

# Checker Backend for Soft and Timing Error Detection and Recovery

X. Vera, J. Abella, O. Unsal, A. González, O. Ergin

Intel Barcelona Research Center

Intel Labs - Universitat Politècnica de Catalunya

Barcelona, Spain

{xavier.vera, jaumex.abella, osmanx.unsal, antonio.gonzalez, oguzx.ergin}@intel.com

**Abstract**—Current microprocessors are becoming more vulnerable to cosmic particle strikes and parameter variations. Particle strikes may cause soft (transient) errors, whereas high variability (due to process, temperature and voltage) may transform non-critical paths into critical paths, resulting in timing errors. This paper proposes a design that exploits the benefits of clustering for detecting and recovering from soft and timing errors in the backend. We propose to use some of the regular backend-clusters for error detection and correction, which would work as a checker backend(s).

## I. INTRODUCTION

Alpha particles released by radioactive impurities and neutrons coming from outer space are known to cause transient errors in contemporary microprocessors [1]. Single and multi-bit upsets may arise when these particles hit intermediate capacitive nodes of processor storage components such as SRAM bitcells and latches. Since these transient errors occur due to an incorrect charge or discharge of an intermediate capacitive node, they do not cause permanent failure in the hardware and hence are termed soft errors in the literature. On the other hand, systematic and random variations in process, supply voltage, and temperature are posing a major challenge to future microprocessor design [2], [3]. Variability is forcing designers to leave a tremendous amount of performance on the table through excessive over-design in order to avoid timing errors.

This paper proposes a design that exploits the benefits of clustering for detecting and recovering from soft and timing errors. Clustering allows splitting up the processor in small processing units (clusters), which may reduce per cluster variations. Accordingly, Marculescu and Talpes [4] compare a synchronous processor and its GALS [5] counterpart and show that the impact of process variability translates into 2-10% faster local clocks for the GALS case. We propose to use some of the regular backends for error detection and correction, which would work as a checker backend(s). For presentational ease, we consider a 3-backend architecture, where all backends are identical (although this design can be applied to any clustered architecture) and one may be used as a checker. Since each backend has its own hardware, a checker backend will detect timing errors since variations will be different in each of them. A key feature of this new microarchitecture is that the checking capability can be disabled at any time; the

checker backend may be used for increased performance with the other ones, either turned off for saving power or used for cluster hopping for temperature reasons.

## II. RELATED WORK

One well-known design that addresses the problem of detecting soft errors is simultaneous redundant multithreading (SRT) [6]–[8]; however it cannot detect timing errors since it employs the same hardware for executing and checking. Additionally, multithreading capabilities are required.

Replicating parts of the core have been also explored. DIVA [9] uses a simple in-order core as a checker for an out-of-order core. It has to design the checker from scratch since the objective is to catch design errors of a complex design that cannot be verified at design time. Whereas it may detect timing errors if the checker is assumed to be always correct, it cannot detect soft errors on the checker itself. Besides, its small bandwidth may be a performance bottleneck for the out-of-order core. The IBM G5 [10] replicates the frontend and the execution engine, and all instructions are executed twice in parallel. By comparing the output of the instructions, it detects errors. In order to recover from errors, it keeps a copy of the register file. The authors report an increase in total area of 35% due to fault-tolerant techniques. Both the DIVA checker and the fault tolerant techniques used in the IBM G5 cannot be used for performance.

Our implementation has the advantages from previous SRT mechanisms and DIVA-like processors, whereas it avoids the main drawbacks. The main differences are that (i) it uses the already existing backends, thus no special design is needed, (ii) it can detect timing errors due to variations since instructions are checked in a different hardware than where they were executed, (iii) the performance loss is shown to be very small (below 5% for a 2-regular and 1 checker backend architecture), and (iv) fault tolerance may be disabled at any time, thus traded for throughput, temperature or power. Other differences of our approach with respect to SRT are that (v) it can recover from most of the errors whereas SRT only detects them, (vi) it reexecutes only committed instructions, (vii) threads in SRT require synchronization, and (viii) an SMT processor is not required.

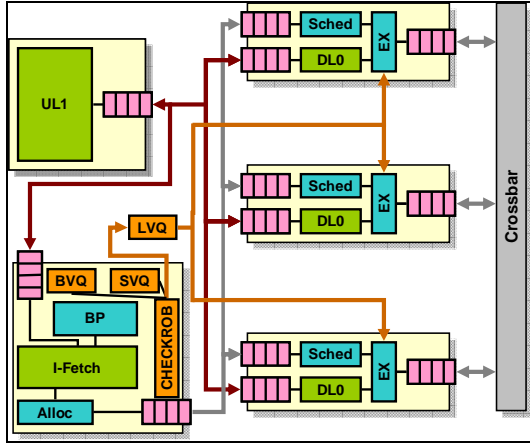


Fig. 1. Microarchitecture.

### III. DESIGN OF A FAULT TOLERANT MICROARCHITECTURE

#### A. Overview

Single event upsets (SEU) can cause single or multiple bit-flips on the pipeline registers and transients faults on the combinational circuits. However, timing variations may only cause transient faults on the logic. A solution to this problem is adding redundancy (e.g., duplicating) high-level components and then compare the outputs [9], [10]. Despite its obvious benefits, the idea has been avoided for years because it has a significant cost in area. Instead, we propose to use the *already redundant* backends in clustered architectures to detect errors.

We use clustering to split up the processor in smaller sections (*clock domains*) and reduce the number of critical paths, which will reduce the impact of variations at cluster level. The domains can either use separate reference clocks or share the same. If the clusters operate at different clocks, we synchronize all inter-cluster communications by means of simple FIFOs [5].

In particular, we cluster the processor in the following domains: (a) a frontend domain that includes the trace cache, the reorder buffer (ROB), and the branch prediction, renaming, decoding, and steering logic; (b) a memory domain that includes the second level cache memory, and (c) multiple execution clusters (*backends*), with each of them including a local DL0 cache, the integer execution pipeline (scheduler, register file and execution unit), and the floating-point pipeline (floating-point scheduler, register files and execution units). Backends communicate values among them using special copy instructions.

In order to protect backends from particle strikes, we need to extend them in such a way that instructions can be executed more than once, and outputs be compared. A small reorder buffer (checkROB), and three small load, store and branch value queues (LVQ, SVQ and BVQ) enable detecting errors when fault tolerance is activated. If recovery is needed, we employ a checkpoint buffer that keeps values of released

registers.

The key features of this microarchitecture are (see Figure 1):

- Any of the backends can be used as a checker backend. This checker backend will run as the other clusters with small differences: (i) it does not access the cache, (ii) jump and return instructions do not change the PC, and (iii) it uses the checkROB instead of the ROB. Notice that the checker backend will reexecute all instructions that are executed in all other backends.
- Needless comparisons increase overhead without any further coverage. Thus, we propose to validate only stores, preventing errors to propagate to memory. This allows masking errors that do not affect the outcome of the program. We use the Store Value Queue (SVQ) to keep stores waiting to be validated. Each entry of the SVQ has the address and the data.
- Since caches are already protected by parity or ECC, the checker backend does not need to duplicate accesses to the cache. Thus, we can save some overhead by reusing loads through a Load Value Queue (LVQ); loads that are replicated read from it, verifying that the address is correct. Each entry of the LVQ keeps the address and the data.
- Instructions are only fetched once (fetching belongs to the frontend domain which is not protected). Thus, in order to guarantee control flow, outcome of branches executed in the checker backend are compared against their execution in the regular backend through the Branch Value Queue (BVQ). Each entry of the BVQ has a bit for indicating whether the branch was *taken/not taken* and another field for the target address, if any.
- For recovery purposes, we use a small checkpoint buffer. It keeps the contents of the released registers for the last 64 instructions. Our simulations running an x86 functional simulator show that it allows recovering from more than 90% of the errors.

#### B. Design

Fault tolerance comprises two different mechanisms: error detection and recovery. Next, we explain in detail how our proposed solution works for detecting errors. We also discuss a recovering mechanism that allows continuing execution once an error is detected. For the remaining of the paper, we will refer to commit when instructions are removed from the reorder buffer (ROB), and retire when registers are released. Note that in a conventional microarchitecture both committing and retiring occur simultaneously, but we require them to happen at a different time.

**Error detection.** In a nutshell, instructions execute as usual employing only the backends that are not used for checking and are reexecuted in the checker backend once they commit. Reexecution at commit time avoids checking instructions on the wrong path, which decreases the pressure on the checker backend (therefore, the performance penalty is smaller).

Upon commit time, instructions committing are removed from the ROB and allocated (as if they came from decode)

to the checker backend. This can be seen as creating two data-independent threads from a single stream of instructions. By allocating and renaming the instructions twice, instructions reexecute independently. Thus, allocating an instruction to the checker backend consists of: (i) allocating an entry in the checkROB, (ii) renaming the instruction, and (iii) allocating an entry in the preschedulers of the checker cluster. Notice that registers are not released until they retire from the checkROB; otherwise, a rollback would not be possible when detecting an error. If there is no available space in any of the structures (checkROB, LVQ, SVQ and BVQ), the instruction commit stalls.

Besides regular allocation to the checker backend, loads, stores and branches require a special treatment.

- **Loads.** When a load is allocated to the checker backend, the entry is removed from the load/store queue (LSQ) (like in conventional commit) and the result (address and data) of the load inserted into the LVQ. This way, reexecuted loads do not need to access the cache. However, since the LVQ is a simple FIFO, the replicated load instructions reexecuted in program order w.r.t. to other loads. Once the load reexecutes, it reads (and removes) the value from the LVQ. If the addresses do not match, an error is signaled. Note that this also holds for uncached loads (loads from I/O devices).
- **Stores.** We do a similar job for the stores; once a store is allocated to the checker backend, stores are removed from the LSQ and inserted into the SVQ waiting for comparison (recall that our proposal only validates stores). This implies that loads executing in the regular backends will have to compare addresses against the SVQ for disambiguation. We claim that it adds negligible complexity since they already compare against addresses of the stores in the LSQ (which is larger than the SVQ) and the write-buffer. Once the store is reexecuted, it checks the address and data with the corresponding entry of the SVQ. Once both are validated, the store is ready to be retired. We will discuss later that we may want to delay the actual writing to memory for recovery purposes.
- **Branches.** Once a branch is allocated to the checker backend, we insert the branch outcome (taken/not taken and/or target address) into the BVQ so control flow can be validated once the branch is retired after reexecution.

Verification is done for stores by comparing address and data, thus guaranteeing that the state of the memory is not corrupted due to a fault. Since the checker backend does not fetch instructions, we also have to make sure that the path taken is correct; thus, we also verify branches by comparing their outcome (target address and taken/not taken). Exceptions are serviced once validated (i.e., an exception occurs in both regular and checker backend).

**Recovery.** Once an instruction finishes its execution on the checker backend, it is ready to retire from the checkROB. However, since only stores are validated, we do not know whether this instruction executed properly. As we explained

before, if there was an error, we expect it to propagate and be later detected by a store. Next, we describe our proposal for having error recovery, although any mechanism could be used (e.g., mechanism described in [8]).

In order to have the ability to roll back to a state that allows correcting the error and resuming execution, we need to keep the values that are released and delay stores long enough so memory is not modified. Although delaying the release will increase the pressure on the register file, our experiments show that the total performance loss is small, and thus, we do not need to increase the size of the register file.

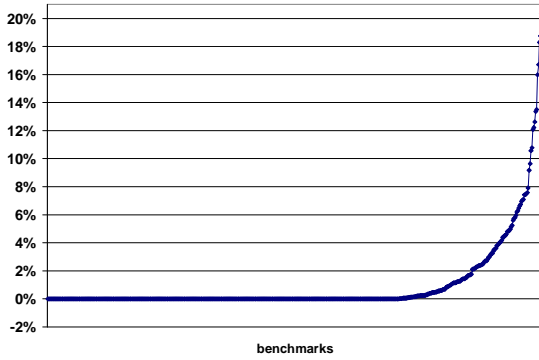
A circular buffer (*checkpoint buffer*) stores the logical register being released (from both the checker and the regular backend) and the actual value of the registers. This way, when the error is later detected by a store, if the error happened in any instruction kept in the checkpoint buffer, we will be able to recover. Memory correctness is ensured by delaying stores until they are released from the checkpoint buffer. We implement the delay mechanism by adding some entries to the SVQ (our experimentation shows that 32 extra entries are enough) and an extra pointer. The SVQ with error recovery capabilities is implemented as a circular buffer, with one pointer to the first non-validated store, and another one, pointing to the first non-allowed-to-write store.

Our evaluation using an x86 functional simulator for many traces, shows that a checkpoint buffer of 64 entries (this is, the distance from the point an error is generated and detected can be as large as 64 instructions) allows recovering from 90% of the errors.

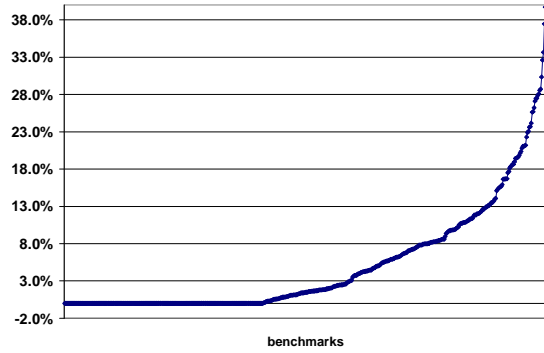
When an error is signaled, the processor rolls back (it recovers the state of the regular and the checker backends undoing all changes through the checkpoint buffer), decreases the frequency (just in case it is a timing error) and reexecutes from that point. If the error is fixed, the processor resumes its normal operation. Otherwise, if the error persists after several trials, the error would probably be due to hard errors or a transient error that we cannot recover from. A mechanism (like a hardware-test) could be run in order to detect a hard error and fix it (i.e., disabling the component), and the execution could resume afterwards.

#### IV. EVALUATION

For assessing the proposed microarchitecture, we evaluate a 3-backend processor, where one of the backends is used for checking purposes. As it is described above, the only extra hardware we add is the LVQ, SVQ, BVQ, checkROB, checkpoint buffer and some comparators to compare the addresses of loads, and output of stores and branches. We employ the same renaming table which is already in place for the different clusters, but differently so the checker cluster does not share the renaming with the regular ones. We also simulate a dedicated renaming logic for the checker backend, although the checker backend could use the existing rename ports with lower priority. Figure 2(a) shows a s-curve representing the slowdown when running in checking mode compared to conventional 2-backend architecture (no



(a) Slowdown when compared to 2 clusters



(b) Slowdown when compared to 3 clusters

Fig. 2. Performance numbers

Benchmark suite	#traces	Desc./Examples
Encoder (enc)	62	Audio/video encoding
SPECfp2K (Sfp)	41	Specs
SPECint2K (Sint)	35	Specs
Kernels (kernels)	52	VectorAdd, FIRs
Multimedia (MM)	85	WMedia, photoshop
Office (office)	75	Excel, word, powerpoint
Productivity (prod)	45	Internet contents creation
Server (server)	53	TPC-C
Workstation (ws)	49	CAD, rendering

TABLE I  
WORKLOADS.

DL0/UL1/mem latency	3/13/190 cycles
Integer Execution	32 entries scheduler, 4 issue
Floating-Point Execution	32 entries scheduler, 4 issue
Commit Bandwidth	6
ROB/checkROB	512/128
LSQ/LVQ/SVQ/BVQ	256/8/(16+32)/16

TABLE II  
CONFIGURATION USED FOR SIMULATION.

errors injected). We run over 500 traces detailed in Table I in a microarchitecture-level performance simulator. The main configuration parameters of the assumed microarchitecture are shown in Table II.

We can see that in 92% of the benchmarks the performance loss is less than 5%. This shows that one checker backend can keep pace with the other 2 backends, barely harming performance. Only for those applications with very large IPCs the loss of performance is higher; if the application is critical, we can disable the fault tolerance capability. The average slowdown is  $\mu = 1.0\%$  with  $\sigma = 2.8\%$ . Compared to the 3-backend architecture (see Figure 2(b)) where all three backends are devoted to throughput, we have obtained an average slowdown of  $\mu = 4.8\%$ , with  $\sigma = 12.0\%$ . 64% of the applications have a slowdown below 5%, and 80% below 10%.

## V. CONCLUSIONS

Using a checker backend is an effective way to deal with transient and timing errors in forthcoming processors. This kind of architecture allows detecting errors and recovering from them with low design and performance cost: (i) it does not need any design effort since it uses the regular backends, (ii) the extra hardware is very small, and (iii) checking incurs in a small performance penalty. Compared to previous approaches [6], [9], [10], it can handle both transient and timing errors with a much lower performance and cost penalty. The fact that it is reconfigurable (trading fault tolerance for performance) makes it very attractive for different environments where performance, power and reliability requirements may change.

## REFERENCES

- [1] R. Baumann, "Soft errors in advanced computer systems," in *Proceedings of IEEE Design and Test of Computers*, 2005.
- [2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the Design Automation Conference*, June 2003.
- [3] K. Bowman, S. Duvall, and J. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 183–190, 2002.
- [4] D. Marculescu and E. Talpes, "Variability and energy awareness: a microarchitecture-level perspective," in *Proceedings of the 42nd annual conference on Design automation (DAC '05)*, 2005, pp. 11–16.
- [5] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *In Proceeding of 2002 International Symposium on High Performance Computer Architecture (HPCA8)*, 2002.
- [6] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.
- [7] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th annual International Symposium on Computer Architecture (ISCA)*, 2002.
- [8] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th annual International Symposium on Computer Architecture (ISCA)*, 2002.
- [9] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of International Symposium on Microarchitecture (MICRO-32)*, 1999.
- [10] L. Spainhower and T. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5/6, pp. 863–873, 1999.