Light-weight one-cycle timing error correction based on hardware software co-design

Li Zhanhui¹, Ding Yong¹, Yan Xiaolang¹,², Meng Jianyi²a), and Xiang Xiaoyan²

¹ Institute of VLSI Design, Zhejiang University, Hangzhou, 310027 China
² State Key Laboratory of ASIC and System, Fudan University, Shanghai, 201203 China

a) mengjy@fudan.edu.cn

Abstract: A light-weight one-cycle EDAC (timing error detection and correction) technique is proposed to eliminate timing margins resulting from process, voltage, temperature and aging (PVTA) variations. The collaborative approach applies TEDPI (timing error deletion programming interface) to pre-detect most of the errors through an offline error prediction model, and pre-correct them at compilation time by using a specially designed error avoidance instruction. Experimental results based on a three-stage commercial processor show that TEDPI has improved peak performance of the traditional EDAC system by 15.6% and reduced the energy consumption by 4.4% with less than 0.71% area overhead.

Keywords: timing error, timing error prediction model, timing speculation, error correction penalty

Classification: Integrated circuits

References

[1] S. Nassif, et al.: “High Performance CMOS Variability in the 65nm Regime and Beyond,” IEDM Tech. Dig. (2007) 569 (DOI: 10.1109/IEDM.2007.4419002).

[2] K. Bowman, et al.: “Circuit techniques for dynamic variation tolerance,” DAC (2009) 4 (DOI: 10.1145/1629911.1629915).

[3] D. Ernst, et al.: “Razor: a low-power pipeline based on circuit-level timing speculation,” MICRO-36 (2003) 7 (DOI: 10.1109/MICRO.2003.1253179).

[4] S. Das, et al.: “RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance,” IEEE J. Solid-State Circuits 44 (2009) 32 (DOI: 10.1109/JSSC.2008.2007145).

[5] I. Kwon, et al.: “Razor-Lite: A Light-Weight Register for Error Detection by Observing Virtual Supply Rails,” IEEE J. Solid-State Circuits 49 (2014) 2054 (DOI: 10.1109/JSSC.2014.2328658).

[6] K. A. Bowman, et al.: “Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance,” IEEE J. Solid-State Circuits 44 (2009) 49 (DOI: 10.1109/JSSC.2008.2007148).

[7] D. Bull, et al.: “A Power-Efficient 32 bit ARM Processor Using Timing-Error Detection and Correction for Transient-Error Tolerance and Adaptation to PVT Variation,” IEEE J. Solid-State Circuits 46 (2011) 18 (DOI: 10.1109/JSSC.
[8] K. A. Bowman, et al.: “A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance,” IEEE J. Solid-State Circuits 46 (2011) 194 (DOI: 10.1109/JSSC.2010.2089657).

[9] M. Fojtik, et al.: “Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction,” IEEE J. Solid-State Circuits 48 (2013) 66 (DOI: 10.1109/JSSC.2012.2220912).

[10] S. Kim and M. Seok: “Variation-Tolerant, Ultra-Low-Voltage Microprocessor With a Low-Overhead, Within-a-Cycle In-Situ Timing-Error Detection and Correction Technique,” IEEE J. Solid-State Circuits 50 (2015) 1478 (DOI: 10.1109/JSSC.2015.2418713).

[11] D. Ernst, et al.: “Razor: circuit-level correction of timing errors for low-power operation,” IEEE Micro 24 (2004) 10 (DOI: 10.1109/MM.2004.85).

[12] C. M. Huang, et al.: “An energy-efficient resilient flip-flop circuit with built-in timing-error detection and correction,” VLSI-DAT Inter. Sym. (2015) 1 (DOI: 10.1109/VLSI-DAT.2015.7114574).

[13] R. Ye, et al.: “On the premises and prospects of timing speculation,” ATECE EDA Con. (2015) 605 (DOI: 10.7873/DATE.2015.0440).

[14] V. J. Reddi and D. Brooks: “Resilient Architectures via Collaborative Design: Maximizing Commodity Processor Performance in the Presence of Variations,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 30 (2011) 1429 (DOI: 10.1109/TCAD.2011.2163635).

[15] O. Assare and R. Gupta: “Timing analysis of erroneous systems,” CODES (2014) 7 (DOI: 10.1145/2656075.2656101).

[16] J. Xin and R. Joseph: “Identifying and predicting timing-critical instructions to boost timing speculation,” MICRO-44 (2011) 128 (DOI: 10.1145/2155620.2155636).

[17] J. Sartori and R. Kumar: “Compiling for energy efficiency on timing speculative processors,” DAC (2012) 1297 (DOI: 10.1145/2228360.2228602).

[18] G. Hoang, et al.: “Exploring circuit timing-aware language and compilation,” ACM SIGPLAN Notices (2011) 46.3.345 (DOI: 10.1145/1961295.1950405).

[19] K. Chakraborty, et al.: “Efficiently tolerating timing violations in pipelined microprocessors,” DAC (2013) 102 (DOI: 10.1145/2463209.2488860).

[20] G. Hoang, et al.: “Exploring circuit timing-aware language and compilation,” ACM SIGPLAN Notices (2011) 46.3.345 (DOI: 10.1145/1961295.1950405).

[21] V. J. Reddi, et al.: “Eliminating voltage emergencies via software-guided code transformations,” IEEE TACO 7 (2010) 12 (DOI: 10.1145/1839667.1839674).

[22] The CSKY Corporation: CK802 Introduction (2012) http://www.c-sky.com/product.php?typeid=105.

[23] P. Ndai, et al.: “Trifecta: A Nonspeculative Scheme to Exploit Common, Data-Dependent Subcritical Paths,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 18 (2010) 53 (DOI: 10.1109/TVLSI.2008.2007491).

[24] S. Roy and K. Chakraborty: “Predicting timing violations through instruction-level path sensitization analysis,” DAC (2012) 1074 (DOI: 10.1145/2228360.2228555).

[25] Y. Chen, et al.: “Cascaded carry-select adder (C²SA): a new structure for low-power CSA design,” IEEE. ISLPED (2005) 115 (DOI: 10.1145/1077603.1077634).
1 Introduction

In advanced technology nodes, variations in process, voltage, temperature and aging (PVTA) significantly impact circuit functionality and performance [1, 2]. To ensure correct operations under all possible variations, suitable amount of timing margins must be added to the circuits [3]. These margins are usually set to be a worst-case combination of variations, which results in significant losses in performance, energy and cost.

The in-situ error detection and correction (EDAC) techniques [4, 5, 6, 7, 8, 9, 10, 11, 12, 13] are proposed to eliminate the timing margins. They monitor timing violations by specially designed circuits, and then take measures to recover functionality of the design if such a violation occurs. Examples of EDAC techniques include Razor [3, 4, 5, 9], TIMBER [14], TDTB [6], DSTB [6, 8], time borrowing [10, 12, 14], and etc. These techniques either correct timing errors with large cycle penalty, which in return degrades energy efficiency and throughput [10, 15]; or require massive hardware cost and high design complexity to reduce the penalty, which makes them unsuitable for commercial processors [16].

Based on the locality of timing errors in instruction level [17, 18, 19, 20, 21, 22], this work proposes a novel hardware-software co-designed EDAC approach to achieve one-cycle error correction. It applies TEDPI (timing error deletion programming interface) to pre-detect most of the errors through an offline timing error prediction model, and pre-correct them at compilation time by using a specially designed timing error avoidance instruction. TEDPI is able to eliminate most of the errors with small runtime overhead, which significantly improves the overall throughput. Meanwhile, since the hardware is only responsible for errors missed-detected by software, it can be designed to be simple and robust with small cost and design complexity. Both the qualitative analysis through mathematical derivation and quantitative analysis on a commercial processor [23] demonstrate that TEDPI is feasible and effective.

2 Motivation

2.1 Locality of timing errors in instruction level

The locality can be described in two aspects.

(1) Instructions with errors. Only some specific instruction sequences and operand values can bring about timing errors [18, 19, 24]. The former determines which set of critical paths will be exercised, while the latter determines the input vectors of these critical paths. Timing errors will not occur until the input vectors trigger long-distance transition propagation in the critical paths. Meanwhile, there is a high possibility that an instruction will cause errors repeatedly due to the locality of operand values [16, 18, 22, 25].

(2) Locations of errors. Most of the timing errors are distributed in execution units [17, 18, 19, 22, 24]. For example, the authors in [17] present that errors in the LEON3 processor are distributed between ALU (arithmetic and logic unit, 24%), address generation (25%), result bus (29%) and control logic (22%). The key reason is that critical paths are usually distributed in data paths of execution units,
which are mainly composed of large combinational logic, such as adders and multipliers.

2.2 Examples of the locality

Fig. 1 shows that no more than 40 dynamic instructions are responsible for most of the errors in the 3-stage commercial CK802 processor [23], and that more than 94% errors are caused by ALU and address generation of LSU (load store unit)/BJU (branch and jump unit). The CK802 processor is based on a China RISC ISA, which has 89 32-bit and 67 16-bit instructions. The experimental platform will be described in Section 4.2.

To have insight into the locality, Fig. 2 presents hot-errant code pieces from the benchmarks. They bring about timing errors in the ALU adder, whose critical paths are in the carry propagation logic [25, 26].

(1) calc_crc8/div32. Calculation of the loop iterator ($R_3$) in calc_crc8 always brings about errors because minuend of the $\text{subi}$ is always a small value less than

![Fig. 1. Distribution of positive errant instructions (PEIs) in CK802. AG (address generation), NPEI (number of PEIs).](image)

![Fig. 2. Code pieces from EEMBC (embedded microprocessor benchmark consortium) that cause errors.](image)
eight. But the addi in div32 does not bring about errors all the time because the value of R4 varies at large scale. From this we can find that errors are closely relative to the operand values of an instruction.

(2) WriteOut. Although cmphs does not bring about long-distance carry propagation, it still causes errors. This is because cmphs has data dependency with the preceding ld.w, and the data of ld.w is from output, which is quite time consuming. From this we can find that errors are closely relative to the instruction flows and data dependencies among instructions.

In the following descriptions, an instruction which has the potential to exercise the critical paths is called errant instruction. The errant instructions which actually bring about errors are called positive errant instruction.

3 The proposed TEDPI

TEDPI is based on the locality of timing errors. Since most of the errors are distributed among critical paths in execution units and are caused by a small portion of instructions, an online timing error avoidance instruction TEAI is proposed to exercise the critical paths for these instructions and prevent the potential errors from happening. Meanwhile, since not all the errant instructions bring about errors, an offline timing error prediction model TEPM is applied to find each positive errant instruction at compilation time and then generate a TEAI before the errant instruction.

3.1 TEAI to avoid the errors at runtime

TEAI serves as an alarm instruction or a timing error messenger. It tells the processor that its succeeding instruction will bring about timing errors. Then the processor can take measures to prevent the future errors from happening.

When detecting a TEAI, the processor continuously pipes down its succeeding instruction for multiple (M+1) times. The first M times pre-exercise and stabilize the critical paths, ignore errors, and cannot update architecture status; while the last times are allowed to update the status (Fig. 3(a)). This eliminates errors caused by the errant instruction. A similar technique is multi-reissue [8] (Fig. 3(b)). The key difference is that TEAI is an integral part of our TEDPI, which is controlled by software to avoid the errors before they really happen; while multi-reissue is only a hardware-level error correction technique, which corrects the errors after they really happen.

The pre-exercise times M can be determined by the number of errant pipeline stages. When an instruction only bring about errors in one stage, it needs to pre-exercise the pipelines once, so M is one; when it brings about errors in successive stages, such as errors in EX1 and EX2 stages, it has to pre-exercise the pipelines at least twice: the first times to avoid errors in EX1, and the second times to avoid errors in EX2, so M is two.

To avoid the compatibility problem caused by adding instructions to an ISA, the existing NOP-like instructions, such as \( \text{mov} r0, r0 (M = 1) \), \( \text{mov} r1, r1 (M = 2) \), can be reused to realize TEAI.
3.2 TEPM to find errors offline

The key work for TEPM is to generate the most suitable amount of TEAIs into the binary code to achieve the highest throughput at given frequency. The frequency is determined by the hardware, and is fed back to TEPM.

TEPM must generate TEAIs before hot instructions with high error rate, and the reason will be described in Section 4.2.2. To detect each hot instruction and calculate its error rate, TEPM integrates the clustered timing model [17] into a cycle-accurate pipeline simulator. The model collects stage-level transition information from the simulator, and then uses the information to calculate the maximum path delay. Knowing the maximum path delay, TEPM is able to predict whether the currently executing instruction will cause timing errors at given frequency. In order to account for variations among chips, the model adds an extra margin to each calculated path delay.

Supposing that the predicted error status for each execution is $P_r(i)$ (“1”: error occurs, and “0”: no error occurs), then the error rate $P_e(f)$ for each instruction can be calculated by Eq. (1), where $TT$ (total execution times) is achieved from the pipeline simulator, and $TE$ (total error times) is achieved through accumulating $P_r(i)$ for each execution.

$$P_e(f) = \frac{TE}{TT} = \frac{\sum_{i=1}^{TT} P_r(i)}{TT}$$

4 Experiment analysis

4.1 Qualitative analysis

$Def 1$ $F_{POFF}$: frequency point of the first error.

$Def 2$ $F_{MAX}$: maximum frequency point that an EDAC system can work.

$Def 3$ $F_{Opt}$: optimal frequency point that provides the highest throughput.
Assuming that the average cycles needed to execute an instruction without errors is $CPI$ (cycles per instruction), then the average execution time for each instruction is $CPI/f$. Assuming that the average error correction penalty for each error is $ECP$, then the average execution time for each instruction with an error rate of $P_e(f)$ is $(CPI + P_e(f) \times ECP)/f$. Consequently, the performance speed up ratio against $F_{POFF}$ can be achieved through Eq. (2), from which we can derive Eq. (3) and Eq. (4).

$$EDAC_{speed-up}(f, ECP) = \frac{CPI}{F_{POFF}} \left( \frac{CPI + P_e(f) \times ECP}{f} - 1 \right)$$

(2)

$$\frac{\partial EDAC_{speed-up}}{\partial ECP} = - \frac{CPI}{F_{POFF} \times ECP} \frac{f}{P_e(f)}$$

(3)

$$\frac{\partial EDAC_{speed-up}}{\partial f} = \frac{CPI}{F_{POFF}} \left( \frac{1}{CPI + P_e(f) \times ECP} - \frac{f}{ECP \times P_e(f)} \right)$$

(4)

Eq. (3) is always negative, which means that $EDAC_{speed-up}(f, ECP)$ will always decrease when $ECP$ increases. This is because a higher $ECP$ increases the total error correction cycles at given error rate.

Eq. (4) should be positive to ensure that the performance will increase with the scaling of $f$. By setting “$\text{Eq. (4)} \geq 0$”, we can achieve Eq. (5), which shows that a smaller $ECP$ results in higher $F_{Opt}$, thus higher performance.

$$F_{Opt} \leq \left( \frac{P_e(f_{opt})}{CPI/ECP + P_e(f_{opt})} \right)$$

(5)

Consequently, we can conclude that a smaller $ECP$ not only improves the performance speed-up ratio of EDAC systems at given frequency, but also pushes the optimal working frequency and throughput to be higher.

4.2 Quantitative analysis

4.2.1 Experimental setup

The commercial 32-bit CK802 processor [23] shown in Fig. 4 is used to implement TEDPI. The pre-exercise times $M$ for most of the instructions is one because of simple pipeline architecture. An TEAI-discarding mechanism is applied to speed up the execution of TEAI. When TEAI is going to be piped down to EX, its opcode is replaced with that of its succeeding instruction to pre-exercise the pipeline.

![Fig. 4. Pipeline architecture of CK802.](image-url)
one-cycle earlier. This guarantees that TEAI’s succeeding instruction is usually executed with an one-cycle penalty.

Razor-Lite register [5] is applied to detect errors, and multi-reissue [8] is utilized to correct errors that escape software. The signoff frequency of CK802 in SMIC 40ll after routing is 70 MHz. After implementing EDAC, the logic area of CK802 increases from 14027 to 15688 um2. After implementing TEAI, the area is further increased by 112 um2, which is less than 0.71%.

To simplify TEPM and achieve high prediction accuracy, we apply the RTL-level model as the pipeline simulator, and the critical paths extracted from the gate-level netlist as the clustered timing model. This provides a near 100% prediction accuracy, and is several times faster than gate-level simulation. Meanwhile, as TEPM is performed offline on an Intel 24-core server, it is simply discussed in the following experiments.

The benchmark is EEMBC-1.1, and all the programs are compiled with the “-O2” optimization effort. The TEAI generation algorithm shown in Fig. 5 is embedded into the CK802 compiler, which originates from GCC.

4.2.2 TEAI insertion threshold

The TEAI insertion threshold should be carefully determined to guarantee that TEAIs are generated before hot instructions with high error rate. A small threshold results in large number of TEAIs, which leads to high code size expansion (Section 4.2.5) and many useless executions of TEAI (maybe performance degradation). A high threshold results in many miss-detected instructions, which prevents TEDPI from achieving the highest throughput as some instructions are corrected by hardware with higher cycle penalty.

TEDPI needs \( \alpha + M + CPI \) cycles to execute a positive errant instruction (\( \alpha \) cycles to execute TEAI, \( M \) cycles to pre-exercise the pipeline), while multi-reissue needs \( CPI + P_e(f) \times ECP_{multi-reissue} \) cycles (Section 4.1). The total cycles needed by TEDPI should be smaller to achieve higher performance. We can get that \( P_e(f) \), or the TEAI insertion threshold, should be higher than \( (\alpha + M)/ECP_{multi-reissue} \). As described in Section 4.2.1, \( M \) in CK802 is one, and \( \alpha \) is usually zero, we can get
that the TEAI insertion threshold should be higher than \( \frac{1}{ECP_{\text{multi-reissue}}} \) (about 15%).

Fig. 6 shows that when increasing the TEAI insertion threshold from \( \frac{1}{ECP_{\text{multi-reissue}}} \) to \( \frac{2}{ECP_{\text{multi-reissue}}} \) at \( F_{\text{Max}} \), the number of TEAIs is reduced by about 50% on average, and the throughput stays almost unchanged; when further increasing the threshold to \( \frac{3}{ECP_{\text{multi-reissue}}} \), the number of TEAIs stays still, but the throughput decreases for some programs. Consequently, the threshold is set to be \( \frac{2}{ECP_{\text{multi-reissue}}} \).

### 4.2.3 Performance improvement

Fig. 7 presents the performance speed-up ratios of TEDPI and multi-reissue against \( F_{\text{signoff}} \). The Ratio is calculated through Eq. (2) when replacing \( F_{\text{POFF}} \) with \( F_{\text{signoff}} \). CPI in Eq. (2) represents the average cycles needed to execute an instruction without errors and TEAIs, and is a constant value during frequency scaling. Fig. 7 shows that 1) TEDPI always improves the performance, while multi-reissue slows down it for several programs; 2) TEDPI has a higher peak performance and a higher \( F_{\text{Opt}} \). The key reason is that \( ECP_{\text{TEDPI}} \) (about 1.3 cycles) is lower than \( ECP_{\text{multi-reissue}} \) (about 7 cycles).

As Fig. 8 shows, the error rate for ip.pkt, canrdr, ttsprk and rspeed increases quickly when \( f \) is increased from \( F_{\text{POFF}} \) to 102 MHz, which causes the error
correction penalty of multi-reissue to be larger than the performance benefit of frequency scaling. So multi-reissue slows down the performance for these programs. Since TEDPI eliminates 99% of the errors with a one-cycle penalty (the miss-detection rate of TEDPI is below 1%), it always improves the performance. The miss-detection refers to errors that TEDPI fails to eliminate, and such errors are corrected by hardware.

We can conclude that a smaller $ECP$ improves the performance speed-up ratio, and pushes the optimal frequency to be higher, which coordinate with the conclusions in Section 4.1.

4.2.4 Energy reduction

Fig. 9 presents that TEDPI consumes less energy than multi-reissue during frequency scaling. The energy consumption is measured by $EPI$ (energy per instruction) in Eq. (6). When $f$ is near $F_{POFF}$, $EPI_{TEAI}$ is higher than $EPI_{multi-reissue}$. The key reason is that TEAIs bring about unnecessary energy consumption when there are no errors (such as instruction fetching and decoding of TEAIs, and redundant clock transitions caused by TEAIs), and this is not needed by multi-reissue. When $f$ is further scaled, the increasing errors bring about many pipeline flushes and instruction replays for multi-reissue, and result in massive energy consumption for error correction. On the other hand, TEAIs avoid most of the errors, and eliminate majority of the pipeline flushes. As a result, $EPI_{TEDPI}$ increases slower than $EPI_{multi-reissue}$ during frequency scaling.
4.2.5 Analysis of TEDPI

This section gives a brief analysis of TEDPI on: 1) the error correction penalty, 2) the code size expansion, 3) the adverse impact on normal program execution without errors. The results are presented in Fig. 10.

(1) Error correction penalty. Fig. 10(a) presents that at $F_{Opt}$, more than half of the programs have a one-cycle error correction penalty (Eq. (7)). The key reason is that error rates for most of the errant instructions are quite high at $F_{Opt}$, and most of the TEAIs are useful in avoiding errors. Since the errant instructions in conven, fbital, ospfv2 and rgbhpg have relative lower error rates (64% for conven, 48% for fbital, 60% for ospfv2 and 62% for viterb), the average error correction penalty for them is higher than 1.5 cycles, but is smaller than the seven cycles of multi-reissue.

$$EPI = \frac{\text{Total energy consumed}}{\text{Number of instructions without errors and TEAIs}}$$ (6)

$$ECP = \frac{\text{Total cycles at } f_{Opt} - \text{Total cycles at } F_{signoff}}{\text{Number of errors at } f_{Max}}$$ (7)

(2) Code size expansion. Fig. 10(b) presents that the code size increase rate (Eq. (8)) caused by TEDPI is no more than 1% due to the locality of errors described in Section 2.1. Since the number of static positive errant instructions in each program is quite small (no more than 40), a 80-Byte space in the instruction memory is enough to store them. However, size of the text section is steadily above 12000 Bytes for all the programs. As a result, the code size increase rate is quite small.

Fig. 10. Analysis of TEDPI.
\[ \text{Code size increase rate} = \frac{\text{Total code size of TEAIs}}{\text{Total size of the text section}} \] (8)

(3) Adverse impact on normal program execution. Fig. 10(c) presents that TEDPI has reduced the performance of CK802 without errors by 4.6% on average. TEDPI statically inserts TEAIs in each program, and these TEAIs have to be executed even if no error happens, which results in performance degradation. However, as the total execution times for TEAIs in each program are quite small, the performance degradation is limited (below 6%).

5 Conclusion

The EDAC technique is an effective way to eliminate timing margins resulting from process, voltage, temperature and aging (PVTA) variations. This paper proposes a cost-effective EDAC approach based on hardware software co-design. TEDPI pre-detects most of the errors through TEPM, and pre-corrects them at compilation time by using TEAI. Both the qualitative analysis through mathematical derivation and the quantitative analysis on CK802 demonstrate the effectiveness and feasibility of TEDPI. In our future work, we will try to generate TEAI at run-time by using online error profiling and dynamic code generation techniques to free our TEDPI from TEPM.

Acknowledgments

This work was supported by the 1) the National 863 Project Key Technology Research on High Efficiency Near Threshold Voltage Integrated Circuit of China (Grant No. 2015AA016601), 2) Key Project of the State Key Lab of ASIC and System Sub-Threshold Voltage Error Tolerance Processor Research (Grant No. 2015ZD005), 3) Shanghai Natural Science Foundation Project Transient Timing Adaptive Low Voltage Processor Research (Grant No. 15ZR1402700), 4) the National Science & Technology Support Program of China (Grant No. 2013-BAH03B01), and 5) the National High Technology Research and Development Program of China (Grant No. 2015AA016704c).