Compiler-guided instruction-level clock scheduling for timing speculative processors

Yuanbo Fan, Tianyu Jia, Jie Gu, Simone Campanoni, Russ Joseph
Department of Electrical Engineering and Computer Science
Northwestern University, Evanston, IL
{yuanbo,tianyujia2015}@u.northwestern.edu,jgu@northwestern.edu,{simonec,rjoseph}@eecs.northwestern.edu

ABSTRACT

Despite the significant promise that circuit-level timing speculation has for enabling operation in marginal conditions, overheads associated with recovery prove to be a serious drawback. We show that fine-grained clock adjustment guided by the compiler can be used to stretch and shrink the clock to maximize benefits of timing speculation and reduce the overheads associated with recovery. We present a formulation for compiler-driven clock scheduling and explore the benefits in several scenarios. Our results show that there are significant opportunities to exploit timing slack when there are appropriate channels for the compiler to select clock period at cycle-level.

1 INTRODUCTION

An explosion of ultra low-power applications raise the profile of aggressive system-wide optimizations including techniques which exploit dynamic timing slack [2, 3, 11, 12]. Dynamic timing slack (DTS) refers to the unused portion of the clock period in which all signals in the design have already propagated through logic paths and wait until the clock edge. At any given cycle DTS will appear if critical paths are not currently exercised. If there are enough continuous cycles with non-zero DTS, there is an opportunity to lower the supply voltage to decrease the system-wide energy without compromising performance. Given the emergence of applications with extremely power constrained profiles in the wearable computing, IoT, and implantable device spaces, there has been significant interest in aggressive schemes like those which exploit DTS. There are a variety of possible solutions to take advantage of DTS. Among these solutions, circuit-level timing speculation proved to be promising.

Circuit-level Timing Speculation (TS) allows a system to simultaneously exploit DTS and eliminate process, voltage, and temperature (PVT) margins, but may impose significant recovery costs. Under timing speculation, the supply voltage can be lowered without changing the clock frequency to improve the energy profile. Because the system is operating outside of conventional design margins, occasionally, signals arrive after the clock edge and the incorrect values are latched, producing timing errors. Timing speculative systems must therefore include mechanisms to detect these errors before they can be committed to visible state and recover so that architecturally correct execution can resume. The area, energy, and performance overheads associated with recovery are the biggest factors that have limited widespread adoption of timing speculation.

In this work, we describe a general framework for compiler guidance of a fast digital phase-locked loop (PLL) for intelligent clock generation which can be adjusted to reduce the total recovery cost of low-power timing speculative pipelines. Our approach relies on a programmable clock generator which can stretch and shrink the clock period at a cycle-to-cycle granularity. Our compiler inserts clock scaling instructions directly into the instruction stream to match the activity in the pipeline. The scaling instructions can either shrink the clock to convert extra timing slack into performance improvements or stretch the clock to avoid timing errors and their associated recovery costs. The compiler is guided by a mathematical framework which considers the cost of inserting clock control into the program and an algorithm which balances the benefits of clock adjustment with undesirable impact of code growth. This approach can be effective in statically scheduled pipelines commonly used in low-power embedded systems. We make the following contributions: (1) We introduce a novel scheme for compiler driven clock-control which drastically lowers the effective error rate and improves the efficiency of statically scheduled timing speculative pipelines; (2) We propose a mathematical framework for clock scheduling which casts the decisions as an optimization problem and considers the costs and benefits of inserting instructions into program; (3) We apply supervised learning to evaluate the error rates within the compiler. We believe this is the first compiler based approach to use machine learning to model properties of timing speculation.

2 BACKGROUND AND RELATED WORK

2.1 Timing Speculation

Timing speculative architectures have been proposed to reduce many of the design margins that appear in conventional systems. The significant impact that process, voltage, and temperature variations (PVT) have on logic delay demands that these margins be quite wide. Consequently, these safeguards which guarantee reliable operation in the worst case impose a severe tax on the efficiency of the entire system and are unnecessary in the common case. The Razor project [4, 5] is an influential example of a timing speculative design. It introduced an in situ error detection mechanism built around a special Razor Flip-flop. The original proposal was applied to a low-power pipeline [5]. Subsequent work examined ways to extend support aggressive superscalar pipelines [4]. These designs explored a wide variety of ways to detect and recover from errors. Given the significant cost for error recovery there have been many proposals for either reducing the effective error rate through both static and dynamic approaches [19]. Compiler support for timing speculative architectures has focused on ways to reduce error rates [16].

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196013

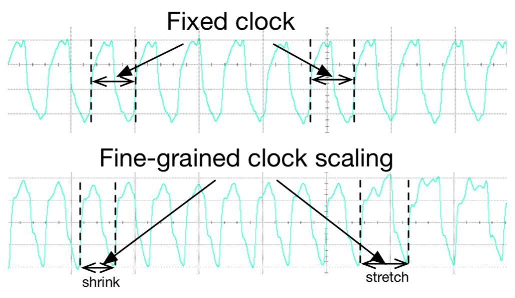


Figure 1: Measured clock waveform with fine-grained clock adjustment on the test chip.

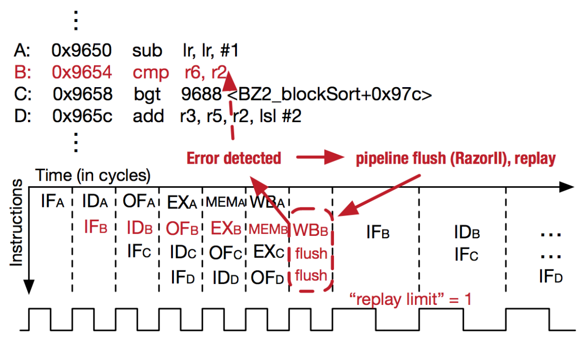
2.2 Compiler Driven Clock Management

While most systems that apply Dynamic Voltage Scaling (DVFS) do so under guidance of the operating system, there are strong arguments for having the compiler drive the voltage/frequency selection. Different voltage and frequency operating points can be appropriately matched to suit characteristics of long loops or program phases resulting in good power/performance tradeoffs. Xie et al examined the benefits and limitations of compile time DVFS [18]. Wu et al introduce a framework for run-time DVFS in dynamic compilation system [17]. Dynamic compilation can be fine-grained and responsive to changes in the run-time environment yielding significant benefits not otherwise possible. These approaches are all limited by inherent switching latencies associated with voltage regulators and conventional PLLs which require thousands of clock cycles to switch between voltage/frequency operating points [14]. In this work, we exploit phase selection with an ultra fast PLL (approximately sub-cycle response time), allowing us to stretch/shrink the clock period at a cycle-to-cycle granularity as shown in Figure 1. By adjusting the clock at orders of magnitude faster than DVFS, we can exploit instruction-level timing characteristics.

3 OVERVIEW AND MOTIVATION

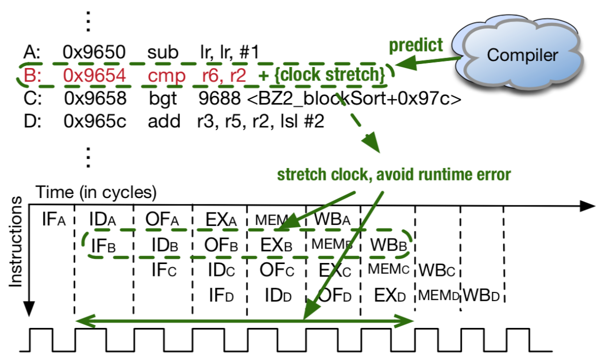
Existing timing speculation systems typically rely on extra hardware to detect errors. These systems recover from an error detected by flushing the pipeline and replaying the related instructions in a slow and safe recovery mode [4]. The large overhead associated with recovery prevents the system from operating under more aggressive voltage and/or clock frequency. Figure 2a shows the flow of instructions in a simple timing speculative pipeline. The `cmp` instruction encounters a timing error during its execution. The error is detected and the system recovers by replay. To ensure correct operation and avoid any additional timing errors during replay, the pipeline operates at half the clock frequency. This guarantees forward progress and correct execution but makes recovery expensive. However, if the supply voltage decreases beyond some point the error rates increase sharply and the system spends too much time in recovery. At this point the energy costs associated with recovery dominate and negate the energy savings of operating at a lower voltage.

Prior work has shown that dynamic timing slack and circuit-level errors are correlated with specific instruction sequences and data usage patterns [6, 10]. Therefore, instruction-level error models can be built and used to estimate the likelihood of an error. A compiler can then rely on such a model to analyze compiled code to identify which instructions are most prone to errors. Moreover, models can be parameterized to predict error rates under different operating conditions.



```
⋮
A:  0x9650  sub   lr, lr, #1
B:  0x9654  cmp   r6, r2
C:  0x9658  bgt   9688 <BZ2_blockSort+0x97c>
D:  0x965c  add   r3, r5, r2, lsl #2
⋮
```

(a) Error detection and recovery in RazorII pipeline.



```
⋮
A:  0x9650  sub   lr, lr, #1
B:  0x9654  cmp   r6, r2  + {clock stretch}
C:  0x9658  bgt   9688 <BZ2_blockSort+0x97c>
D:  0x965c  add   r3, r5, r2, lsl #2
⋮
```

(b) Error prediction and compiler-guided clock stretch

Figure 2: Compiler-guided clock scaling for predicted error.

A static compiler can influence the clock period by inserting directions for the programmable PLL into the instruction stream. At run-time, the hardware decodes the clock adjustment directives and configures the PLL. The way that clock control directives are encoded in the instruction stream will influence when and where it is advantageous to scale the clock period. Extra instructions produce memory traffic, consume cache capacity, and expend valuable fetch and execute bandwidth. Consequently, the benefit of scaling the clock at any given point in the program have to be weighed against the cost. For example, architectures with sparse instruction encodings represent the extreme case where controls can be embedded at every instruction. By entirely embedding control information within existing instructions there would be no overhead and the benefits of clock adjustment would be maximized. On the other hand, if the architecture encoding was denser, it would be necessary to insert special clock adjustment operations into the instruction set. At each point in the program where the compiler wanted to adjust the clock, it would need to insert this special instruction. This would consequently limit the compiler to adjust the clock only when it provides large benefits.

Figure 2b shows what the same `cmp` instruction embedded with a stretch clock attribute. In this instance, the compiler was able to determine that this static instruction was likely to generate timing error, and it uses fine-grained clock adjustment to avoid the error and subsequent recovery operation. This directly improves execution time and energy.

Overall, these capabilities increase the reach of timing speculation and can foster significant energy reduction. With lower average recovery cost per error, the system may operate at more aggressive clock frequency or voltage, as shown in Figure 3.

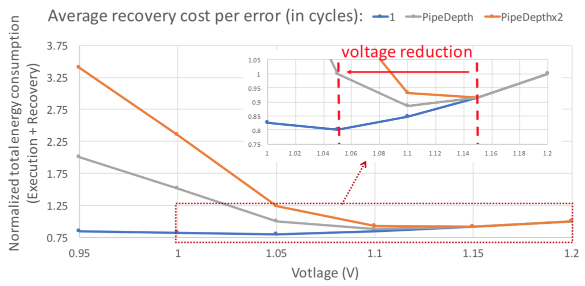


Figure 3: Lowered voltage due to reduced recovery cost per error for gcc. The recovery overhead per error in TS pipeline [4] is between $PipelineDepth$ and $PipelineDepth \times 2$.

4 PROPOSED SCHEME

4.1 Problem Formulation

4.1.1 Problem: Previous work has shown that in many processor pipelines timing critical paths are not triggered on every clock cycle [2, 3, 12]. Furthermore instructions within a pipeline have different circuit-level timing requirements. This is a consequence of varying degrees to which instructions stress critical paths in the design. In conventional designs, the clock frequency remains constant over thousands or millions of cycles due to the slow response rate of DVFS. This results in dynamic timing slack from cycle to cycle. For non-speculative systems, the fixed clock period is determined under the worst-case circuit-timing analysis, which guarantees the correct operation but wastes power and hurts performance. By contrast, timing speculative systems can operate at a more aggressive voltage level or clock frequency that is optimized for common cases. In practice, the steep error rate curves mean that designs can over-scale to a very limited degree and often operate very close to the point of fist failure (PoFF) [4, 5].

We assume a timing speculative in-order processor in which the clock period can be scaled at a very fine granularity (cycle-by-cycle). We then assume that the compiler can select the clock period on a cycle-by-cycle basis by appropriately inserting control information at some fixed cost per stretch/shrink of the clock. Then the execution time of an input trace is function of its CPI, the base clock period, the error rates of instructions in the program, clock control embedded by the compiler, and the recovery cost per error. This allows us to formulate an optimization problem for minimizing the total execution time for given program at specified voltage level in the presence of timing errors.

4.1.2 Formulation: Given an instruction stream, clock assignment for each static instruction can be found to minimize the total execution time. G_i is the overhead when switching from one clock period to another. This is determined by the way the system implements instruction-level clock scaling, including both the cost associated with embedding control information into program and the hardware overhead. For example, in an ideal case, if the control information for clock scaling is embedded inside the instruction without any extra cost, and the overhead of changing from one clock period to another is negligible, G is 0. R_i is defined as the overhead (in cycles) associated with error recovery for an instruction. For Razor-like TS processors, $R \propto$ `Pipeline Depth` and it may also be related to the CPI. We define Err_i as the error rate of instruction i during program execution. Any error introduced by this instruction in any stage is treated as an error produced by this instruction. For a static program, dynamic execution sequence is mostly fixed due to the fact that instructions are statically scheduled in in-order processors. If the instruction has different preceding instructions due to control flow convergence, we apply a conservative analysis. The maximum error rate among all possible control flows within two conditional branches is applied. Finally, w_k is a fraction which represents the frequency that instruction k is executed at runtime. To sum up, the total execution of an instruction trace can be represented as 1.

$$T = \min_{p \subset P}\{\sum_{i=0}^{N}(CPI_i \times p_i + G_i(p_{i-1}, p_i) + R_i \times Err_i(p_i)) \times w_i\} \quad (1)$$

For a given static instruction trace, values of P_i which minimize this equation provide the optimal execution time and implicitly trade-off clock shrinking and stretching with the overhead of inserting clock management instructions. Also, since this mechanism is flexible about the length of trace, the clock management can be done for a small piece of instruction sequence or large static program depending on the overall benefit. Therefore, it works for complex structures, such as nested loops or recursive functions.

4.2 Clock Scheduling Algorithm

Our Clock Scheduling Algorithm is used by the compiler to choose optimal clock scheduling for the static instruction trace. Based on the equation described before, the clock period selected for each instruction affects both the transition cost from previous to current clock (G_i) and the error rate of the current instruction.

Algorithm 1 applies dynamic programming to solve this problem. Assume that the program length is N (instructions) and the number of available clock periods provided by the multi-phase ADPLL is M. memo[k,p] represents the minimum execution time of instruction sequence [k:N] while the previous clock period selected is p. The complexity of this algorithm is $O(NM^2)$. Since M is usually small in most of systems, the complexity can be practically treated as $O(N)$.

Algorithm 1 Instrution-level Clock Scheduling Algorithm

```
1:  procedure instr_exec_time($k, P_{k-1}, P_k$)
2:      return $CPI_k \times P_k + G_k(P_{k-1}, P_k) + R_k \times Err_k(P_k)$
3:  end procedure
4:  procedure min_total_exec_time
5:      I: instruction sequence [1:N]
6:      P: set of clock phases $\{P_1, P_2, ...P_M\}$
7:      C: default clock period
8:      Initialize table memo[1...N, 1...M]
9:      $\forall p \in P, memo[N+1, p] = 0$
10:     for i = N to 1: do // from the last instr. to the first
11:         for s = $P_1$ to $P_M$ do // all possible clock set before i
12:             memo[i, s] = min$_{j \in P}$(instr_exec_time(i, s, j)
13:             + memo[i+1, j])
14:         end for
15:     end for
16:     return memo[1, C] // min. exec. time for the trace
17: end procedure
```

4.3 Instruction-level Delay Model

The connection between certain high-level instruction patterns and low-level circuit timing characteristics provides an opportunity to effectively model errors in the design. In in-order processors, this correlation is even stronger due to the fact that instruction sequence directly determines the execution order in the pipelines. By taking advantage of this property, supervised learning may be used to extract important "features" from labeled data and build adaptive error models automatically.

We specifically select features: (1) that are available at compile time and (2) that may trigger errors in the circuit. One of our goals is to be able to identify instruction-level program characteristics that can be easily tracked by the compiler and yet correlate well to errors which are rooted in circuit-level structure. One reasonable way to do this is to use a recent segment from the instruction stream which represents instructions that are currently present in the pipeline.

During the dynamic execution of an instruction stream, the sequence can be labeled as either "Error" (1) or "No Error" (0). As the design is over-clocked step by step, different sequences are labeled in each step. Based on this information, the delay range of instruction sequences can be derived. Then, these labeled instruction sequences are used as training dataset to build single/separate models with supervised learning algorithm. This instruction timing data reflects circuit-level timing characteristics of both hardware design and real environment such as Process, Temperature and Voltage (PVT) variations, thereby models that are trained by this data do not only make predictions for delay introduced by program, but also dynamic conditions that the program is currently running under. In our experiments, we applied various algorithms (including Decision Tree, Multi-Layer Perceptron and Support Vector Machine) for training models. Among these, Decision Tree shows good accuracy with relatively low training/testing cost, therefore, we decided to use Decision Tree for the evaluations in Section 6.

With the guidance of the models, the compiler is able to identify both critical and non-critical instructions and schedule the clock at cycle-level accordingly based on the micro-architectural implementation of instructions. For cases that some instructions are overlapping in the same cycle, the compiler selects the conservative clock to avoid errors.

4.4 Code Portability under PVT variations

In this approach, a static compiler intervention is proposed to integrate clock scheduling with the program stream for minimizing the total execution time. Success depends on the quality of the model. The optimized code should improve the system operation as long as the model is a close match to the conditions observed at run-time.

As in the original Razor designs ([5] [4]), we assume that the processor includes a simple hardware control loop which monitors error rate and applies dynamic voltage scaling (DVS) to maintain a low error rate (e.g. around 1%). In cases when the environmental conditions do not match the target model (e.g. ambient temperature rises or the process variation profile does not match the training data), we would expect that the error rate would briefly spike. The control logic would adjust the voltage to correct from the error and the system would resume operation at a stable voltage and low error rate. As we shown in Section 6.1, our models are fairly robust under small to moderate variations. The feedback controller can help in cases when the changes are more pronounced.

4.5 Optimization with 3-phase Clock Scheduling

To illustrate the proposed approach, we optimize the program with 3-phase clock selection on a TS processor. There are three clock periods that are available for the compiler to schedule, and they are 50%, 85% and 100% of the nominal clock period. Figure 4 shows the

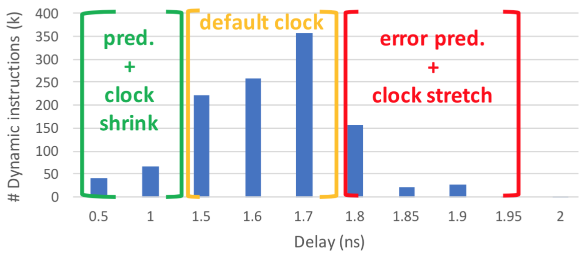
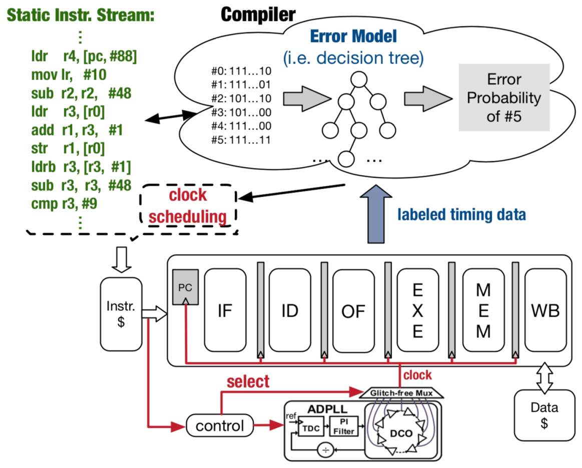


Figure 4: 3-phase clock selection for basicmath.



```
    ⋮
ldr   r4, [pc, #88]
mov lr,  #10
sub  r2, r2,  #48
ldr   r3, [r0]
add  r1, r3,  #1
str   r1, [r0]
ldrb  r3, [r3,  #1]
sub  r3,  r3,  #48
cmp r3, #9
    ⋮
```

Figure 5: Overview of the proposed scheme.

distribution of dynamic instruction delays for basicmath. As we can see, the majority of dynamic instructions have circuit-level delay less than 85% of the nominal, which is used as the default clock period for the program execution. The mechanism that is used to embed clock selection information into program is to insert a clock selection instruction. In our evaluation, we assume that clock control instructions consume fetch bandwidth and hence increase the runtime of the program. However, if the clock selection information can be directly embedded into the instruction or there is excess fetch bandwidth, this overhead can be avoided.

We use supervised learning to build adaptive error models that predict suitability of instructions to be clocked at 50%, 85%, and 100% of the clock period. Based on these model, the compiler inserts clock selection instructions to stretch or shrink the clock. For incorrectly predicted errors, the overhead of the False Position (FP) prediction will be wasted timing slack. For False Negatives (FN) (errors that are not predicted), the full recovery (i.e. pipeline flush and replay) is required to guarantee the correct execution.

5 METHODOLOGY

5.1 Processor and Workload

We evaluate our approach by modeling a six-stage single issue ARMv7 pipeline supporting timing speculation. This design is suitable for battery-power or energy-scavenging systems. The pipeline closely models a 55nm test chip that that we fabricated to study the potential for dynamic clock adjustment as proposed in this paper. We build a gate-level model of the processor pipeline and capture the gate-level simulation to study dynamic timing slack and timing errors. We further validated our simulated model with the test-chip to understand model fidelity.

We extend the LLVM-ARM backend [13] so that the compiler is able to predict errors for instructions based on static sequences and insert clock scaling instructions accordingly. The multi-phase ADPLL is able to scale clock at cycle level, and the compiler selects the conservative clock period for consecutive instructions. We cross-compile and run several benchmarks from the MiBench [8] and the SPEC benchmark suite [9] using LLVM with highest optimization level. All phases in benchmarks are sampled by SimpPoint [15] with interval size of 100M instructions. The overall benefit is weighted by all phases.

5.2 Clocking Assumptions

We designed and evaluated a multi-phase All-Digital Phase-Locked Loop (ADPLL) associated with the pipeline, as described in Figure 5. The entire design is fabricated on a 55nm CMOS silicon chip. The ADPLL provides fine-grained clock adjustment via phase selection, as shown in Figure 1. Similar implementations of dynamic clock phase selection were proposed in [1, 12]. We assume that clock scaling directives come directly from instruction fetch and can be evaluated directly by the ADPLL. We assume that the fetch bandwidth remains the same as the baseline design and therefore the overhead of each clock scaling instruction is one extra fetch cycle.

Our test chip includes the ADPLL, and we use validation of the test chip to show the practicality of fine grained clock adjustment. The entire ADPLL occupies only 2.48% of the total chip area, and is able to provide the range from 60% to 150% of the reference clock rate, and consumes less than 5% of the total power consumption in the test chip.

6 EVALUATION

6.1 Model Accuracy

The error model plays an very important role in this scheme by determining which instructions produce errors at compile time. The prediction results include four categories: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). To be more specific, the TP represents the correctly predicted "Error", while the TN is the correctly predicted "No Error". The accuracy of model is defined as "correctly classified rate" ($\frac{TP+TN}{TP+FP+TN+FN}$). For FN, the full recovery may be required in hardware and the overhead is proportional to the depth of pipeline, while the overhead of FP is only the stretched portion of clock period which is normally around 20%. Therefore, the model is biased based on the overhead of misclassification for FP and FN.

Figure 6 shows the classification breakdown of the model built and tested under lowered voltage, and we can see that more than 95% of the instructions are correctly predicted by the model. Also, compared to a "naive" model that always predicts "no error", the model provides very low rates for high-overhead FN cases (« 1%), which significantly lower the recovery cost associated with errors through the clock stretching at runtime. To validate the model under various scenarios, Figure 7 compares the accuracy among different PVTs and features. "Diff-PVT" represents the case when the optimization is targeted at PVT0 (nominal): {Process: Typical, Voltage: 1.2V, Temperature: 25C}, but the scheme is applied under PVT1: {Process: Slow, Voltage: 1.08V, Temperature: 125C}. We observed that, the model accuracy depends heavily on the representativity of training dataset. Fortunately, profiling with training inputs provides good datasets to build high-accuracy error models. Also, as long as the instruction pattern has same label across different PVT

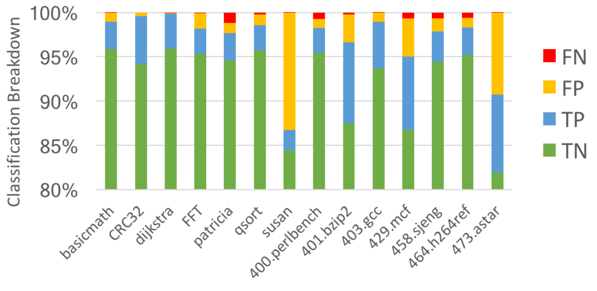


Figure 6: Breakdown of classification results under PVT0.

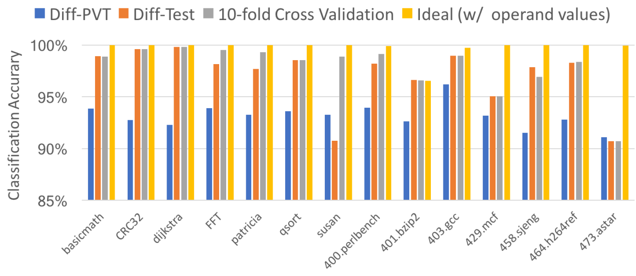


Figure 7: Comparison with various conditions and features.

conditions, the model is able to make the correct prediction. We also tested the model by using both separate dataset and 10-fold cross validation. The results show that, the model can achieve better than 93% accuracy across all conditions. Since circuit-level timing also has a strong correlation with the data usage, when operand values are added as extra features to train/test the model, the accuracy can be as high as 99.7%. This suggests that further improvements could be made if the compiler had good knowledge about operand values (e.g. through value profiling).

6.2 Optimization for Minimum Energy with 3-phase Clock Scheduling

We evaluated the energy savings with simple three-phase clock scheduling. With the proposed error prediction at compile time and clock stretch at runtime, the runtime error rates and recovery overhead are dramatically reduced. This maximizes the overall benefits of the TS pipeline and push it to operate at a more aggressive voltage. As Table 1 shows, voltage level is further reduced by 12% compared to 5% reduction of the point where the TS system has the minimum energy consumption, with only averagely 0.2% runtime error rate.

Clock stretching can effectively lower runtime error rates thereby leading to the operation under a more aggressive voltage level. Moreover, clock shrinking improves performance by reducing unnecessary cycle time. As Figure 8 shows, the total energy saving is as high as 36%, as normalized to the baseline design which assumes a TS processor operates at nominal conditions (voltage and clock frequency). Results for TS at the minimum energy point only shows less than 12% on average due to the high recovery cost thereby more conservative voltage. For some low recovery overhead TS systems proposed recently ([7, 19]), it is possible to further push the operating voltage. However, it normally comes with extra complexity in the design (i.e. clock tree) as well as area and power cost. Since the overhead associated with clock stretching is much lower

Table 1: Comparison of error rates under aggressive supply voltage for minimum total energy consumption

Benchmark	Vol. w/ TS (V)	Vol. w/ clk sched. (V)	Err. w/o clk sched. (%)	Err. w/ clk sched. (%)
basicmath	1.12	1.08	3.13	0.62
CRC32	1.12	1.09	5.41	0.04
dijkstra	1.12	1.08	3.94	0.14
FFT	1.13	1.08	2.86	0.02
patricia	1.13	1.08	4.24	0.70
qsort	1.12	1.08	2.97	0.40
susan	1.12	1.08	2.32	0.05
400.perlbench	1.12	1.08	3.48	0.03
401.bzip2	1.13	1.08	9.27	<0.01
403.gcc	1.11	1.05	5.32	0.08
429.mcf	1.12	1.07	9.03	0.07
458.sjeng	1.13	1.08	4.03	0.57
464.h264ref	1.11	1.06	3.63	0.09
473.astar	1.11	1.06	8.83	0.19

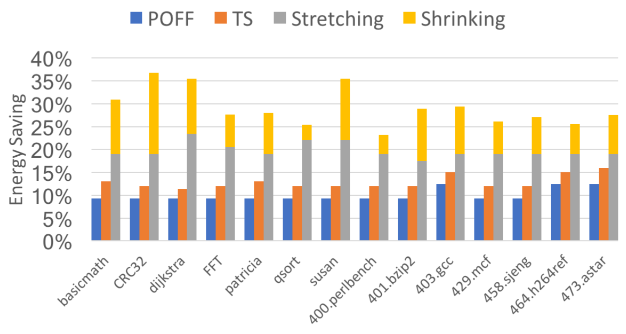


Figure 8: Our Energy savings compared to the PoFF.

(< 0.5 cycle), with the guidance of the error model and clock scheduling at compilation time, the average recovery cost is even lower than the proposed low cost recovery. In this work, the hardware model and baseline are designed based on conventional Razor style systems ([4]). The proposed approach is independent of the error detection and correction mechanisms. Moreover, different from previous works on TS which detects and recover errors on static paths, this technique schedules clock with the consideration of program dependent timing error and provides performance improvement by reducing program introduced timing slack.

7 CONCLUSION

This work introduces a compiler guided scheme for fine-grained clock period management of timing speculative processors. This approach leverages a compiler with knowledge of instruction-level dynamic timing slack with a fast, programmable PLL to enhance the benefits of timing speculation. We introduce a mathematical frameworks and algorithm that help the compiler decide when and where to insert clock control information into the instruction stream. With a simple three-phase clock scheduling approach our technique achieves on average 27.3% energy savings over a range of benchmarks.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grants CCF-1116610 and CCF-1618065.

REFERENCES

[1] K. A. Bowman, S. Raina, J. T. Bridges, D. J. Yingling, H. H. Nguyen, B. R. Appel, Y. N. Kolla, J. Jeong, F. I. Atallah, and D. W. Hansquine. 2016. A 16 nm All-Digital Auto-Calibrating Adaptive Clock Distribution for Supply Voltage Droop Tolerance Across a Wide Operating Range. *IEEE Journal of Solid-State Circuits* 51, 1 (Jan 2016), 8–17. https://doi.org/10.1109/JSSC.2015.2473655

[2] H. Cherupalli, R. Kumar, and J. Sartori. 2016. Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 671–681. https://doi.org/10.1109/ISCA.2016.64

[3] Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. 2015. Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment. *Proceedings of the 2015 Design, Automation & Test in Europe* (2015), 381–386.

[4] S. Das, C. Tokunaga, S. Pant, W. H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw. 2009. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *IEEE Journal of Solid-State Circuits* 44, 1 (Jan 2009), 32–48. https://doi.org/10.1109/JSSC.2008.2007145

[5] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, Trevor Mudge, Beal Ave, and Ann Arbor. 2003. Razor : A Low-Power Pipeline Based on Circuit-Level Timing Speculation. December (2003).

[6] Yuanbo Fan and Russ Joseph. 2017. D2M: Data-driven Model for Fast and Accurate Timing Error Simulation in Statically Scheduled Microprocessors. In *Proceedings of the Summer Simulation Multi-Conference (SummerSim '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 4, 13 pages. http://dl.acm.org/citation.cfm?id=3140065.3140069

[7] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. 2012. Bubble Razor: An architecture-independent approach to timing-error detection and correction. In *2012 IEEE International Solid-State Circuits Conference*. 488–490. https://doi.org/10.1109/ISSCC.2012.6177103

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[9] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[10] Giang Hoang, Robby Bruce Findler, and Russ Joseph. 2011. Exploring Circuit Timing-aware Language and Compilation. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 345–356. https://doi.org/10.1145/1950365.1950405

[11] T. Jia, Y. Fan, R. Joseph, and J. Gu. 2016. Exploration of associative power management with instruction governed operation for ultra-low power design. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898021

[12] T. Jia, R. Joseph, and Jie Gu. 2017. Greybox design methodology: A program driven hardware co-optimization with ultra-dynamic clock management. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062255

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[14] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. 2013. Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (May 2013), 695–708. https://doi.org/10.1109/TCAD.2012.2235126

[15] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (June 2003), 318–319. https://doi.org/10.1145/885651.781076

[16] John Sartori and Rakesh Kumar. 2012. Compiling for Energy Efficiency on Timing Speculative Processors. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1301–1308. https://doi.org/10.1145/2228360.2228602

[17] Qiang Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Youfeng Wu, Jin Lee, and D. Brooks. 2006. Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE Micro* 26, 1 (Jan 2006), 119–129. https://doi.org/10.1109/MM.2006.9

[18] Fen Xie, Margaret Martonosi, and Sharad Malik. 2003. Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/781131.781138

[19] J. Xin and R. Joseph. 2011. Identifying and predicting timing-critical instructions to boost timing speculation. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 128–139.