Using Tag-Match Comparators for Detecting Soft Errors

Gülay Yalçın, and Oğuz Ergin
Dept. of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey
{gulay, oergin}@etu.edu.tr

***Abstract*—Soft errors caused by high energy particle strikes are becoming an increasingly important problem in microprocessor design. With increasing transistor density and die sizes, soft errors are expected to be a larger problem in the near future. Recovering from these unexpected faults may be possible by reexecuting some part of the program only if the error can be detected. Therefore it is important to come up with new techniques to detect soft errors and increase the number of errors that are detected.**

Modern microprocessors employ out-of-order execution and dynamic scheduling logic. Comparator circuits, which are used to keep track of data dependencies, are usually idle. In this paper, we propose various schemes to exploit on-chip comparators to detect transient faults. Our results show that around 50% of the errors on the wakeup logic can be detected with minimal hardware overhead by using the proposed techniques.

***Index Terms*—Comparators, computer architecture, error detection coding, fault tolerance, microprocessors**

I. Introduction

EVER decreasing feature sizes and supply voltages lead to an increased number of transient errors in contemporary microprocessors [2][12]. These errors, which don't result in a permanent fault on the chip (and hence are termed "soft errors"), may cause a system crash if the error cannot be detected. If an error is detected, processor can recover from the error by simply flushing the pipeline and refetching the instructions starting from the faulting one. Therefore, detecting errors that would otherwise go unnoticed and create a silent data corruption is important in processor pipelines.

Different techniques are proposed to cope with soft errors in microprocessors. Protection is usually achieved by employing redundancy; this redundancy may be in time [1], in area [17] [4][6][7] or in information [4][10]. Symptom based detection schemes are also proposed [15].

Instruction scheduling is one of the most critical components of the current superscalar out-of-order microprocessors. After passing through instruction fetch, decode and rename stages, each instruction is dispatched to an issue queue and waits for its operand values before it can be scheduled to an appropriate function unit. Since each and every instruction has to be informed about the availability of register values that hold the generated results, every completing instruction broadcasts the register tag of the produced result to the issue queue. Instructions compare their source register tags with the incoming broadcasted tags on every clock cycle to check if their operands are available. Content Addressable Memory (CAM) cells are used for comparing the stored operand tags and tags broadcasted on the broadcast busses. These CAM cells are built by combining regular SRAM bitcells and comparators that employ dynamic logic. Tag-match comparators generate a mismatch signal most of the time and are employed inefficiently [11].

Instructions that get delayed inside the issue queue are vulnerable to particle strikes. Some solutions such as flushing the instruction queue upon cache misses were proposed to reduce this vulnerability at the expense of some performance loss of refilling the pipeline [16].

In this paper, we propose several schemes to use the tag-match comparators, which are already available inside the processors, to detect soft errors that may occur both on the data and the logic circuits. We first show that some instructions have identical operand tags and use this information to validate the match signal generated by two comparators that are wired to broadcast busses. Afterwards, the same idea is extended to the instructions that have only one operand by replicating the operand into the second tag field. Finally we use the observation that most of the instructions enter the issue queue with at least one operand ready [5] and exploit the unemployed comparators by replicating the unready tag into two source fields.

II. Architectural Soft Error Vulnerability

When a particle strikes a processor component, it is not for sure that it will create an error. Even if the particle causes a bit flip, the bit whose content is changed may not be holding valid information or its content may not be affecting the final program output. Bits that affect the final program outcome are called the bits needed for "Architecturally Correct Execution" and therefore these bits are called the ACE bits [9]. Percentage of the ACE bits inside a processor component determines its vulnerability to soft errors. Architectural soft error vulnerability factor of a processor component is found using

Manuscript submitted: 28-Jun-2007. Manuscript accepted: 28-Jul-2007. Final manuscript received: 01-Oct-2007

G. Yalçın is with the Department of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey (e-mail: gulay@etu.edu.tr).

O. Ergin, is with the Department of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey (phone: +90-312-292-4059; fax: +90-312-292-4081; e-mail: oergin@etu.edu.tr).

the following formula:

$$AVF = \frac{\text{Average number of ACE bits resident in a hardware structure in a cycle}}{\text{Total Number of bits in the Hardware Structure}}$$

Some bits of the instructions are not vulnerable to soft errors. For example if the instruction does not use an immediate operand, immediate field reserved in the issue queue is not used by the instruction and hence is not vulnerable to any particle strikes.

III. Structure of the Instruction wakeup Logic

The instruction scheduling logic employs a number of comparator circuits for each instruction to check for data dependencies. Fig. 1 shows the structure of an issue queue entry for a processor that has four forwarding paths and hence allows four generated results to be broadcast to the issue queue simultaneously. Each instruction compares both of its tags with the value on each of the forwarding buses on each cycle. Therefore, if there are N forwarding busses, N comparator circuits are needed for each tag storage. If any of the comparators produce a match signal, the bit that indicates that the tag is available, is set. If both tags become available, the instruction is marked as "ready to schedule". The number of bits required for the tag storage depends on the number of physical registers available inside the processor. If there are M physical registers, size of a tag field is $\log_2 M$. This number also shows the number of inputs for the comparators shown in Fig. 1.

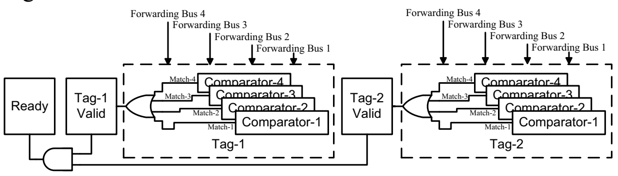

Fig. 1. Structure of the wakeup logic of a single entry of the issue queue.

IV. Using Identical Tag Information for Error Detection

Many of the instructions have more than one source operand in common workloads and occasionally an instruction uses the same register value for more than one of its operands. In those cases when both source tags are identical, all of the match signals generated by the comparators of source tags have to be identical in each cycle until the instruction is issued. If any one of these match signals does not match the corresponding match signal of the other tag, this must be a result of an error either on the source tags or on the comparison logic (or even the forwarding logic altogether). Errors occurring on the tag part of the instructions with identical tags can be detected by simply identifying those instructions and comparing the match signals of the comparators at each cycle.

We propose adding an *Identical_tags* identifier bit for each entry inside the issue queue to detect soft errors. This bit is set at the register rename stage when both tags indicate the same register as the source operand and is reset whenever the instruction is issued. At each cycle, when the comparators produce a match or mismatch signal, the logic circuit shown in Fig. 2 is used to detect an error that may have occurred on the stored tags or the wakeup logic. Note that the combination of XOR gates and the OR gate in Fig. 2 is itself a 4-bit comparator and can be implemented by using any kind of logic, including the dynamic logic of the comparators used in the issue queue. The number of inputs needed for the error checker comparator is equal to the number of forwarding busses available inside the issue queue (equal to 4 in the example).

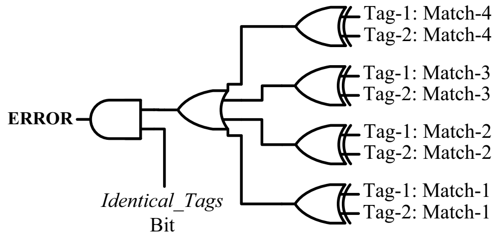

Fig. 2. The error detection circuit when identical tags are identified.

Identical_tags bit itself is not vulnerable to soft errors since a particle strike on this bit at worst results in an erroneous error signal. When an error is detected, all the instructions that come after the faulting instruction are squashed and refetched. Therefore a wrong error signal at worst causes some performance loss due to an unneeded recovery.

V. Exploiting Unused Tags for More Error Coverage

Most of the instructions don't use more than one operand [5]. These instructions only use a single tag and the comparators of the second tag field are not used until the instruction is issued and a new instruction is placed inside the entry. It is possible to protect the used tag field against soft errors by copying the used tag value into the second tag field and using the available comparators for error checking. After the instruction is replicated the same hardware mechanism proposed for protecting the instructions that have identical tags can be used. The *Identical_tags* bit is set and the same error checking logic depicted in Fig. 2 is used to detect soft errors that occur on the tag data and wakeup logic.

Copying the single tag into both tag fields does not alter the normal operation of issue logic since this portion of the issue queue is not used for instructions with single operand. As long as the *Identical_tags* bit is reset after the instruction leaves the issue queue, error checking will be accomplished safely.

VI. Exploiting the Tag Readiness for Improving Error Detection Coverage

It is previously reported that a large percentage of instructions that use two operands, enter the issue queue with at least one operand ready [5][13]. In order to extend the effectiveness of the proposed soft error detection circuit, the unavailable tag of those instructions that enter the issue queue with one available operand, can be copied into the field reserved for the second tag and the comparators of the second field can be used to protect the stored tag as long as the instruction resides in the issue queue. However copying the tag of unavailable operand inside the second tag is not a safe operation since the second tag field now contains valid information. In order to use the proposed error checking method on such instructions, second tag of the instruction has to be stored in a separate payload RAM like it was done in [13]. Note that although this new storage space used for saving the second tag contains the same number bits, it is not wired to the incoming forwarding buses and it does not contain any comparison circuitry.

It is important to keep track of the location of the operand tags in this scheme. The processor has to remember which tag was unready and was placed inside the payload area so that the instruction can be issued without any problems. A simple solution to this problem is to always protect one of the tags if the other one is already available. This way the tag stored in the payload area will always belong to the same operand and the processor will always know where to read its operands from. However, deciding which operand tag will be protected at design time does not work well in all benchmarks. In some programs it is usually the first operand that is ready, while in others it is the second one. Therefore we chose to cover all of the instructions that enter the issue queue with one ready operand by introducing another bit (called the *Stored_tag* bit) that shows which tag is stored in the payload area. This bit is used along with the *Identical_tags* identifier bit. When an instruction that uses two operands enter the issue queue with one of the operands already available *Identical_tags* bit is set and the tag that is not available is copied in the other CAM field. Available tag is stored inside the payload area and *Stored_tag* bit is reset to 0 if the stored value is the tag of the first operand and the bit is set to 1 if the stored tag belongs to the second operand.

In order to use this scheme along with the previous schemes, a small modification is necessary. When an instruction has two identical tags, the tag value has to be moved into the payload area so that the issue/RF read logic reads the correct operand at issue time. Since both tags are identical the value of *Stored_tag* bit is not important. For the instructions that only have a single operand, no changes are necessary as the instruction will always read its single operand from a fixed location. Unfortunately the *Stored_tag* bit is vulnerable to particle strikes and is an ACE bit when the instruction that uses it has two operand values and one of these operands is ready at the time of dispatch. If this bit is flipped when it holds valid information, issue logic will use the payload area as the source for the wrong tag. For other instructions this bit is not an ACE bit.

VII. Results and Discussions

In order to get an accurate idea of the error detection coverage achieved by using the proposed technique we used M-Sim [14], a significantly modified version of the Simplescalar 3.0d simulator [3]. 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 1 billion instructions and simulated 200 million committed instructions. Table I shows the simulated processor configuration.

TABLE I
SIMULATION PARAMETERS

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 registers
Function Units and Latency	Integer ALU (1/1), load/store unit (2/1), integer multiply (3/1), integer division (12/12), floating-point addition (2/1), floating-point multiplication (4/1), floating-point division (12/12).
L1 I-Cache	32 KB, 2-way set-associative, 32 byte line, 1 cycle hit time
L1 D-Cache	32 KB, 4–way set–associative, 32 byte line, 1 cycle hit time
L2 Unified Cache	512 KB , 8–way set–associative, 64 byte line, 6 cycles hit time
BTB	512 entry, 4–way set–associative
Branch Predictor	2K entry bimodal
Memory	8 bytes wide, 100 cycles first chunk, 1 cycle inter chunk
TLB	16 entry (I) – 4-way set-associative, 32 entry (D) – 4-way set associative, 30 cycles miss latency

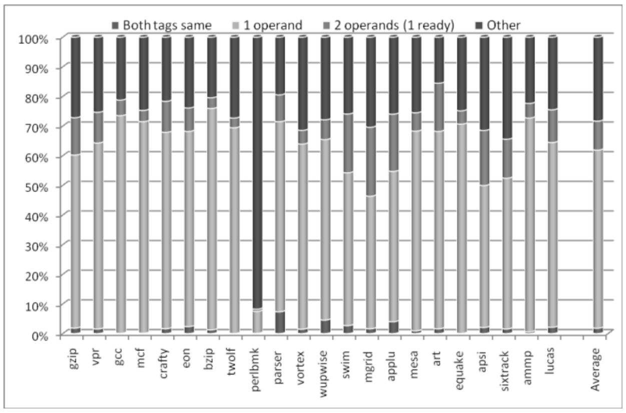

Fig. 3. Operand statistics of dispatched instructions

Fig. 3 shows the operand statistics for all instructions dispatched to issue queue for 20 programs of the spec CPU2000 benchmark suite. For each benchmark program, the bar shows the percentages of instructions that can be used for our error detection scheme. Bottom part of the bar shows the percentage of the instructions that has identical operand tags. Most of the benchmarks don't have many of this kind of instructions; on the average across all benchmarks 1.78% of the instructions have both tags identical. However some benchmarks show more of these instructions; for example more than 7% of the instructions in *parser* have identical operand tags. Fig. 3 reveals that most of the instructions in spec 2K workloads operate with a single operand. This behavior was also observed by Ernst in [5]. Most of the instructions in common workloads don't use both operand tag fields. On the average across all benchmarks, 60.1% of the instructions operate with one operand. Operand tags of all these instructions can be protected against soft errors by using the proposed technique.

Extending the error detection coverage to the instructions that use both source operands requires more hardware investment. Fig. 3 shows that 2-operand instructions that enter the issue queue with one of the operands ready, amount to 9.8% of all committed instructions on the average across all benchmarks. In total, by using slightly different detection schemes, it is possible to detect soft errors on the tags of around 71.7% of the instructions. However the percentage of instructions that are covered by the proposed schemes does not directly correspond to the percent reduction in the architectural soft error vulnerability of the tag part of the issue queue. AVF reduction in the tag part of the issue queue can be computed by looking at the number of cycles spent and the number of vulnerable bits occupied by each instruction. Therefore the instruction's vulnerability can be computed by the following formula (for the tag part):

$$AVF_{instruction} = number\ of\ cycles\ spent \times number\ of\ ACE\ bits\ occupied$$

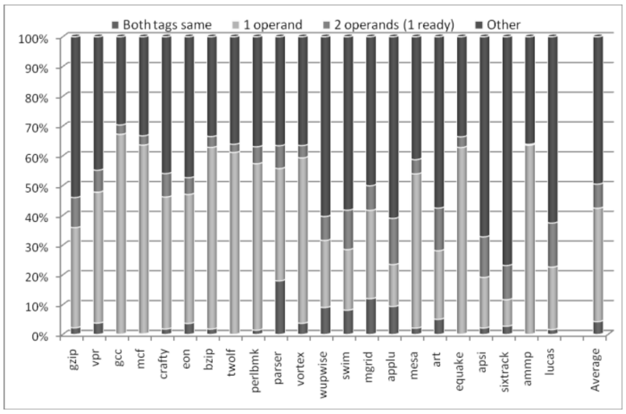

Fig. 4. Contribution of the instruction types to the tag field vulnerability

Instructions that don't use the any operands are not vulnerable particle strikes on the tag part. The instructions that use a single operand have only the bits of one tag occupied (7 in our processor) and instructions that use both tags are the most vulnerable ones. In order to evaluate the benefits of our schemes, we multiplied the number of cycles spent in the issue queue for each instruction and multiplied this value with the number of vulnerable tag bits of that instruction. Fig. 4 shows the breakdown of contribution of each instruction type to the vulnerability of the tag field.

Although the instructions that use two operands are small in number when compared with single-operand instructions, they contribute to the soft error vulnerability more than the other instructions as shown in Fig. 4. For example, instructions that have two identical operand tags are 1.78% of all instructions as shown in Fig. 3; however these instructions hold 6.86% of the vulnerable bits in the tag field of the issue queue. On the average across all benchmarks around vulnerability of the tag field of the issue queue can be reduced by 50% by using the proposed techniques. It is previously reported that around 10% of all soft errors occur in the issue queue of a processor on average across all spec 2000 benchmarks [7]. Vulnerability of the tag field of the issue queue amounts to 24.7% of the structure. Therefore our proposed techniques can detect roughly 1.25% of all the errors that occur in the whole processor.

Our techniques introduce some energy dissipation overhead to the issue queue. A new circuit which is actually a small 4-bit comparator is added to each entry of the issue queue and one tag-size comparator has to be used in the rename stage to detect instructions that have identical tags. This new circuitry along with the data replication energy dissipation will slightly increase the dynamic and static energy dissipation of the processor. However since the techniques described in this paper rely on the use of already available comparators, energy dissipation overhead is a lot less than full data replication.

VIII. Conclusion

Comparators that are used in the wakeup logic of current microprocessors are not used efficiently. We showed that by replicating tag data, it is possible to use these idle comparators for detecting soft errors that occur on the tag storage area of the issue queue. Since most of the instructions don't even use a second operand, a large percentage of instructions benefit from our schemes. We showed that around 50% of the errors that occur on the storage area and the wakeup logic of the tags of instructions can be detected by including a single bit and replicating the value inside the tag storage.

References

[1] Austin, T. M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", in *MICRO*, 1999
[2] Baumann, R., "Soft Errors in Advanced Computer Systems", in *IEEE Design & Test of Computers*, 2005
[3] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Technical Report, Det. of CS, Univ. of Wisconsin-Madison, June 1997.
[4] Ergin, O., et al., "Exploiting Narrow Values for Soft Error Tolerance", *IEEE Computer Architecture Letters* (CAL), Vol. 5, 2006
[5] Ernst, D., Austin, T., "Efficient Dynamic Scheduling Through Tag Elimination", in *ISCA*, 2002
[6] Hu, J., et al., "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability", in DSN 2006
[7] Li, X., et al., "SoftArch: An Architectural-Level Tool for Modeling and Analyzing Soft Errors", in *DSN,* June 2005
[8] Memik, G, et al., "Increasing Register File Immunity to Transient Errors", in *DATE'05*, 2005
[9] Mukherjee, S. S., et al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor", in *MICRO*, 2003
[10] Phelan, R., "Addressing Soft Errors in ARM Core-based Designs", White Paper, ARM, December 2003
[11] Ponomarev, D. V., et al., "Energy Efficient Comparators for Superscalar Datapaths", IEEE Transactions on Computers, Vol. 53, No. 7, July 2004
[12] SIA, International Technology Roadmap for Semiconductors 2005, http://www.itrs.net/Links/2005ITRS/Home2005.htm.
[13] Sharkey, J. J., et al., "Instruction Packing: Toward Fast and Energy-Efficient Instruction Scheduling", in ACM TACO, Vol. 3, No. 2, June 2006, pp. 156–181.
[14] Sharkey, J., "M-Sim: A Flexible, Multithreaded Architectural Simulation Environment", Technical Report CS-TR-05-DP01, Dept. of CS, SUNY - Binghamton, October 2005.
[15] Wang, N., and Patel, S. J., "ReStore: Symptom Based Soft Error Detection in Microprocessors", in *DSN*, 2005
[16] Weaver, C., et al., "Techniques to Reduce the Soft Errors Rate in a High-Performance Microprocessor", in *ISCA*, 2004
[17] Zhang, W., et al., "ICR: In-Cache Replication for Enhancing Data Cache Reliability", in *DSN*, 2003