A Result Propagation Scheme for Redundant Multithreaded Systems

Bernhard Fechner
Department of Computer Science
FernUniversität in Hagen
58084 Hagen, Germany

Abstract - *Traditional fault-tolerance techniques like triple modular redundant (TMR)-systems can detect and correct a single transient fault with majority voting among the replicated units at about 200% space increase. The performance of two processors is sacrificed for the detection and localization of faults. Simultaneous multithreading (SMT) offers the detection of transient faults at only 20-40% performance degradation. Most of these techniques rely on the duplicated execution of the same process on different hardware threads. Due to SMT, the execution can be parallel or time-shifted. The fact that e.g. the direction of a branch or the value of a load encountered by the thread which is executing ahead is exploited by using structures like the branch outcome queue (BOQ) or the value outcome queue (VOQ). In this work we will first extend the idea of the BOQ/ VOQ to a more sophisticated structure, so that results being produced by the leading thread can be consumed faster by the trailing thread. Additionally, we propose how the scheme can help to detect transient and permanent bus faults. We validated the scheme by using a FPGA implementation, including a branch target address and instruction cache. The results show a space increase of less than 4% and a speedup of 20%.*

Keywords: Design and Test for Reliability, Redundant Multithreading.

1 Introduction

The development of commercial microprocessors is a quest for high performance. Processing speed can be gained by increasing the clock frequency and integration density. Examples are dual-core systems on a die or Simultaneous Multithreading (SMT) [3]. The Semiconductor Industry Association (SIA) roadmap shows an increase of integration density to 22nm until 2016[1]. Shrinking the thickness of gate and interlayer dielectrics leads to increased leakage current in CMOS devices. More current is collected at interconnects which raises the processors temperature. Thus, the lifetime of the concerned transistors is shortened. At 90nm and below [4] a problem occurs at sea-level, which is known from aerospace applications: The collision of high-energetic neutrons from deep-space with silicon, causing a partial or total failure of the concerned circuit. At large heights, fault rates in SRAMs increase by 3-10, approximately by 1.3 per 1000ft [14]. To decrease the energy consumption, e.g. the clock is dynamically adjusted and the current potential between logic 0 and 1 is reduced. Thus, the signal-to-noise ratio and the critical charge Q_{crit} needed to change the state of a flip-flop or latch is decreased. It is commonly believed, that soft-errors mainly occur in memory elements (flip-flops and capacitors). At minimum feature sizes below 90nm, this is not the case any more. With high integration, protons are able to induce Single Event Upsets (SEUs), leading to a higher rate of transient faults, especially in deep-space applications. The consequence is an increased susceptibility for Single Event Upsets. Downtime costs by SEUs already have dramatically increased in the last years. The soft-error rate in combinatorial circuits will increase by $\sim 10^5$ from 1992 until 2011 [5]. For the next decade a total error rate of 10^4 FIT (Failure In Time=1 error in 10^9 hours of operation=one year MTBF) $\Rightarrow$ fault rate $10^{-5} / h$ in combinatorial circuits is forecasted [6].

This paper makes the following contributions:

- It clarifies some obscurities from preceding papers proposing queues to forward results like branch targets, loads and stores between redundant threads and proposes a faster result-forwarding scheme.
- We show how existing structures like the branch target address cache (BTAC) can be integrated, omitting the duplicate storage of branch targets.
- We propose how the structure can help to detect transient and permanent bus faults in the start-up phase of the processor.
- We validate the idea by using a FPGA implementation and analyze the efficiency (speed and space) in comparison with the original outcome-queue concept.

The rest of the paper is organized as follows: Related work and the fault model are presented in Section 2.

[1] http://public.itrs.net/Files/ITRS_Overview.pdf

Section 3 presents the main idea of this work, the *temporal memory* for branches and values. Section 4 describes the implementation methodology and presents the results. Section 5 concludes the paper.

2 Related Work

To limit the extensive space, energy and performance needs of triple modular redundant systems, numerous approaches exist. Duplex systems (DMR) are a common technique to detect transient faults by duplicating two instead of three components. Lockstepping is one possibility to implement a duplex system. Here, two processors are provided with the same clock. Due to time-dependences, the processors must have the same mask revision if no additional logic helps to produce the results at the same time. It is simple and cost-efficient to implement and therefore used in many commercial processor designs (e.g. Compaq Himalaya [16] or IBM G5 [13]). Virtual duplex systems are using time instead of structural redundancy. Here, identical processes are not concurrently executed on processing nodes (like TMR, DMR) but two times - one after the other - on one node. Recently, Simultaneous Multithreading [1][2][3][13] has been discovered to detect transient faults. Apart from variations (e.g. [17][18][19][20]), two identical copies of the same program are run as independent threads either on a multithreaded processor and the outputs are compared for mismatch. SMT processors need 22% less energy per instruction due to better resource utilization [15]. If less energy is consumed, the processor can be clocked faster or can achieve the same performance while consuming less power. Thus, the transistor lifetime and reliability will increase. If duplicate processor cores are used fault detection, the forfeiture is the performance of an entire processor. If multithreaded processors are used, only 20 to 40% of the processor's performance [18][19][20] will be sacrificed.

AR-SMT [9] used SMT to run redundant copies of the same program in one active(A) and one redundant(R) thread on the same processor. Here, the idea to exploit previously computed results (e.g. branch outcomes) of the A-thread by the R-thread with the help of a delay buffer was introduced. Reinhardt and Mukherjee proposed a detection scheme for transient faults similar to AR-SMT. Simultaneous and Redundantly Threaded processors [8][10] (SRT) also uses two redundant threads to run the same program, one *leading* and one *trailing* thread. Both threads are separated by a *slack,* where the slack represents the number of instructions by which the leading thread executes ahead of the trailing one. The R-thread also computes branch targets etc. Since the A-thread was fetching values etc. from the main memory they are stored in the cache, so that the R-thread experiences a speedup. Another method not used by AR-SMT for the sake of reliability is to let the R-thread not fetch the values and branch targets over the bus and entirely use the values in the outcome queues. To provide fault detection for Chip Multi-Processing (CMP), SRT was extended in context of a study to the so called Chip-level Redundantly Threaded multiprocessor (CRT). Similar to SRT, CRT uses leading and trailing threads on separate processors. CRT achieved a reduction of the probability that one fault will corrupt both threads at the same instruction. Slipstream [11], the CMP fault detection scheme corresponding to AR-SMT maximizes performance rather than to provide robust fault detection. With the concept of Simultaneous and Redundantly Threaded processors with Recovery (SRTR), Vijaykumar et al. [12] addressed the issue of providing robust and efficient recovery from transient faults in a uniprocessor SMT-system. Gomaa et al. [7] introduced a scheme called Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR), porting SMT transient fault recovery to multiprocessors.

The applied fault model assumes one fault at a time for a component and transient faults in the form of Single Event Upsets (SEUs) in memory elements. Multiple faults are not addressed since they are extremely seldom. SEUs are modeled by bit-flips (flip-to 0 and flip-to 1) of the corresponding latches or memory cells [21]. Furthermore, we assume transient and permanent faults in the form of stuck-at faults (stuck-at 0 and stuck-at 1) for the external bus.

3 Temporal Memory for Branches and Memory Accesses

Until today, all structures within redundantly multithreaded (RMT) systems used a FIFO buffer to forward results between the leading and the trailing thread. The FIFO is needed to preserve the time-dependence between results. Figure 1 illustrates the delay buffer concept to forward the results between the commit phase of the A-stream and the fetch phase of the R-stream [9]. Results (branches, loads and stores) produced by the A-stream during its commit phase are inserted in the delay buffer and consumed by the R-thread. In fact, during its fetch phase, the R-thread accesses both, the main memory and the delay buffer. Stores can not be committed to main memory until the R-thread wants to commit the store, since a computation of the R-thread could depend on a memory value which is in the meantime overwritten by the A-thread.

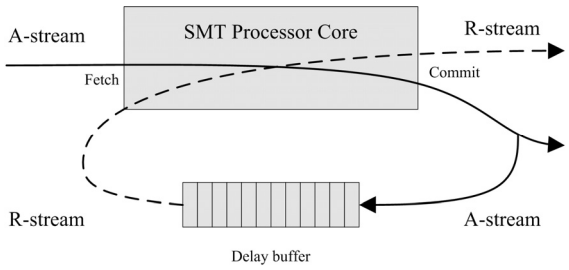

Figure 1. The delay buffer used within AR-SMT

Whereas Figure 1 suggests using a FIFO for the exchange of results between threads, we propose a more sophisticated structure which allows keeping the time-dependence between instructions, but exploits multiple commitments of modern superscalar, simultaneously multithreaded processors. Supposing a real-world implementation, the FIFO queue between threads will be a bottleneck, since we must have the possibility to simultaneously write back multiple results. Simulations will not reveal this bottleneck. If multiple instructions are committed, the queue will perform poorly, since we can only write back one result at a time. This disadvantage can partly be balanced by accessing the queue asynchronously if the logic is fast enough. As a consequence, the implementation will be more complex. Another disadvantage of a queue is that target addresses of loops will be stored again and again in the outcome queue, no matter if the branch target or value was previously computed or not. This will consume power and valuable space in the queue and thus keep results from being forwarded to the trailing thread because the queue will be occupied much earlier. Furthermore, structures like the Branch Target Address Cache (BTAC) will work in parallel to the delay buffer. The A-thread will always enter the outcome in the BTAC *and* the delay buffer. This is a waste of space and unnecessary and unwanted redundancy. Since both circuits are active in each cycle, the power consumption will increase. Furthermore, both structures are very vulnerable, since they contain only valid targets. Therefore, both structures must be secured against SEUs, leading to a further space increase, because error detection mechanisms must be implemented for both structures. We are putting it all together by solving performance problems from delay-buffer queues and by integrating the BTAC and the Branch Target Instruction Cache (BTIC) into the result feed-forward structure. We call this structure *temporal memory*. It is shared between both hardware threads. We do not have to separate thread-specific entries since we have a duplicated execution of identical instruction streams. Naturally, the temporal memory is a vulnerable structure and should be secured against transient faults by commonly used error detection/ error correction codes. For data and instructions we consider separate structures. Table 1 shows a temporal memory entry for instructions/ branch targets, Table 2 the entry for a data value.

Table 1. A temporal memory entry (branches)

Name	Width	Description
PC	32	Program counter at branch.
DEST	32	Branch target+4.
INST	32	Instruction at branch target.
FREE@acc	4	Free entries at time of access.
Total	100	Total bits used.

Table 2. Temporal memory entry (data)

Name	Width	Description
ADDR	32	Memory access register.
VAL	32	Data value.
FREE@acc	4	Free entries at time of access.
Total	68	Total bits used.

Separate structures were chosen to limit the number of read/write ports and the space increase. Figure 2 shows how the temporal memory can be seamlessly integrated to speedup a SMT processor core applying redundant multithreading. We integrate a start-up entry-point memory and a counter, whose function will be described in the next subsection.

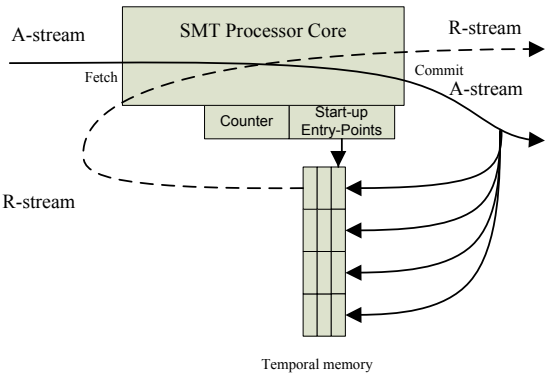

Figure 2. Temporal memory system integration

We distinguish two different modes the memory can be accessed in:

1. Start-up operation after reset and
2. Lookup-mode during normal operation.

3.1 Start-up Operation

The start-up mode serves to initialize the temporal memory and to detect transient and permanent bus faults. All we have to do is to configure the temporal memory after reset with the jump addresses of the system startup-code. The entry points are stored in a small programmable on-chip storage (*Start-up Entry-Points* in Figure 2). When the leading thread starts to fetch instructions, it will try to store them in the temporal memory. As the entry is already occupied by data after a reset (as the number of occupied entries is equal to the number of available entries), the entry point addresses of branch, load and store instructions fetched over the bus will be found in the temporal memory by using the program counter (PC) or memory addresses (ADDR) as index. If we know the size of the temporal memory, we can use an access counter to indicate the end of the start-up phase. This is exactly

when the counter reaches zero. It will be initialized with the number of entries in the memory and decremented each time it is accessed. The fetched instructions and issued PCs/ memory addresses will be compared with the entries in the temporal memory. If they match, no fault occurred. If not, a transient or permanent fault corrupted the issued (addresses) or fetched values (instructions or data). When encountering such an error, the system will restart. To cover as much errors as possible the addresses and the data being fetched should be disjunct concerning bit values at the equal positions so the Hamming distance between both words is maximal. It is plain to see, that not all transient or permanent errors can be detected by using this scheme, if the system start-up code is not written in a way taking this error detection into account. Even if the data is disjunct, most addresses or at least the most significant bits of the address will not change during start-up, since the system start-up code will reside in a limited area. This error detection scheme is only thought as a feature, which we can apply at marginal cost increase (a counter and a small programmable ROM).

3.2 Lookup-Mode

If the access counter reaches zero, the start-up phase has finished and the system will enter normal lookup-mode. This mode does not differ much from the start-up mode except that no values will be loaded from the start-up entry point memory into temporal memory and the counter will be used in a different way. We initialize the counter with the number of available entries in the memory. The leading thread has to be stalled if all entries in the table are occupied. Thus, the memory size should be chosen in a way that no stall occurs. The counter will be decremented, at each write of the leading thread. It is incremented, when the trailing thread accesses the table. In the following, we clarify the accesses of the leading and the trailing thread.

Leading thread access:

Write access: in this mode, precomputed branch targets and values are written to the temporal memory if a branch, load or store is decoded. The number of free entries (the counter value) is entered in the field FREE@acc. If simultaneous writes occur, we use the same values for FREE@acc. This will help to save multiple addition units. Then the counter is decremented by the number of write accesses. The branch target (DEST), the program counter (PC) and the instruction at the branch target (INST) of the entry are updated. We use the PC as index in the table. Then, we compare the data value or the branch target to insert with the according entry. If they are equal, we do not store the value. If not, we enter the value in the first free entry (if available) and decrement the counter. What do we do to resolve the time-dependence between threads? For example, the leading thread could produce a value and enter it in the memory. After some instructions another value for the same variable will be entered again. If the value was not consumed by the trailing thread in the meantime, we have equal program counters but different values. This is no problem, since we use the field for the counter of free elements, FREE@acc, to determine the time of the access. The counter and FREE@acc cannot be equal until the element was read by the trailing thread.

Read access: Since the memory replaces the BTAC and the BTIC, each issued PC will be used to search an entry in the memory. If it is found, the stored address will be used as future PC. Since we use a subset of the MIPS ISA and do not allow self-modifying code, we do not have to handle conditional jumps at the same PC with different branch targets.

Trailing thread access:

If a branch, load or store is decoded, the counter of free entries will be used to find the corresponding entry in the table. If entries with the same FREE@acc value are encountered, they can be read simultaneously if enough read ports are available. After the entries are accessed, the counter will be decremented by the number of parallel reads. We must not fetch the concerned instructions or data over the bus. Thus, we can save power and cycles by omitting bus accesses. The consequence is that instructions fetched by the leading thread can be corrupted by a transient fault and the error will not be detected since we only fetched one value. For a better comparison of both schemes, we considered the fetching of instructions and data over the bus by both threads.

4 Results

To measure the space requirements, we discuss the synthesis results of a FPGA implementation using VHDL. As a target, we chose the Xilinx Virtex-E XCV1000, bg560, speed grade -8 FPGA [22]. The reasons why we selected this kind of FPGA were:

1. This FPGA is used in our development platform.
2. The FPGA does not support content addressable memories (CAMs) which can be easily used to implement temporal memory. By using FPGAs supporting such memories, the space increase will be even smaller and the circuit will perform faster.

Both concepts - the outcome queue and the temporal memory - were implemented by using the Xilinx ISE software version 6.3. Table 3 shows the resource usage after synthesis. The column ‘OQ’ holds the number of slices used by the outcome queue implementation. The column ‘Temp’ holds the number of slices used by the temporal memory result propagation scheme. We included the number of external IOBs (IO Blocks) because

logic was mainly routed to external IOBs by the synthesis tool. Remember, that we are able to spare the BTAC *and* BTIC. This will further reduce the cost in comparison with the outcome-queue concept.

Table 3. Resource usage of the two examined result propagation schemes

	OQ	Temp	Increase	Total
Number of External IOBs	93	130	~9%	404
Number of Slices	369	807	~3%	12288
Min. period (ns)	9.8	11.8	~20%	

As size for the outcome queue and the temporal memory we assumed 16 entries. To implement a structure which is able to hold instructions and/or data we have to consider either another queue or a tag within the queue. We used dedicated structures for the outcome queue and the temporal memory. Furthermore, we assumed a fixed instruction width of 32 bit. In relation to the total number of FPGA slices and IOBs (12692), the overall space increase is less than 4%. As Figure 2 suggests, we used four parallel write and one read port to the temporal memory for the leading thread. We assumed that the redundant thread is able to issue one fetch at a time. Thus, we implemented one read port for the R-thread. If the trailing thread is able to issue multiple fetches at a time, the performance will increase further.

5 Conclusion

To fight transient faults induced by Single Event Upsets is one of the major challenges for microarchitects and manufacturers in the present and future. Redundant multithreading is one means to detect transient faults. This work presented a space and time efficient scheme to forward results in redundantly multithreaded systems. The scheme can be seamlessly integrated in most microprocessors with support for redundant multithreading. It is able to eliminate the bandwidth bottleneck of queue-like structures used to forward results between the active and the redundant thread and to keep the time-dependence between threads. To prove and validate our approach, we synthesized the proposed circuit in VHDL with normal place and route effort. As target we selected a Xilinx Virtex-E XCV1000bg560 speed grade -8 FPGA [22]. The overall space increase was less than 4% in comparison with the original outcome-queue concept, but we are able spare the BTAC and the BTIC. The FPGA implementation yielded a maximal clock rate of t_{TM}=84.4 MHz for the proposed result propagation scheme and t_{OQ}=102 MHz for the outcome queue concept. Thus, the outcome queue implementation was about 20% faster. One explanation for this performance is that we did not apply special search algorithms to speed up the matching and the FIFO was better synthesized by the synthesis tool, implemented by using pointers and an asynchronous control. If we assume $\#_{pcommit} = 5$ as the percentage of instructions than can be concurrently committed on additional $\#_{w\_commit} = 4$ write and consumed on $\#_{r\_commit} = 2$ read access ports, we have a maximal speedup of $\#_{pcommit} \cdot \#_{w\_commit} \cdot \#_{r\_commit} = 40\%$. Thus, we have a performance increase of 20%.

4 References

[1] J.S. Emer, *Simultaneous Multithreading: Multiplying Alpha Performance*, Microprocessor Forum, Oct. 1999.

[2] N. Tuck, D.M. Tullsen. *Initial Observations of the Simultaneous Multithreading Pentium 4 Processor*. Proceedings of 12th Intl Conference on Parallel Architectures and Compilation Techniques, September 2003.

[3] D.M. Tullsen, S.J. Eggers, H.M. Levy, *Simultaneous Multithreading: Maximizing On-Chip Parallelism*, Proc. 22nd Annual Int'l Symp. on Computer Architecture, pp. 392-403, Jun. 1995.

[4] T. Juhnke: *Die Soft-Error-Rate von Submikrometer-CMOS-Logikschaltungen*. Fakultät Elektrotechnik und Informatik, Technischen Universität Berlin, Dissertation, 2003.

[5] E. Normand, *Single Event Upset at Ground Level*. IEEE Transactions on Nuclear Science, Vol. 43, No. 6, December 1996.

[6] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, L. Alvisi. *Modeling the effect of technology trends on soft-error rate of combinational logic*. Int'l. Conference of Dependable Systems and Networks, June 2002.

[7] Gomaa, M. et al: *Transient-Fault Recovery for Chip Multiprocessors*. In Proc. of the 30th Annual Int'l. Symp. on Computer Architecture, pp. 98-109, June 2003.

[8] S. Reinhardt, S.S. Mukherjee. *Transient-Fault Detection via Simultaneous Multithreading*. In Proc. of the 27th Annual Int'l. Symp. on Computer Architecture, pp. 25-36, June 2000.

[9] E. Rothenberg. *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors*, In Proc. of Fault-Tolerant Computing Systems, pp. 84-91, 1999.

[10] S.S. Mukherjee, M. Kontz, S. Reinhardt. *Detailed Design and Evaluation of Redundant Multithreading Alternatives*, Proc. of the 29th Int'l. Symp. on Computer Architecture, May 2002.

[11] S., K., Purser, Z., E. E. Rotenberg: *Slipstream Processors: Improving Both Performance and Fault-Tolerance*. In Proc. of the 9th Int'l. Symp. on Architectural Support for Programming Languages and Operating Systems, pp. 257-268, November 2000.

[12] Vijaykumar, T.N., Pomeranz, I, Cheng, K.: *Transient Fault Recovery using Simultaneous Multithreading*, Proc. of the 29th Int'l. Symp. on Computer Architecture, pp. 87-98, May 2002.

[13] T. J. Slegel, et al. *IBM's S/390 G5 Microprocessor Design*, IEEE Micro, 19(2):12–23, Mar/Apr 1999.

[14] H. Kobayashi, et. al. *Soft Errors in SRAM Devices Induced by High Energy Neutrons, Thermal Neutrons and Alpha Particles*, IEDM Tech. Digest, Dec. 2002, pp. 337-340.

[15] J. Seng, D.M. Tullsen, G.Z. Cai. *Power-Sensitive Multithreaded Architecture*, Int'l. Conference on Computer Design, pp. 199-206, Sept. 2000.

[16] Alan Wood, *Data Integrity Concepts, Features, and Technology*, White paper, Tandem Division, Compaq Computer Corporation.

[17] Todd M. Austin, *DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design* Proceedings of the 32nd Annual International Symposium on Microarchitecture, 1999.

[18] Eric Rotenberg, *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor*, Proceedings of Fault-Tolerant Computing Systems (FTCS), 1999.

[19] Steven K. Reinhardt and Shubhendu S. Mukherjee, *Transient Fault Detection via Simultaneous Multithreading*, International Symposium on Computer Architecture, 2000.

[20] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt, *Detailed Design and Evaluation of Redundant Multithreading Alternatives*, submitted for publication.

[21] J.P. Hayes: *Fault Modeling*, IEEE Design& Test, pp. 88-95, April 1985.

[22] Xilinx: *Virtex-E 1.8 V Field Programmable Gate Arrays*, 2002.http://direct.xilinx.com/bvdocs/publications/ds022.pdf