Exploiting the Dependency Checking Logic of the Rename Stage for Soft Error Detection

Oğuz Ergin[1], Gülay Yalçın[2], Osman S. Ünsal[3], Mateo Valero[2,3]

[1]TOBB University of Economics and Technology
Department of Computer Engineering
Söğütözü Cad. No:43 Söğütözü, Ankara, Turkey
oergin@etu.edu.tr

[2]Universitat Politecnica de Catalunya (UPC)
{gyalcin,mateo}@ac.upc.edu

[3]Barcelona Supercomputing Center (BSC)
osman.unsal@bsc.es

Abstract. Register renaming is a widely used technique to remove false data dependencies in superscalar datapaths. Rename logic consists of a table that holds a physical register mapping for each architectural register and a logic for checking intra-group dependencies which consists of a number of comparators that compares the values of destination and source registers. Previous research has shown that the full capacity of the dependency checking logic is not used at each cycle. In this paper we propose a technique that makes use of the unused capacity of the dependency checking logic of the rename stage in order to detect soft errors that occur on the register tags while the instructions are passing through the frontend of the processor.

1 Introduction

Soft errors caused by cosmic particles and radiation from packaging are an increasingly important problem in modern superscalar processors. Particle strikes that occur both on the memory structures and the computational logic may cause system crashes if these errors are not detected [1][12]. Parity bits and ECC are widely used in cache memories and some other important parts of the processors [12].

Modern microprocessors use aggressive techniques like out-of-order execution and dynamic scheduling for boosting performance. In order to feed these aggressive techniques that leverage instruction level parallelism, superscalar datapaths try to fetch more than one instruction each cycle. For example Intel's Pentium 4 processor [6] and each processor core in Intel's Core Duo architecture [5] fetches up to 3 micro-instructions per cycle from the instruction cache, while the Alpha 21264 fetches 4 instructions per cycle [8]. In practice, the processor cannot fill the whole fetch width due to the speculative nature of branch instructions and the fetch stops after the first taken branch which leads to the underuse of processor resources.

Almost all contemporary processors use register renaming in order to cope with false data dependencies with the exception of Sun's Ultra Sparc [16]. Use of register renaming mandates the use of a mapping table and a renaming logic where a free register is assigned to each result-producing instruction and the dependent instructions get this information in the same cycle. This logic includes dependency checking logic, which contains a number of comparators to compare each and every destination register tag with the source register tags of the subsequent instructions that are renamed in the same cycle. Because of the fact that the processor pipeline is not filled to its capacity every cycle, the comparators of the dependency checking logic are not always utilized.

In this paper we propose a new technique that leverages the inefficient utilization of the comparison logic in the renaming stage of the pipeline to detect transient errors that occur in the frontend of the processor. When the full pipeline width is not used it is possible to protect the register tags of the instructions by replicating the register tags into the unused fields of the subsequent instruction slots. We then use this redundant information by employing the already available comparator circuits of the rename logic to detect any errors that occur until the instruction reaches the rename stage.

2 Register Renaming

Register renaming is a widely used technique to remove false data dependencies. The false data dependencies occur because of the insufficient number of architectural registers that the processor offers to the compiler. When the compiler runs out of registers, it uses the same architectural register multiple times in short intervals, which creates a false write-after-write (WAW) or write-after-read (WAR) dependency between the instructions that are in fact not related at all. Modern processors solve this problem by employing a large physical register file and mapping the logical register identifiers produced by the compiler to these physical registers. Consequently a processor that makes use of the register renaming technique needs more physical registers than the number of architectural registers to maintain forward progress [16].

A mapping table is maintained in order to point out the location of the value that belongs to each architectural register. This mapping table is called the "Register alias table (RAT)" or in short "rename table" and it contains an entry for each architectural register that holds the corresponding physical register that holds the last instance of the architectural register [6].

Each result-producing instruction that enters the renaming stage of the processor checks the availability of a physical register from a list of free registers. If a free register is available, the instruction grabs the register and updates the corresponding entry in the rename table. In some implementations of the register renaming, the instruction has to read and hold the previous mapping of its destination architectural register in order to recover from branch mispredictions or free the physical register that holds the previous value of the architectural register [6]. Each instruction also has to read the physical register identifiers that correspond to the architectural registers that it uses as source operands from the rename table.

In a superscalar processor, the rename table has multiple ports to allow the renaming of multiple instructions per cycle. Every instruction that is renamed together in the same cycle needs to acquire a free register from the free list, update the rename table and read the mappings for its source operands concurrently. Since some of the instructions that are renamed in the same cycle are dependent on each other with WAR or WAW hazards, instructions may try to write to the same entry of the rename table in the same cycle or an instruction may need to wait for a previous instruction to update the rename table before it can read the mappings for its source operands. This kind of sequential access to the rename table may either increase the cycle time or may not be even possible due to some design choices. Therefore register renaming stage of the pipeline includes a dependency checking logic to detect intragroup dependencies.

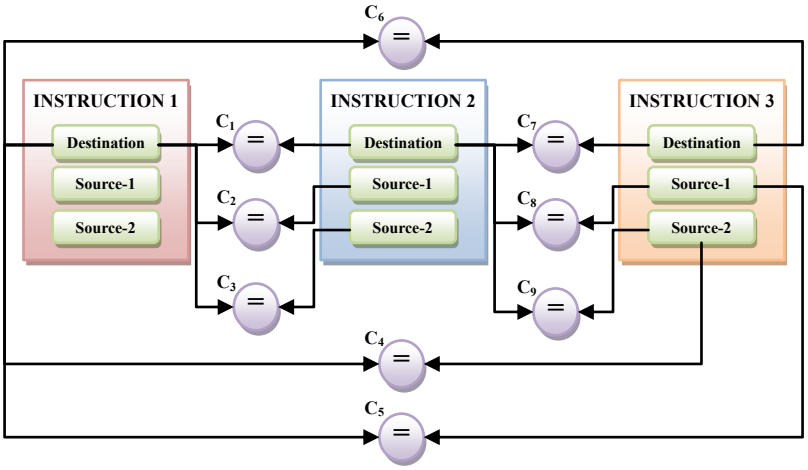

Fig. 1. Comparison circuitry of the rename logic

Fig. 1 shows the structure of the dependency checking logic for a machine that renames 3 instructions concurrently. There are multiple comparator circuits that compare the destination and source tags of all concurrently renamed instructions. Each instruction's destination architectural register tag is compared to the source operand tags of all of the subsequent instructions. In case of a match, the physical register mapping corresponding to the source operand of the subsequent instruction is obtained from the destination physical register field of the preceding instruction rather than being read from the rename table. This way a serial write and read operation is avoided. Similarly, in order to avoid updating the same rename table entry multiple times in a single cycle, destinations of all of the instructions are compared against each other. If a match is detected, only the youngest instruction is allowed to update the rename table. The match/mismatch signals that are produced by the comparators $C_1 \ldots C_9$ are fed into the priority decoders to control the access of the instructions to the mapping table.

3 Using Dependency Checking Logic for Soft Error Detection

Although superscalar processors are designed for high throughput, pipeline width is not fully used from time to time. As the pipeline of the processor is not filled with instructions to its capacity, the number of simultaneously renamed instructions is reduced. This fact was previously observed by Moshovos and was exploited to reduce the complexity and the power dissipation of the rename table [9].

When the number of concurrently renamed instructions is below the processor rename width, dependency checking logic of the rename stage is not employed to its capacity. The comparators that are wired to the empty instruction slots during this period stay idle and generally are not used for any purpose. Therefore during the times when the processor is not using all of its rename slots, these comparators can be used to detect soft errors that occur on the register tags of the instructions when they are passing through the frontend of the processor.

Value replication is a long time known and used technique for reliability. Although it is possible to detect single bit errors in a value by adding a single parity bit, replicating the value can detect and possibly correct multiple errors if there are enough copies of the value. Errors can be detected with one redundant copy of a value and can be corrected with two redundant copies through simple voting [3]. More than three copies of the same data leads to a stronger protection as there will be more chances to recover from an error.

Instruction replication was proposed and implemented in different ways to cope with soft errors in the past. Redundant multithreading was proposed to replicate the whole thread running on the processor to detect any soft errors [11][13][18]. While it offers a system level protection, replicating each and every instruction in a program results in some performance degradation as processor resources are divided into two to execute instructions from both the leading and the trailing threads.

Selectively replicating most vulnerable instructions and processor structures have been proposed as a good tradeoff between performance degradation and fault coverage. However, most of the previous approaches have concentrated on protecting the backend structures of the processor such as the functional units, the issue queue or the reorder buffer [3][7][17].

In this paper we propose to replicate the register tags of the instructions into the register tag fields of the unused instruction slots and use the idle comparators of the dependency checking logic to detect and correct soft errors that occur in the frontend of the processor.

3.1 Protecting the tags of a single instruction

When there is only one instruction flowing through the pipeline, all of the hardware resources can be used for this single instruction. It is possible to detect the errors on both the source tags and destination register identifier if the pipeline width is at least 4, without adding any additional comparator circuit. In order to maximize the error coverage, the tags of the single running instruction is copied to the empty fields of the unused slots as shown in Fig. 2 as early as possible in the pipeline. This copy operation is mostly likely to happen right after the instruction leaves the fetch buffer and

enters the instruction pipeline. The instruction slots that hold the redundant information are marked as "bogus" in order to let the rename logic know that these tags do not belong to real instructions.

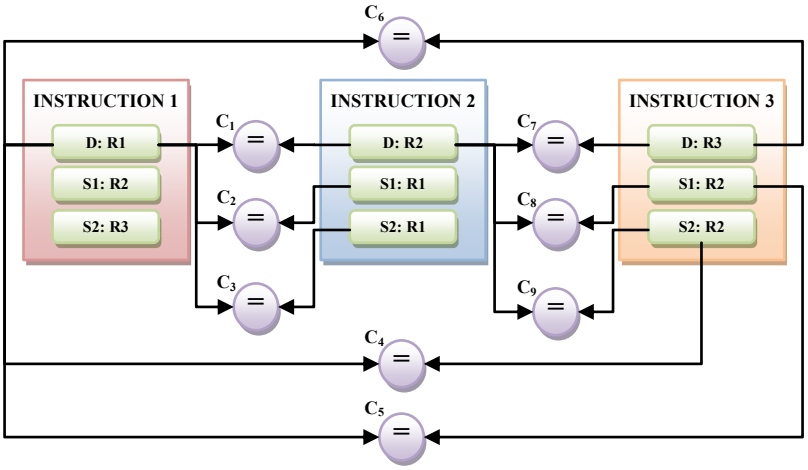

Fig. 2. Error detection example for single instruction renaming

After the single instruction's tags are replicated in the empty slots, outputs of the already wired comparators at the rename stage are checked to see if an error occurred by the time instruction arrived at the rename stage. As seen in Fig. 2, if the outputs of the comparators C_2 and C_3 mismatch it can only be the result of an error on the destination register identifier of the instruction. Similarly, the outputs of the comparators C_8 and C_9 are checked in order to detect an error on the first source tag of the instruction. As for the destination tag, in fact even a mismatch signal in C_2 or C_3 indicates that an error has occurred. Using the output of the both comparators actually gives a chance to correct the error if one assumes a single event upset model.

In order to cover both source tags and the destination tag of the single instruction the processor needs to be capable of renaming at least 4 instructions each cycle. Although the Fig. 2 is shown for a 3-wide machine for simplicity, the second source tag of the first instruction is copied to the destination field of the third instruction as it would be done in a 4-wide machine.

3.1.1. Single Event Upset Model

If we assume that the fault-rate is sufficiently low, then we can statistically ignore the probability of a multiple-bit flip. This is typically called the Single Event Upset model. Under this assumption, the bit flip can not only be detected but corrected as well. Consider Fig. 2, and assume that there was a bit flip affecting R1. There are two possible cases:

a) The bit flip could occur in the destination field of instruction 1; then *both* C_2 and C_3 comparators would signal a mismatch, indicating that there was a strike to the destination field of instruction 1. The field is updated by copying the field of either S1 or S2 of instruction 2, which hold the copies of R1.

b) The bit flip could occur on the copies of R1 held in the S1 or the S2 fields of instruction 2. In this case a mismatch in either C_2 or C_3 would indicate a flip in S1 or S2 respectively. The error could then be corrected by updating the faulty field or would not be corrected at all since the actual tag value is free of errors.

3.1.2. Multiple Events Upset Model

If the fault-rate is high, then multiple bit-flips could occur. The fault-recovery for this model is slightly more complicated; however even for this case we can detect the fault. In this model, it is not for sure that the "correction" will lead to the actual value of the tag. For example if C_2 indicates a mismatch and or C_3 indicates a match it is logical to think that an error occurred on the first source field of instruction 2. However it is also possible that both the real tag and the copy residing in the second source field may be erroneous. Assuming that the actual tag is free of errors in this case may be wrong although the probability of actual tag being correct is high. Therefore it makes sense to flush the pipeline and refetch the instruction if any of the comparators give a mismatch signal.

It is possible to correct errors that occur on the tags of the single instruction. However this comes at the cost of sacrificing the protection on one of the tags since there are not enough comparators to check three tags in a 4-wide machine. In order to correct an error on R1 for the example shown in Fig. 2, the tag R1 has to copied to the destination field of the instruction 2. This way there will be 3 comparisons for R1 and simple voting can be employed. Note that different from regular voting used to correct or detect errors, more copies of the same value are held here since we can only compare a destination tag to a source or destination tag of another instruction but we cannot compare two source tags with each other using only the already available hardware resources. Although not shown in Fig. 2, source register R3 is also copied to the source fields of the instruction slot 4 for protection in a processor with a 4-wide rename stage.

When the destination tag is replicated to all of the fields of the second instruction, an error in the actual tag will result in a mismatch signal at C_1, C_2 and C_3 at the same time. By copying one of the replicated values back into the actual tag area corrects the problem. However if there is even 1 match signal, it probably means that the replicas are corrupted since having two errors, one on the actual value and one on a replica, that will result to the same faulty value is a low probability. Yet again, it may be a good choice to flush the pipeline and refetch the instruction in such a case.

3.2 Protecting the tags of two instructions renamed in the same cycle

The processor renames one instruction at a cycle only if there is a problem. This problem may be a cache miss, a taken branch or a processor resource that causes a bottleneck temporarily. When the processor starts to execute the program faster, the number of instructions renamed per cycle increases. However as the throughput of the processor increases, the benefits of our technique decrease as more comparators start to be employed for their real purpose.

For a 4-wide processor, when only two instructions are renamed together, it is possible to correct an error that occurs on both of the destination tags of the instructions but we cannot detect or correct any errors occurring on the source tags at the same time. Alternatively we can copy the destination tag of one of the instructions to the source tag fields of instruction 3 and one source tag of the same instruction both to the destination tag of the third instruction and the source tags of the fourth instruction slot. This way it is possible to protect the destination tag and one of the source tags of one of the instructions.

3.3 Renaming more than 2 instructions simultaneously

Our proposed technique can also provide partial soft error detection coverage if three instructions are renamed simultaneously. In this case only an error on the destination tag of one of the instructions can be detected by copying this tag to all of the fields (destination + sources) of the fourth instruction slot. In a 4-wide processor, the proposed technique won't be able to provide any soft error detection coverage since there won't be any empty slots or idle comparators.

4 Simulation Methodology

Table 1. Configuration of the Simulated Processor

Parameter	Configuration
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	32 entry issue queue, 48 entry load/store queue, 128–entry ROB, 128-entry Register File
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 1 cycles hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache unified	512 KB , 8–way set–associative, 64 byte line, 6 cycles hit time
BTB	512 KB, 4–way set–associative
Branch Predictor	4K entry bimodal
Memory	8 bytes wide, 100 cycles first part, 2 cycles interpart
TLB	16 entry (I) – 4-way set-associative, 32 entry (D) – 4-way set associative, 30 cycles miss latency

We used M-Sim [15] (a significantly modified version of the well known and widely used Simplescalar 3.0d simulator [2]) to get the percentage of the instructions whose tags are protected by the proposed technique. M-Sim simulates an Alpha 21264 machine and accurately models pipeline structures such as the issue queue, the reorder buffer, and physical register file which are implemented separately inside the simulator. For each benchmark, we skipped 200 million instructions and simulated 200 million committed instructions. Table I shows the simulated processor configuration.

5 Results and Discussions

In order to achieve high soft error detection coverage on the register tags with the proposed technique, the number of simultaneously renamed instructions per cycle needs to be as small as possible. This observation is against the general rule that the faster the processor gets the less empty the pipeline is. However since the invested hardware is minimal in our technique, detecting even the small number of errors would be beneficial.

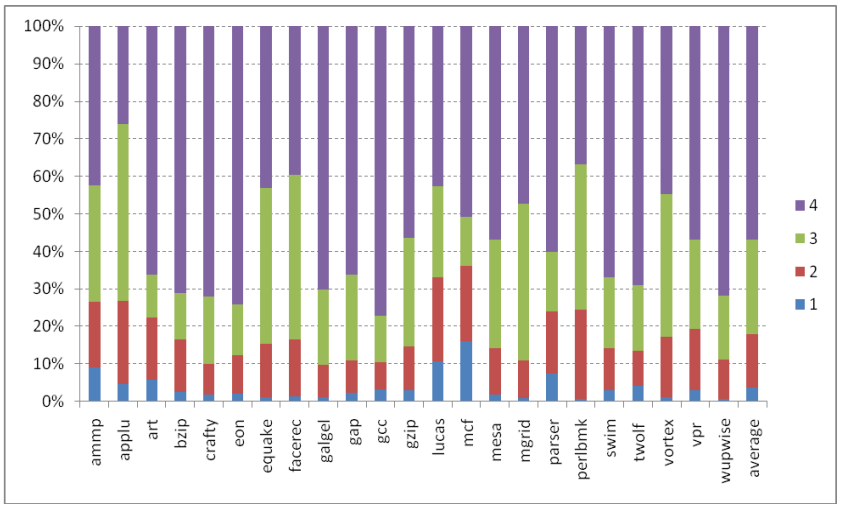

Fig. 3. Number of concurrently renamed instructions

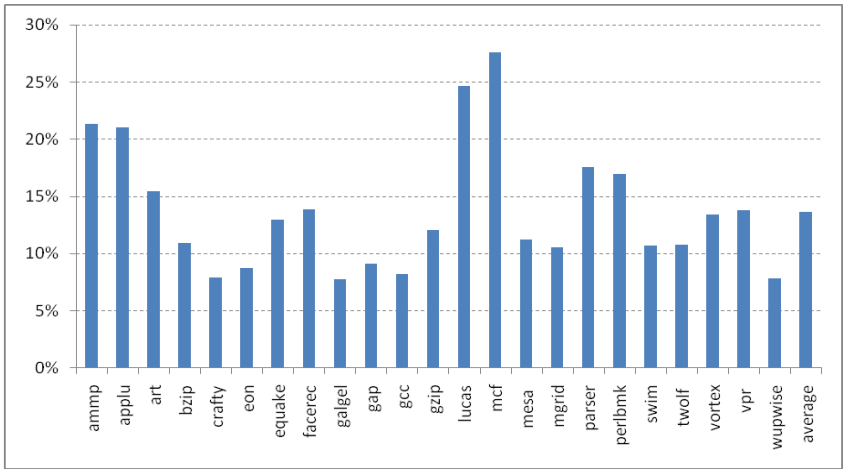

Fig. 4. Reduction of soft error vulnerability in the register tags

Fig. 3 shows the number of concurrently renamed instructions for spec 2000 benchmarks. Having no instructions in the rename stage does not have any benefits

for the proposed techniques. In fact the tag fields of the instructions are not vulnerable at all when they do not contain any valid information [10]. Therefore the results from the empty rename stages that show zero renamed instructions are omitted and the rest of the cycles are divided into 4 groups in a 4-way machine. The figure shows that the simulated processor frequently does not use the full rename width on average. On the average across all spec 2000 benchmarks in more than 40% of the utilized cycles the full processor renaming capacity is not used. This result is consistent with the findings of Moshovos in [9].

Fig. 4 shows the vulnerability reduction achieved in the register tags of the instructions by applying the proposed technique. This vulnerability reduction comes either in terms of soft error detection or correction. On the average across all spec 2000 benchmarks it is possible to protect 14% of the ACE bits [10] that belong to the tag fields of all instructions. For some benchmarks such as *mcf* which has a large percentage of cycles with only one renaming instruction, the protection offered by the proposed technique can be as high as 27%.

6 Conclusion and Future Work

In superscalar processors multiple instructions are fetched, decoded, renamed and dispatched each and every cycle. In order to solve the intra group data dependencies, comparator circuits are employed in the rename stage of the pipeline which check if the destination of an older instruction is equal to the sources of a younger instruction. Frequently, because of control dependencies, full fetch width of the machine is not used and the comparators in the rename logic are kept idle until another set of instructions arrive. In this paper we proposed a scheme to detect and correct soft errors by replicating the instruction tags in the frontend of the processor and later checking the equality of these replicated tags at the rename stage of the pipeline by employing the unused comparator circuits. Our results show that, on the average across all spec benchmarks, around 14% of the register tags can be protected against any soft errors in the frontend of the processors by employing the proposed technique.

The results of our study shows an upper bound for the benefits that can be achieved by using the dependency checking logic of the rename stage since it is assumed that each and every instruction uses the destination and source tags. In reality, many instructions don't use some or all of these tag fields and this observation was used for different purposes by many researchers [4][14][19]. By using the unused space inside the instruction slots, it is possible to use the on-chip comparators for detecting more errors on the register tags. The investigation of how to use the unused register tag space for soft error detection is left for future work. Another future direction could be the use of instruction slots when no instructions are renamed in a cycle. These cycles can be used to check the errors that occur on the register tags of the instructions that passed through the rename stage in the previous cycles. Investigation of schemes that makes use of this observation is also left for future work.

Acknowledgments

This work was supported in part by the Scientific and Technological Research Council of Turkey (TUBITAK) through the research grant 107E043, by the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) through the "Reliable Embedded Processors" research cluster and by the European Union (FEDER funds) under contract TIN2007-60625.

References

1. Baumann, R., "Soft Errors in Advanced Computer Systems", in *IEEE Design & Test of Computers*, 2005.
2. Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Technical Report, Det. of CS, Univ. of Wisconsin-Madison, June 1997.
3. Ergin, O., Unsal, O., Vera, X., and González, A., "Exploiting Narrow Values for Soft Error Tolerance", *IEEE Computer Architecture Letters* (CAL), Vol. 5, 2006.
4. Ernst, D. and Austin, T., "Efficient Dynamic Scheduling Through Tag Elimination", in *ISCA*, 2002
5. Gochman, S., et al., "Introduction to Intel Core Processor Architecture", *Intel Technology Journal*, Vol. 10 Issue 2, May 2006.
6. Hinton, G., et al., "The Microarchitecture of the Pentium 4 Processor ", *Intel Technology Journal*, Vol. 5, Issue 1, February 2001.
7. Hu, J. S., et al., "Resource-driven optimizations for transient fault detecting superscalar microarchitectures." In Proc. Of Tenth Asia-Pacific Computer Systems Architecture Conference (ACSAC 05), pages 24–26, 2005.
8. Kessler, R.E., "The Alpha 21264 Microprocessor", *IEEE Micro*, 19(2) (March 1999), pp. 24-36.
9. Moshovos, A., "Power Aware Register Renaming", Computer Engineering Group Technical Report 01-08-2, University of Toronto, 2002.
10. Mukherjee, S. S., et al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor", in *MICRO*, 2003
11. Mukherjee, S. S.,et al. "Detailed Design and Evaluation of Redundant Multithreading Alternatives", in *ISCA*, 2002
12. Phelan, R., "Addressing Soft Errors in ARM Core-based Designs", White Paper, ARM, December 2003.
13. Reinhardt, S. K., and Mukherjee, S. S., "Transient Fault Detection via Simultaneous Multithreading", in *ISCA*, 2000.
14. Sangireddy, R., "Reducing Rename Logic Complexity for High-Speed and Low-Power Front-End Architectures", *IEEE Transactions on Computers*, Vol. 55 No. 6, pp. 672-685, June 2006.
15. Sharkey, J., "M-Sim: A Flexible, Multithreaded Architectural Simulation Environment", Technical Report CS-TR-05-DP01, Dept. of CS, SUNY - Binghamton, October 2005.
16. Sima, D., "The Design Space of Register Renaming Techniques", *IEEE Micro*, Vol. 20, No. 5, pp. 70-83, September/October 2000.
17. Sridharan V., Kaeli D., Biswas A., "Reliability in the Shadow of Long-Stall Instructions", in SELSE-3.
18. Vijaykumar, T. N., et al., "Transient-Fault Recovery Using Simultaneous Multithreading", in *ISCA*, 2002.
19. Yalçın, G., and Ergin, O., "Using Tag-Match Comparators for Detecting Soft Errors", *IEEE Computer Architecture Letters*, Vol 6, 2007.