On the Exploitation of Narrow-Width Values for Improving Register File Reliability

Jie Hu, *Member, IEEE*, Shuai Wang, *Student Member, IEEE*, and Sotirios G. Ziavras, *Senior Member, IEEE*

***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 making a duplicate of the value within the same data item; this *in-register duplication (IRD)* eliminates the requirement for 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. A detailed architectural vulnerability factor (AVF) analysis shows that IRD significantly reduces the AVF from 8.4% in a conventional unprotected register file to 0.1% in an IRD register file. Our experimental evaluation using the SPEC CINT2000 benchmark suite also shows that IRD provides superior read-with-duplicate (RWD) and error detection/recovery rates under heavy error injection as compared to previous reliability schemes, while only incurring a small power overhead.**

***Index Terms*—In-register duplication (IRD), narrow-width value, register file, reliability, soft errors.**

I. Introduction

ALONG with dramatic performance improvements driven by advancing silicon 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 [1], [2]. 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 [3]. 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) [4] is used to achieve highly reliable fault-tolerant computing at high hardware cost. Recently, proposals targeting at soft error problems in microprocessors have suggested utilizing the inherent resource redundancy in simultaneous multithreading (SMT) microprocessors and chip-multiprocessors (CMPs) to enhance the datapath reliability with concurrent error detection [5]–[10]. 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 [11]–[16].

Since: 1) the register file read is within the datapath loose loops [17]; 2) error-flipped intermediate computation results in the register file are very likely to propagate to later computations or to memory hierarchies; and 3) a large register file is a major die-area consumer increasing its exposure to high-energy particle strikes [18], designing high-performance error-resilient register files is of critical importance. Notice that most dual-instruction execution (DIE) processors include the register file within the sphere of replication (SoR) [7]. This is mainly due to the unbearable access latency and power overhead of ECC-protected register files [19], [20]. 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 [19] 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 (with values that 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 [21]–[24]. 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, the existing leading-0/1 detection logic within the functional units is utilized for narrow-width check. Detected narrow-width results that can be represented by no more than 32 bits automatically duplicate themselves in 64-bit processors by muxing (copying) the lower 32 bits into the higher 32 bits before being latched by the pipeline registers. We call this scheme in-register duplication (IRD). IRD stores two copies of the narrow-width value in the same register and transmits these 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

Manuscript received September 07, 2007; revised January 26, 2008. First published May 02, 2009; current version published June 19, 2009. A preliminary version of this work was presented at the International Conference on Dependable Systems and Networks (DSN-2006)—Dependable Computing and Communications Symposium (DCCS), VLSI Session, 2006.

The authors are with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102 USA (e-mail: jhu@njit.edu; sw63@njit.edu; ziavras@njit.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2009.2017441

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.

To evaluate the effectiveness of our proposed IRD scheme, we conduct both architectural vulnerability factor (AVF) measurements for register files with and without IRD, and experimental evaluation under software-implemented soft error injection. Our experimental evaluation using 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. In the meantime, IRD only incurs a small 8.8% increase in the register file power consumption. Based on a detailed register lifetime model, our AVF analysis shows that our IRD scheme achieves a dramatic reduction of 98.8% in register file AVF, from 8.4% to 0.1%, on the average. Under error injection with accelerated error rates of $10^{-5}/10^{-4}$ per selected bit per cycle, IRD schemes detect virtually all errors in narrow-width and regular values being read in. To avoid signaling unnecessary errors in the duplicate copy, IRD is further tuned to only check the parity bit of the lower 32-bit half for error detection and utilize the duplicate in the upper half for error recovery. Our experimental results show that IRD detects 99.7% of the erroneous reads for narrow-width values and successfully recovers 99.7% and 99.2% of detected errors at error rate 10^{-5} and 10^{-4}, respectively, using the uncorrupted duplicate, which makes our in-register duplication a very cost-effective design for highly reliable register files.

The rest of this paper is organized as follows. We discuss related work in Section II, and review some basics of register renaming and narrow-width register values in Section III. We elaborate on our in-register duplication design in Section IV and introduce AVF analysis model in Section V. In Section VI, we evaluate the reliability, performance, and power consumption of the proposed IRD design. Section VII concludes this paper.

II. 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 [25] and IBM z900 [26] with G5 processors. Other designs use asymmetric redundancy to include a watch-dog processor [27] or a low-performance checker processor in DIVA [28] to verify the correctness of the execution on the main processor.

Targeting at the increasing processor vulnerability to soft errors at new technologies, reliable schemes exploiting simultaneous multithreading (SMT) architectures have been extensively studied for both single processors and chip-multiprocessors, such as AR-SMT [5], Slipstream [6], SRT [7], [8], and SRTR [9]. To reclaim the performance loss due to excessive resource contentions in these redundant multithreading (RMT) approaches, recent work [29] has proposed to exploit register bit reuse and register value reuse to reduce resource redundancy in the register file. SlicK [30] on the other hand explored a partial redundant multithreading mechanism that exploits value and control-flow locality to avoid redundant execution of highly predictable slices. In the meantime, many research efforts have also been spent on exploiting the redundant resources in superscalar processors for instruction-level redundant execution against transient faults. In [11], 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 occupies only a single reorder buffer (ROB) entry. On the other hand, the dual-instruction execution scheme (DIE) in [12] 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 [13], SHREC [14], and PER-IRTR [16], try to alleviate the resource contention in DIE processors to recover the performance loss.

In [31], 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 [19] 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 [32] studied the tradeoffs 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.

III. Basics of Register Renaming and Narrow-Width Register Values

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). Since the register file size limits the effective size of the instruction window, it presents a major constraint on ILP exploitation in superscalar microprocessors. We implemented MIPS R10000 [33] style register renaming, where the architectural and physical register files are combined. Fig. 1 gives the superscalar microprocessor model simulated in this paper. Notice that a physical register is susceptible to soft errors only after a value is written into the register and before it is freed.

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 [21]–[24]. Different from the previous work, we exploit

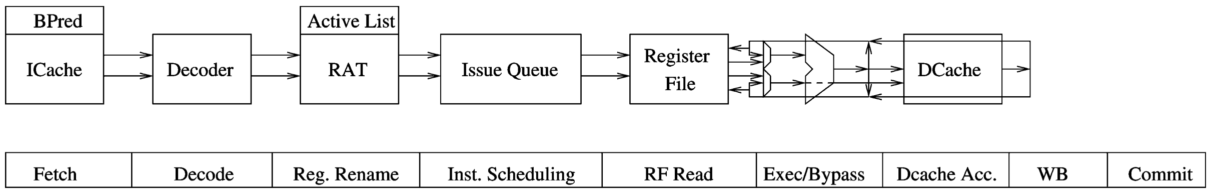


Fig. 1. Datapath and the pipeline stages of the simulated superscalar microprocessor.

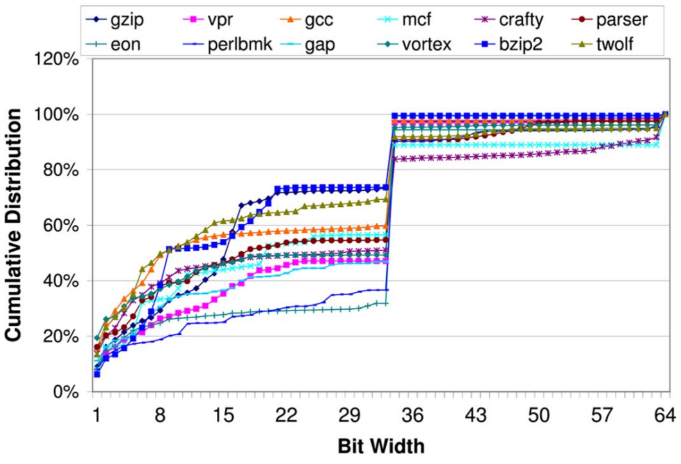
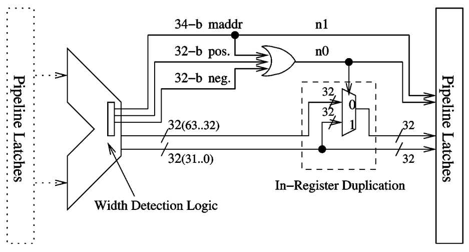


Fig. 2. Cumulative distribution of the Register value width.



(a)

n_1	n_0	meaning
0	0	regular value
0	1	32-bit pos./neg. narrow-width value
1	1	34-bit narrow-width memory address
1	0	reserved

(b)

Fig. 3. (a) Augmented functional unit datapath with narrow-width flag generation and in-register duplication logic. (b) The meaning of the value of narrowness flag bits n_1n_0.

narrow-width register values for improving the register file reliability against soft errors.

The detailed cumulative data-width distribution given in Fig. 2 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 $\mathrm{bit} = 34$ 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, which we exploit in this work for designing high-performance error-resilient register files using in-register duplication.

IV. 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.

A. *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 extract the internal signals from the existing leading-0/1 detection logic within the functional units [34] (in order to minimize its timing overhead in deeply pipelined designs at new technology generations [35]), indicating 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 Fig. 3(b). The block diagram in Fig. 3(a) shows a slightly modified datapath with added logic for setting the flag bit n_0 and in-register duplication. Notice that a narrow-width value will have the flag bit n_0 set to 1. The in-register duplication logic (the Mux in the figure) is controlled by flag bit n_0 to either perform duplication for a narrow-width value (by copying the lower 32-bit half into the higher 32-bit half) or bypass duplication for a regular value.

B. *Exploiting IRD 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 register. 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. IRD enforces at any time two copies of the narrow-width value to be stored in

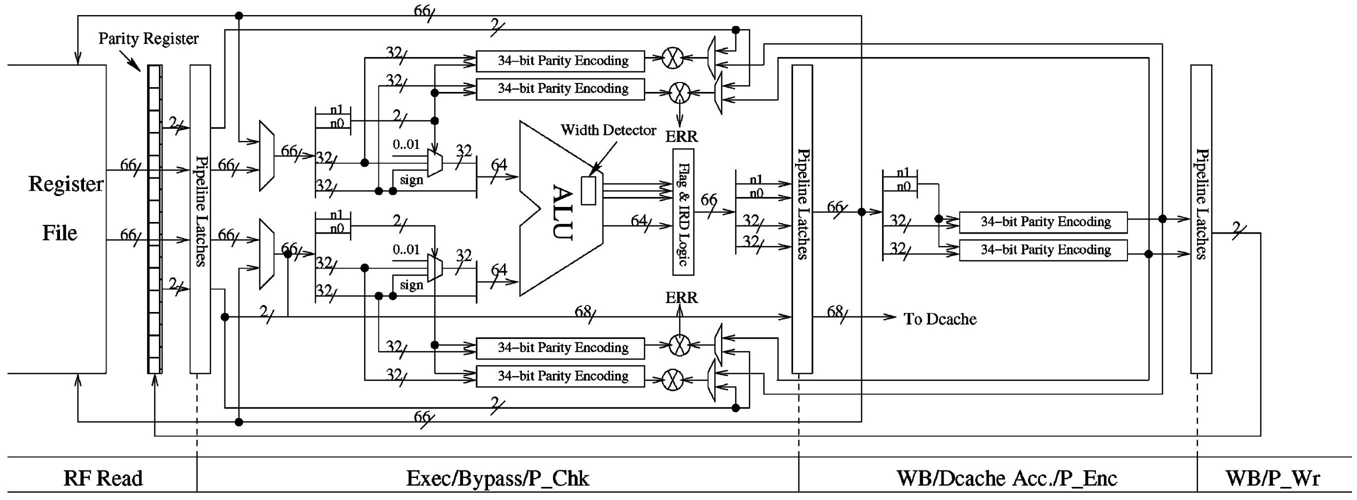


Fig. 4. Augmented datapath integrating IRD and parity coding to support both error detection and error recovery.

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 TRD and conventional redundancy-based reliable designs. IRD incurs much less hardware complexity compared to schemes utilizing idle or predicted dead registers for duplicating a data value, where the register renaming logic needs to be 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 [19]. IRD needs none of the previous 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 [19], 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 IRD.

Since the probability of the two copies of the narrow value being corrupted at exactly the same bit position is negligible, 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 vulnerability factors (AVFs) [36] 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. Notice that narrow-width operands are restored into full-width regular values at the inputs of the functional units. As shown in Fig. 4. the restoration logic, basically a Mux, is controlled by the 2-bit narrowness flag either to sign extend the lower 32-bit half or to reform the memory address for narrow-width values, or to bypass the upper 32-bit half for regular values.

C. Integrating IRD and Parity Coding

IRD itself is expected to be very effective in soft-error detection however, is not capable of recovering from an error. Since IRD 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 [31]. 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 are included in the parity coding for both 32-bit halves, thus covered by the same parity bits for the data value. We assume that parity encoding/checking takes one clock cycle.

To integrate parity coding with IRD, we need to add a separate pipeline stage to perform parity encoding after the execution stage. Fig. 4 shows the modified datapath supporting both error detection and recovery. The parity bit for each 32-bit half (and narrowness flag) 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 [17] 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 arithmetic logic unit (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 (and 2-bit flag). Parity checking basically regenerates the parity bit for each 32-bit half (and 2-bit flag) 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 latter stage of P_Chk. If the two parity bits for the lower 32-bit half (and 2-bit flag) 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 (and 2-bit flag) 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 the error is detected, an exception is raised for the higher level system(s) to solve the problem.

Since parity encoding takes one additional clock cycle, one design issue raised here is when to write back the result value and the parity bits. To avoid increasing the complexity of the register file read/write ports or the bypass network, we propose to use a special bit-addressable parity register to hold two parity bits for each entry in the register file, as shown in Fig. 4. The parity bits are written into the parity register at P_Wr stage.

IRD with parity coding is expected to be very effective in detecting and recovering single-bit errors in narrow-width values. In the presence of multi-bit errors (at a rate of several orders of magnitude lower than single-bit errors), a more aggressive detection scheme combining parity checking and duplicate comparison can be employed. Due to the extremely low possibility that the two copies are corrupted by multi-bit errors at exactly the same bit locations, multi-bit errors in narrow-width values can be effectively detected by duplicate comparison. However, IRD may lack the capability of recovering from detected multi-bit errors.

D. Protecting Regular Values

As a side benefit of in-register duplication, regular values (those that 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 (and flag bits) 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 [19] 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.

V. New Models for Register File AVF Estimation

To estimate the register file AVF, we will need to calculate the ACE (architectural correct execution) and un-ACE cycles of each register value residing within the register file. Inspired by a previous work [37] that uses lifetime analysis to compute the AVF for address-based structures exemplified by a data cache, a data translation buffer, and a store buffer, we exploit the register lifetime model as the basis for AVF estimation. As shown in Fig. 5, the lifetime of a physical register starts with the `Idle` state when it is in the free list. Once the register is allocated to rename a logical (destination) register at the renaming stage, it changes from the `Idle` state to the `Busy` state. The register

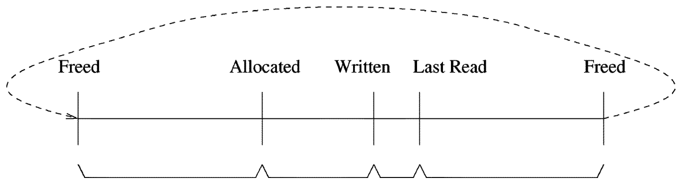


Fig. 5. Lifetime model of a physical register.

stays in the `Busy` state till the result value is written into it. The time between the write and last read to the register is referred to as the `Live` phase. After its last read, the register enters its `Dead` state. The physical register is then freed when its unmapping instruction commits. Freeing a physical register puts it back to the free list and returns it to the `Idle` state. The register lifetime model clearly indicates that except the `Live` state, all the other states in a register's lifespan are un-ACE, i.e., the register in these states has no impact on the correctness of the processor architectural state. This is simply because the register either does not contain valid data or its valid value will not be used by any later computation if the register is in the `Idle`, `Busy`, or `Dead` states. Thus the `Live` phase presents an upper bound of the register's ACE cycles. Consequently, a conservative design for a reliable register file would be protecting the register value during its `Live` phase, while most existing proposals [19] allocating copy registers at the renaming stage are clearly overkill designs.

For more accurate ACE calculation in the register file, a more detailed and comprehensive analysis model of the `Live` register value is required. We propose here a new register value classification for ACE calculation purposes. In this classification, a register value is either speculative (i.e., produced by a speculative instruction) or non-speculative (i.e., produced by a non-speculative instruction). Obviously, a speculative register value will never be committed and thus it is un-ACE. A non-speculative register value can be dynamically dead (DD) during execution because it is either first-level dynamically dead (FDD) or transitively dynamically dead (TDD). Different from previous study [38], [36] that identifies dynamically dead (both first-level and transitively) instructions for issue queue AVF estimation, this study sees new challenges in determining FDD or TDD register values. A register is first-level dynamically dead if: 1) all its consumer instructions are speculative ones, referred to as FDD_S or 2) it is not read by any instruction before being freed, referred to as FDD_N. Notice that FDD_N registers have a zero `Live` cycle. TDD registers can be further divided into three groups: 1) TDD_S due to all consumers falling into the TDD_S, FDD_S, and/or speculative ones; 2) TDD_N due to FDD_N and TDD_N consumers; and 3) TDD_NS due to a combination of TDD_S and TDD_N consumers. If a register cannot be determined as dynamically dead, it is conservatively assumed to be ACE. The detailed register classification is given in Fig. 6. All registers in categories other than *ACE Reg* are un-ACE registers.

The question is: does the `Live` time of an ACE register correspond to all ACE cycles? Not necessarily. The reason is quite straightforward: an ACE register can be consumed as the source operand in producing both ACE registers and un-ACE registers. A further breakdown of the `Live` time of an ACE register is

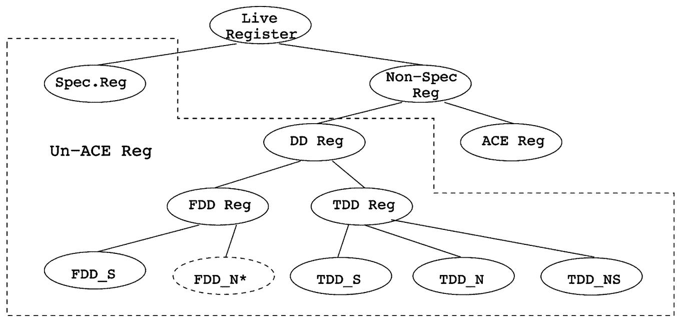


Fig. 6. Register level ACE analysis and register value classification for Live registers. (DD: dynamically dead, FDD: first-level dynamically dead, TDD: transitively dynamically dead).

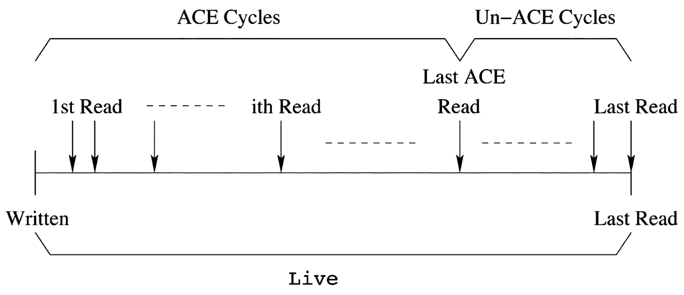


Fig. 7. Extracting un-ACE cycles from the `Live` phase of an ACE register.

given in Fig. 7. During its `Live` time, a register is to be accessed once or multiple times by its consumers. The last read to the register ends its `Live` phase. If the last ACE read (by an ACE instruction) to the register is different from its last read, then the time between the last ACE read and the last read is considered as un-ACE cycles, while the remaining is ACE cycles independent of possible un-ACE reads before the last ACE read. Thus, the AVF of a register (of this renaming instance) is calculated as the percentage of its ACE cycles over its overall lifetime. Notice that each bit of an ACE register within its ACE cycles is counted as ACE for simplicity without further exploring the masking effects of ALU operations performed on the register value.

For a base register file without any error detection/protection scheme, errors happening during a register's ACE cycles are likely to result in silent data corruptions (SDCs). Thus, the AVF of the base register file is also its SDC AVF [36], [2]. If each register entry is protected by parity coding and only single-bit soft errors are assumed, then all the errors can be detected by the parity checking logic; however, they cannot be recovered. In other words, parity coding the register file converts all SDC errors into detected unrecoverable errors (DUEs). Consequently, the SDC AVF of a parity protected register file turns into zero and its AVF is now DUE AVF. With our proposed IRD scheme, the ACE cycles of a register holding a narrow-width value are almost halved from the base case, since the upper 32 bits of the register only contains the duplicate of the narrow-width value. The 2-bit narrowness flag should be also considered as ACE bits in this case. Notice that in the IRD register file, any single-bit error to a register storing a narrow-width value will be detected and corrected by the duplicate. Thus those "ACE" cycles will not contribute to the AVF of the IRD register file. Only a regular value (still protected by parity coding) will introduce DUE AVF. Due to the high percentage of narrow-width register values, we expect that our IRD scheme will significantly reduce the register file AVF and thus its vulnerability to soft errors.

TABLE I
PARAMETERS FOR THE SIMULATED MICROPROCESSOR

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/DCache	64KB, 2 ways, 64B blocks, 2 cycle latency
L2 UCache	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
Power Parameters	
Technology	70nm
Supply Voltage	0.9V

VI. EVALUATION

A. Evaluation Framework

We derive our simulators from SimpleScalar V3.0 [39] to model a contemporary high-performance microprocessor similar to Alpha 21464 [18]. Table I shows the detailed configuration of the simulated microprocessor. For this 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 [40]. 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.

B. Duplication Rates and Performance Impact

The ability to recover register values from detected errors depends on the availability and correctness of duplicate copies. We use the write-with-duplicate (WWD) rate as a first-order estimation to measure the capability of a reliable scheme to duplicate the register values, and use the read-with-duplicate (RWD) rate as a first-order estimation to approximate reliable reads of register values against soft errors. Fig. 8 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

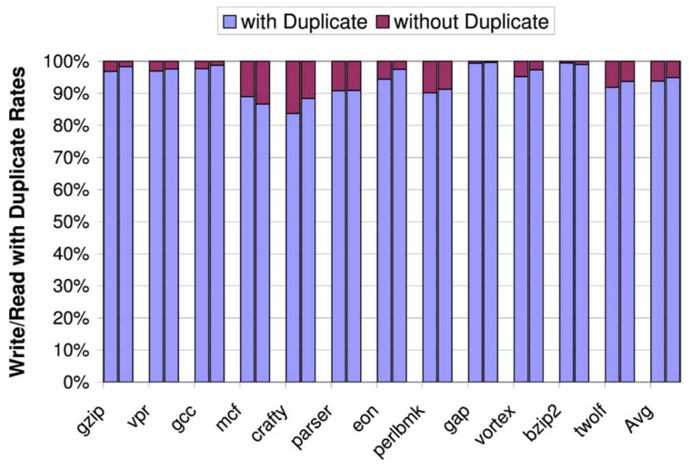


Fig. 8. WWD rate (left bar) and RWD rate (right bar) of the IRD scheme.

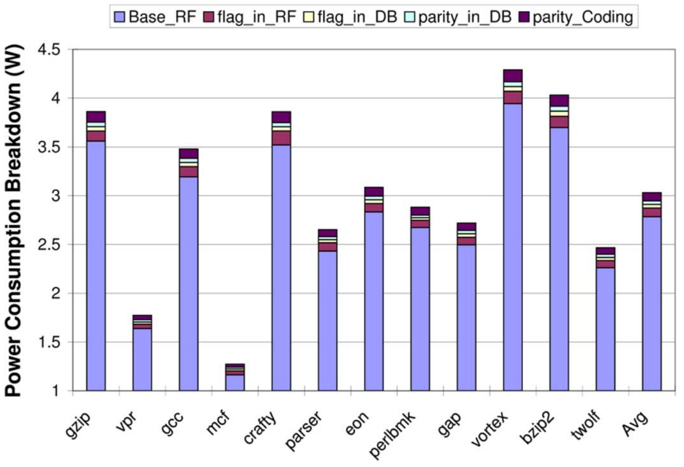


Fig. 10. Breakdown of the power consumption in the IRD register file.

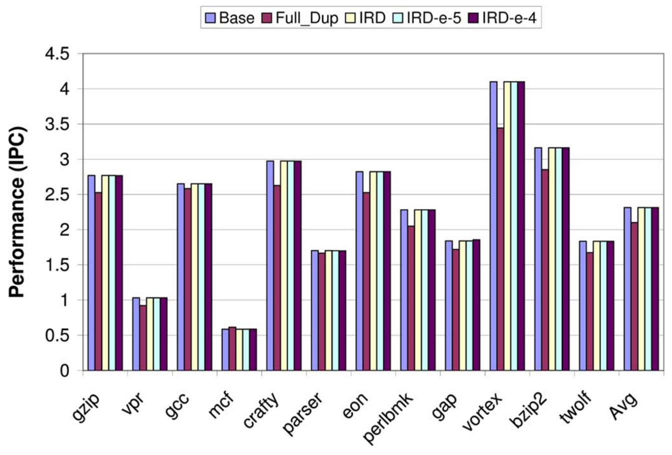


Fig. 9. Performance comparison of various register file schemes.

in [19]. In the mean time, our in-register duplication scheme avoids the hardware complexity of the latter schemes.

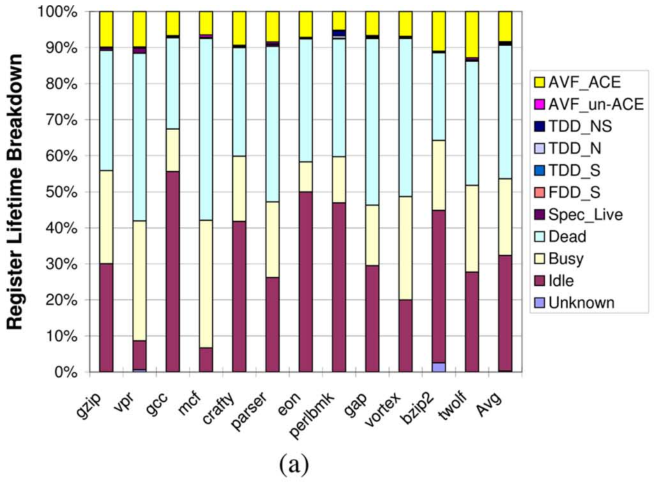
Considering a full-duplication scheme (`Full_Dup`) that allocates a copy register for each result register at the register renaming stage, the hardware implementation is much more complexity-effective than the CE/AG schemes [19] since the copy register is implied. The (`Full_Dup`) scheme achieves a rate of 100% for both WWD and RWD; however, it suffers from a significant performance loss, 7.7% on the average, as shown in Fig. 9. Our IRD `IRD` scheme duplicates the narrow-width value within the same register and requires no additional copy register, thus incurring no performance overhead. Figs. 8 and 9 together show that our schemes are very effective in providing a high error coverage for applications where the performance and cost are highly constrained.

C. *Power Efficiency of the IRD Register File*

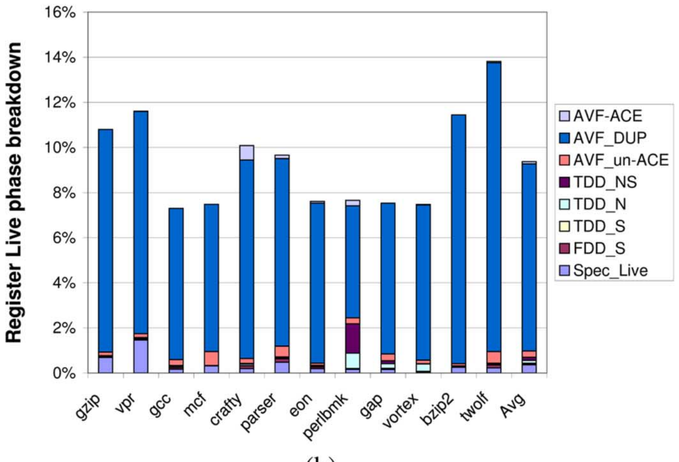
To evaluate the impact of the IRD scheme on register file power consumption, we extend the Wattch power model [41] to include power profiling for the physical register file. Compared to the base register file without any protection scheme, our IRD requires an additional 2-bit narrowness flag to be transferred and stored with each register value. Integrating parity coding has added two additional parity bits for each register entry. Notice that the parity bits are not stored with the value in the register file for the reason discussed in Section IV-C. Since we exploited input narrowness information and functional unit internal signals for simple fast narrow-width detection, the major power overhead due to the IRD scheme will be from transferring this 2-bit narrowness flag on the result bus and writing/reading the flag to/from the register file. Similarly, the parity scheme also introduces power penalties due to parity encoding and decoding (parity check) as well as transferring two parity bits (one for each 32-bit half plus 2-bit narrowness flag) on the result bus. We have augmented the power models of the register file and result bus to include the 2-bit flag and two parity bits. We borrow the published parity encoding/decoding power numbers from [42] to approximate the power consumption of the (34, 1) parity scheme in our IRD register file. All the power numbers are scaled to the 70-nm technology for the simulated microprocessor. Fig. 10 breaks down the power consumption of the IRD register file. On the average, the IRD scheme alone only causes a 4.5% power increase (due to the narrowness flag in the register file and data bus, `flag_in_RF` and `flag_in_DB`) to the base register file, `Base_RF`, and the parity scheme is responsible for additional 4.3% increase (1.3% for `parity_in_DB` and 3.0% for `parity_Coding`). Overall, the IRD register file is only 8.8% higher in power consumption than the base one, while significantly improving register file reliability.

D. *Register File AVF Estimation*

To estimate the AVF of the register file, we introduce a register AVF analysis window with 50 000 entries to record information (e.g., lifetime information, reads, and source registers, etc., required for ACE calculation) for the past 50 000 register values produced by non-speculative instructions. Notice that whether a non-speculative register value is dynamically dead (un_ACE) or not can only be determined by future instructions on its dependence chains. Following the AVF analysis model proposed in Section V, we present the AVF analysis results of the Base register file in Fig. 11(a). Among the register lifetime, on the average, the `Idle, Busy`, and `Dead` states account for 32%, 21%, and 37%, respectively. `Unknown` represents those that cannot be determined upon completion of the simulation, which is a very small portion, less than 1% in the figure. The remaining 9% is contributed by the `Live` state, which is the sum of the live time of `Spec_Live`, `FDD_S`, `TDD_S`, `TDD_N`, `TDD_NS`, and `ACE` registers. `ACE` is further broken down into `AVF_ACE` and `AVF_un-ACE` cycles. As shown in the Fig. 11(a), components of the `Live` time other than `AVF_ACE` form a total of less than 1%. Thus, the AVF of the register file in the simulated microprocessor when running SPEC 2000 CINT benchmarks is 8.4%, the percentage of `AVF_ACE` in the overall register lifetime. If it is the base register file without any protection scheme, its AVF only consists of SDC AVF, which is 8.4% in this case. While protected by parity coding, the register file converts SDC AVF into DUE AVF. Reliable register file designs shall target at reducing `AVF_ACE` for register file AVF reduction and reliability improvement. The employment of our proposed IRD scheme successfully eliminates the `AVF_ACE` cycles of a register holding a narrow-width value, since the error will be detected by the parity logic and corrected using the duplicate stored in the upper half of the same register, under the assumption of the single-bit error model. The eliminated `AVF_ACE` portion is referred to as `AVF_DUP` in the IRD register file, which is no longer ACE. The remaining part constitutes the true ACE and contributes to DUE AVF. As shown in Fig. 11(b), our IRD scheme removes 98.8% of the original `AVF_ACE` cycles, which dramatically improves register file reliability by reducing the AVF from 8.4% in the base register files to 0.1% in our IRD register file.



(a)



(b)

Fig. 11. Register lifetime breakdown for AVF measurement in: (a) a base register file, and (b) our IRD register file, a zoom-in view of its Live phase breakdown. (a) Base register file. (b) IRD register file.

E. Error Model and Soft Error Injection

To further evaluate the error resilience of our schemes, we also conducted soft error injection during the execution-driven simulation. Software-based error injection flips one bit or multiple bits in a selected register value. Since the multiple-bit error rate is several orders of magnitude lower than the single-bit error rate [20], 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} [20]), i.e., single-bit soft error rate. During error injection, if the selected register is receiving a new 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 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.

F. Error Behavior Under Soft Error Injection

We evaluated the IRD schemes with a wide range of error rates (per bit per cycle) from 10^{-7} (suggested in [20]) to 10^{0}. Due to the limited simulation of 100 million instructions, error injection with a rate of 10^{-7} injects very few errors and errors are rarely being read in. At this error rate, only single-bit errors can happen to a register and AVF measurement itself can be a very good tool for analyzing the reliability of the IRD scheme. To exercise the IRD register file's resilience to double- or multi-bit errors as in extremely harsh environments, a higher error rate is required for the injection. For illustration purposes, we only present our results for error rates of 10^{-5} (shown as "e-5") and 10^{-4} ("e-4"). Notice that these again are accelerated rates for very rare single-event upsets (SEUs).

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

TABLE II
CHARACTERIZATION OF ERRONEOUS READS FOR INPUT OPERANDS

	Error Sources				Error Types			
	e-5		e-4		e-5		e-4	
	RF	FWD	RF	FWD	SE	ME	SE	ME
gzip	86.3%	13.7%	83.8%	16.2%	100.0%	0.0%	98.9%	1.1%
vpr	97.1%	2.9%	95.2%	4.8%	100.0%	0.0%	100.0%	0.0%
gcc	78.1%	21.9%	89.8%	10.2%	100.0%	0.0%	100.0%	0.0%
mcf	93.0%	7.0%	96.8%	3.2%	99.1%	0.9%	97.6%	2.4%
crafty	90.6%	9.4%	86.6%	13.4%	100.0%	0.0%	99.7%	0.3%
parser	91.3%	8.7%	90.3%	9.7%	98.6%	1.4%	98.2%	1.8%
eon	90.3%	9.7%	86.9%	13.1%	100.0%	0.0%	100.0%	0.0%
perlbmk	87.8%	12.2%	87.5%	12.5%	100.0%	0.0%	100.0%	0.0%
gap	91.5%	8.5%	92.0%	8.0%	100.0%	0.0%	100.0%	0.0%
vortex	86.4%	13.6%	86.4%	13.6%	100.0%	0.0%	100.0%	0.0%
bzip2	85.7%	14.3%	83.4%	16.6%	100.0%	0.0%	98.4%	1.6%
twolf	96.1%	3.9%	93.0%	7.0%	100.0%	0.0%	100.0%	0.0%
Avg	89.5%	10.5%	89.3%	10.7%	99.8%	0.2%	99.4%	0.6%

*RF: Register File, FWD: Forward Network, SE: Sing-bit Error, ME: Multi-bit Error

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. Table II shows that the majority of the erroneous reads, for both soft error rates of e-5 and e-4: 1) around 89% on the average, are due to the corrupted value read from the register file (`RF`) and 2) the remaining 11% are due to wire flips when the value is being forwarded by the bypass network (`FWD`). Thus, the register file is still the major source of erroneous reads. A second breakdown of erroneous reads in Table II shows that most readin errors are single-bit errors (`SE`), 99.8% (99.4%) 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 [43], during which the same register entry is rarely hit by multiple errors.

G. Error Detection and Recovery From Detected Soft Errors

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 parity coding scheme is not capable of detecting an even number of bit errors in a 32-bit half. Fig. 12 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.

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. 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 also 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. Fig. 13 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 Fig. 9. The performance overhead due to error recovery is negligible at these two error rates.

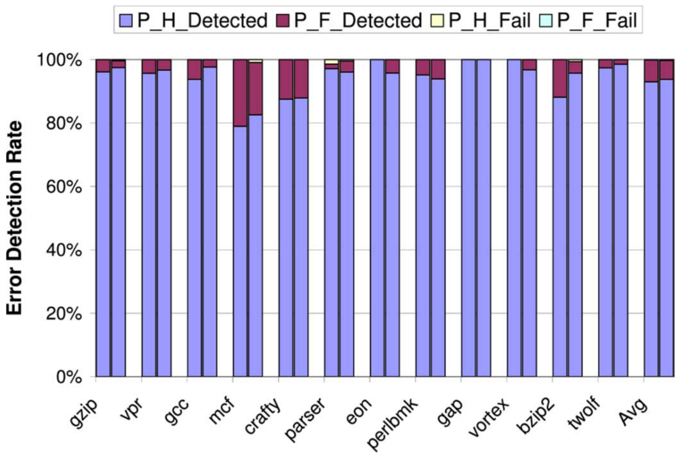


Fig. 12. Soft error detection in the IRD scheme by parity checking. (Left bar for e-5 and right bar for e-4).

Overall, these results confirm that our in-register duplication scheme that exploits narrow-width values is very effective in detecting and recovering from soft errors occurring in the register file, the bypass network, or the result writeback bus, while only incurring some minor microarchitectural modifications. It is important to notice that this high error detection/recovery rate in the IRD register file is achieved under the extremely high error injection rates that are unlikely to happen in the real world. Thus, for realistic applications experiencing significantly low

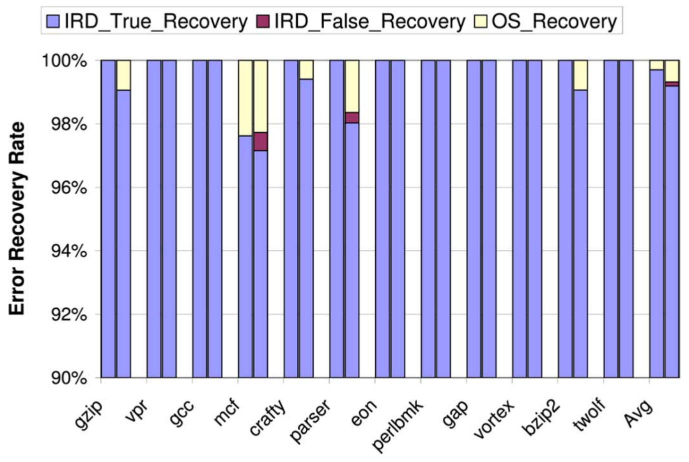


Fig. 13. Error recovery rate of detected errors in IRD scheme, under error injection rates of 10^{-5} (left bar) and 10^{-4} (right bar) per selected bit per cycle.

error incidents, it is of crucial importance that reliable designs only incur minimal cost/overhead in terms of hardware, performance, and power consumption.

VII. Conclusion 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 IRD scheme creates 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. AVF measurement based on a new analysis model has shown that our IRD scheme achieves an extremely low AVF of 0.1% in the register file, a 98.8% reduction over a base one. Evaluation via software-based error injection shows that our IRD scheme demonstrates superior error detection and recovery rates at minimum hardware cost, making it a suitable design choice 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. Another interesting direction is to apply the idea of in-register duplication for protecting the data cache.

References

[1] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin, "IBM experiments in soft fails in computer electronics (1978–1994)," *IBM J. Res. Development*, vol. 40, no. 1, pp. 3–18, Jan. 1996.

[2] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft errors rate in a high-performance microprocessor," in *Proc. ISCA-31*, 2004, pp. 264–275.

[3] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2002, pp. 389–398.

[4] R. E. Lyons and W. Vanderkulk, "The use of tripple-modular redundancy to improve computer reliability," *IBM J.*, vol. 6, no. 2, pp. 200–209, Apr. 1962.

[5] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Jun. 1999, pp. 84–91.

[6] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: Improving both performance and fault tolerance," in *Proc. 9th Int. Conf. Arch. Support for Program. Lang. Operat. Syst.*, 2000, pp. 257–268.

[7] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proc. 27th Annu. Int. Symp. Comput. Arch.*, Jun. 2000, pp. 25–36.

[8] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. 29th Annu. Int. Symp. Comput. Arch.*, May 2002, pp. 99–110.

[9] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery via simultaneous multithreading," in *Proc. 29th Annu. Int. Symp. Comput. Arch.*, May 2002, pp. 87–98.

[10] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proc. Int. Symp. Comput. Arch.*, Jun. 2003, pp. 98–109.

[11] A. Mendelson and N. Suri, "Designing high-performance and reliable superscalar architectures: The out of order reliable superscalar (o3rs) approach," in *Proc. Int. Conf. Depend. Syst. Netw.*, Jun. 2000, pp. 473–481.

[12] J. Ray, J. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proc. 34th Annu. IEEE/ACM Int. Symp. Microarch.*, Dec. 2001, pp. 214–224.

[13] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy," in *Proc. 31st Annu. Int. Symp. Comput. Arch.*, Jun. 2004, pp. 376–386.

[14] J. Smolens, J. Kim, J. C. Hoe, and B. Falsafi, "Efficient resource sharing in concurrent error detecting superscalar microarchitecture," in *Proc. ACM/IEEE Int. Symp. Microarch. (MICRO)*, Dec. 2004, pp. 257–268.

[15] J. S. Hu, G. M. Link, J. K. John, S. Wang, and S. G. Ziavras, "Resource-driven optimizations for transient-fault detecting superscalar microarchitectures," presented at the 10th Asia-Pac. Comput. Syst. Arch. Conf. (ACSAC), Singapore, Oct. 2005.

[16] M. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection," in *Proc. 32nd Annu. Int. Symp. Comput. Arch. (ISCA)*, Jun. 2005, pp. 172–183.

[17] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *Proc. HPCA-8*, Feb. 2002, pp. 270–281.

[18] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, and M. J. Smith, "Design of an 8-issue superscalar RISC microprocessor with simultaneous multithreading," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2002, pp. 266–267.

[19] G. Memik, M. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," presented at the DATE, Munich, Germany, May 2005.

[20] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Soft error and energy consumption interactions: A data cache perspective," in *Proc. ISLPED*, 2004, pp. 132–137.

[21] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proc. HPCA-5*, Jan. 1999, pp. 13–22.

[22] G. H. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," in *Proc. MICRO-35*, 2002, pp. 395–405.

[23] M. H. Lipasti, B. R. Mestan, and E. Gunadi, "Physical register inlining," in *Proc. ISCA-31*, Jun. 2004, pp. 325–335.

[24] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *Proc. MICRO-37*, Portland, OR, 2004, pp. 304–315.

[25] Compaq, Palo Alto, CA, "HP nonstop himalaya," 1997 [Online]. Available: http://nonstop.compaq.com/

[26] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar./Apr. 1999.

[27] M. Namjoo and E. McCluskey, "Watchdog processors and detection of malfunctions at the system level," CRC, Tech. Rep. 81-17, Dec. 1981.

[28] T. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Proc. 32nd Annu. IEEE/ACM Int. Symp. Microarch.*, Nov. 1999, pp. 196–207.

[29] S. Kumar and A. Aggarwal, "Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors," in *Proc. 12th Int. Symp. High-Perform. Comput. Arch.*, Feb. 2006, pp. 212–221.

[30] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi, "Slick: Slice-based locality exploitation for efficient redundant multithreading," in *Proc. 12th Int. Conf. Arch. Support for Program. Lang. Operat. Syst. (ASPLOS-XII)*, 2006, pp. 95–105.

[31] S. Kim and A. Somani, "Area efficient architectures for information integrity checking in cache memories," in *Proc. Int. Symp. Comput. Arch. (ISCA)*, May 1999, pp. 246–255.

[32] G. Memik, M. H. Chowdhury, A. Mallik, and Y. I. Ismail, "Engineering over-clocking: Reliability-performance trade-offs for high-performance register files," in *Proc. Int. Conf. Depend. Syst. Netw. (DSN)*, 2005, pp. 770–779.

[33] K. C. Yager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.

[34] D. R. Lutz and D. N. Jayasimha, "Early zero detection," in *Proc. Int. Conf. Comput. Des. (ICCD)*, 1996, pp. 545–550.

[35] M. S. Hrishikesh, D. Burger, S. W. Keckler, P. emkishore Shivakumar, N. P. Jouppi, and K. I. Farkas, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inver ter delays," in *Proc. 29th Annu. Int. Symp. Comput. Arch.*, May 2002, pp. 14–24.

[36] 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. 36th Annu. IEEE/ACM Int. Symp. Microarch.*, Dec. 2003, pp. 29–40.

[37] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," in *Proc. 32nd Annu. Int. Symp. Comput. Arch. (ISCA)*, 2005, pp. 532–543.

[38] J. A. Butts and G. Sohi, "Dynamic dead-instruction detection and elimination," in *Proc. 10th Int. Conf. Arch. Support for Program. Lang. Operat. Syst. (ASPLOS-X)*, 2002, pp. 199–210.

[39] D. Burger and T. M. Austin, "The Simplescalar tool set, Version 2.0," Comput. Sci. Dept. Univ. Wisconsin-Madison, Tech. Rep. 1342, 1997.

[40] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS X*, Oct. 2002, pp. 45–57.

[41] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. Symp. High-Perform. Comput. Arch.*, 2000, pp. 83–94.

[42] R. Phelan, "Addressing soft errors in arm core-based SOC," ARM Ltd., ARM White Paper, Dec. 2003.

[43] G. S. S. J. Adam Butts, "Use-based register caching with decoupled indexing," in *Proc. 31st Annu. Int. Symp. Comput. Arch. (ISCA)*, 2004, pp. 302–313.

Jie Hu (S'02–M'04) received the B.E. degree in computer science and engineering from Beijing University of Aeronautics and Astronautics, Beijing, China, in 1997, the M.E. degree in signal and information processing from Peking University, Beijing, China, in 2000, and the Ph.D. degree in computer science and engineering from the Pennsylvania State University, University Park, in 2004.

He has been an Assistant Professor with the Electrical and Computer Engineering Department, New Jersey Institute of Technology, Newark, since 2004. His research interests include the areas of computer architecture, power-aware systems design, power-efficient memory hierarchy, high-performance microprocessors, complexity-effective processor microarchitecture, power-efficient reliable systems, compiler optimizations for performance and power consumption, and reconfigurable computing architecture.

Shuai Wang (S'07) received the B.S. degree in computer science from Nanjing University, Nanjing, China, in 2003. He is currently pursuing the Ph.D. degree in electrical and computer engineering from the New Jersey Institute of Technology, Newark.

He is a member of the Computer Architecture and Parallel Processing Lab (CAPPL), New Jersey Institute of Technology. His research interests include power/thermal-aware systems design, reliable circuits and systems, reconfigurable computing architectures, and embedded systems.

Sotirios G. Ziavras (S'83–M'90–SM'96) received the Diploma in electrical engineering from the National Technical University of Athens, Athens, Greece, in 1984, the M.Sc. degree in computer engineering from Ohio University, Athens, in 1985, and the D.Sc. degree in computer science from George Washington University, Washington, DC, in 1990.

He was a Distinguished Graduate Teaching Assistant with George Washington University. He carried out research in supercomputing for computer vision at the Center for Automation Research in the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He is currently a Professor with the Electrical and Computer Engineering (ECE) Department, New Jersey Institute of Technology, Newark, and also the ECE Associate Chair for Graduate Studies. He has authored about 140 research papers. He is listed, among others, in *Who's Who in Science and Engineering*, *Who's Who in America*, *Who's Who in the World*, and *Who's Who in the East*. His main research interests include advanced computer architecture, reconfigurable computing, embedded computing systems, parallel and distributed computer architectures and algorithms, and network router design.