

# In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability

Jie Hu, Shuai Wang, and Sotirios G. Ziavras  
Department of Electrical and Computer Engineering  
New Jersey Institute of Technology  
Newark, NJ 07102  
{jie.hu,sw63,ziavras}@njit.edu

## Abstract

*Protecting the register value and its data buses is crucial to reliable computing in high-performance microprocessors due to the increasing susceptibility of CMOS circuitry to soft errors induced by high-energy particle strikes. Since the register file is in the critical path of the processor pipeline, any reliable design that increases either the pressure on the register file or the register file access latency is not desirable. In this paper, we propose to exploit narrow-width register values, which present the majority of the generated values, for duplicating a copy of the value within the same data item, called in-register duplication (IRD), eliminating the requirement of additional copy registers. The datapath pipeline is augmented to efficiently incorporate parity encoding and parity checking such that error recovery is seamlessly supported in IRD and the parity checking is overlapped with the execution stage to avoid increasing the critical path. Our experimental evaluation using the SPEC CINT2000 benchmark suite shows that IRD provides superior read-with-duplicate (RWD) and error detection/recovery rates under heavy error injection as compared to previous reliability schemes.*

## 1. Introduction

Along with the dramatic performance improvement driven by advancing technologies, future microprocessors are becoming even more vulnerable to soft errors induced by energetic particle strikes such as alpha particles (emitted by decaying radioactive impurities in packaging and interconnect materials) and high-energy neutrons induced by cosmic rays [35][33]. This increasing vulnerability is primarily due to the continuously reducing logic depth, lowering supply voltage, decreasing nodal capacitance, and increasing clock frequency and on-chip integration density at new technologies [28]. Thus, designing new generation microprocessors against soft errors has arisen as a major requirement along with performance and power considerations.

Traditionally, triple-modular redundancy (TMR) [15] is used to achieve highly reliable fault-tolerant computing at

high hardware cost. Recently, proposals targeting soft error problems in microprocessors have suggested utilizing the inherent resource redundancy in simultaneous multi-threading (SMT) microprocessors and chip-multiprocessors (CMPs) to enhance the datapath reliability with concurrent error detection [26, 31, 25, 32, 7, 19]. Some other research has proposed that designers exploit the redundant resources in high-performance superscalar out-of-order cores to enable a reliable processor through instruction-level redundant execution [18, 24, 22, 30, 9, 8].

Since 1) register file read is within the datapath loose loops [3], 2) error-flipped intermediate computation results in the register file are very likely to propagate to later computations or to memory hierarchies, and 3) the large register file is a major die-area consumer increasing its exposure to high-energy particle strikes [23], designing high-performance error-resilient register files is of critical importance. Most dual-instruction execution (DIE) processors include register file within the sphere of replication (SoR) [25]. This is mainly due to the unbearable access latency and power overhead of ECC-protected register files [17, 12]. Notice that including the register file within the SoR effectively halves the size of the register file. Assuming a parity-protected register file, recent work [17] proposed to use idle/free registers to accommodate duplicate values and the copy registers can be preempted for regular register renaming to avoid any performance loss. Further, it proposed to use predicted dead registers to improve the duplication rate at a small performance overhead, based on the assumption that the register value is first written back to the memory before the register is reused. However, the error coverage of these schemes is significantly limited compared to the full-duplication scheme.

The presence of narrow-width data (whose value can be represented by fewer bits than the full data width of the processor) in general-purpose applications is well understood and has been utilized for power and performance optimizations [4][13][14][6]. In this paper, we propose to exploit the produced narrow-width register values for designing high-performance error-resilient register files, and protecting the result writeback bus and the bypass network. In the proposed new processor microarchitecture, each integer functional unit is augmented with a simple fast leading-0/1 detector for narrow-width check. Detected narrow-width re-

sults that can be represented by no more than 32 bits automatically duplicate themselves by muxing the lower 32 bits into the higher 32 bits before being latched by the pipeline registers. We call this scheme *In-Register Duplication (IRD)*. In-register duplication stores two copies of the narrow-width value in the same register and transmits two copies of the value using the bandwidth for a single data value over the writeback bus and forwarding bus. Thus, IRD eliminates the need for additional copy registers that maintain redundant copies of the register value for error detection and recovery. It also protects the data transfer paths from/to the register file and the functional units for narrow-width values. To our best knowledge, this work presents the first effort to exploit narrow-width values for reliable register file design against soft errors.

Experimental evaluation using the SPEC CINT2000 benchmark suite shows that without sacrificing any performance IRD achieves a write-with-duplicate (WWD) rate of 94% at the output of functional units and a read-with-duplicate (RWD) rate of 95% at the inputs of functional units. We evaluated two schemes for error detection in IRD, based on duplicate value comparison and parity checking. Under error injection with accelerated error rates at  $10^{-5}/10^{-4}$  per selected bit per cycle, both schemes virtually detect all errors in narrow-width values being read in. In addition, parity checking also identifies all errors in regular values. To avoid signaling unnecessary errors in the duplicate copy, IRD only checks the parity bit of the lower 32-bit half for error detection and utilizes the duplicate in the upper half for error recovery. Our experimental results show that IRD detects 99.7% of the erroneous reads for the narrow-width values and successfully recovers 99.7% and 99.2% at error rate  $10^{-5}$  and  $10^{-4}$ , respectively, of the detected errors using the uncorrupted duplicate, which makes our in-register duplication a very cost-effective design for highly reliable register files.

The rest of the paper is organized as follows. We discuss related work in Section 2, and review some basics of register renaming and the correlation between register file size and performance in Section 3. A detailed analysis of narrow-width values is given in Section 4. We elaborate on our in-register duplication design in Section 5 and introduce the error injection model in Section 6. In Section 7, we study the error behavior of the register file system under error injection and evaluate IRD for reliable design. Section 8 concludes this work.

## 2. Related Work

Fault-tolerant designs based on modular redundancy have been widely used to build highly reliable systems. For example, cycle-by-cycle lockstepping of dual-processors and comparison of their outputs are employed for error detection in Compaq Himalaya [1] and IBM z900 [29] with G5 processors. Other designs use asymmetric redundancy to include a watch-dog processor [21] or a low-performance checker processor in DIVA [2] to verify the correctness of the execution on the main processor.

Targeting the increasing processor vulnerability to soft errors at new technologies, temporal redundancy based

reliable schemes exploiting simultaneous multithreading (SMT) architectures have been extensively studied for both single processors and chip-multiprocessors, such as AR-SMT [26], SRT [25][19], SRTR [32], and Slipstream [31]. Lately, many research efforts have been spent on exploiting the redundant resources in superscalar processors for instruction-level redundant execution against transient faults. In [18], each instruction is executed twice and the results from duplicate execution are compared to verify the absence of transient errors in functional units. However, each instruction only occupies a single re-order buffer (ROB) entry. On the other hand, the dual-instruction execution scheme (DIE) in [24] physically duplicates each decoded instruction to provide a *Sphere of Replication* including the instruction issue queue/ROB, functional units, physical register files, and the interconnect among them. Due to the substantially increased pressure on the hardware resources, dual-instruction execution in general suffers from significant performance loss. Follow-up work such as DIE-IRB [22], SHREC [30], and PER-IRTR [8], try to alleviate the resource contention in DIE processors to recover the performance loss.

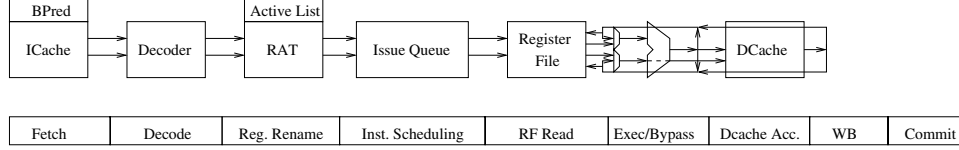
In [11], the protection schemes were particularly tuned to protect frequently accessed cache lines which are more error-prone, in order to reduce the area overhead. Our focus in this work is to design reliable register files. Previous work [17] has exploited utilizing free registers or predicted dead registers to maintain a replica of the value in the register file to increase its error resilience. Recent work [16] studied the trade-offs between performance and reliability of the register file when over-clocking is applied to increase the operation frequency. Different from their work, our in-register duplication scheme is based on the detection and capture of narrow-width register values such that redundant copies are generated within a single 64-bit data item to improve the reliability of the register file system, eliminating the need for copy registers and related hardware enhancements.

## 3. Basics of Register Renaming in Superscalar Microprocessors

### 3.1. Register Files and Register Renaming

Superscalar microprocessors dynamically exploit instruction-level parallelism (ILP) to issue multiple instructions per cycle for improved performance. Register renaming is one of the fundamental techniques employed in superscalar microprocessors to increase the ILP by eliminating the two false data dependences, write-after-read (WAR) and write-after-write (WAW). We implemented MIPS R10000 [34] style register renaming, where the architectural and physical register files are combined. Figure 1 gives the superscalar microprocessor model simulated in this paper.

In the register renaming stage, the logical register ids of the source operands in a decoded instruction are used to access the register alias table (RAT), a.k.a. register mapping table. The table entry indexed by the logical register id contains the physical register id that the source register



**Figure 1. Datapath and the pipeline stages of the simulated superscalar microprocessor.**

was renamed to. For the destination register, a free physical register is allocated from the register free list and the RAT is updated as follows: the old physical register id is read out from the RAT and stored in the active list entry allocated to the instruction, and then the new physical register id is written to the same RAT entry indexed by the logical destination register id. The destination register is said to have been remapped to the new physical register and the old physical register is said to have been unmapped. In case the register free list is empty, the renaming stage is stalled till some physical register is freed. Notice that a physical register cannot be freed until an instruction that previously unmapped this physical register is committed. Furthermore, a physical register is susceptible to soft errors only after a value is written into the register and before it is freed.

### 3.2 Register File Performance Sensitivity

The register file size limits the effective size of the instruction window thus presenting a major constraint on the ILP exploitation in superscalar microprocessors. Figure 2 (a) shows the performance comparison of an 8-wide superscalar microprocessor (the detailed configuration is given in Table 1) when the integer register file size varies from 40 to 512 entries. Significant performance improvement is achieved when the size increases from 40 to 48, 64, and 80. However, further increasing the size beyond 80 registers, the performance improvement is diminished. Notice that for this study we have assumed a uniform access latency for the integer register file at different sizes. A reliable design based on full register duplication will either require large size register files or significantly limit the success rate of duplication. In the following study, we focus on a medium size, 128-entry integer register file, which simulates the register pressure in wide-issue datapaths while avoiding the unnecessary performance overhead of the full-duplication scheme.

## 4. Narrow-Width Register Values

In high-performance 64-bit microprocessors, many generated register values during the execution of general-purpose applications do not require the full width of 64 bits. Values that can be represented by less than 64 bits are called narrow-width values in this paper. The presence of narrow-width values has been well studied and exploited for performance and power optimizations [4][14][13][6]. Different from the previous work, we exploit narrow-width register values for improving the register file reliability against soft errors.

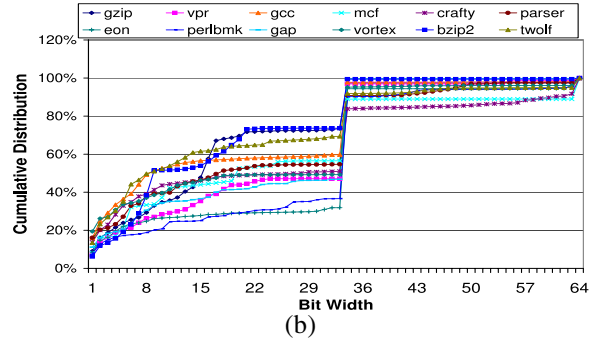
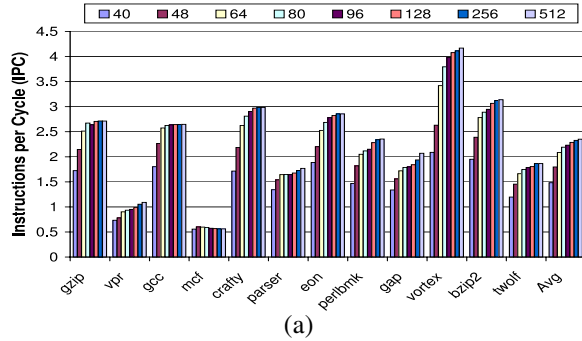
We present a similar study on the data-width distribution of integer register values for SPEC CINT2000 benchmarks

(Alpha binaries) as in [4][13]. In the following discussion, we use the terms register value and data value interchangeably with integer register value. The detailed cumulative data-width distribution is given in Figure 2 (b). The data-width ( $W$ ) of a given register value is determined as follows:  $W = 64 - (LS - 1)$ , where  $LS$  is the number of leading 0's or leading 1's. A data value  $V$  at width  $W$  can be represented by  $W$  bits instead of 64 bits in the following way:  $V = b_{W-1}b_{W-2}...b_0$ , where  $b_{W-1}$  is the sign bit,  $b_{W-1} \oplus b_{W-2} = 1$ , and  $W \leq 64$ . Figure 2 (b) shows that on the average 54% of the generated register values have a data-width no more than 32 bits, and the difference between 32 bits and 33 bits is negligible. However, there is a significant 40% jump from 33 bits to 34 bits. This is because the memory address in the Alpha ISA uses 33 bits (plus 1 sign bit = 34 bits) and memory operations account for a large portion of the executed instructions. Please notice that 1) the operations generating these memory addresses are different from the address calculation in a load/store instruction, and 2) compiler options or large-size programs may change the data width of memory addresses. Overall, around 94% of the integer values can be represented by no more than 34 bits, an average for SPEC CINT2000 benchmarks.

We also observed that small values present a significant percentage. For example, 46% of the generated values can be represented by no more than 16 bits. This percentage increases to 51% for data-width  $\leq 21$  bits. Exploiting these small values enables duplicating multiple replicas of the original data value within the same register, thus achieving  $N$ -modular redundancy ( $N \geq 3$ ) for the value. However, such a scheme complicates the design in order to capture multiple levels of narrow-width values and provide different error detection and recovery mechanisms. In this work, we exploit the large percentage of narrow-width values, especially with data-width  $\leq 34$  bits, for designing high-performance error-resilient register files using in-register duplication.

## 5. Exploiting Narrow-Width Register Values

In this section, we present our reliable register file design that exploits the generated narrow-width register values. Information redundancy is the basic idea for protecting memory structures against soft errors. Instead of duplicating each register value into two registers, we exploit the majority of narrow-width values ( $\leq 32$  bits and 34-bit memory addresses) to perform in-register duplication.



**Figure 2. (a) Performance sensitivity to the register file size. (b) Cumulative distribution of the register value width.**

### 5.1. Narrow-Width Value Detection

Based on the data-width analysis presented in the previous section, our design is particularly tuned to capture three types of narrow-width values: 32-bit positive values ( $0^{32}0x^{31}$ ), 32-bit negative values ( $1^{32}1x^{31}$ ), and 34-bit memory addresses ( $0^{30}01x^{32}$ ), where  $x$  can be either a “1” or a “0”. From now on, we only refer to these three types as narrow-width data.

To capture narrow-width values, we add a narrow-width detector between the output of a functional unit and the input of the pipeline latch. It detects whether the newly generated result from the functional unit is a 32-bit positive value, a 32-bit negative value, or a 34-bit memory address (positive value). After detection, two flag bits ( $n_1n_0$ ) associated with each register value are set to indicate the narrowness of the current value. The meaning of these two  $n_1n_0$  bits is given in Figure 3 (b). The block diagram in Figure 3 (a) shows the datapath augmented with the narrow-width value detector. The detector consists of three sub-detectors: a 32-bit positive value detector, a 32-bit negative value detector, and a 34-bit memory address detector. Notice that a narrow-width value will have flag bit  $n_0$  set to 1. We utilize flag bit  $n_0$  to control in-register duplication for a narrow-width value (by copying the lower 32-bit half into the higher 32-bit half) or to bypass duplication for a regular value.

An alternative implementation to reduce the cycle time impact is to incorporate the narrow-width detection within the functional units by monitoring the narrowness of the inputs and the internal carry signals. However, such a design requires modification of the standard ALU implementation as well as different detection circuitry for different operations, which is what we try to avoid here. In our design, the functional units are not modified. Due to the simple detector circuit [6], the impact on the clock cycle time should be negligible.

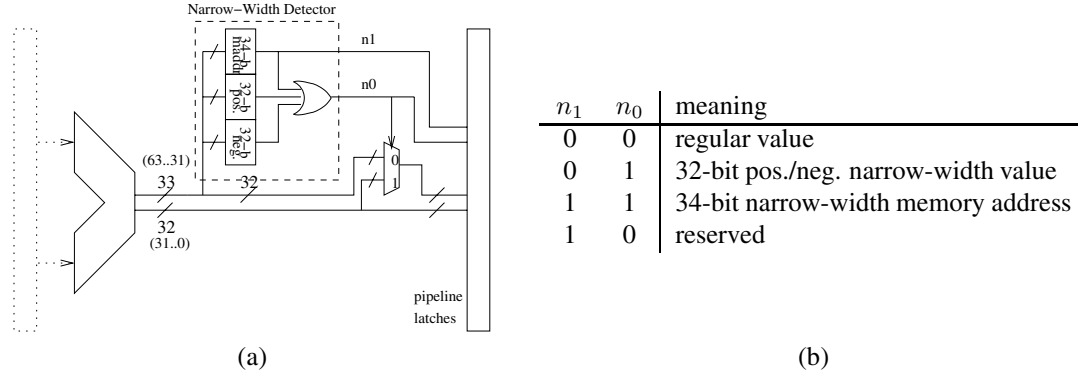
### 5.2. Exploiting In-Register Duplication for Error Detection

Once a narrow-width register value is detected, in-register duplication is automatically performed by copying the lower 32-bit half into its higher 32-bit half such that two copies of the value will be latched into the pipeline regis-

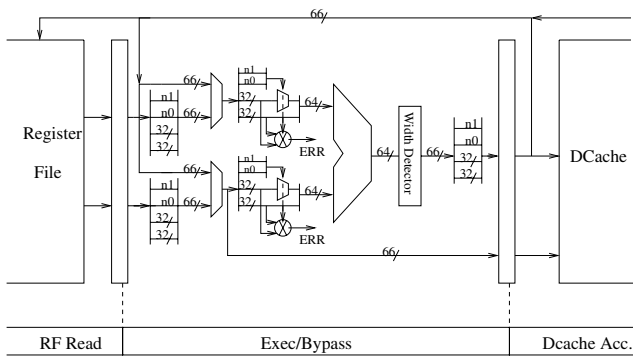
ter. The incentive of our reliable register file design is not only to protect the register file against soft errors, but also to guarantee reliable data transmission over the writeback and bypass networks. In-register duplication enforces at any time two copies of the narrow-width value to be stored in the register file, latched by the pipeline register, or transferred between the register file and the functional units.

It is important to notice the significant difference between our in-register duplication and conventional redundancy-based reliable designs. In-register duplication incurs much less hardware complexity compared to schemes utilizing idle or predicted dead registers for duplicating data value, where the register renaming logic is redesigned for copy register allocation, the instruction queue is augmented to hold copy register ids, and the number of register file writeports is doubled or a set of copy ports is required [17]. In-register duplication needs none of the above hardware modifications. More importantly, our scheme also protects the result writeback bus and the bypass network by transferring two copies of the value without increasing the bandwidth requirement. In the schemes presented in [17], a data value hit by errors when transferring over the writeback bus will result in two corrupted copies being stored in the register file due to the use of copy ports. Since around 50%-70% of the input operands are retrieved from the bypass network, hardening both the bypass network and the writeback bus against soft errors is of critical importance, which is naturally supported by in-register duplication.

Since the probability of the two copies of the narrow value being corrupted at exactly the same bit positions is extremely low, the two copies can be compared against each other to verify the absence of soft errors very effectively. A follow-up question is when to perform this comparison. Notice that soft-error corrupted data only matters when used later in computation or written out to the memory hierarchy. The probability of resulting in crashed execution or erroneous outputs can be estimated by the architectural vulnerable factors (AVFs) [20] of the microarchitectural blocks that the corrupted value is going through. We choose to perform error detection (comparing the upper 32 bits against the lower 32 bits of the input operand) at the execution stage when the operands are fed to the functional units. Figure 4 shows the slightly changed datapath back-end stages to support in-register duplication and soft error detection. After narrow-width detection, each value is augmented with



**Figure 3. (a) Augmented functional unit datapath with narrow-width detectors and value duplication. (b) The meaning of the value of narrowness flag bits  $n_1n_0$ .**



**Figure 4. Microarchitectural support for in-register duplication and error detection.**

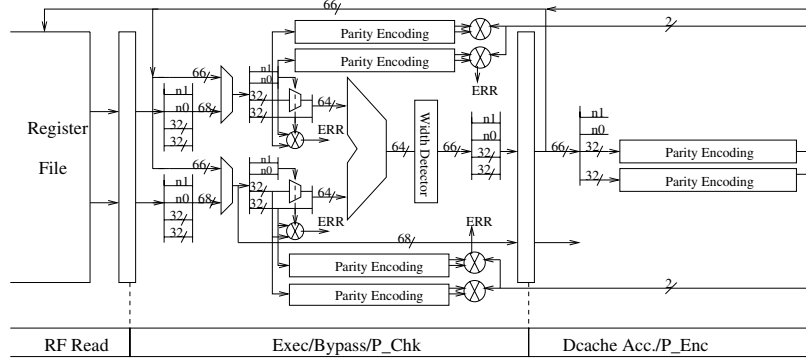
the 2 additional flag bits  $n_1n_0$  indicating the narrowness of the value. The data bus and the register file are extended to transmit and store 66-bit values instead of the original 64-bit ones. The source operands retrieved either from the register file or from the bypass network are in 66-bit format. If the  $n_1n_0$  bits indicate a narrow-width value, sign extension is performed to restore the original 64-bit value and error detection is carried out simultaneously by comparing the upper and lower 32-bit halves. If the two halves match each other, no error is detected. Otherwise, an “ERR” signal is raised to denote errors. The data cache here is treated in the same way as regular functional units; its data bus is also protected by in-register duplication for narrow-width values.

### 5.3. Integrating In-Register Duplication and Parity Coding

In-register duplication is expected to be very effective in soft-error detection by comparing the two copies of the narrow-width value stored within the same 64-bit data, however, is not capable of recovering from an error. Since in-register duplication already maintains two redundant copies of the value, providing ECC coding (e.g., Hamming coding) for each 32-bit half is either over-designed or not feasible

considering the ECC coding/checking latency and power consumption [11]. We choose to use simple and fast parity coding to supplement each 32-bit half with an additional parity bit. Notice that the flag bits can be also covered by the same parity bits for the data value or a separate parity bit, which we do not explore further in this work. We assume parity encoding/checking takes one clock cycle.

There are at least two strategies to perform error detection and recovery in our in-register duplication scheme with parity coding. The first strategy is to compare the two 32-bit halves of the input operand to detect possible errors just as in the base IRD scheme. Simultaneously, parity checking is carried out for both halves. If the two 32-bit halves match each other, no error is detected and the parity checking result is ignored. Otherwise, if the lower half passes the parity check, the error(s) is(are) considered only to have corrupted the upper half which we do not care about and no error recovery is needed. If the lower half fails parity check and the upper half passes the check, then the upper half is used to recover the detected errors. In case both halves fail the check, an exception is raised to inform the upper levels of the system. The second strategy eliminates the comparison and directly performs parity checking for both halves. If the lower half passes the check, we do not care about the duplicate in the upper half since the value is restored from the lower half. Otherwise, we try to use the upper half to recover the corrupted lower half in a similar way as in the first strategy. Effectively, these two strategies are the same in terms of error recovery capability. However, the first strategy using comparison shall detect more errors than the second one. This might not be always good; e.g., errors happening in the upper half of the operand have no impact on the correctness of the execution, however, will be detected and raised for action in the first strategy. A similar problem exists in both of these two strategies as in any protection scheme using parity coding: a successful parity check does not guarantee the integrity of the data value and a false recovery may be performed by using the corrupted yet not detected redundant copy. Whether to recover the erroneous data value in the register file or not after error detection should be justified by the error rates in real applications due to the potentially significant hardware complexity needed for such a recovery.



**Figure 5. The augmented datapath integrating in-register duplication and parity coding to support both error detection and error recovery.**

To integrate parity coding with in-register duplication, we need to add a separate pipeline stage to perform parity encoding after the execution stage. Figure 5 shows the modified datapath supporting both error detection and recovery. The parity bit for each 32-bit half is generated in the parity encoding ( $P\_Enc$ ) stage. Parity checking ( $P\_Chk$ ) for input operands is overlapped with the first cycle of the execution stage such that the branch resolution loose loop [3] is not increased. This also guarantees that detected errors in input operands are signaled before the erroneous result is written back to the register file since many ALU operations take just one cycle to complete. Input operands read from the register file come with the parity bits for the two 32-bit halves. Parity checking basically regenerates the parity bit for each 32-bit half and compares it against the one with the data value. However, operands retrieved from the first stage of the bypass network do not have parity bits generated yet. In such a case, both parity encoding ( $P\_Enc$  stage) and parity regenerating (in  $P\_Chk$  stage) are performed simultaneously and the parity bits from the  $P\_Enc$  stage are bypassed to the  $P\_Chk$  stage for parity bit checking since the comparison happens in the later stage of  $P\_Chk$ . Here, we use the second strategy presented early. If the two parity bits for the lower 32-bit half match, no error is detected. Otherwise, the lower half has been corrupted by errors and a stall cycle is inserted. Now if the parity bits for the upper 32-bit half match, then the upper half is copied back to the lower half to recover the corrupted data. The instruction is then replayed with the recovered inputs. However, if the upper half is also corrupted and error-detected, an exception is raised for the higher level systems to solve the problem.

Since parity encoding takes one additional cycle, one design issue raised here is when to write back the result value and the parity bits. If the value and parity bits are written back to the register file in two consecutive cycles, additional write ports must be provided for writing parity bits, which violates the spirit of in-register duplication since it deteriorates the access latency and power consumption of the register file. Otherwise, the result writeback can be delayed by one cycle till the parity bits are available. However, delaying writeback increases the stages of the bypass network, thus increasing the complexity and number of wires of the bypass network. To address these issues, we propose to use a special bit-addressable parity register to hold two parity

bits for each entry in the register file. Notice that the data cache interface uses the first strategy (comparing two 32-bit halves for error detection) to avoid storing error-corrupted data into the cache.

#### 5.4. Protecting the Regular Values

As a side benefit of in-register duplication, regular values (cannot be represented by 32 bits plus 2 flag bits) are also protected by the 2 parity bits. For a detected regular value, the 2 flag bits  $n1n0$  are reset to 00. During the  $P\_Enc$  stage, two parity bits are generated for the two 32-bit halves in the same way as for narrow-width values. Once a regular value reaches the input of a functional unit, the flags bits  $n1n0$  (=00) enforce parity checking for both 32-bit halves to verify the absence of soft errors. If any half fails the parity check, an error signal is raised. However, the hardware itself is not capable of recovering the error-corrupted regular value. Notice that a similar scheme as in [17] can be applied to exploit free registers for duplicating a replica of the regular value, which provides recovery capability. In such a scheme, the mapping information between the original register and the copy register shall be maintained in order to locate the copy register during recovery. Due to significant modifications required in the register renaming logic, the register file, and the issue queue, we do not explore further this idea in the following discussion.

#### 6. Error Model and Soft Error Injection

To evaluate the error resilience of our schemes, we conducted soft error injection during execution-driven simulation. The software error injection flips one bit or multiple bits in a selected register value. Since the multiple-bit error rate is several magnitudes lower than the single-bit error rate [12], we assume a single-bit error model in this study. Our error injection scheme simulates single-event upsets (SEUs) in the register file, the bypass network, and the result writeback bus. At each clock cycle, a uniformly distributed random function is called to locate a register and a specific bit in that register. Then, an error is injected with a given probability (e.g.,  $10^{-7}$  [12]), i.e., single-bit soft error rate. During error injection, if the selected register is receiving a new

Processor Core	
Int/FP issue queue	128 entries
Load/Store Queue	256 entries
Active list (ACL)	512 entries
Int/FP Register File	128/512 registers
Datapath width	8 instructions per cycle
Function Units	8 IALU, 2 IMULT/IDIV, 4 FALU
	2 FMULT/FDIV/FSQRT, 4 MemPorts
Branch Predictor	
Branch Predictor	tournament predictor with a 4K meta-table, a 4K bimodal predictor table, and a 2-level gshare predictor with 12-bit history
	2048-entry, 4-way BTB, and 32-entry RAS
Memory Hierarchy	
L1 I/D-Cache	64KB, 2 ways, 64B blocks, 2 cycle latency
L2 U-Cache	4MB, 8 ways, 128B blocks, 12 cycle latency
Memory	225 cycles first chunk, 12 cycles rest
TLB	fully-assoc., 128 entries, 30-cycle miss penalty

**Table 1. Parameters for the simulated micro-processor.**

value which is also being bypassed to the next execution stage, we flip the bit wire in the bypass network instead of the bit cell in the register file. Thus error detection and recovery can be immediately exercised at the *P\_Chk* stage. If the selected value is transmitting over the result bus, error injection also flips the corresponding bit wire in the result bus and the error is propagated to the register entry in the register file once the value is written. Otherwise, a bit cell in the register file is flipped to reflect the error-corrupted bit value. Notice that each register file write clears out the errors previously injected into that particular register entry.

To avoid crashing the simulation, each injected error is logged using a bitmap for each register entry instead of flipping the real bit value and the error history is also recorded. During the simulation, the soft error bitmap and error history information of a given register value are used to perform error detection and recovery at the execution/*P\_Chk* stage.

## 7. Evaluation

### 7.1. Experimental Setup

We derive our simulators from SimpleScalar V3.0 [5] to model a contemporary high-performance microprocessor similar to Alpha 21464 [23]. Table 1 shows the detailed configuration of the simulated microprocessor. For experimental evaluation, we use SPEC CINT2000 suite compiled for the Alpha instruction set architecture using “-arch ev6 -non\_shared” option with “peak” tuning. We use the reference input sets for this study. Each benchmark is first fast-forwarded to its early single simulation point (*gap* uses the standard single simulation point instead of the very large early single simulation point) specified by SimPoint [27]. We use the last 100 million instructions during the fast-forwarding phase to warm-up the caches if the number of skipped instructions is more than 100 million. Then, we simulate the next 100 million instructions in detail.

### 7.2. Duplication Rates and Performance Impact

The ability to recover register values from detected errors depends on the availability and correctness of the duplicate copies. We use the write-with-duplicate (WWD) rate as first-order estimation to measure the capability of a reliable scheme to duplicate the register values,

$$WWD\_Rate = \frac{\#Write\ w/\ Duplicate}{\#Total\ Writes},$$

and use the read-with-duplicate (RWD) rate as first-order estimation to approximate the reliable read of register values against soft errors,

$$RWD\_Rate = \frac{\#Reads\ w/\ Duplicate}{\#Total\ Reads}.$$

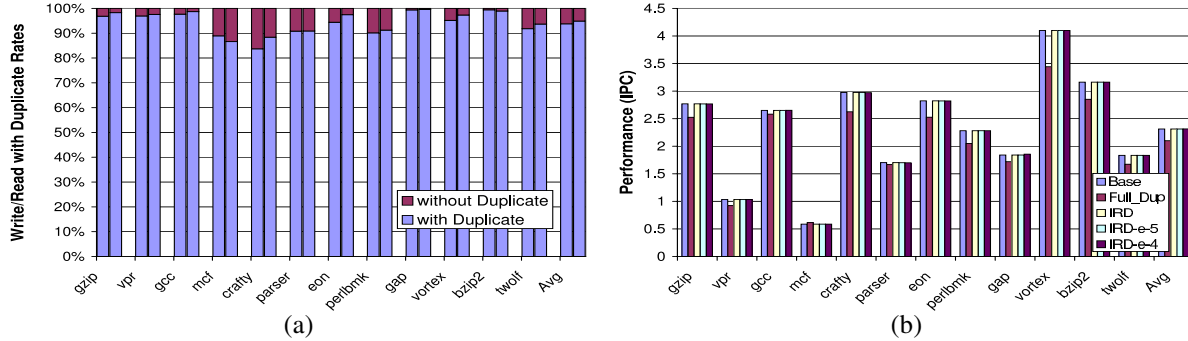
Figure 6 (a) shows that by exploiting narrow-width values alone, in-register duplication achieves a WWD rate of 94% and a RWD rate of 95%, an average across SPEC CINT2000 benchmarks, without any performance loss. This RWD rate is significantly improved over the results (78% for CE, and 84% for AG at a 0.2% performance loss) reported in [17]. In the mean time, our in-register duplication scheme avoids the hardware complexity of the latter schemes.

Considering a full-duplication scheme (Full\_Duplication) that allocates a copy register for each result register at the register renaming stage, the hardware implementation is much more complexity-effective than the CE/AG schemes [17] since the copy register is implied. The Full\_Duplication scheme achieves a rate of 100% for both WWD and RWD, however, suffers from a significant performance loss, 7.7% on the average, as shown in Figure 6 (b). Our in-register duplication IRD scheme duplicates the narrow-width value within the same register and requires no additional copy register, thus incurring no performance overhead. Figures 6 (a) and (b) together show that our schemes are very effective in providing a high error coverage for applications where the performance and cost are highly constrained.

### 7.3. Error Behavior under Soft Error Injection

To further evaluate the effectiveness of the reliable schemes proposed in this work, we perform experiments for error detection and recovery under error injection. For error injection, we use a uniformly distributed random function to locate a particular bit in a selected register at each clock cycle, and then use a second random function to control the error injection with respect to a given soft error rate. We evaluated the IRD schemes with two soft error rates of  $10^{-5}$  (shown as “e-5”) and  $10^{-4}$  (“e-4”), given the limited simulation of 100 million instructions. Notice that these again are accelerated rates for the very rare single-event upsets (SEUs).

Understanding the error behavior is of critical importance towards designing a complexity- and cost-effective



**Figure 6. (a) Write-with-duplicate rate (left bar) and read-with-duplicate rate (right bar) of the in-register duplication scheme. (b) Performance comparison of various schemes.**

reliable system. Table 2 shows the characterization of soft errors being injected into the register file system. We classify the targets of error injection into two groups, active and inactive register entries. Active registers include registers in their lifetime of between being written and being freed. Errors happened in the bypass network or the result bus are also counted in this group. Errors in active registers once being read in by later dependent instructions have the potential to crash the execution or corrupt the memory hierarchy. On the other hand, inactive registers are free registers and registers allocated but not written yet. Errors injected into inactive registers have no impact on the correctness of the program execution since inactive registers do not contain a valid value. From Table 2, we find that on the average 56% of the errors happen in the inactive registers and another 44% in the active registers, which are of concern. In some cases, if an error happens in exactly the same bit position as some previous instance, the bit error is cleared automatically. We call this self-recovery. The results from Table 2 show that self-recovery is rare.

Erroneous input operands can be either read from the register file or retrieved from the bypass network. This experiment tries to identify the contributions of these two error sources. Notice that erroneous reads are instances of retrieved input operands with errors, which are different from the cumulative bit errors in the input operands. For example, an input value with multiple bits flipped due to soft errors (multiple-bit errors) is only counted as one instance of erroneous read. Figure 7 (a) shows that the majority of the erroneous reads, 92% (91%) at soft error rate e-5 (e-4), on the average, are due to the corrupted value read from the register file. The remaining 8% (9%) are due to wire flips when the value is being forwarded from the bypass network. Thus, the register file is still the major source of erroneous reads. Figure 7 (b) breaks down erroneous reads into those with single-bit errors and multiple-bit errors. Our results also show that most readin errors are single-bit errors, 99.7% (99.2%) at this very high error rate e-5 (e-4). This is because the live time (between writeback and the last read) of a register value is quite short [10], during which the same register entry is rarely hit by multiple errors.

#### 7.4. Error Detection and Recovery from Detected Soft Errors

The in-register duplication (IRD) scheme alone provides error detection for narrow-width values by comparing the two 32-bit halves. If the two halves match each other, the data value is assumed to be correct. Otherwise, the data is assumed to be corrupted by soft errors. However, this detection scheme is not applicable to regular values. Once integrated with parity coding, IRD checks the parity bits for both 32-bit halves at the first stage of execution. If any half fails this check, erroneous data is detected. This scheme covers both narrow-width values and regular values. However, this (even) parity coding scheme is not capable of detecting an even number of bit errors in a 32-bit half. Figure 8 compares the error detection capability of these two options. Both figures show results at two error rates, e-5 and e-4. In Figure 8 (a), IRD\_Detected represents the detection rate for narrow-width values using duplicate value comparison, Fail\_Detection is for the case where the error patterns are exactly the same for the upper and lower halves and this comparison-based detection fails, and Regular\_Value represents erroneous reads of regular values that cannot be detected by this scheme. The results show that using duplicate value comparison IRD detects all readin errors in narrow-width values. Figure 8 (b) presents results for IRD using parity checking. P\_H\_Detected and P\_F\_Detected correspond to detected readin errors in narrow-width values and regular values, and P\_H\_Fail and P\_F\_Fail represent undetected readin errors, respectively. IRD using parity checking detects all readin errors in regular values and only fails less than 0.3% of the time for narrow-width values.

As discussed in Section 5.3, duplicate value comparison and parity checking for both 32-bit halves detect more readin errors than necessary. Notice that our in-register duplication scheme restores the full 64-bit value of a narrow-width input by only using its lower 32-bit half. This is to say, that for narrow-width values, the IRD scheme is further tuned to use the parity checking result of the lower 32-bit half to detect soft errors and the parity checking of the upper half to determine whether it can be used to recover the value once the lower half is detected as error-corrupted. As shown in Figure 9 (a), using this revised IRD detection scheme,

	Error Rate: e-5					Error Rate: e-4				
	Total	Active	Inactive	Self-recovery		Total	Active	Inactive	Self-recovery	
gzip	339	146	43.1%	193	0	3603	1584	44.0%	2019	0
vpr	947	529	55.9%	418	0	9555	5134	53.7%	4421	10
gcc	399	123	30.8%	276	0	3760	1193	31.7%	2567	0
mcf	1716	856	49.9%	860	0	16931	8370	49.4%	8561	10
crafty	320	119	37.2%	201	0	3432	1293	37.7%	2139	0
parser	549	262	47.7%	287	1	5904	2855	48.4%	3049	6
eon	372	162	43.5%	210	0	3564	1490	41.8%	2074	1
perlbmk	457	179	39.2%	278	0	4416	1683	38.1%	2733	0
gap	535	288	53.8%	247	0	5497	2895	52.7%	2602	0
vortex	246	107	43.5%	139	0	2396	1210	50.5%	1186	0
bzip2	333	119	35.7%	214	0	3171	1209	38.1%	1962	8
twolf	516	213	41.3%	303	0	5390	2182	40.5%	3208	0
Avg	560.8	258.6	43.5%	302.2	0.08	5634.9	2591.5	43.9%	3043.4	2.9

Table 2. A characterization of soft errors injected into the register file system.

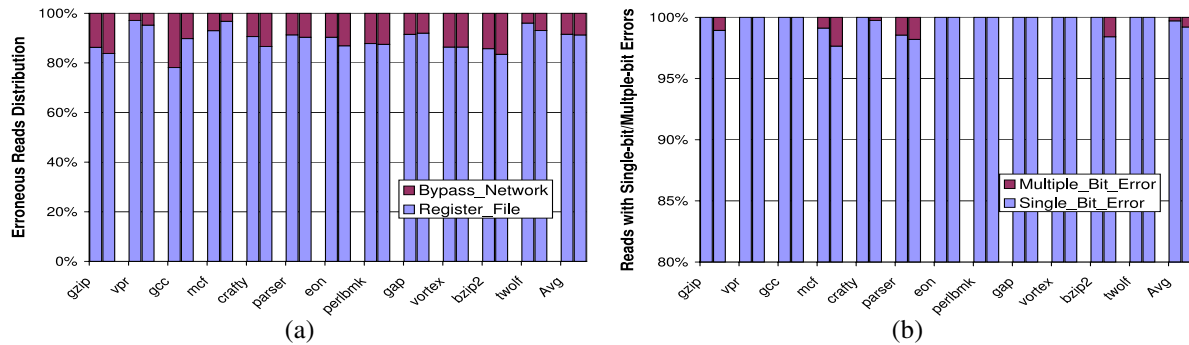


Figure 7. A Characterization of erroneous reads for input operands: (a) distribution of the error sources, (b) breakdown of erroneous reads with single-bit and multiple-bit errors. (Left bar for e-5 and right bar for e-4)

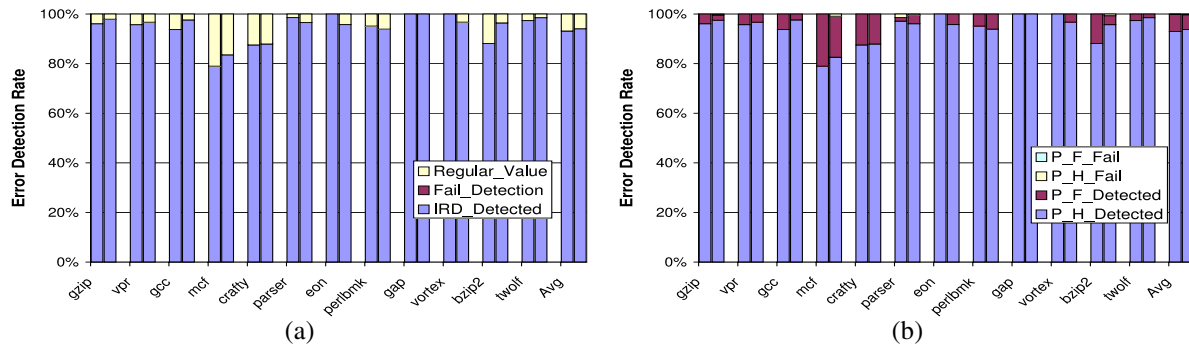


Figure 8. Soft error detection in the IRD scheme by (a) duplicate value comparison, and (b) parity checking. (Left bar for e-5 and right bar for e-4)

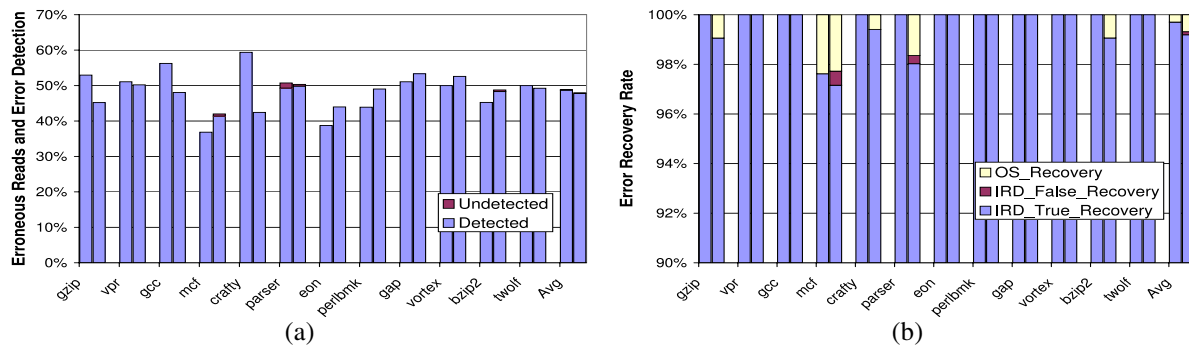


Figure 9. (a) Normalized erroneous reads and error detection for narrow-width values in the IRD scheme, and (b) error recovery rate of detected errors in IRD scheme, under error injection rates of  $10^{-5}$  (left bar) and  $10^{-4}$  (right bar) per selected bit per cycle.

the number of erroneous reads of narrow-width values is reduced to 49% of those presented in Figure 8. Of these readin erroneous narrow-width values, IRD detects 99.7% of the errors, which is very encouraging. Once errors are detected, IRD makes the following decision: if the duplicate in the upper 32-bit half passes the parity check, IRD uses the duplicate for error recovery; otherwise, IRD generates an ERROR exception and lets the operating system handle error recovery. We introduce an additional 1000 cycles for this ERROR exception handler. Notice that each detected erroneous regular value will trigger this ERROR exception. However, during IRD recovery, if the duplicate was also corrupted but yet succeeded in parity checking (even number of bit errors), IRD is forced to perform a false recovery using the corrupted duplicate. Figure 9 (b) shows, that, of the detected errors in narrow-width input operands, IRD recovers 99.7% (99.2%) of the errors with non-corrupted duplicates, `IRD_True_Recovery`. The false recovery rate, `IRD_False_Recovery`, is 0% (0.1%) at error rates e-5 (e-4). The operating system takes care of the remaining 0.3% (0.7%) of the detected errors. A performance comparison was shown early in Figure 6 (b). The performance overhead due to error recovery is negligible at these two error rates.

Overall, these results confirm that our in-register duplication scheme exploiting narrow-width values is very effective in detecting and recovering soft errors occurring in the register file, the bypass network, or the result writeback bus, while only incurring some minor microarchitectural modifications.

## 8. Conclusions and Future Work

We propose in this work to exploit narrow-width register values for designing high-performance reliable register files. Instead of allocating an additional copy register for storing the duplicate, our in-register duplication (IRD) scheme duplicates a replica of the narrow-width value in its upper 32-bit half, thus eliminating the hardware complexity required for acquiring and maintaining copy registers in previous schemes. Evaluation via software error injection, our IRD scheme has demonstrated superior error detection and recovery rates at minimum hardware cost, making it a suitable design in high-performance, highly reliable microprocessors. For future work, we plan to extend the current IRD framework to also support hardware recovery for error-corrupted regular values. We are also working on the power evaluation of the IRD scheme. Another interesting direction is to apply the idea of in-register duplication for protecting the data cache.

## References

- [1] Hp nonstop himalaya. <http://nonstop.compaq.com/>.
- [2] T. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proc. the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.
- [3] E. Borch et al. Loose loops sink chips. In *Proc. of HPCA-8*, pages 270–281, February 2002.
- [4] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of HPCA-5*, January 1999.
- [5] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.
- [6] O. Ergin et al. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proc. of MICRO-37*, pages 304–315, Portland, OR, 2004.
- [7] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. the International Symposium on Computer Architecture*, pages 98–109, June 2003.
- [8] M. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, June 2005.
- [9] J. S. Hu, G. M. Link, J. K. John, S. Wang, and S. G. Ziavras. Resource-driven optimizations for transient-fault detecting superscalar microarchitectures. In *Proc. of Tenth Asia-Pacific Computer Systems Architecture Conference (AC-SAC 05)*, Singapore, October 24–26 2005.
- [10] G. S. S. J. Adam Butts. Use-based register caching with decoupled indexing. In *Proceedings of 31st Annual International Symposium on Computer Architecture (ISCA'04)*, pages 302–313, 2004.
- [11] S. Kim and A. Somani. Area efficient architectures for information integrity checking in cache memories. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 246–255, May 1999.
- [12] L. Li et al. Soft error and energy consumption interactions: A data cache perspective. In *Proc. of ISLPED'04*, pages 132–137, 2004.
- [13] M. H. Lipasti et al. Physical register inlining. In *Proc. of ISCA-31*, pages 325–335, June 2004.
- [14] G. H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proc. of MICRO-35*, 2002.
- [15] R. E. Lyons and W. Vanderkulk. The use of tripple-modular redundancy to improve computer reliability. *IBM Journal*, April 1962.
- [16] G. Memik, M. H. Chowdhury, A. Mallik, and Y. I. Ismail. Engineering over-clocking: Reliability-performance trade-offs for high-performance register files. In *International Conference on Dependable Systems and Networks (DSN'05)*, pages 770–779, 2005.
- [17] G. Memik et al. Increasing register file immunity to transient errors. In *Proc. of DATE 2005*, Munich, Germany, May 2005.
- [18] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures: The out of order reliable superscalar (o3rs) approach. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2000.
- [19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. the 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002.
- [20] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.
- [21] M. Namjoo and E. McCluskey. Watchdog processors and detection of malfunctions at the system level. Technical Report 81-17, CRC, December 1981.
- [22] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. In *Proc. the 31st Annual International Symposium on Computer Architecture*, June 2004.
- [23] R. P. Preston et al. Design of an 8-issue superscalar risc microprocessor with simultaneous multithreading. In *Proc. IEEE International Solid-State Circuits Conference*, 2002.
- [24] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 214–224, December 2001.
- [25] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.
- [26] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. the International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.
- [27] T. Sherwood et al. Automatically characterizing large scale program behavior. In *Proc. of ASPLOS X*, October 2002.
- [28] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.
- [29] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.
- [30] J. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitecture. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, December 2004.
- [31] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.
- [32] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery via simultaneous multithreading. In *Proc. the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.
- [33] C. Weaver et al. Techniques to reduce the soft errors rate in a high-performance microprocessor. In *Proc. of ISCA-31*, 2004.
- [34] K. C. Yager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.
- [35] J. F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978 - 1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996.