REPAIR: Hard-Error Recovery via Re-Execution

Jyothish Soman, Negar Miralaei, Alan Mycroft and Timothy M. Jones
Computer Laboratory, University of Cambridge, Email: firstname.lastname@cl.cam.ac.uk

***Abstract*—Processor reliability at upcoming technology nodes presents significant challenges to designers from increased manufacturing variability, parametric variation and transistor wearout leading to permanent faults. We present a design to tolerate this impact at the microarchitectural level—a chip with *n* cores together with one or more shared instruction re-execution units (IRUs). Instructions using a faulty component are identified and re-executed on an IRU. This design incurs no slowdown in the absence of errors and allows continued operation of all *n* cores after multiple hard errors on one or all cores in the structures protected by our scheme. Experiments show that a single-core chip experiences only a 23% slowdown with 1 error, rising to 43% in the presence of 5 errors. In a 4-core scenario with 4 errors on every core and a shared IRU, REPAIR enables performance of 0.68× of a fully functioning system.**

I. Introduction

Relentless technology scaling has led to device fabrication processes approaching the single-digit nanometre range. However, design for reliability has been unable to catch up [6], leading to significant challenges for processor designers such as wearout, parametric variation and manufacturing defects [2]. Each of these affect the lifetime and operability of the processor. Hard or permanent errors can manifest themselves across the full life-cycle of a processor, starting from design and fabrication to operation, potentially affecting correct program execution.

Dealing with permanent faults after leaving the fab is a complex matter. Traditional reliability solutions involving high-cost redundant spares are now unacceptable [10]. Prior work has considered using natural architectural redundancy to address this [14], but some components (e.g., often the divider unit) are one-per-core. On the other hand, spare components can be added for use in the event of an error [3], but this requires adding redundancy into each processor structure, in each core within the chip, when it is unlikely that it will all be used. In addition, both methods require the design of per-structure error-correction logic with associated implementation and verification costs.

On the other hand, schemes such as architectural core salvaging [11] use the natural redundancy across cores to avoid structures with defects. StageNet [7] also leverages architectural redundancy by providing a network of pipeline stages that can be configured to work around faulty stages, creating logical cores that are distributed about the chip. Similarly Romanescu and Sorin [6] present a scheme to cannibalize cores at pipeline granularity, arranging cores into groups so that spare pipeline stages can be lent to nearby cores. Rodrigues et al. [12] present a method to re-execute the faulty instruction in the same core using idle cycles to correct errors in the functional units.

Runtime methods to detect errors typically re-execute the instruction either locally or on an external unit. DIVA [6] incorporates a functional checker at the commit stage of a superscalar pipeline. Results from computations on the main core are sent to the DIVA checker for comparison with the values generated there, and differences flagged up as an exception that can be handled by flushing and restarting. BlackJack [13] runs duplicate threads on a single SMT core. In both methods, providing a second execution of a program allows the comparison of application state at key points and differences discovered. Thread relocation [8] uses a hypervisor-based system to handle errors by mapping software to appropriate cores for execution.

Finally, recent work has considered approximate compute where errors are allowed to occur and tolerated rather than corrected [5], [9]. Although this is suitable for a certain class of application, the majority of programs require exact computation and cannot survive errors in the underlying fabric.

Our method takes a different approach by keeping faulty cores running and performing real computation, even when there is no natural redundancy in the faulty microarchitectural structures. We augment our system with a small component containing only functional units and buffers that we call an instruction re-execution unit (IRU). Fault maps on each core, initialized by power-on self-test or periodic built-in self-test, record faulty components within the core. We monitor each instruction's resource usage as it traverses through the pipeline and re-execute it on the IRU if it touches a component marked as faulty. This requires minimal modification to a standard out-of-order superscalar pipeline and, additionally, imposes no performance penalty unless and until a hard error occurs. In this way we can tolerate a large number of errors in different components within each faulty core, yet still use that core to generate correct results. We call our technique REPAIR because it provides "***R**ecovery from **E**rrors in **P**rocessors by **A**llowing **I**nstruction **R**e-execution*".

We evaluate REPAIR as an addition to a single-core system, and as a shared resource within a multicore chip, showing that the expected performance for a single core is $0.81\times$ peak performance and for multicore with four errors in every core is $0.68\times$. Compared to existing techniques, REPAIR a) provides coverage for components with no natural redundancy; b) removes the need to implement complex per-structure error-recovering logic; and c) provides a single resource for error correction, amortizing the overheads across all components that make use of it. REPAIR of itself only provides protection for the ALU, register and pipelining components of a core; it could be complemented by structural redundancy to provide full hard-error coverage.

II. REPAIR

The addition of REPAIR to a multicore is sufficient to allow it to survive in the presence of errors, and to allow faulty components to continue functioning by targeted re-execution of possibly faulty instructions. The added redundancy is shared across multiple cores, meaning low hardware overhead.

REPAIR adds a per-core detection/correction unit (DCU) and a shared instruction re-execution unit (IRU) to the traditional core structure. The DCU interacts with the processor at the dispatch and commit stages, and also queries the TLB for the IRU. One DCU is required per core, but multiple cores can share one IRU.

1) Detection / Correction Unit and Processor Interfacing: REPAIR identifies instructions that will use or have used faulty processor structures through the use of fault maps. These indicate the processor structures that have hard errors within them and are populated via power-on self-test, or through built-in self-test (e.g., [4]). A fault map is a simple array of bits for each structure where each bit represents the presence of a fault in one entry of that component.

Figure 1 gives an overview of how the DCU differentiates between dispatch-based checking and commit-based checking in the superscalar pipeline. For every instruction passing through the processor pipeline, the DCU collects the destination register, ROB entry, LSQ entries, register scoreboard and issue queue entry related to the instruction. If any of these is found to be faulty as the instruction dispatches then it, and all later instructions, are stalled and the pipeline is allowed to drain of older instructions. For the faulting instruction (FI), its entries in the reorder buffer and other queues (and those of any younger instructions that have dispatched in the same cycle) are annulled to avoid them being executed erroneously. Once the pipeline is empty (all earlier instructions have been executed and no event was caused by them which would lead to FI being squashed), the instruction is sent to the IRU, with all its dependent values, executed there and results are brought back. The processor then commits FI and resumes operation by dispatching the next instructions. If an instruction is broken into a series of micro-ops during decode, only the faulting micro-op needs sending to the IRU.

The DCU also checks instructions at commit, to detect usage of faulty components in the ALU, LSU and register file ports (usage of which could not be predicted before). If the instruction is found to have used faulty components, then commit blocks and the FI are re-executed on the IRU. In this case, the values generated by the IRU and core are compared. If the same, then commit proceeds; on a mismatch, all later instructions are squashed and the IRU result is committed. In contrast to catching errors at dispatch, when we detect a micro-op that is faulting at commit, we cannot send it by itself to the IRU without adding significant complexity to the decode engine to allow us to restart within a macro-instruction. To avoid this, we simply send all younger micro-ops from the same macro-instruction to the IRU and, once complete, fetch can start again with the next macro-instruction.

REPAIR expects valid, decoded instructions to be presented to the rename stage, from where it can take over. Faults in

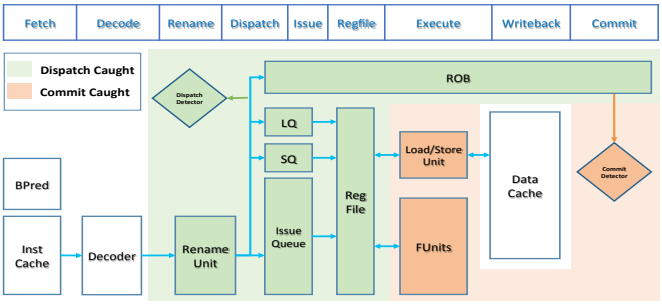


Fig. 1: Detection stages within a core.

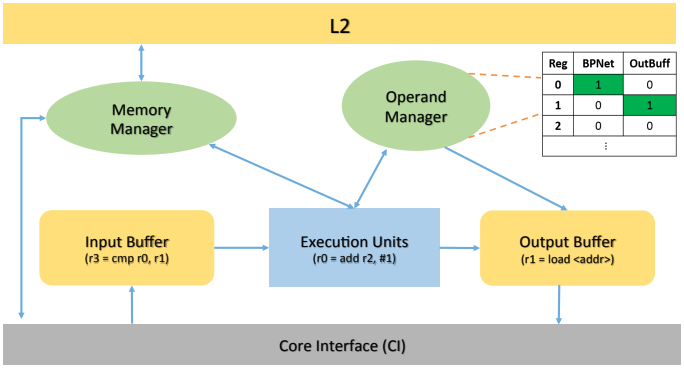


Fig. 2: Overview of the IRU.

REPAIR can be tolerated provided they do not affect in error correction, for example, if cores have faults in their load units and REPAIR has an error in the IRU's ALU.

2) Instruction Re-execution Unit: The IRU is a simple circuit capable of only executing instructions and accessing the L2 cache. It is capable of storing a limited number of previous results internally, allowing it to execute consecutive instructions from the same core back-to-back. The basic IRU is shown in Figure 2 and consists of an interface to the main cores, input and output buffers, execution units, an operand manager, to deal with dependences between instructions, and a memory manager, to perform loads and stores. The execute unit of the IRU contains one copy of each type of functional unit appearing in a standard core.

The IRU only executes instructions from a single core at any point of time. The core interface (CI) is responsible for managing order and grouping of instructions coming from the main cores. The CI also arbitrates (round robin), when more than one core wishes to use the IRU. The CI also requests TLB translations from the DCU of the appropriate core when load or store instructions require re-execution.

The CI writes instructions and data into the input buffer, from which they are executed in order. If memory accesses are required, the memory manager requests and completes address translation from the CI, and then accesses the L2 cache. The IRU is viewed as a cacheless core for cache coherence. Results are placed in the output buffer, along with the destination architectural register ID. Storing the register ID allows coalescing of multiple writes to the same register, reducing the size of data to be sent back to the originating core. Although the IRU can execute any number of instructions

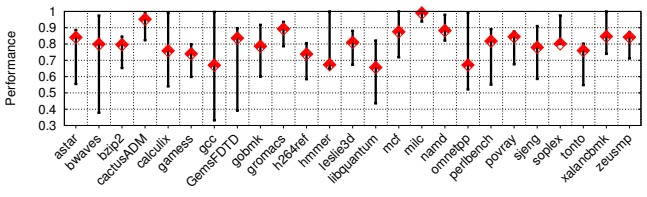
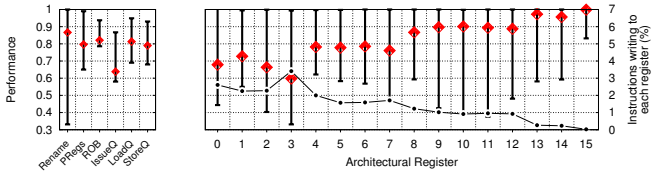


Fig. 3: Performance of REPAIR across single-core systems, each with a single error.



(a) Arrays (b) Rename map

Fig. 4: Performance across different architectural arrays and within the rename map; also the fraction of instructions committed using each architectural register as a destination.

from a core, we only use it with sequences of micro-ops from a single macro-instruction at any one time.

REPAIR also maintains precise exceptions by marking any instruction that causes an exception within the output buffer. Once this instruction has been transferred back to the core, exception handling deals with the issue as normal.

III. Results

We evaluated REPAIR using the gem5 simulator [1] running the ARMv7-A ISA. Our cores have out-of-order superscalar pipelines, and resemble the Cortex-A15. Each core has a private 32KB L1 data cache and 32KB L1 instruction cache. There is a 1MB L2 cache for the single-core experiments or 2MB shared L2 for the multicore simulations. Our benchmarks are taken from the SPEC CPU2006 suite and compiled with gcc 4.6.3. We used all benchmarks from this suite, apart from *dealII*, *lbm*, *sphinx3* and *wrf* which did not compile correctly for our environment. For the 4-core experiments we created 20 workloads (G1 ...G20) each consisting of four benchmarks to be run concurrently. The benchmarks were uniformly distributed across the 20 workloads. Benchmarks are run for a total of 250 million instructions (62.5 million per core for multi-core), after fast-forwarding for 1 billion instructions and warming the cache and branch predictor for 100 million instructions. We use weighted speedup as a performance metric. In the results presented in Figures 3-6, maximum, median and minimum performance are show for every benchmark/grouping.

Our experiments require us to execute on cores that have errors. The size of our design space (approximately 250,000 single-core configurations with 2 errors, 125M with 3 errors, etc.) meant that we could not exhaustively simulate every point within it. Therefore we first randomly created 50 single-core systems each with a single unique fault, which represents 10% of our single-error space. We then created single-core systems with 2 errors by randomly adding faults to the single-fault systems, and likewise for 3, 4 and 5 errors. Our faulty multicore systems were created in the same manner.

1) Single-Core REPAIR: REPAIR can efficiently tolerate single-bit errors in the core with an average performance of just 0.81×. Figure 3 shows the results for our 50 single-core systems, each with a randomly placed (within coverage area) single-bit error. The results show a performance range of 0.33× to 1.00× with an average performance of 0.81×, corresponding to a slowdown of around 23%. The maximum performance of 1.00× is achieved when there is an error in a component that is unused by a particular benchmark (e.g., a floating point register). The worst performance occurs within *gcc* when there is an error in the rename map entry for architectural register 3, which is used as destination register for 13.5% of all instructions executed in this experiment.

The application *milc* is of interest in that it shows the least performance degradation in the presence of errors, across experiments, with the worst performance being only 0.94×. This is explained by *milc* experiencing significant pipeline stalls due to L2 cache misses. In fact its baseline CPI is 2.60 and although in the worst case it suffers over 14 errors per thousand instruction, its overall CPI increases to just 2.69.

a) Differences Between Arrays: Errors in different architectural arrays yield a range of slowdowns as shown in Figure 4a.Within the queues, ROB and physical registers, variability in performance comes from the differences between applications, and not from the position of the error, since each entry within these arrays is equally likely to be written to. However, within the rename map the performance variability comes from the position of the error and application behavior, since benchmarks do not write to each architectural register equally. This means that the rename map has the highest variability, although the median performance is 0.87×. The issue queue has the worst median performance of 0.64× because it is small (only 32 entries) and is used by every instruction. Errors in the first eight registers impact performance more significantly than the remainder. This is an artifact of the ARM architecture, in particular the Thumb-2 16-bit instructions which can only directly access these first eight registers (and are hence compiler-favored). Again the highest performance loss comes from a fault in register 3, which is due to gcc's register allocation favouring register 3.

b) Number of Errors: Figure 5 shows the median performance as the number of errors within the core grows up to 5, where average performance over all benchmarks is 0.70×, or a 43% slowdown. Some applications, such as *milc*, are barely affected by the increased work that REPAIR must perform to keep the core functioning correctly. Others incur a more substantial performance impact (e.g., *GemsFDTD* with a drop from 0.85× to 0.67×). Unfortunately, *GemsFDTD* has a very low baseline CPI and therefore experiences a drop in performance with each new error that is introduced, whereas other applications (e.g., *mcf* which has a high number of L2 misses) have a lower initial CPI and can better absorb the performance impact of faults.

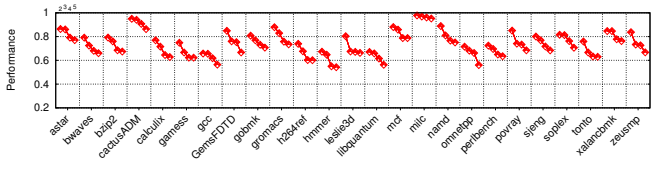
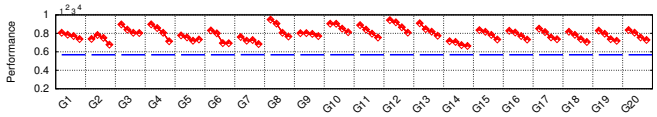
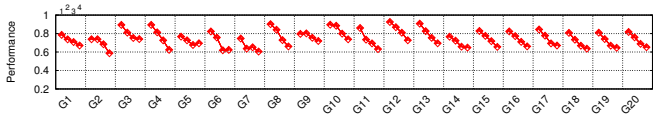


Fig. 5: Single core performance with increasing errors.



(a) Two faulty cores



(b) Four faulty cores

Fig. 6: Performance on a 4-core system as the number of errors per core increases. Also shown is the performance when the faulty core is simply switched off and the scheduler avoids it.

2) Multicore REPAIR: We now show the impact of adding REPAIR to a cluster of four cores, sharing a single IRU. For comparison, we implemented a time-sliced scheduler to distribute benchmarks across healthy cores, simulating faulty cores as switched off. Our basic scheduler uses a quantum of 1 ms and does not actually move programs around but pauses each core when it is "off". This means the L1 and branch predictor remain warm and there is no overhead to scheduling, slightly favouring this comparison scheme over an actual implementation.

Figure 6 shows the results of using REPAIR on this system with 2 or 4 erroneous cores, as the number of errors per core increases. Our comparison scheduler is also shown, although it is missing in Figure 10(b) because with 4 faulty cores and no REPAIR, the whole system would have to be turned off! As in the previous section, more errors leads to worse performance, but still REPAIR is able to keep the system functioning without significant slowdowns. With a single error in two cores, performance is, on average, just 0.86×, dropping to just 0.83× when these two cores have four errors. Were REPAIR not present, the two faulty cores would have to be turned off and our scheduling scheme used, which achieves an average of 0.59× performance (higher than the expected value of 0.5×). When all four cores contain a single error, average performance is 0.84×, dropping to 0.68× (or a slowdown of 47%) when each of the four cores contains four errors.

Intuitively, we would expect that performance degrades monotonically as the number of errors increases. However various groups (e.g. G2 running on two faulty cores) gives performances of 0.74×, 0.78×, 0.75× and 0.68× when these cores have 1, 2, 3 and 4 errors respectively. The increase in performance of the two-error scenario is because the second core, in the baseline, is cache-unfriendly—it uses a large fraction of the L2. When core 2 is faulty, the frequency of its L2 accesses is reduced and so the other applications make better use of the shared L2, raising overall performance.

IV. Conclusion

We have proposed the REPAIR architecture to extend the useful life of a processor chip in the presence of an increasing number of hard errors (and to give graceful performance degradation during this process). It allows standard processor cores to continue to operate in the presence of faults, simply routing potentially incorrectly executed instructions to a instruction re-execution unit to be re-run. Results show a four-core processor with 4 errors in every core has a performance of 0.68× a fully functioning system. We conclude that the REPAIR architecture is an effective technique for extending the life of a processor chip with no performance impact in the absence of errors.

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 High Performance Computing Service (http://www.hpc.cam.ac.uk/) 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://www.repository.cam.ac.uk/handle/1810/249207.

References

[1] N. Binkert and B. et al, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

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

[3] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, "Tolerating hard faults in microprocessor array structures," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 51–60.

[4] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation," in *Proceedings of MICRO*, 2007.

[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proceedings of ASPLOS*, 2012.

[6] D. Gizopoulos, M. Psarakis, S. Adve, P. Ramachandran, S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *DATE, 2011*.

[7] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "The StageNet fabric for constructing resilient multicore systems," in *Proceedings of MICRO*, 2008.

[8] O. Khan and S. Kundu, "Thread relocation: A runtime architecture for tolerating hard errors in chip multiprocessors," *IEEE Transactions on Computers*, 2010.

[9] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Proceedings of DATE*, 2010.

[10] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *Proceedings of HPCA*, 2009.

[11] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proceedings of ISCA*, 2009.

[12] R. Rodrigues, A. Annamalai, and S. Kundu, "A low-power instruction replay mechanism for design of resilient microprocessors," *ACM Trans. Embed. Comput. Syst.*

[13] E. Schuchman and T. N. Vijaykumar, "BlackJack: Hard error detection with redundant threads on SMT," in *Proceedings of DSN*, 2007.

[14] ——, "Rescue: A microarchitecture for testability and defect tolerance," in *ACM SIGARCH Computer Architecture News*, 2005.