Exploiting Existing Comparators for Fine-Grained Low-Cost Error Detection

GULAY YALCIN, Barcelona Supercomputing Center

OGUZ ERGIN and EMRAH ISLEK, TOBB University of Economics and Technology

OSMAN SABRI UNSAL and ADRIAN CRISTAL, Barcelona Supercomputing Center

Fault tolerance has become a fundamental concern in computer design, in addition to performance and power. Although several error detection schemes have been proposed to discover a faulty core in the system, these proposals could waste the whole core, including many error-free structures in it after error detection. Moreover, many fault-tolerant designs require additional hardware for data replication or for comparing the replicated data. In this study, we provide a low-cost, fine-grained error detection scheme by exploiting already existing comparators and data replications in the several pipeline stages such as issue queue, rename logic, and translation lookaside buffer. We reduce the vulnerability of the source register tags in IQ by 60%, the vulnerability of instruction TLB by 64%, the vulnerability of data TLB by 45%, and the vulnerability of the register tags of rename logic by 20%.

Categories and Subject Descriptors: C.2.1 [Computer Systems Organization]: Dependable and Fault-tolerant Systems and Networks

General Terms: Reliability, Error Detection

Additional Key Words and Phrases: Reliability, CAM logic

ACM Reference Format:

Gulay Yalcin, Oguz Ergin, Emrah Islek, Osman Sabri Unsal, and Adrian Cristal. 2014. Exploiting existing comparators for fine-grained low-cost error detection. *ACM Trans. Architect. Code Optim.* 11, 3, Article 32 (October 2014), 24 pages.

DOI: <http://dx.doi.org/10.1145/2656341>

1. INTRODUCTION

Components in shipped chips may fail for several reasons, such as permanent (i.e., aging, wear-out, design defects, infant mortality) or transient faults (i.e., soft errors) [Baumann 2005; Anglada and Rubio 1988]. As technology scales (i.e., smaller transistors, lower voltage), variability and degradation in the performance of transistors make them more vulnerable to faults. Therefore, fault tolerance is one of the major aspects in processor design so that the system can detect a fault, recover from the fault, and repair/reconfigure around the failed component.

Modern microprocessors use aggressive techniques like out-of-order execution and dynamic scheduling for boosting performance. However, due to a higher number of

This work was partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under research grants 112E004. The work is performed in the framework of COST ICT Action 1103 “Manufacturable and Dependable Multicore Architectures at Nanoscale.”

Authors’ addresses: G. Yalcin, O. S. Unsal, and A. Cristal, C/Gran Capita 2–4, Office 303, Barcelona Spain; emails: {gyalcin, ounsul, acristal}@bsc.es; O. Ergin and E. Islek, Sogutozu Avenue 43, Sogutozu, Ankara, Turkey; emails: oergin@etu.edu.tr, emrahislek@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/10-ART32 \$15.00

DOI: <http://dx.doi.org/10.1145/2656341>

hardware components, out-of-order processors are more prone to faults than in-order ones. Fortunately, in out-of-order systems, if the error can be detected in a short time, it is possible to recover the error and to avoid a system crash by flushing the processor pipeline and refetching the faulty instruction. Therefore, it is important to come up with techniques that are capable of signalling the bit flip effectively in a short time.

As the instructions spend more time in processor structures, their probability of getting hit by a particle increases. There were some previous efforts to reduce this vulnerable period by flushing the pipeline when instructions are stalled for a long period [Weaver et al. 2004]. However, this scheme cannot be utilized to tolerate permanent faults. Parity checking and error-correcting codes (ECCs) can be used to detect and correct faults in the data arrays of the processor [Baylis 1998]. ECCs rely on encoding some information from the stored data and checking this information upon reading the value. Parity and ECC are widely used for cache memories. However, they are not suitable to use for data path components due to a high encoding and decoding delay of ECC. For example, the parity-protected register files of Intel's 90nm Itanium processor need an extra cycle to calculate parity [Fetzer et al. 2006]. Time and space redundancies are other techniques for fault detection. Either a value is replicated into more storage space and later checked with simple voting or a value is generated multiple times with a single or multiple resources. This redundancy can be achieved either in coarse granularity or in fine granularity.

Coarse-grained fault detection mechanisms for multicore systems [Austin 2000; Li et al. 2008b; Mukherjee et al. 2002; Reinhardt and Mukherjee 2000; Slegel 1999; Wang et al. 2006; Wood et al. 2006] have the capability of discovering the faulty core and omitting it for future executions. In particular, lockstepping is a popular error detection scheme widely implemented in systems requiring high reliability such as the IBM S/390 G5 [Slegel 1999] or the HP NonStop servers [Wood et al. 2006]. Lockstepping executes an instruction stream redundantly in two synchronized and lockstepped processors and checks if both produce identical results. In lockstepping, consecutively divergent results signify a permanent fault in one of the cores and the faulty core is extracted from the system for future executions. However, coarse-grained error detection schemes lead to waste of a whole core after error detection, although there are many error-free units in it since the coarse-grained error detection schemes are inadequate to detect the faulty unit in the core. For instance, assuming that a core is faulty due to an error in a register, omitting the whole core, wastes the other error-free registers as well as other error-free units in the core (e.g., TLB, ALU, or decode unit). Hence, *fine-grained error detection mechanisms*, which can detect the faulty units or entries, are required, so that only these units or entries can be omitted after error detection. Thus, a faulty core can be utilized longer, which increases the system lifetime without letting the user know about the self-repairing process.

On the other hand, it is a technical challenge for researchers to develop simple (in terms of complexity effectiveness) and cost-effective (in terms of less performance degradation and power consumption) error detection schemes. For instance, replication of the data and comparison to check the consistency of the replicated fields require additional hardware in the design and additional execution time. Moreover, fine-grained error detection schemes are significantly more costly than their coarser counterparts despite their greater lifetime extensions. Thus, it is essential to come up with solutions to reduce the cost of fine-grained error detection schemes.

In this study, our goal is providing fine-grained, low-cost error detection by leveraging already existing structures in the processor logic in order to detect faulty units. To this end, we exploit already existing content-addressable memory (CAM) logic and comparators in order to detect faults.

CAM cells are utilized in the issue queue (IQ), rename logic, and translation lookaside buffer (TLB) in order to search the broadcasted value in the stored entries. These CAM cells are built by combining regular SRAM bitcells and comparators that employ dynamic logic. The comparators generate a mismatch signal most of the time and are inefficiently employed [Ponomarev et al. 2004]. In this study, we show that these structures can be exploited for fault detection without any performance degradation and with minimum energy overhead and hardware extension. Note that, although the comparators in the rename stage can be implemented differently than the CAM logic, they can still be used to detect soft errors. In the past, comparators and CAM logic were used interchangeably by researchers since talking about the concepts by talking about a comparator that compares two different data is easier to understand by the reader; thus, we also use these concepts interchangeably in this study.

First, in out-of-order execution, the tags of the results of the executed instructions are broadcasted to the issue queue so that the dependent instructions are aware of the fact that their source operand is ready. CAM cells in the IQ are used for comparing the stored operand tags and tags broadcasted on the broadcast buses. We propose several schemes utilizing these CAM comparators to detect faults that may occur both on the stored tags and on the logic circuits. To this end, we present several error detection schemes for the instructions (1) having identical operand tags, (2) having only one operand, and (3) entering the issue queue with at least one operand ready. Second, in TLBs, which are on-chip memory structures caching page table entries, CAM logics compare the broadcasted virtual addresses and the stored virtual addresses in order to see if the translation information is in the TLB. Since page address numbers are generally consecutive for an application, several numbers of bits can be identical between two adjacent entries in the TLB. In this study, we exploit CAM logics in TLBs and the identical bits between two adjacent entries in order to detect errors in virtual addresses. Third, rename logic includes 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. In this study, we present a scheme protecting the register tags of the instructions by replicating the register tags into unused fields of the subsequent instruction slots when the full pipeline width is not used.

The main contributions of the article are as follows:

- We reduce the vulnerability of an out-of-order, superscalar processor against transient and permanent faults without resenting any performance degradation in the common, error-free execution.
- We leverage already existing hardware structures for low-cost, fine-grained fault detection.
- We present the design details of how CAM logic and comparators in TLB, IQ, and rename logic can be utilized for fault tolerance.

We use the gem5 full system simulator to test our schemes [Binkert et al. 2011]. According to our results, we reduce the vulnerability of the source register tags in the IQ by 60%, the vulnerability of the instruction TLB by 64%, the vulnerability of the data TLB by 45%, and the vulnerability of the register tags of rename logic by 20%. Our schemes do not reduce the number of instructions committed per cycle (IPC) but require a small, extra logic that presents a negligible hardware overhead (i.e., less than 8% of the IQ and 3% of the TLB) in the system. We present the fault detection design for the IQ, TLB, and Register Renaming in Section 2, Section 3, and Section 4, respectively. In Section 5, we evaluate our schemes. We present the related work in Section 6, and we conclude in Section 7.

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

2. FAULT TOLERANCE IN ISSUE QUEUE

The IQ is one of the most vulnerable structures in the processor. In this section, we first present the principles of the wakeup logic in the IQ. Then we explain how the CAM logic in the IQ can be utilized for fault detection.

2.1. Instruction Wakeup Logic

Instruction scheduling is one of the most critical components of the current super-scalar out-of-order microprocessors. After passing through instruction fetch, decode, and rename stages, each instruction is dispatched to the IQ and waits for its operand values to be ready 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 (i.e., an instruction that completed the execute stage, not necessarily committed) 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. When the instructions are issued to the function units, the information stored inside the corresponding entry is read from the issue queue and transferred to the function unit that will execute the instruction.

The issue queue of the microprocessor is designed using both the SRAM bitcells and CAM cells. CAM cells are used for comparing the stored operand tags and tags broadcasted on the broadcast buses [Ponomarev et al. 2003]. The CAM cells are, in fact, regular SRAM bitcells with comparator circuits attached to them [Ponomarev et al. 2004]. The source tags of instructions are compared to all the tags that are broadcast to the queue each cycle. A microprocessor that issues up to four instructions per cycle may produce four results each cycle. The tags of all of these results are broadcast to the issue queue so that the dependent instructions are aware of the fact that their source operand is ready. In the RISC architectures, there are up to two source tags for each instruction, which mandates the use of two source tag fields inside each entry of the issue queue even though they may not be used.

Figure 1 shows the structure of the tag storage space inside the issue queue in a processor where the register tags of up to four results are broadcast to all of the entries. Each of the source register tag fields contains four separate comparators in order to compare the stored tag value with the values on all of the forwarding buses. Since both source tags may contain any value, all of the forwarded tag values need to be broadcast to two tag field columns separately. In a processor that contains N forwarding buses inside its issue queue's wakeup logic, N comparator circuits are used in each tag storage space. When the stored tag value matches the value on any of the values on the forwarding buses, the corresponding comparator circuit produces a match signal and the valid bit of the corresponding tag is set. After both of the valid bits are set, the ready bit of the instruction is set, and the instruction becomes available to be selected for execution.

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

The number of bits stored inside the tag fields is a function of the number of registers present inside the processor. The use of register renaming together with superscalar and out-of-order designs increases the necessity for more physical registers than architectural registers, which increases the register tag size in the issue queue. If there are R physical registers in a processor, the number of bits stored inside the tag field is $\log_2 R$. This number also shows the number of bits for each tag value forwarded to the issue queue and the number of inputs for each comparator circuit.

2.2. Exploiting IQ Comparators for Reducing AVF

We propose several schemes to detect faults that may occur both on the data and the logic circuits by utilizing the CAM comparators that are already available inside the IQ. We first show that, for the instructions having identical operand tags, we can use this information to validate the match signal generated by two comparators that are wired to broadcast buses. Afterward, we extend the same idea to the instructions that have only one operand by replicating the operand into the second tag field. Finally, for the instructions entering the issue queue with at least one operand ready, we exploit the unemployed comparators for fault tolerance by replicating the unready tag into two source fields.

2.2.1. 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 the 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 a mismatch signal, the logic circuit shown in Figure 2 is used to detect an error that may have occurred on the stored tags or the wakeup logic. Note the combination of XOR gates and that the OR gate in Figure 2 is itself a 4-bit comparator and can be implemented by using any kind of logic, including the dynamic logic comparators [Ponomarev et al. 2004] used in the issue queue. The number of inputs needed for the error checker comparator is

Fig. 3. Example of an instruction using the same register value for two of its operands.

equal to the number of forwarding buses available inside the issue queue (equal to 4 in the example).

Under the assumption of a single-bit fault model, the Identical_tags bit itself is un-ACE 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.

Figure 3 shows an example of the proposed scheme. When an instruction in the form Opcode X, R56, R56 enters the issue queue, where R56 shows the physical register tags of the sources for the instruction, the number 56 is placed in storage space reserved for both source tag 1 and source tag 2. In this case, since the stored value is the same for both storage components, the output of the comparators should satisfy $C_1 = C_5$, $C_2 = C_6$, $C_3 = C_7$, and $C_4 = C_8$. In this case, in order to detect the soft error using these comparator circuits, the Identical_tags bit is set and the detection circuit depicted in Figure 2 is employed. Note that the instruction does not need to produce a result value and that its opcode is not important in order to use this technique. The only necessary condition for instruction is to have identical source register tags.

2.2.2. Exploiting Unused Tags for More Error Coverage. Most of the instructions do not use more than one operand [Ernst and Austin 2002]. 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 Figure 2 is used to detect soft errors that occur on the tag data and wakeup logic.

As long as the Identical_tag bit is reset after the instruction leaves the accomplished safety, 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 a single operand. Also, the tag is copied at the register rename stage when the Identical_tags identifier bit is set so that no extra update operation is required in the issue queue.

Figure 4 shows the example of an instruction with only one source register using the proposed scheme. This instruction must be in the form Opcode X, R56 (e.g., NOT AX in x86 instruction set), and it does not need to have a destination register. After the tag is replicated, the tags are identical and the technique described in Section 2.2.1 can be applied.

2.2.3. Exploiting the Tag Readiness for Improving Error Detection Coverage. It was previously reported that a large percentage of instructions that use two operands enter the issue

Fig. 4. Example of an instruction with only one source register.

queue with at least one operand ready [Ernst and Austin 2002; Kim and Lipasti 2003; Sharkey et al. 2006]. 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, the second tag of the instruction has to be stored in a separate payload RAM as was done in Sharkey et al. [2006]. Note that although this new storage space used for saving the second tag contains the same number of 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 always protecting the same 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 from where to read its operands. 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 it is the second one in others. 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 enters the issue queue with one of the operands already available, the Identical_tags bit is set, and the tag that is not available is copied in the other CAM field. The available tag is stored inside the payload area, and the Stored_tag bit is reset to 0 if the stored value is the tag of the first operand and or 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 the 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

Fig. 5. Example of an instruction with two source registers, one of them ready at dispatch time.

payload area as the source for the wrong tag. For other instructions, this bit is not an ACE bit. Note that the available tag is copied to the RAM area and the unavailable tag is also copied to the second tag area at the end of the rename stage so that the issue queue does not require any additional read/write port for error detection.

Figure 5 shows an example of the proposed extension. In the example, physical register R17 is already available at dispatch time, but the value of physical register R56 is not available. Since R56 needs to be wired to the comparators to understand the produced values, the tag R17 is moved to the payload area, and R56 is replicated to the second source tag field at dispatch time. The Stored_tag bit is set to 1 in order to indicate that the payload area holds the second source of the instruction that occupies the issue queue entry.

It is possible to extend soft error detection protection to the instructions that enter the issue queue with two unavailable operands by employing the replication scheme when one of the operands becomes available after the instruction is inserted into the issue queue. This can be achieved by monitoring the valid bits of the source operand tags every cycle and moving the tag that becomes available to the payload area. However, this requires some circuit-level modification to the CAM cell as additional ports are needed to read the ready tag and write it into the payload area. This can be done either by increasing the number of read ports of the CAM cells and the number of write ports of the SRAM cells of the payload area by one or through some custom logic similar to the checkpointing logic used in the rename tables and register files [Ergin et al. 2006]. Because of its circuit-level complexity, we did not explore this case as an extension to our soft error detection schemes quantitatively.

3. FAULT TOLERANCE IN TLB

In this section, we explain the fundamentals of the TLB and then present how already existing structures in the TLB can be used for error detection.

3.1. Translation Lookaside Buffer

Historically, virtual memory [Hennessy and Patterson 2012; Jacob and Mudge 1998] was invented to relieve programmers from the burden of ensuring that a program does not exceed the physical memory by giving the illusion of very large memory to the process. Moreover, virtual memory allows the physical memory to be shared among many processes, and it also allows the same program to run in any location in physical memory, which simplifies loading the program for execution.

In virtual memory, the virtual addresses produced by processors are translated to physical addresses through the page table, which keeps track of where the virtual pages are loaded in the physical memory. However, since the page table is usually so large, it is stored in main memory, which makes every memory access at least twice as long (one for page table and one for the data). In order to reduce the extra memory

access overhead, the *translation lookaside buffer* (TLB) caches the mostly used address translations to avoid page table access for them.

TLBs are on-chip memory structures that cache page table entries. Since TLBs translate both instruction and data stream addresses (i.e., typically known as the instruction TLB and the data TLB in split TLBs), TLBs are accessed every clock cycle. The virtual page address (VPA) is broadcasted to the TLB, which is compared with the stored VPAs in each entry via comparators. If the translation information is in the TLB (i.e., TLB hit), the system can translate the virtual address to a physical address without accessing the page table. If the translation information is not in the TLB (TLB miss), the system searches the translation in the page table and inserts it into the TLB.

The TLB is vulnerable to hardware faults such as transient and permanent faults. Assuming that there was a fault in virtual page number, it may be converted to another address and the address translation may result in accessing an incorrect physical address, which may even cause a system crash. In another scenario, the system cannot find the address in the TLB although the uncorrupted address mapping is cached in the TLB, which causes performance degradation in the system. Also, the TLB takes a significant part in the die area (i.e., $\sim 5.3\%$ of an out-of-order core, which has L1 data and instruction cache in it [Alp]; this ratio is higher in simple in-order cores than out-of-order cores). Therefore, it is considerably vulnerable to hardware faults. Thus, it is essential to provide fault tolerance for TLBs to detect faults and recover from them.

3.2. Error Detection in TLB

Obviously, each entry in a TLB is different from each other at a time, so that only one entry may hit for a broadcasted VPA. However, generally, page address numbers are consecutive for an application. Thus, several numbers of bits can be identical between two adjacent entries in a TLB. In this study, we will exploit the comparators and the identical bits between two adjacent entries in order to detect errors in VPAs. The fundamentals of this error detection scheme are as follows:

- We divide VPAs into two parts as VPA_U and VPA_L .
- We divide the comparators into two parts as $Comparator_U$ and $Comparator_L$.
- Each entry marks if the VPA_U in the entry is identical with the VPA_U of the above entry.
- If VPA_U s are identical in two adjacent entries, their $Comparator_U$ s should give the same result (i.e., hit or miss) for every address comparison.

In order to decide the effective number of bits for the division of VPAs and comparators, we conduct a simulation in the environment we define in Section 5.1 to quantify generally what number of bits are identical in the TLB. In the experiment, every time a VPA is inserted into a TLB (e.g., a VPA is inserted into entry i), we compare the virtual address with the above neighbor of the new entry (i.e., VPA in $i-1$) to evaluate how many most significant bits of VPAs are identical between two entries. According to the result we present in Figure 6, on average, for 40% of the entries, at least 25 most significant bits are identical with their above neighbor (i.e., at most five least significant bits are different). More significantly, for 80% of the entries, at least 20 most significant bits are identical with the above neighbor.

In Figure 7, we present the architecture design of a TLB for fault tolerance. In order to increase the coverage, we divide VPAs as 20 bits for VPA_U and 10 bits for VPA_L . Similarly, in each entry, we divide the comparators into two parts as $Comparator_U$ for the most significant 20 bits and $Comparator_L$ for the least significant 10 bits. We also extend each TLB entry with two bits: *upperHit* and *identical*. *Identical* bit signifies that the VPA_U of the entry is identical with the VPA_U of the adjacent above entry. *upperHit*

Fig. 6. The number of identical bits between two adjacent entries in the TLB.

Fig. 7. The architecture design of a reliable TLB. We divide both the comparators and VPAs into two as ($Comparator_U - Comparator_L$) and ($VPA_U - VPA_L$). We also extend each TLB entry with two bits: identical and upperHit.

bit indicates that VPA_U of the entry was identical with the last broadcasted VPA_U . In the rest of this section, we will explain how these bits are set and reset to detect faults.

3.2.1. Insert into TLB. During the virtual-to-physical address translation, if the translation information is not in the TLB (TLB miss), the system searches the translation in the page table and inserts it into the TLB. Since the TLB is highly utilized, an entry (not the recently used one) is evicted from the TLB before inserting a new one. In Figure 8, we show how the extended bits are updated when an entry is inserted into the TLB.

During the insertion of the address, we need to update the identical bits of two entries, the inserted entry and the entry below the inserted entry (i.e., entry i and $i + 1$ in the figure). In order to update this information, in a system with serialized TLB accesses, we do not need to make another comparison. We can use the information when the system searched the VPA in the TLB when it was missed. Every time a virtual address is searched, the hit/miss information of the VPA_U is saved to upperHit bits. When

Fig. 8. Setting *identical* bit during insertion to the TLB. If the upperHit is set in the entry above, the identical bit is set in the inserted entry. If the bottom entry has an upperHit, the identical bit is set in it.

Fig. 9. The circuit design of error detection in the TLB. If the VPA_U is identical with the above entry, both should hit or miss for any comparison for the most significant part.

the address is inserted following a miss, if the upperHit is set in the entry above, the identical bit is set in the inserted entry. Similarly, if the bottom entry has an upperHit is set, the identical bit is set in the bottom entry. On the other side, in a system that allows out-of-order execution of translations, after the TLB miss is resolved, another comparison is required in order to set upperHit bits. Since a TLB miss is a rare and time-consuming event, the overhead of this extra comparison is not drastic. Note that in order to update upperHit bits without increasing the number of ports in a TLB, we inserted the dedicated circuits between two additional bits.

3.2.2. Error Detection, Recovery, and Repair in TLB. In our reliability design, if VPA_U s of two neighbor entries are identical, their $Comparator_U$ s should hit or miss when an address is searched in the TLB. Otherwise, there is a fault in one of the VPA_U s (either transient or permanent fault). In Figure 9, we present an error detection circuit in a reliable TLB and the truth table of the circuit.

After detecting that there is an error, a reliable TLB should discover if the error is transient or permanent. Also, it is essential to diagnose the faulty entry if the error is permanent in order to omit the entry for future executions. To be able to distinguish if the detected error is transient or permanent, we save the possibly faulty entries, and we flush the pipeline and these possibly faulty entries. If an error is detected again in the related entries in the future execution, it signifies that the error was permanent, and the faulty entry should be omitted from the architecture. Otherwise, the error was

transient, and it was already fixed. In order to save the erroneous entry numbers, we can follow two possible methodologies. In the first one, we can add one more bit to each entry, *errorCandidate*, and set this bit after error detection in the related entries. Every time the entry is detected as error free, the bit is reset. In the second option, after error detection, the operating system (OS) tracks the error candidate entries. In this study, we follow the second option since it does not present an area overhead. However, we argue that when the error rate is very high (e.g., once in every 100 memory accesses), this area overhead should be paid in order to avoid the execution time overhead presented by the OS.

When we detect that one entry is permanently faulty, we may have several candidates for the source of the fault. If we have three adjacent candidates at a time, it is trivial that the one in the middle is permanently faulty.¹ If we have only two candidates, we need to wait for the next error detection. To this end, we flush the faulty entries and pipeline again. We wait for the next error detection. In order to avoid the error detection giving the same two entries the next time, we force the TLB to insert an entry other than these candidates after the flush.

After detecting the permanently faulty entry, first it is deleted from the TLB, and if the entry has a virtual address hit, it is ignored. Second, the TLB is reconfigured so that no memory mappings are inserted into this entry via OS support or hardware switches. If a valid bit exists in TLB entries, this could be easily implemented by permanently setting the entry to invalid assuming that this operation is supported by the system.

3.2.3. Limitations and Possible Improvements. It is obvious that a fault in the TLB can be detected if the fault is in VPA_U s while the proposed method cannot detect the faults in VPA_L s or physical page addresses. In order to increase the error detection capability, we can switch the meaning of identical bit periodically as VPA_L s are identical in two adjacent entries. Also in this case, the upperHit bit signifies that the $Comparator_L$ hits. In this case, a permanent fault in the lower part can be detected and the processor stays functional after omitting the entry. The switching period can be adjusted according to the error rate of the architecture. In this study, in order to keep the design simple, we limit the reliability performance by VPA_U s.

Another limitation of this study is that when the associativity level is lower in a set-associative cache, the reliability performance of our proposal becomes lower. For instance, for a four-way set associative cache, VPA_U s of four addresses belonging to the same set can be protected by using the comparison results obtained during the same lookup operation. Obviously, this scheme cannot be applied when the associativity level is 2 or lower (i.e., direct mapping).

4. FAULT TOLERANCE IN RENAME LOGIC

In this section, first we explain the principles of rename logic, and then we present our fault tolerance design on rename logic.

4.1. Register Renaming

Register renaming is a technique utilized in many out-of-order processors in order to cope with false data dependencies occurring between register operands of subsequent instructions in a straight-line code [Rau and Fisher 1993; Sima 2000]. 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

¹*Proof:* for entries $(i - 1)$, i , and $(i + 1)$, when $identical_i$ and $identical_{i+1}$ are set while $upperHit_i$ does not match either $upperHit_{i-1}$ or $upperHit_{i+1}$, these three entries become candidates. Assuming that only one of these three entries can be faulty, $upperHit_{i-1}$ and $upperHit_{i+1}$ match while VPA_U s are identical in entries $(i - 1)$ and $(i + 1)$. Thus, entries in $(i - 1)$ and $(i + 1)$ are error free.

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. Out-of-order 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 [Sima 2000].

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 mapping table is called the “register alias table (RAT)” or, in short, the “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 [Sager et al. 2001]. This logic includes 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. Note that those registers are architectural registers. Because of the fact that the processor pipeline is not filled to its capacity every cycle, the comparators of the renaming logic are not always utilized.

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 there is an available free register, the instruction grabs that 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 [Sager et al. 2001]. 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 either may increase the cycle time or may not even be possible due to some design choices. Therefore, the register renaming stage of the pipeline includes a dependency checking logic to detect intragroup dependencies.

Figure 10 shows the structure of the dependency checking logic for a machine that renames three 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 multiple updates of an entry in the rename table at a 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 C1...C9 are fed into the priority decoders to control the access of the instructions to the mapping table.

Fig. 10. Rename logic.

4.2. Fault Detection in Rename Logic

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. 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 faults that occur on the register tags of the instructions when they are passing through the front end 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.

In this section, we explain the error detection scheme when there are one, two, and more than two instructions renamed concurrently at a cycle.

4.2.1. Single Instruction Is Renamed at a Cycle. 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 four, without adding any additional comparator circuit. In order to maximize the error coverage, the tags of the single running instruction are copied to the empty fields of the unused slots, as shown in Figure 11, as early as possible in the pipeline. This copy operation is most 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.

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 Figure 11, if the outputs of the comparators C2 and C3 mismatch, it can only be the result of an error on the destination register identifier of the instruction. Similarly, the outputs of the comparators C8 and C9 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 C2 or

Fig. 11. Error detection example for single instruction renaming.

C3 indicates that an error has occurred. Using the output of 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 four instructions each cycle (i.e., four-wide processors). Although Figure 11 shows a three-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 four-wide machine.

4.2.2. Two Instructions Are Renamed at a Cycle. The processor renames one instruction at a cycle only if there is an event causing a performance degradation such as 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 four-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.

4.2.3. Renaming More Than Two 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 four-wide processor, the proposed technique will not be able to provide any soft error detection coverage since there will not be any empty slots or idle comparators.

5. EVALUATION

5.1. Simulation Setup

We use the GEM5 full-system simulator [Binkert et al. 2011] using the spec cpu2006 [Henning 2006] benchmark suite. We execute the applications for either 4 billion

Table I. Simulation Configuration

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	4 Int Add (1/1), 1 Int Mult (3/1)/Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1 FP Mult (4/1)/Div (12/12)/Sqrt (24/24)
L1 Icache	32KB, 2-way set associative, 32-byte line, 1-cycle hit time
L1 Dcache	32KB, 4-way set associative, 32-byte line, 2-cycle hit time
L2 cache unified	512KB, 8-way set associative, 64-byte line, 6-cycle hit time BTB, 512KB, 4-way set associative
TLB	16 entry (I) full associative, 32-entry (D) full associative, 30-cycle miss latency

instructions or until application termination with the test dataset with the hardware parameters presented in Table I.

We use the architectural vulnerability factor (AVF) [Mukherjee et al. 2003] to evaluate the reliability performance of the proposed error detection schemes. In the next section, we explain the details of AVF analysis. Then we present our experimental results.

5.2. Architectural Vulnerability Factor

Even though a particle strike occurs on a processor component and a bit is flipped as a result of this strike, the program running on the processor may not get affected. This is because the values of many bits inside the processor components are not required for correct execution. However, some bits stored inside the components are critical, and an error on these bits will be observed in the final program outcome. The bits that are needed for architecturally correct execution (ACE) of the running program were termed ACE bits previously by Mukherjee et al. [2003]. Since the ACE bits are the ones that are important for correct program outcome, it is important to protect exactly these bits inside the processor components.

Each structure in a processor contains a variable number of ACE bits over program execution time. The number of ACE bits present in a processor component at a given time determines the level of the structure's vulnerability to soft errors. If the number of ACE bits in the structure is high, it is more probable that a bit flip will result in an error in the final program outcome. The level of a component's vulnerability to soft errors is termed the AVF and is calculated with the following equation:

$$AVF = \frac{\text{Average Number of ACE Bits}}{\text{Total Number of Bits}}. \quad (1)$$

In some structures, such as the program counter, the AVF is almost 100% since a faulty value is very likely to cause a crash in the system. However, in some other structures, such as the branch predictor, a particle strike has no effect in the final program outcome and hence the AVF is 0%. For instance, in a TLB, only the bits of the valid entries are ACE bits.

In a processor design process, in order to verify that the processor meets the reliability targets, the AVF of the structures can be used as a metric. Although recently it was argued that AVF analysis just provides an upper bound with a high error margin [Wang et al. 2007], Biswas et al. [2007] showed that AVF's error margin can be reduced by adding more details to the model. Recently, an online AVF estimation technique was also proposed to estimate the AVF of structures after design time while the processor is actually in use [Li et al. 2008a].

Fig. 12. Operand statistics of dispatched instructions.

5.3. Results

In this section, we present the reduced vulnerability of the IQ, TLB, and rename logic.

5.3.1. Evaluation IQ. Figure 12 shows the operand statistics for all instructions dispatched to the issue queue for applications in the spec CPU2006 benchmark suite. For each benchmark program, the bar shows the percentages of instructions that can be used for our error detection scheme. The bottom part of the bar shows the percentage of the instructions that have identical operand tags. Most of the benchmarks do not have many of these kinds of instructions; on average across all benchmarks, 2.1% of the instructions have both tags identical. However, some benchmarks show more of these instructions; for example, more than 6% of the instructions in gromacs have identical operand tags. Figure 12 reveals that most of the instructions in spec 2K6 workloads operate with a single operand. This behavior was also observed by Ernst and Austin [2002]. Most of the instructions in common workloads do not use both operand tag fields. On average across all benchmarks, 55.1% 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. Figure 12 shows that two-operand instructions that enter the issue queue with one of the operands ready amount to 23.7% of all committed instructions on average across all benchmarks. In total, by using slightly different detection schemes, it is possible to detect soft errors on the tags of around 70.8% of the instructions. However, the percentage of instructions that are covered with the proposed schemes does not directly correspond to the percent reduction in the AVF 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):

$$AVF_{ins} = \text{number of cycles spent} \times \text{number of ACE bits occupied}. \quad (2)$$

Instructions that do not use any operands are not vulnerable to particle strikes on the tag part. The instructions that use a single operand have only the bits of one tag occupied (seven in our processor), and instructions that use both tags are the most

Fig. 13. Vulnerability reduction in the tag fields of the IQ.

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 with the number of vulnerable tag bits of that instruction. Figure 13 shows the percentage of the total vulnerability reduction in the source tags of the instructions in the IQ. On average, across all benchmarks, vulnerability of the tag field of the issue queue can be reduced by 60% by using the proposed techniques. It was previously reported that around 10% of all soft errors occur in the issue queue of a processor on average [Li et al. 2005]. Vulnerability of the tag field of the issue queue amounts to 24.7% of the structure. Therefore, our proposed techniques can detect roughly 1.48% of all the errors that occur in the whole processor.

5.3.2. Evaluation TLB. Alpha architecture uses a split TLB for instructions and data. Hence, we present our experimental results for the instruction TLB (ITLB) and data TLB (DTLB) separately.

In Figure 14, we show the percentage of the entries in TLBs that can be protected by the proposed scheme when we track the most significant 20 bits in TLBs. Instructions present more similarity between entries than data since instruction numbers are generally ordered, and their page numbers are usually consecutive. On average, virtual page addresses of 96% of entries in ITLB and 68% of entries in DTLB are protected by our reliability scheme.

In Figure 15, we present the AVF reduction of the virtual page addresses in the ITLB and DTLB. We ran experiments on spec cpu2006 applications to determine the ITLB and DTLB vulnerability without any reliability scheme; our results indicate 97% and 98% vulnerability for the ITLB and DTLB, respectively (not presented in the graph). Since the TLB is generally full, it is expected to be highly vulnerable to faults. According to our results, vulnerability of virtual page addresses in the ITLB is reduced to 35%. Similarly, the vulnerability of virtual page addresses in the DTLB can be reduced to 54%. On average across all benchmarks, the vulnerability of the virtual addresses in ITLBs is reduced by 64% (i.e., from 98% to 35%) and in DTLBs it is reduced by 45% (i.e., from 97% to 54%) by using the proposed technique. Note that during the vulnerability calculations of TLBs with and without the proposed technique, we did not take into account the case explained in Biswas et al. [2005], in which a fault in the virtual address is benign unless the faulty address is accessed in the TLB. In this study, we assumed

Fig. 14. Percentage of protected entries in a reliable TLB when identical bit indicates that 20 bits are common in virtual addressees.

Fig. 15. The vulnerability of virtual addresses when identical bit indicates that 20 bits are common in virtual addressees.

that all faulty addresses have the probability of being accessed. We follow this manner due to our findings in our earlier fault injection study [Yalcin et al. 2011]. Although soft errors in virtual addresses of TLBs are benign as it is remarked in Biswas et al. [2005], more than 40% of the injected permanent faults lead to errors.

The TLB keeps 5.3% of the area in a superscalar out-of-order core (note that this ratio is higher for simple in-order cores), while virtual addresses account for 45% of the TLB. Thus, our proposed techniques can roughly reduce the vulnerability of the whole processor by 1.27%.

5.3.3. Evaluation Rename Table. 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

Fig. 16. Number of concurrently renamed instructions.

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 a small number of errors would be beneficial.

Figure 16 shows the number of concurrently renamed instructions for spec 2006 benchmarks. In the experiment, we excluded the time of rename stalls when the out-of-order window's resources (i.e., ROB and IQ) are full since in those times mostly no instructions are renamed. Having no instructions in the rename stage does not have any benefit 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 [Mukherjee et al. 2003]. Therefore, the results from the empty rename stages that show zero renamed instructions are omitted, and the rest of the cycles are divided into four groups in a four-way machine. The figure shows that the simulated processor frequently does not use the full rename width. On average across all spec 2006 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 [2002].

Figure 17 shows the vulnerability reduction achieved in the register tags of the instructions by applying the proposed technique. This vulnerability reduction comes in terms of either fault detection or correction. On average across all spec 2006 benchmarks, it is possible to protect 20% of the ACE bits that belong to the tag fields of all instructions. For some benchmarks such as milc, which has a large percentage of cycles with only one renaming instruction, the protection offered by the proposed technique can be as high as 31%.

5.4. Time and Energy Impact

Our techniques introduce some energy dissipation and a slight time overhead to the IQ and TLB. In Table II, we present these overheads and compare the schemes with single-bit parity. We compare our scheme with single-bit parity since it is faster than any ECC scheme with less energy consumption for encoding and decoding. Note that single-bit parity can detect only an odd number of faults.

First, we evaluate the time impact of the scheme in a system running at 1GHz in which one cycle is 1 nanosecond. In the table, we present the encoding and decoding time of the schemes. For the parity protection of the IQ, parity should be calculated for

Fig. 17. Reduction of soft error vulnerability in the register tags.

Table II. Time and Energy Impact of the Proposed Error Detection Scheme Compared to Single-Bit Parity

	Encode/Decode Time (% of a cycle)			
	IQ		TLB	
	Encoder	Decoder	Encoder	Decoder
Parity	0.12 cycle	0.12 cycle	0.18 cycle	0.18 cycle
CAM Logic	-	0.09 cycle	-	0.06 cycle
	Encode/Decode Energy			
	IQ		TLB	
	Encoder	Decoder	Encoder	Decoder
Parity	0.09%	0.09%	0.15%	0.15%
CAM Logic	-	0.06%	-	0.06%
	Additional Structures			
	IQ (RAM)		TLB (upper hit update)	
	Energy	Area	Energy	Area
CAM Logic	4%	4%	1.9%	3.0%

7 bits, which takes 120ps for passing three XOR gates. For the TLB, the parity of 20 bits should be calculated in which the critical path consists of five XOR gates taking 180ps. When we use CAM logic for error detection, we do not need any encoding. For error detection (i.e., decoding), the IQ utilizes the circuit design presented in Figure 2, while the TLB utilizes the design presented in Figure 9. The required time for these circuits is less than 10% of a cycle, and these operations can be accomplished in parallel with accessing the related structures. Thus, it does not require any additional cycle in the system.

Second, we present the energy consumption of these encoding/decoding circuits as the percentage of the total energy consumption of related structures (i.e., IQ or TLB). Both the single-bit-parity scheme and CAM logic present less than 1% energy overhead for encoding and decoding.

Besides the error detection circuit, we need additional structures for the TLB and IQ. For the IQ, we extend each line with a 7-bit RAM area as we explained in Section 2.2.3.

The data that should be written to this RAM area is prepared at the end of the rename stage. Similarly, in the TLB, we extend each line with the circuit in order to update the upperHit bit located in the above line as we explained in Section 2. In Table II, we also present the energy and area overhead of these additional structures together with the encoder/decoder circuits. Our scheme presents only 1.9% energy overhead with and only 3% area overhead in the TLB, which is negligibly low. On the other side, our scheme presents 4% area and energy overhead in the IQ. The highest portion of this overhead belongs to the additional RAM area. In order to avoid this overhead, the reliability gained by using the tag readiness can be sacrificed.

6. RELATED WORK

Several coarse-grained error detection schemes have been studied in the literature in detail, such as redundancy-based error detection [Austin 2000; Mukherjee et al. 2002; Reinhardt and Mukherjee 2000; Slegel 1999; Wood et al. 2006] and symptom-based error detection [Li et al. 2008b; Wang et al. 2006]. Redundancy-based schemes execute the replicated instruction streams in two different cores and compare if both cores produce the same result. In symptom-based error detection schemes, the execution is monitored to detect if there is any symptoms of hardware errors such as fatal traps or high mispredictions. In the repair process of these coarse-grained error detection schemes, the faulty core is omitted for the future executions.

In order to improve the lifetime performance of processors, fine-grained fault tolerance schemes [Bower et al. 2004, 2005; Constantinides et al. 2006; Gupta et al. 2008; Romanescu and Sorin 2008; Shivakumar et al. 2012; Srinivasan et al. 2005; Yalcin and Ergin 2007] have been proposed that detect the faulty unit in a core in order to avoid the waste of the entire core.

Shivakumar et al. [2012] present three types of microarchitectural redundancy within a core: components redundancy, array redundancy, and dynamic queue redundancy. These redundancies can be exploited for self-repairing in the presence of a fault to improve the lifetime of the processor. In this study, we leveraged array redundancy by utilizing the error-free entries and omitting the erroneous entries in the TLB after error detection.

Core Cannibalization Architecture [Romanescu and Sorin 2008], a self-repair mechanism for multicore architectures, cannibalizes faulty cores for pipeline stages by using resources from fault-free cores. Although it improves the lifetime of a core, its error detection granularity is coarser than our proposal.

Bower et al. [2004] proposed Self-Repairing Array Structures, which tolerate hard faults in array structures by using spare rows instead of faulty rows. However, in order to detect the faulty row, this method should mirror every write to a row to a dedicated check row.

Srinivasan et al. [2005] also proposed a lightweight approach to leverage existing redundancy in microprocessors. However, error detection is not addressed in the study.

Error correction codes (ECCs) can be used for storage structures in microarchitectures for self-repairing. However, ECC calculation presents time and energy overhead for both reads and writes. Also, generally, they are only capable of correcting a couple of bits instead of the entire data.

Carretero et al. [2009] proposed a self-test for register dataflow. They use a variety of signatures to verify the control logic of the issue queue and register files and the operation of ALUs. Similarly, Meixner and Sorin [2007] proposed a compiler technique to compute dataflow graphs and use them for error detection. However, both schemes require additional hardware for generating signatures and for verification. Also, they are not adequate to detect the faulty structure.

7. CONCLUSION AND FUTURE WORK

Fine-grained error detection schemes are essential to maintain the functionality of the processor in the presence of a hardware fault. In this study, we show that already existing CAMs and comparators in the IQ, TLB, and rename logic can be exploited to provide a fine-grained, low-cost error detection scheme for processors. We reduce the vulnerability of an out-of-order, superscalar processor without presenting any performance degradation in the common, error-free execution.

We reduce the vulnerability of the source register tags in the IQ by 60%, the vulnerability of the instruction TLB by 64%, the vulnerability of the data TLB by 45%, and the vulnerability of the register tags of rename logic by 20%.

REFERENCES

- R. Anglada and A. Rubio. 1988. An Approach to Crosstalk Effect Analysis and Avoidance Techniques in Digital CMOS VLSI Circuits. *International Journal of Electronics* 6, 5 (1988), 9–17.
- T. M. Austin. 2000. DIVA: A Dynamic Approach to Microprocessor Verification. *Journal of Instruction-Level Parallelism* 2 (2000), 1–6.
- R. Baumann. 2005. Soft Errors in Advanced Computer Systems. *IEEE Design and Test of Computers* 22, 3 (2005), 258–266.
- D. J. Baylis. 1998. *Error Correcting Codes: A Mathematical Introduction*. Chapman and Hall.
- N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (August 2011), 1–7.
- A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. 2005. Computing Architectural Vulnerability Factors for Address-Based Structures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 532–543.
- A. Biswas, P. Racunas, J. S. Emer, and S. S. Mukherjee. 2007. Computing Accurate AVFs Using ACE Analysis on Performance Models: A Rebuttal. *Computer Architecture Letters* 7, 1 (2007), 21–24.
- F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. 2004. Tolerating Hard Faults in Microprocessor Array Structures. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 51–60.
- F. A. Bower, D. J. Sorin, and S. Ozev. 2005. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*.
- J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. Gonzlez. 2009. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 105–115.
- K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. 2006. BulletProof: A Defect-Tolerant CMP Switch Architecture. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*. 3–14.
- P. DeMone. HP's Struggle for Simplicity Ends at Intel. Retrieved October 27, 1999, from <http://www.realworldtech.com/hp-intel-itanium/3/>.
- O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. 2006. Early Register Deallocation Mechanisms Using Checkpointed Register Files. *IEEE Transactions on Computers* 55, 9 (2006), 1153–1166.
- D. Ernst and T. Austin. 2002. Efficient Dynamic Scheduling Through Tag Elimination. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. 37–46.
- E. S. Fetzer, D. Dahle, C. Little, and K. Safford. 2006. The Parity Protected, Multithreaded Register Files on the 90-nm Itanium Processor. *IEEE Journal of Solid-State Circuits* 41, 1 (January 2006), 246–255.
- S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. A. Mahlke. 2008. StageNetSlice: A Reconfigurable Microarchitecture Building Block for Resilient CMP Systems. In *CASES* (2008). 1–10.
- J. L. Hennessy and D. A. Patterson. 2012. *Computer Architecture—A Quantitative Approach*. (5th ed.). Morgan Kaufmann.
- J. L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News* 34 (2006), 1–17.
- B. Jacob and T. Mudge. 1998. Virtual Memory in Contemporary Microprocessors. *IEEE Micro* 18, 4 (July 1998), 60–75.
- I. Kim and M. H. Lipasti. 2003. Half-Price Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. 28–38.

- M. Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. 2008b. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of ASPLOS*.
- X. Li, S. V. Adve, P. Bose, and J. A. Rivers. 2005. SoftArch: An Architectural-Level Tool for Modeling and Analyzing Soft Errors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*. 496–505.
- X. Li, S. V. Adve, P. Bose, and J. A. Rivers. 2008a. Online Estimation of Architectural Vulnerability Factor for Soft Errors. *SIGARCH Computer Architecture News* 36, 3 (June 2008), 341–352.
- A. Meixner and D. J. Sorin. 2007. Error Detection Using Dynamic Dataflow Verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 104–118.
- A. Moshovos. 2002. Power-Aware Register Renaming. Computer Engineering Group, Technical Report 01-08-2, University of Toronto.
- S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. 2002. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of ISCA*. 99–110.
- S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*.
- D. V. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. 2004. Energy Efficient Comparators for Superscalar Datapaths. *IEEE Transactions on Computers* 53, 7 (2004), 892–904.
- D. V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, and P. M. Kogge. 2003. Energy-Efficient Issue Queue Design. *IEEE Transactions on VLSI Systems* 11 (2003), 789–800.
- B. R. Rau and J. A. Fisher. 1993. Instruction-level Parallel Processing: History, Overview, and Perspective. *Journal on Supercomputers* 7, 1–2 (May 1993), 9–50.
- S. K. Reinhardt and S. S. Mukherjee. 2000. Transient Fault Detection via Simultaneous Multithreading. *SIGARCH Computer Architecture News* 28, 2 (2000), 25–36.
- B. F. Romanescu and D. J. Sorin. 2008. Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults. In *PACT*. 43–51.
- D. Sager, Desktop Platforms Group, and Intel Corp. 2001. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal* 1 (2001), 2001.
- J. J. Sharkey, D. V. Ponomarev, K. Ghose, and O. Ergin. 2006. Instruction Packing: Toward Fast and Energy-Efficient Instruction Scheduling. *ACM Transactions on Architecture and Code Optimization* 3, 2 (June 2006), 156–181.
- P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. 2012. Exploiting Microarchitectural Redundancy for Defect Tolerance. In *Proceedings of the 30th International Conference on Computer Design*. 35–42.
- D. Sima. 2000. The Design Space of Register Renaming Techniques. *IEEE Micro* 20, 5 (September 2000), 70–83.
- T. J. Slegel, N. J. Wang, and S. J. Patel. 1999. IBM's S/390 G5 Microprocessor Design. *IEEE Micro* 19 (1999), 12–23.
- J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. 2005. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 520–531.
- N. J. Wang, A. Mahesri, and S. J. Patel. 2007. Examining ACE Analysis Reliability Estimates Using Fault-Injection. *SIGARCH Computer Architecture News* 35, 2 (May 2007), 460–469.
- N. J. Wang, Student Member, and Sanjay J. Patel. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE TDSC* 3 (2006), 188–201.
- C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. 2004. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. 264–275.
- A. Wood, R. Jardine, and W. Bartlett. 2006. Data Integrity in HP NonStop Servers. In *Workshop on SELSE*.
- G. Yalcin and O. Ergin. 2007. Using Tag-Match Comparators for Detecting Soft Errors. *IEEE Computer Architecture Letters* 6, 2 (2007), 53–56.
- G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero. 2011. FIMSIM: A Fault Injection Infrastructure for Microarchitectural Simulators. In *Proceedings of the International Conference on Computer Design (ICCD'11)*.

Received September 2013; revised June 2014; accepted July 2014