High Performance Fault Tolerance Through Predictive Instruction Re-Execution

Jyothish Soman and Timothy M. Jones

Computer Laboratory, University of Cambridge, {jyothish.soman,timothy.jones}@cl.cam.ac.uk

***Abstract*—Processor designers face the challenge of defect formation, leading to permanent faults, during fabrication and operation. Permanent or hard fault tolerance is an important problem in computing systems, solutions to which can help improve yield during fabrication and reduce the cost of transistor mortality during the service life of the processor.**

This paper presents PreFix, a method to handle hard errors to keep a faulty core running and correctly executing instructions. Instead of turning off faulty structures, PreFix predicts early on whether an instruction is likely to use faulty components, then refines this prediction later in the pipeline to actually detect when an error has occurred. Instructions marked as possibly-faulty in the front-end are queued for duplicate execution on a separate core. At commit, results from the original and duplicate instructions are compared. Upon a mismatch, the original instruction is patched up, the pipeline flushed and execution continues. Using PreFix, faulty components can continue performing useful work when their errors do not manifest in architecturally visible state changes. This enhances processor lifetime with minimal performance overhead.

I. Introduction

Fabrication and operational constraints have led to decreasing reliability and reduced lifetimes for microprocessors [1]. Manufacturing defects, parametric variation and wear-out pose significant reliability challenges [2] across the full life-cycle of a processor, from design and fabrication through to operation in the field. One manifestation of reduced reliability is the formation of hard or permanent faults within the hardware [2], risking the correct execution of applications that run on the processor. Traditional reliability methods do not use the faulty components, and rely on completing the computation elsewhere. For example, Stagenet [3] is a method by which multiple processor pipelines are interleaved to allow for switching faulty parts. In contrast, Necromancer [4] uses faulty (so-called "dead") high-performance cores to accelerate operations on a smaller core. Khan et al. [5] present a method where a hypervisor keeps track of faulty cores and the profile of the threads running in the system. It uses this information to match a core to a thread at runtime. Similarly, a compiler-based method is presented by Meixner and Sorin [6], where code is recompiled so that the faulty hardware is not used. Finally, Rescue [7] presents a method to isolate logic modules, providing for better fault localization.

Other prior work has also shown that at least 30% [8], to up to 65% [9], of faults do not affect the correct execution of a component over millions of cycles. Figure 1 shows the degradation in performance when one of two integer ALUs is faulty, yet allowed to continue operating (but not accepting instructions that would fault), giving an upper bound on performance. When the ALU is faulty for every instruction, there is considerable difference in performance, ranging from a negligible 2% to a much more substantial 30%. When 50% of the instructions pass through, performance reduction is a mere 7%. For the most affected applications, it is vital to keep the ALU functioning, even if it only produces error-free results for a fraction of the time.

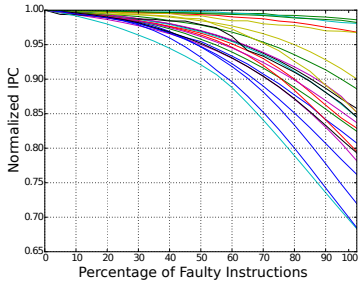


Fig. 1: Performance degradation caused by preventing instructions that would incur errors from passing through a partially faulty ALU.

Our solution, named PreFix, aims to provide a low-overhead error detection and correction technique for tolerating hard faults. PreFix is a method to allow faulty cores to continue correct and high performance execution, despite containing faults, by predicting and verifying the possibility of error on each instruction passing through the core. Instead of turning off faulty structures, it isolates faults based on the results from instructions that use these circuits. We conservatively predict the instructions that will have erroneous results and duplicate their execution on a different core, and correct the results if the errors manifest. In the event that the prediction is wrong, we simply lose performance from stalling these instructions while waiting for their duplicate results. PreFix brings together fault detection techniques with redundant execution on a healthy core to both detect and correct permanent errors. Overall, it enables continued use of faulty components, allowing them to perform useful computation when faults do not propagate.

II. PreFix

PreFix consists of at least two cores: cores with one or more faults are denoted as the Faulty Cores (FC); the rest are healthy and we call them as the Remote Cores (RC). The RC is responsible for re-executing instructions that have been marked as possibly erroneous on the FC. We augment each core with an error prediction unit that gives an indication of

978-1-5386-0362-8/17/1.00

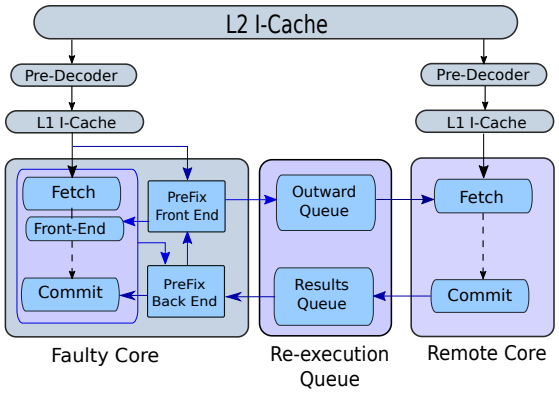


Fig. 2: PreFix overview showing CPU pipeline integration.

whether each instruction is likely to produce an erroneous output or not. PreFix duplicates those that are, placing some in a holding queue and sends the rest to a central queue for duplicate execution on the RC; their results are then placed back in a central queue. PreFix is intended for handling faults in the core's data and control logic, but not for those in buffers. An overview of PreFix is shown in fig. 2. We next describe how instructions flow through the processor pipeline and then give details of PreFix.

A. Instruction Flow

Instructions first interact with PreFix within the pre-decoder that sits between the L1 instruction and L2 caches. The pre-decoder identifies resources that each instruction requires and stores that within the L1 instruction cache, for use in a later stage. When instructions are fetched into the core, they simultaneously pass through the fault prediction unit. This unit uses the information from the pre-decoder and the instruction itself to determine whether the instruction is likely to execute correctly within the core. If an error is possible, then a copy is placed into a holding queue for duplicate execution on the RC. If the instruction cannot be handled by the FC, then it is forwarded immediately; other instructions are held in the queue until the PreFix back-end informs it of possible faults.

Meanwhile, all instructions enter the faulty core's fetch queue and pass as normal through the core's pipeline. Fingerprinting logic monitors execution and use of resources, flagging up an error if an instruction does not produce the correct result. At the commit stage, a corrector module uses the outputs from the fingerprinting, and only allows instructions with the correct result to commit and leave the pipeline. Those with errors are replaced with the results from the duplicate instruction on the re-execution queue; the pipeline is flushed and fetch restarts with the next instruction.

B. Pre-Decoder

The pre-decoder is responsible for extracting early, high-level information from each instruction that enters the instruction cache. Many modern processors contain pre-decoders at this level [10] to reduce the amount of repeated work that the pipeline's decode stage must perform. Every instruction is pre-decoded to extract information about resources required during its traversal through the pipeline.

1) PreFix Frontend: Pre-decoded instructions enter the PreFix front-end in parallel with being sent to the fetch stage. The structures that make up the front-end are shown in fig. 3.

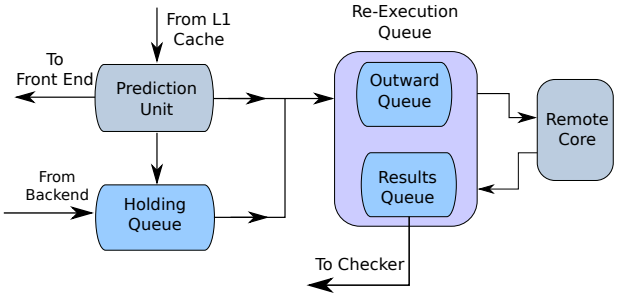


Fig. 3: PreFix front-end showing predictor and the queues.

a) PreFix Predictor: The primary task of the predictor is to classify each instruction into one of the three categories: not faulty (NF), highly likely to fault (HLF, the default), or low likelihood of fault (LLF). The predictor is necessarily conservative; it generates false positives but never says an instruction is fault-free when it isn't, using hardware fault trees. The predictor further contains logic which calculates when an instruction will use a faulty pipeline lane, to deal with errors in specific fetch or decode units. NF and LLF instructions pass through the core's pipeline. LLF and HLF instructions are duplicated, with the HLF duplicates immediately sent to the RC. LLF instructions are only re-executed if the later detectors catch an error. The stream of HLF and LLF instructions are written into the re-execution queue, from which they leave in program order only when their original counterparts commit or are squashed in the main core. The re-execution queue is dual channeled, one sending instructions and operands to the RC (the outward queue) and the other receiving results from the RC, if and when that occurs (the results queue).

b) Fault Trees: The fault tree in our method works over the ISA. It groups instructions by resource usage (i.e., core structures) and predicts whether instructions from each group might use faulty components as they pass through the processor pipeline. Pre-decoders support ISA-based component analysis allowing the creation of fault trees where groups are based on ISA-level characteristics. At the head of the tree, all instructions are part of the super-group, that is the group of instructions using the processor. Further down the tree, the instructions are split into more specialized groups, for example, based on the specific type of functional unit they will use. As the tree becomes larger, the nodes start representing internal circuitry, such as an operation's bit width. Hardware represents the tree in its flattened form as a bit array with the ability of PreFix to detect the fault also stored. Information from the pre-decoder regarding the instruction class and the resource requirements are used to query the fault tree, which is populated using built-in self-test [11].

c) Duplicate Execution: The RC executes duplicate instructions alongside any workload it has to run, using an otherwise-idle redundant thread. The redundant thread obtains duplicate instructions from the re-execution queue and executes them when the RC allows. In each fetch cycle, the RC either fetches from its main thread or the redundant secondary thread (if it has work to do). The RC favors its main thread, giving it more fetch cycles than its redundant counterpart. In our experiments, fetch occurs from the redundant secondary thread only when the primary thread is inactive while waiting for either data or instruction from memory. Instructions marked

as ready in the re-execution queue contain not just their original instruction bits, but also their source operands. This means that the redundant secondary thread is free to execute each instruction in isolation, asynchronously to the original faulty core. Redundant secondary threads are non-speculative in the RC since they are independent of all other instructions and do not use the branch predictor. The results from this duplicate execution are available at commit and are written into PreFix's re-execution queue, for reading by the checker unit if the original instruction actually does experience an error.

2) PreFix Back-end: The PreFix front-end is concerned with predicting whether a fault may occur with each instruction whereas the back-end is responsible for detecting whether an error has actually occurred.

a) PreFix Fault Detector: The PreFix fault detector is responsible for detecting whether an error may have occurred. If so, it communicates with the front-end to ensure that the duplicate of the faulty instruction actually gets executed on the RC. Note that the detection need not be perfect, and over-prediction is acceptable with performance penalties. The PreFix back-end contains both a usage monitor and multiple detectors. The usage monitor runs in parallel with instruction issue and checks instructions that were marked as possibly faulty by the front-end to see if they will actually use any faulty components. If not, then it reclassifies the instruction as NF. This component helps reduce the overheads of PreFix by removing false positives introduced by the front-end. It also serves as a backup to the detector. The usage monitor is responsible for filtering instructions that require checking and the detector is responsible for detecting faults in marked instructions (those classified LLF). Detector designs have been previously proposed, for example, using parity checking [12], and PreFix can work with any of these methods. Instructions passed by the usage monitor are placed in a detector queue, pending the results of the detector. From here they are are either discarded (if no fault is detected) or, in the event of a detected error, sent to the holding queue in the PreFix front-end to ensure they are re-executed on the RC.

In addition, Mitra and McCluskey [13] show that concurrent error detection methods themselves may be subject to errors. Inclusion of the usage monitor, therefore, protects against scenarios in which the detector as well as the actual circuit have complementary errors. Where the detector is itself faulty for certain errors, or does not provide complete coverage, the usage monitor's filtering alone is used to determine whether to re-execute the instruction. In these scenarios, false positives can occur from the PreFix back-end.

b) Corrector Unit: For each instruction that is still marked as an LLF, the corrector is responsible for checking if the results from both executions match. It sits at the end of the pipeline, alongside commit and is responsible for ensuring that instructions with erroneous results do not leave the pipeline and so do not contribute to the architectural state. Instructions that have been marked as faulty by the PreFix back-end after their execution are intercepted by the corrector unit and prevented from committing until they have been validated. To accomplish this, the corrector unit queries the instructions at the head of the holding queue to find the duplicate of the erroneous instruction. If this has already been executed on the remote core through the re-execution queue and redundant secondary thread, then the values from the remote execution are retrieved. On the other hand, if duplicate execution has not yet finished for this instruction, the corrector unit stalls until the remote results come back.

Processor	1GHz, 3-wide, out-of-order
ROB	40 entries
L/S Queues	16 / 16 entries
Issue Queue	32 entries
Registers	128 integer, 128 FP
ALUs	3 Int, 2 FP, 1 Mult/Div
Branch Pred.	Tournament with 2048 entry local, 8192 entry global, 2048 entry chooser, 2048 entry BTB, and 16 entry RAS
L1 Caches	32KiB, 2-way, 64B lines, 2-cycle hit
L2 Cache	2MiB, 8-way, 64B lines, 12-cycle hit
Main Memory	DDR3-1600 11-11-11-28 @ 800MHz

TABLE I: Experimental setup for cores and memory.

1	astar	sjeng	2	bwaves	pds50
3	bzip2	tonto	4	cactusADM	tonto
5	calculix	zeusmp	6	gamess	soplex
7	gcc	bwaves	8	gobmk	cactusADM
9	gromacs	calculix	10	h264ref	gamess
11	hmmer	gcc	12	libquantum	gobmk
13	milc	h264ref	14	namd	hmmer
15	perlbench	libquantum	16	sjeng	mcf

TABLE II: Randomly-selected pairs of benchmarks studied.

As section II-B2a described, the PreFix back-end can generate false positives. To avoid a loss of performance for these false positives, the corrector unit does not assume that a fault has actually occurred. Instead, it compares the results of remote execution with those from the faulty instruction to actually determine the instruction's fault status. If they are the same, then the faulty instruction commits as normal. If they differ, then the results from the remote execution are accepted as correct and copied into the faulty instruction's output registers. The corrector unit stalls the processor, and squashes all later in-flight instructions to avoid subsequent errors from dependent instructions reading the wrong value. At this point, as with a branch mis-prediction, the instructions in the holding and re-execution queues are also squashed. Any instructions currently being re-executed on the remote core are ignored when they finish. Execution on the remote core is independent and asynchronous to that on the faulty core, hence there is no interaction between the two.

III. Experimental Setup

We evaluated PreFix using the gem5 simulator [14] using the ARMv7-A ISA and randomly-selected pairings of applications drawn from the SPEC CPU2006 benchmark suite as shown in table II. The out-of-order cores have private L1 caches and a shared L2. Table I details the core and memory configuration. We compiled each benchmark with gcc 5.2; missing applications would not compile or run correctly in our environment. For each experiment, we fast forwarded and warmed the caches and branch predictor for each benchmark pairs for 500 million instructions and then executed for at least a further 250 million instructions. The weighted speed-up of the IPC of the main threads is taken as the performance indicator. To allow for a viable comparison, the base case is taken as the error free multi-core case. 50 faulty versions of the first core, each one containing exactly 5 errors in different components are used. For benchmarking experiments, faults

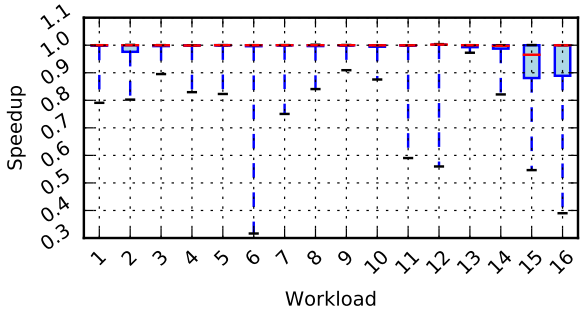
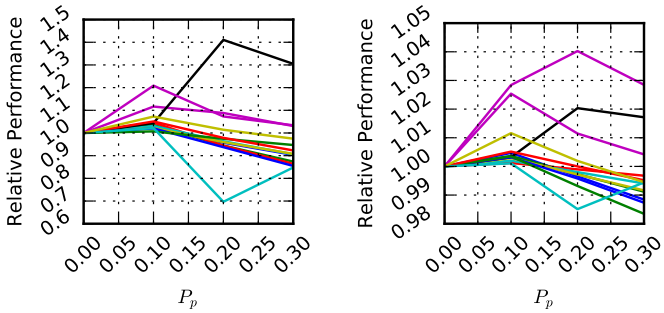


Fig. 4: Results with full PreFix. Frequent errors cause significant slowdowns, but median performance shows little impact.



(a) Faulty Core Performance (b) Dual Core Performance

Fig. 5: Effect of varying the prediction rate on a 2 core system

had a 20% chance of affecting the result of each instruction that used the faulty component.

IV. Results

a) Performance: Figure 4 shows the results of PreFix when the complete system is functional for each workload across all 50 erroneous systems. The x-axis gives the workload number from table II, the y-axis shows normalized performance and we plot the minimum, maximum, median, 25th and 75th percentiles of the distribution across erroneous systems. For most workloads, such as 3, 4, & 5, there is little impact from running on a faulty core with the median performance at 1×, and few outliers as shown in fig. 4. Some workloads present noticeable degradation in certain error classes for certain benchmark pairs. For example, error class 13 causes significant performance degradation in workload 6, but shows relatively less performance degradation in other benchmark pairs. Also, workload 6 shows substantial resilience to other error classes. This clearly shows that performance degradation is related to the error and benchmark pair.

However, most workloads experience a range of slowdowns depending on the types of faults in the simulated systems. The worst performance is 0.3× on workload 6 which is due to a core with faults exclusively in the integer ALU. In contrast to the workloads that are barely affected, in this case both benchmarks have high baseline IPC. Frequent erroneous instructions reduce the IPC of the first workload (on the faulty core) because it must stall at commit to wait for instruction re-execution. Further, these additional instructions reduce the IPC of the second workload (on the remote core) because it does not get the full fetch capacity and cannot tolerate this reduction in bandwidth.

b) Prediction: Figure 5 shows the impact of prediction on the system, where higher performance is better. Initially there is an improvement in the performance with increasing prediction rates, as expected, but the performance starts to degrade as over-prediction starts increasing the number of instructions classified as HLF. Most workloads have an optimal prediction rate of 0.1 and three benchmark pairings have the optimal prediction rate at 0.2. In the experiments shown previously, a prediction rate of 0.2 was used.

c) Area and Power Overhead: Using McPAT [15], we obtained area and power estimates for the PreFix framework which gave an overhead of 3.5% on a 2 core machine, and for a 4 core machine, the overhead decreases to 3.1% of the total processor area, excluding the caches. For power, the overhead varies from 1-4% based on the application-fault profile.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grants EP/K026399/1 and EP/J016284/1. Experiments used the Darwin Supercomputer of the University of Cambridge HPC Service funded by the Higher Education Funding Council for England and the Science and Technology Facilities Council. Additional data related to this publication is available at `https://doi.org/10.17863/CAM.11957`.

References

[1] D. Gizopoulos, M. Psarakis, and E. al, "Architectures for online error detection and recovery in multicore processors," in *DATE*, 2011.

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, 2005.

[3] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "The StageNet Fabric for Constructing Resilient Multicore Systems," 2008.

[4] A. Ansari *et al.*, "Necromancer: Enhancing System Throughput by Animating Dead Cores," in *ACM Computer Architecture News*, 2010.

[5] O. Khan, "Thread Relocation: A Runtime Architecture for Tolerating Hard Errors in Chip Multiprocessors," *IEEE Trans. on Computers*, 2010.

[6] A. Meixner and D. J. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *DSN*, 2008.

[7] E. Schuchman *et al.*, "Rescue: A microarchitecture for testability and defect tolerance," in *ACM Computer Architecture News*, 2005.

[8] M.-L. Li *et al.*, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *HPCA*, 2009.

[9] K. Nepal, N. Alves, J. Dworak, and R. I. Bahar, "Using Implications for Online Error Detection," in *ITC*, 2008.

[10] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," vol. 16, no. 2, pp. 28–41, 1996.

[11] V. Iyengar, K. Chakrabarty, and B. T. Murray, "Built-in self testing of sequential circuits using precomputed test sets," 1998.

[12] M. Nicolaidis, "Carry checking/parity prediction adders and ALUs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2003.

[13] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?" in *ITC*, 2000.

[14] N. Binkert, B. Et al, B. Beckmann, G. Black, Others, and B. Et al, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[15] S. Li *et al.*, "McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architectures," *, 2009. Micro-42.*