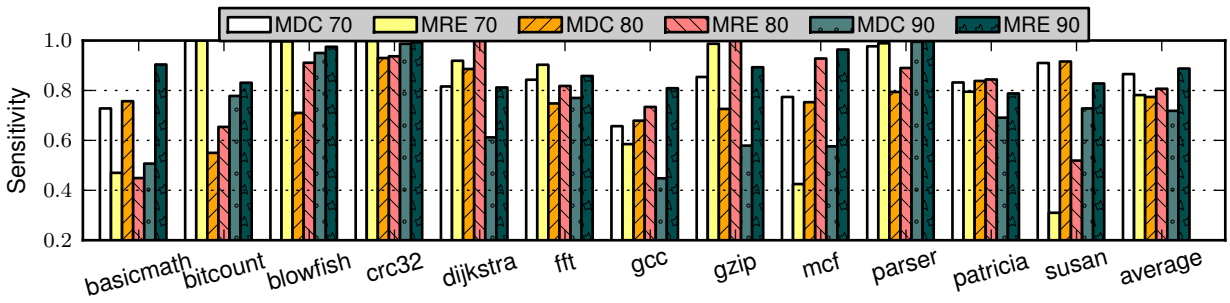


Figure 12: Predictor *Sensitivity* for Fixed Voltage Operation

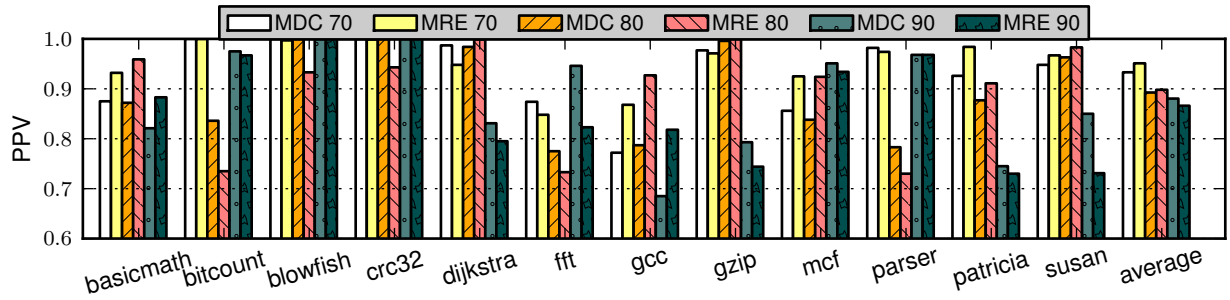


Figure 13: Predictor *PPV* for Fixed Voltage Operation

`MDC2` has a large table size, but it may suffer from conflicts in which multiple PC addresses interfere with each other and produce misleading predictions. However, both techniques show that on average over 70% of the timing errors are successfully identified. This suggests that error padding could effectively eliminate the recovery penalty.

Figure 13 shows the *PPV* results. Both techniques have more than 90% confidence on their timing error prediction for 70% VDD case on average. For some benchmarks like `fft`, there is lower accuracy because its error rates are extremely small. This makes it harder to successfully identify the errors.

We compare the performance (CPI) of our mechanism with *Razor-Resize* in Figure 14. The performance of `MRE` is similar to `MDC2`, so we just show the results of `MDC2` to focus our discussion. The performance penalty is normalized to a baseline design which operates at a fixed maximum voltage and has no timing errors at all. The last cluster shows that on average our mechanism can reduce 13.5% performance loss compared to the baseline design under 70% nominal VDD, and 6.6% under 80% nominal VDD. For 90% nominal VDD, since error rates are very small, the *Razor-Resize* and our mechanism have penalties below 2%. On average our mechanism can reduce the performance penalty caused by timing errors by over 80% compared to *Razor-Resize*.

Also Figure 14 shows that applications reveal different error tolerances and have large differences in performance under different operating voltages. For example, `bitcount` has almost no timing errors for 80% and 90% nominal VDD. Even for 70% nominal VDD, the performance penalty remains small. A counter example is `patricia` which has high timing error rate even for 90% nominal VDD. This indicates that dynamic voltage adaption is necessary to achieve best energy efficiency. We turn to this in the next section.

7.3 Tuning Error Rates

To examine the impact of error prediction and padding under adaptive voltage scaling, we implemented the *Hill Climbing*, *Sampling* and *Fuzzy Controller* tuning mechanisms and compare their energy-delay with a error-free baseline system operating at maximum voltage. The results of energy-delay are shown in Figure 15. Here we can see all the mechanisms achieve low energy-delay with average values less than 50% compared to system with no timing speculation. However, generally *Hill Climbing* cannot beat the other two mechanisms due to slow adaption and oscillation. *Sampling* gives promising results when one considers its relative simplicity.

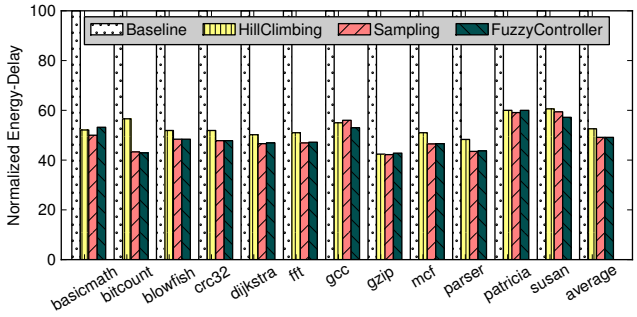


Figure 15: Comparison of Energy-Delay Optimization Algorithms

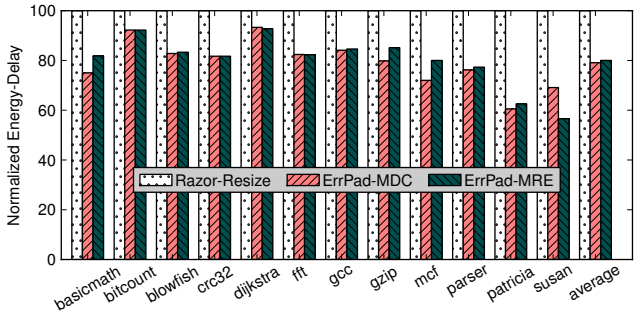


Figure 16: Comparison of Power Savings

To examine the performance of *error padding* to *Razor-Resize* we focus by comparing it with *Sampling*. We show

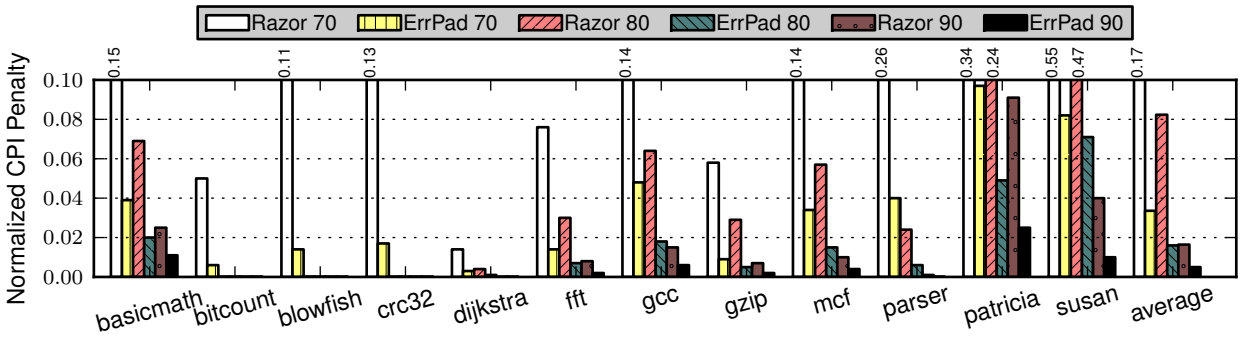


Figure 14: Comparison of CPI

their energy-delay results in Figure 16. We use the same tuning parameters for both *Razor-Resize* and our *error padding* system. Generally error prediction and padding can gain more power savings by scaling beyond the critical voltage and masking the predicted timing errors. Consider `gcc`, both of our predictors reduce energy-delay by approximately 21% compared to *Razor-Resize*. We note that `bitcount` does not generate many timing errors in execution stage (even when aggressively scaling VDD). As a result our method and *Razor-Resize* are comparable and the true bottleneck becomes the SRAM structures. Another interesting benchmark is `patricia`; this application has highly variable timing errors across a large range of PC addresses, so it is difficult to scale operating voltage down due to the large penalty of error recovery. However, our mechanism can reduce energy-delay by 40% by dynamically masking the timing errors.

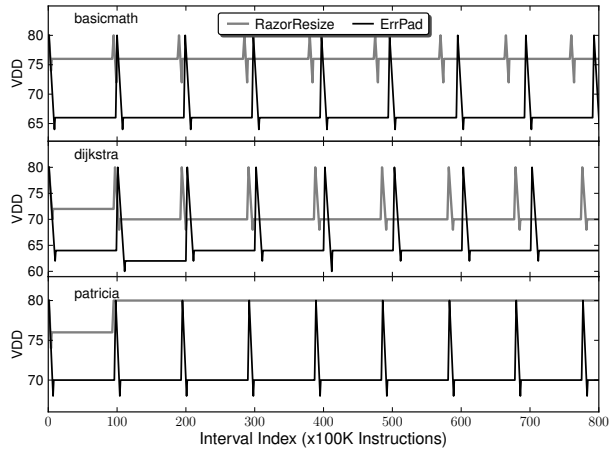


Figure 17: Operating Voltage of Dynamic Tuning According to Time Intervals

Figure 17 shows how the voltage is tuned according to each control interval for three representative benchmarks `basicmath`, `dijkstra` and `patricia`. For `basicmath` we can see our mechanism can sample a large range of voltages because the error padding effectively reduces the timing error penalty and allows the possibility of further scaling. It usually operates at 10% lower voltages with the same tolerance on performance penalty as *Razor-Resize*. For `dijkstra` both *Razor-Resize* and *error padding* can scale to some relatively low level voltage-levels so the effective difference is muted (∼5%). The most striking example is `patricia`. This yields a dramatic difference in operating voltages because this application has a large number of timing critical operations. Because these instructions are easily predicted with our method, we can operate at a much lower voltage level and gain more power-savings without adversely affecting performance.

7.4 The Impact of Parameter Variations

In Section 3.2 we discussed parameter variation and its relationship with timing error locality. To show the effect of parameter variation in a complete system, we generated 100 sample instances based on parameter variation models described in previous work [8,20]. We implement both *Razor-Resize* and *error padding* on these 100 sample chips and show their power-performance results for `basicmath` in Figure 18. The green line shows the "Energy-Efficient Frontier" which is the ideal case for best power-performance trade-offs [4]. From the figure we can see although the parameter variation changes the distribution of critical paths, it cannot change the timing error locality of applications. We can still apply *error padding* and achieve better energy efficiency by hiding the cost of timing error recovery.

8 Discussion

Recent work has proposed *better-than-worst-case* (BTWC) approaches to trade off error rate for power savings and/or performance improvement [2, 3, 7, 12]. The fundamental idea is to add modifications to the processor to dynamically detect and recover from timing errors, allowing more freedom in hardware implementation since worst-case design constraints can be relaxed. In particular, with timing speculation the processor can function at some small non-zero error rate with lower voltage or higher frequency than traditional worst-case designs. This can yield more efficient overall operation than the traditional design methodologies.

Critical Voltage Wall

Timing speculation faces the limitations of the "critical voltage wall" [19] that hinders overscaling the processor beyond some point. At the critical voltage, the number of paths causing timing errors increases dramatically and the benefit of operating at a lower voltage is negated by the penalty due to error recovery. As a result, design-time and run-time optimizations have been proposed to extend the capabilities of timing speculation [1, 13, 15, 35]. The EVAL [35] framework applies microarchitectural techniques and run-time control techniques to reshape the error rate curve using machine-learning algorithms. Gupta *et al.* [15] proposed local recovery and fine-grained dynamic adaption to further maximize power-performance efficiency under process, voltage and temperature variations. Blueshift [13] applies a ground-up based design by optimizing the most-often violated timing paths in order to maximize the potential of timing speculation. Kahng *et al.* [1] proposed a ground-up design approach

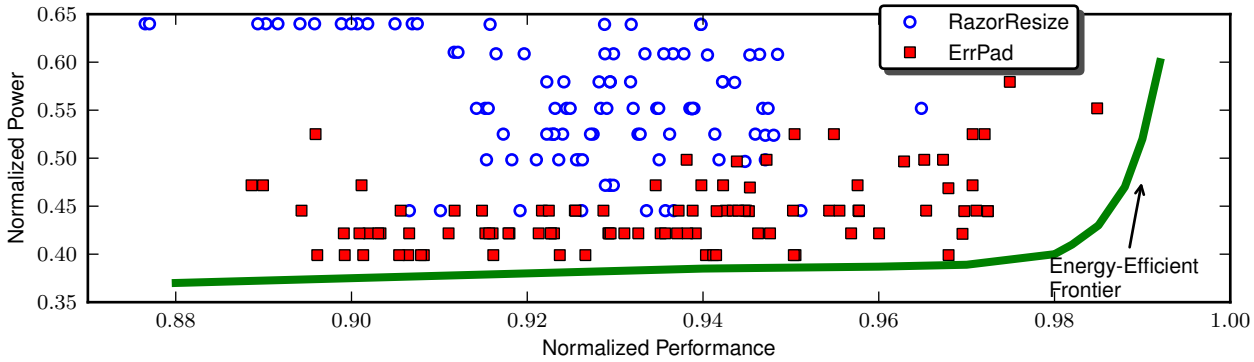


Figure 18: Power-performance under Process Variation

which applies slack redistribution and cell-resizing to reshape the distribution of near-critical paths. Overall this approach achieves significant power savings within target error rate. Our work is orthogonal and can be applied in concert with design-level optimization to reduce the impact of the "critical voltage wall" and boost timing speculation.

Analytic Versus Simulation Driven Models

Most prior work applies analytic error models either derived from empirical error rates [40] or extrapolated from circuit-level simulation [13]. While detailed gate-level timing models have been used in some studies [24], the simulation slowdown is often prohibitively slow. In particular, it typically limits study to very small portions of the processor design (e.g. a single execution unit). Our work used detailed gate-level delay simulation coupled with conventional microarchitectural simulation to provide analysis of timing error locality at the granularity of static instructions. Simulation time is still a bottleneck for this research. We hope that the knowledge that static instruction-level error rates are non-uniform will encourage other researchers to develop novel methodologies and models to accelerate simulation.

Static Versus Run-time Optimization

Very recently, there has been interest in adding compile-time support for timing speculation [10, 36]. Hoang *et al.* showed that instruction sequences can have significant impact on timing error rates within a simple five stage processor [10]. They evaluated a few very simple code transformations that significantly reduce the error rate. Sartori and Kumar investigated the impact of standard compiler optimizations on timing errors in a family of out-of-order architectures [36]. They show that compiling specifically for error resilient architectures can yield significant benefits through overscaling. Recent work by Zandian *et al.* [43] also explores cross-layer optimizations, namely application-specific path profiles, to improve wearout monitoring.

Parameter Variation

Much of the previous work has examined timing speculation as a means to overcoming circuit latency uncertainties related to parameter variations [15, 35]. We do not focus directly on parameter variation in our work. The primary gains we achieve come through exploiting differences in input sensitive timing paths. Although as we show, timing error locality is still a prominent and observable phenomena under parameter variation. This occurs for several reasons. First, parameter variations largely shift or reshape error curves. The critical path that an instruction accesses is largely dominated by its opcode and operand values. Process variation would make some small changes in which instructions were critical at a given operating voltage or alter the point at which the instructions became critical. It would not likely change the fact that a small number of instructions would be responsible for most of the timing errors. Second, if parameter variation did affect the error rate curves of two functional units in different ways, this still would not be likely to have a huge impact on our results. For relatively simple superscalar yet in-order cores (such as the one we model in this work), static instructions frequently execute on the same functional unit. In this case, individual timing-critical instruction would be no more difficult to predict than if there had been no variation. Finally, in designs where there was no strong binding between static instructions and execution units, one could imagine extending the predictor mechanism to make multiple predictions, one for each execution unit. When scheduling decisions are made either during or after slotting, the corresponding prediction could be examined to appropriately pad or not pad the instruction accordingly.

9 Conclusion

In this work, we examined error rate locality in processor execution hardware using detailed gate-level simulation. We find that static instructions have rather predictable and stable error rates (*timing error locality*). In particular, we find that the majority of timing errors come from a small number of static instructions and these static instructions tend to have stable error rates. Based on this insight we developed error prediction and *error padding*, a general pipeline scheduling technique that can be used to reduce the recovery cost in the event of predicted timing error.

We examined a wide range of mechanisms to dynamically predict timing critical instructions. When combined with *error padding*, we can avoid the full cost of timing error recovery in traditional *better-then-worst-case (BTWC)* design. Our simulation shows that error prediction can successfully identify over 80% of the timing errors and reduce the performance penalty caused by timing error recovery by 80% on average. Because error rates are also a function of program phase and other dynamic factors, we also implemented dynamic tuning algorithms that optimizes the energy-delay while maintain tolerable performance degradation level. With the dynamic control mechanism we could achieved 21% improvement in power efficiency compared to a baseline timing speculative design with the same dynamic optimization mechanism.

10 Acknowledgments

We would like to thank anonymous reviewers for their helpful comments. This work was supported by NSF CAREER CCF-0644332 and NSF CSR-0720820.

11 References

[1] R. K. Andrew Kahng, Seokhyeong Kang and J. Sartori. Designing processors from the ground up to allow voltage/reliability tradeoffs. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA-16)*, "Jan" 2010.

[2] T. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual Symposium on Microarchitecture (MICRO-32)*, November 1999.

[3] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 2–7, New York, NY, USA, 2005. ACM.

[4] O. Azizi, A. Mahesri, B. Lee, S. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *ISCA 2010. International Symposium on Computer Architecture*, pages 26–36. ACM, 2010.

[5] 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.

[6] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. In *hpca*, page 0299. Published by the IEEE Computer Society, 2002.

[7] 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.

[8] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos. Modeling within-die spatial correlation effects for process-design co-optimization. In *Proc. of the 6th Int. Symp. on Quality Electronic Design*, 2005.

[9] F. Gabbay and A. Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems*, Aug. 1998.

[10] R. B. F. Giang Hoang and R. Joseph. Exploring circuit timing-aware languages and compilation. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, "March" 2011.

[11] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, MobiCom '95, New York, NY, USA, 1995.

[12] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. 2007.

[13] 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.

[14] 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.

[15] M. Gupta, J. Rivers, P. Bose, G.-Y. Wei, and D. Brooks. Tribeca: Design for pvt variations with local recovery and fine-grained adaptation. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 435 –446, 2009.

[16] 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.

[17] T. Halfhill. Intel's tiny atom. *Microprocessor Report*, 22(4):1, 2008.

[18] http://www.nangate.com. Nangate *45nm* Cell Library.

[19] Janak H. Patel. CMOS process variations: A critical operation point hypothesis. 2008.

[20] S. G. D. K. A. Bowman and J. D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid State Circuits*, 37:183–190, Feb. 2002.

[21] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–11. IEEE, 2010.

[22] D. Lampret. Openrisc 1200 ip core specification, rev. 0.7. *Sep*, 6:9, 2001.

[23] 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.

[24] M. Li, P. Ramachandran, S. Sahoo, S. Hari, S. Adve, V. Adve, and Y. Zhou. SWAT-Sim: Accurate Microarchitecture-Level Fault Models. In *Poster, GSRC Annual Symposium*, volume 29, 2008.

[25] 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.

[26] X. Liang, G.-Y. Wei, and D. Brooks. Revival: A variation-tolerant architecture using voltage interpolation and variable latency. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 191 –202, 2008.

[27] M. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–47, Oct. 1996.

[28] M. H. Lipasti and J. P. Shen. The performance potential of value and dependence prediction. In *in EUROPAR-97*, pages 1043–1052. Springer-Verlag, 1997.

[29] S. McFarling. Combining branch predictors. Tech. Note TN 36, Digital Western Research Laboratory, June 1993.

[30] S. Mukhopadhyay, K. Kang, H. Mahmoodi, and K. Roy. Reliable and self-repairing SRAM in nano-scale technologies using leakage and delay monitoring. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 10–1135. IEEE, 2006.

[31] 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.

[32] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 233–244. IEEE, 2002.

[33] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[34] T. Sakurai and A. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter. *IEEE J. Solid-State Circuits*, 25(2):584–594, 1990.

[35] 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.

[36] J. Sartori and R. Kumar. A case for timing error resilience-aware compilation. In *Proceedings of the 7th Workshop on Silicon Errors in Logic - System Effects (SELSE 2011)*, "March" 2011.

[37] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *Solid-State Circuits, IEEE Journal of*, 20(2):510–522, 2002.

[38] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *in 29th Annual International Symposium on Computer Architecture*, pages 295–306, 2002.

[39] R. Sites. *Alpha architecture reference manual*. Digital Pr, 1998.

[40] R. Teodorescu, B. Greskamp, J. Nakano, S. Sarangi, A. Tiwari, and J. Torrellas. Varius: A model of parameter variation and resulting timing errors for microarchitects. *IEEE Trans on Semiconductor Manufacturing*, 2008.

[41] The Standard Performance Evaluation Corporation, December 2000. http://www.spec.org.

[42] www.synopsys.com. Synopsys Design Compiler.

[43] B. Zandian and M. Annavaram. Cross-layer resilience using wearout aware design flow. *Dependable Systems and Networks, International Conference on*, 0:279–290, 2011.