Resource-Driven Optimizations for Transient-Fault Detecting SuperScalar Microarchitectures

Jie S. Hu[1], G. M. Link[2], Johnsy K. John[1], Shuai Wang[1], and Sotirios G. Ziavras[1]

[1] Electrical and Computer Engineering, New Jersey Institute of Technology
Newark, NJ 07102, USA
[2] Microsystems Design Lab, The Pennsylvania State University
University Park, PA 16802, USA

Abstract. Increasing microprocessor vulnerability to soft errors induced by neutron and alpha particle strikes prevents aggressive scaling and integration of transistors in future technologies if left unaddressed. Previously proposed instruction-level redundant execution, as a means of detecting errors, suffers from a severe performance loss due to the resource shortage caused by the large number of redundant instructions injected into the superscalar core. In this paper, we propose to apply three architectural enhancements, namely 1) floating-point unit sharing (FUS), 2) prioritizing primary instructions (PRI), and 3) early retiring of redundant instructions (ERT), that enable transient-fault detecting redundant execution in superscalar microarchitectures with a much smaller performance penalty, while maintaining the original full coverage of soft errors. In addition, our enhancements are compatible with many other proposed techniques, allowing for further performance improvement.

1 Introduction

Transient hardware failures due to single-event upsets (SEUs) (a.k.a., soft errors) are becoming an increasing concern in microprocessor design at new technologies. Soft errors happen when the internal states of circuit nodes are changed due to 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 [20]. Technology trends such as continuously reducing logic depths, lowering supply voltages, smaller nodal capacitances, and increasing clock frequency will make future microprocessors more susceptible to soft errors. As such, future processors must not only protect memory elements against soft errors with error checking and correcting codes such as parity and ECC, but also protect the combinational logic within the data path in some fashion [15]. Consequently, previous proposals have suggested utilizing the inherent resource redundancy in simultaneous multithreading (SMT) microprocessors and chip-multiprocessors (CMP) to enhance the datapath reliability with concurrent error detection [12, 18, 11, 19, 4, 7]. Some recent 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 [6, 10, 9, 17]. Most of these techniques execute redundant copies of instructions and compare the results to verify the absence of single-event upsets. The goal of such a reliable datapath design is to provide affordable reliability in microprocessors with minimized cost and complexity increase. However, instruction-level redundant execution causes a severe performance penalty (up to 45% in dual-instruction execution (DIE) for a set of SPEC95 and SPEC2000 benchmarks [10]) as compared to an identically configured superscalar core without any redundant execution. This performance loss is mainly due to resource shortages and contention in functional units, issue bandwidth, and the instruction window, caused by the large number of redundant instructions being injected into the superscalar core [17]. Therefore, to provide the perfect soft-error coverage with minimum performance loss and minimum hardware changes is the major challenge in transient-error detecting dual-instruction execution designs.

In this paper, we first analyze the resource contention on different types of functional units in dual-instruction execution (DIE) environment. Our analysis indicates that different types of functional units receive very different contention and that some applications do not cause significant contention on functional units in redundant execution. The performance of integer benchmarks, especially those with high and moderate IPC (instructions per cycle), is significantly constrained by the available integer ALUs. On the other hand, most low IPC floating-point and integer benchmarks suffer more performance loss from the reduced instruction window size due to the redundant instructions. Based on these observations, we propose applying three architectural enhancements for instruction-level redundant execution, namely 1) floating-point unit sharing (FUS), 2) prioritizing the primary instructions (PRI), and 3) early retiring (ERT) of redundant instructions, for full protection against transient faults with negligible impact on area and a much lower performance overhead.

Our floating-point unit sharing (FUS) exploits the load imbalance between integer ALUs and floating-point ALUs and alleviates the pressure on the integer ALUs in DIE environment by offloading integer ALU operations to idle floating-point ALUs. Different from a previous compiler scheme [13], floating-point unit sharing utilizes a conventional unified instruction scheduler to issue operations to both integer and floating-point functional units. Prioritizing the primary stream (PRI) for instruction scheduling exploits the fact that the duplicate copy can directly use the results computed from the primary execution and form asymmetric data dependences between the duplicate stream and the primary stream. PRI has the ability to restrain the duplicate instructions (along the mispredicted path) from competing with the primary instructions, thus has the potential to improve performance in DIE. As dual-instruction execution effectively halves the size of the instruction window, it may significantly reduces the ability of the processor to extract higher ILP (instruction level parallelism) at runtime. This is especially true for most low IPC floating-point benchmarks, as they rely on a large instruction window to expose available ILP. The early-retiring (ERT) approach

exploits the fact that it is not necessary to keep all redundant copies occupying the instruction window once a redundant instruction is issued for execution. ERT immediately removes the redundant instruction (either the original or the duplicate copy) from the instruction window after it is issued to increase the effective window size, and keeps the last issued redundant copy of that instruction in the window until the commit stage, allowing result comparison before retiring the instruction from the instruction window. Our experimental evaluation using SPEC2000 benchmarks shows these three proposed enhancements, when combined, reduce the performance penalty of dual-instruction execution (DIE) from 37% to 17% for floating-point benchmarks, and from 26% to 6% for integer benchmarks, allowing it to be used in a wide variety of situations without significant performance penalties. In addition, our enhancements are compatible with many other proposed techniques, allowing for further performance improvement.

The rest of the paper is organized as follows. Section 2 reviews the background of dual instruction execution and presents our experimental framework. A detailed analysis of resource competition on different functional units, issue bandwidth, and the instruction window is given in Section 3. In Section 4, we propose three architectural enhancements, floating-point unit sharing (FUS), prioritizing the primary instructions (PRI), and early retiring (ERT) of redundant instructions for performance recovery in DIE. We evaluate our proposed schemes for dual-instruction execution in Section 5 and discuss related work in Section 6. Section 7 concludes this paper.

2 Baseline Microarchitectures and Experimental Framework

2.1 Basics about SIE and DIE

We use the same terminologies as in [9], where SIE (single instruction execution) refers to the baseline superscalar processors without any redundant instruction execution, and DIE (dual instruction execution) refers to the same processor while executing a redundant copy of the original code for transient-error detection. To support two-way redundant execution in DIE, each instruction is duplicated at the register renaming stage to form two independent instruction streams (the primary and duplicate instruction streams) that are scheduled individually. The register renaming logic only needs a very slight change to support DIE, as the physical register number to be assigned to an operand in the duplicate instruction is implied by the corresponding field in the primary instruction [10]. At commit stage, the results of these two redundant instructions are compared before retirement to verify the absence of soft errors. If the two results do not match, a soft error is implied, and a recovery must be initiated. The datapath of a DIE superscalar processor is given in Figure 1. The shaded area sketches the *sphere of replication* in DIE datapath. All other components outside this sphere are assumed to be protected by other means such as information redundancy (using parity, ECC, etc.). Therefore, a full coverage of soft errors

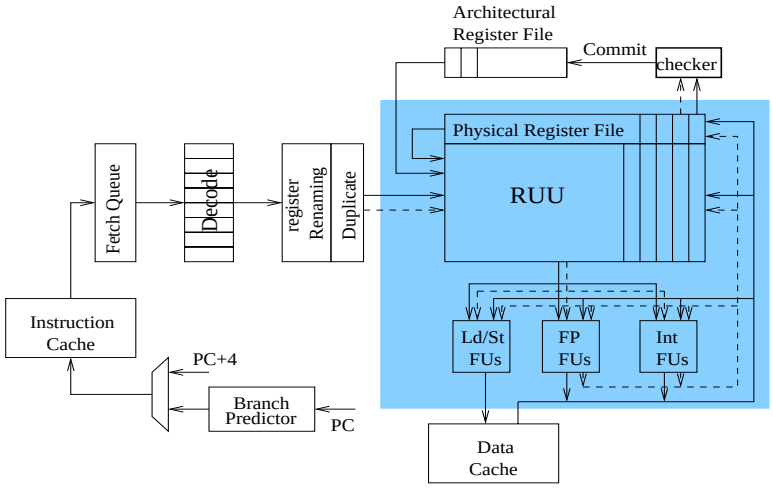

Fig. 1. Datapath of DIE superscalar processors for current transient error detection as proposed in [10]. The Sphere of Replication is sketched by the shaded area.

is achieved for the superscalar datapath through the DIE mechanism. However, this full coverage comes at the cost of severe performance loss (from 2% to 45% for SPEC2000 benchmarks as reported in [10]) due to the hardware resource contention and shortage caused by redundant instruction execution.

2.2 Experimental Setup

Our simulators are derived from SimpleScalar V3.0 [3], an execution-driven simulator for out-of-order superscalar processors. In our implementation, the physical register file is separated from the register update unit (RUU) structure. The baseline superscalar processor, which model a contemporary high-performance superscalar processor, (SIE) is configured with parameters given in Table 1.

For experimental evaluation, we use a representative subset of SPEC2000 suite, including 10 integer benchmarks and 7 floating-point benchmarks. The SPEC2000 benchmarks were compiled for Alpha instruction set architecture with "peak" tuning. We use the reference input sets for this study. Each benchmark is first fast-forwarded to its early single simulation point specified by SimPoint [14]. 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 details.

3 Redundant Instruction Execution Analysis and Motivation

This section presents our detailed analysis of resource contention on different types of functional units and motivates our proposals for performance improve-

Parameters	Value
RUU size	128 entries
Load/Store Queue (LSQ) size	64 entries
Physical Register File	128 registers
Fetch/Decode/Issue/Commit Width	8 instructions per cycle
Function Units	4 IALU, 2 IMULT/IDIV, 2 FALU, 1 FMULT/FDIV/FSQRT 2 Mem Read/Write ports
Branch Predictor	combined predictor with 4K meta-table, a bimodal predictor with 4K table, and a 2-level gshare predictor with 12-bit history BTB: 2048-entry 4-way, RAS: 32-entry
L1 ICache	64KB, 2 ways, 32B blocks, 2 cycle latency
L1 DCache	64KB, 2 ways, 32B blocks, 2 cycle latency
L2 UCache	1MB, 4 ways, 64B blocks, 12 cycle latency
Memory	200 cycles first chunk, 12 cycles rest
TLB	4 way, ITLB 64 entry, DTLB 128 entry, 30 cycle miss penalty

Table 1. Parameters for the simulated baseline superscalar processor SIE.

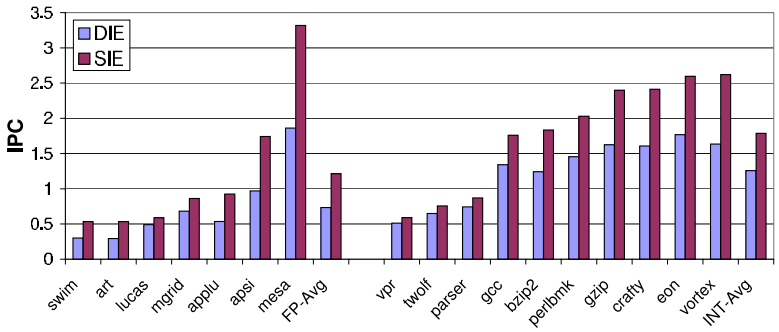

Fig. 2. Performance loss due to dual-instruction execution: DIE vs. SIE.

ment in redundant instruction execution. A performance comparison between DIE and SIE given in Figure 2 shows that, across all these benchmarks, DIE loses performance up to 44% in floating-point benchmarks and up to 38% in integer benchmarks, compared to SIE. On the average, the performance loss of DIE vs. SIE is 37% for floating-point benchmarks and 26% for integer benchmarks. Note that both SIE and DIE have an issue width of 8 instructions per cycle and 11 functional units including memory Read/Write ports as given in Table 1.

3.1 Resource Contention on Different Functional Units

The functional unit pool in the simulated SIE and DIE processors consists of 4 integer ALUs, 2 integer multiplier/dividers, 2 floating-point ALUs, 1 floating-point multiplier/divider/sqrters, and 2 memory read/write ports. Integer ALUs perform following operations: arithmetic, compare, move, control, memory address calculation, logic, shift, and byte manipulation. All operations are pipelined

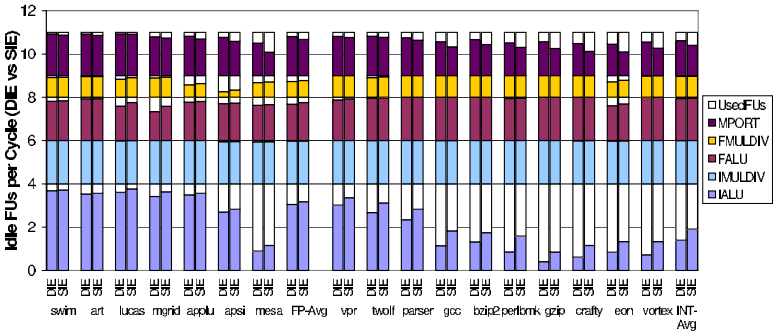

Fig. 3. Functional unit idleness/utilization comparison between DIE and SIE.

except division and sqrt operations. Obviously, a scheme trying to double the effective bandwidth of all functional units might be an overkill since not all the functional units are the contenting resources in DIE. Thus, the critical question we are going to answer here is: *what is the resource contention on these five different functional units?*

To better understand the resource contention caused by different instructions, Figure 3 gives the average idleness/utilization per cycle for each of these five different functional units in DIE comparing to SIE. The clear message conveyed through Figure 3 is that the integer ALUs are the most heavily used functional units for both integer benchmarks and floating-point benchmarks with moderate and high IPC (> 1.5). Notice that the integer ALU utilization in these benchmarks are more than 2 in SIE, which means they will require more integer ALUs than the available 4 integer ALUs in DIE in order to avoid performance loss. Consequently, integer ALU contention will be the big performance hurdle in DIE for those benchmarks provided with enough issue width. On the other hand, low IPC benchmarks do not put too much pressure on integer ALUs in both SIE and DIE, which indicates that the integer ALU contention is not the major cause of performance loss in DIE. Another important observation from Figure 3 is that the majority of floating-point ALUs are remaining idle during the simulation of both integer and floating-point benchmarks. If we can utilize these idle floating-point ALUs to take off some load from the heavily used integer ALUs in high IPC applications, the performance of DIE can be improved by providing a much larger bandwidth for integer ALU operations.

3.2 Competition on Issue Bandwidth

Besides functional unit contention, the increased competition on issue ports poses another source of performance loss in DIE. However, issue port competition is closely related to functional unit contention. Doubling the issue width alone without increasing the functional unit bandwidth or instruction window size has negligible impact on performance [9]. Figure 3 also gives the average raw numbers of instructions issued per cycle (the sum of used functional units) for DIE and

SIE. Except for few benchmarks such as *mesa* (4.5 instructions per cycle) and *eon* (4.1 instructions per cycle), the raw number of issued instructions per cycle is lower than 4 in SIE. Notice that the issue width is 8 instructions per cycle in the simulated SIE and DIE processors. This raises the question:*can we effectively use the remaining issue bandwidth for the redundant stream in DIE?* This can be achieved by an enhanced DIE microarchitecture that forms dependences between the primary and duplicate streams and prioritizes instructions in the primary stream at issue stage, which is detailed in Section 4.2.

3.3 Impact of Half-sized Instruction Window in DIE

The size of the instruction window in DIE is effectively halved when compared to the instruction window in SIE. Superscalar architectures rely on the instruction window to extract ILP of applications at runtime. Applications that need to maintain a large instruction window to expose ILP are very sensitive to the window size variation. In general, a size reduction in the instruction window prevents available ILP from being extracted and thus hurts performance. That is also one of the reasons why the functional units and issue bandwidth can not be saturated in DIE as shown in Figure 3. More importantly, the performance loss of low IPC floating-point and integer benchmarks in DIE is not likely due to the contention on functional unit bandwidth or issue width as indicated by Figure 2 and Figure 3. Instead, we suggest the half-sized instruction window be the performance killer for those low IPC benchmarks in DIE. In this work, we exploit the redundant nature of the duplicate instructions in the instruction window to effectively increase the window size for redundant execution.

4 Resource-Driven Optimizations for DIE

4.1 Exploiting Idle FP Resources for Integer Execution

As modern processors are expected to execute floating-point operations as well as integer operations, most commercial processors have a number of floating-point ALUs available on chip. Our analysis of functional unit utilization in the previous Section indicates that the integer ALUs bandwidth presents a major performance bottleneck for high IPC applications in DIE, while most of the floating-point ALUs are in idle state. We adopt the idea of offloading integer ALU operations to floating-point ALUs in [13] to exploit idle floating-point ALUs thus reducing the pressure on integer ALUs in DIE. In [13] the compiler takes full responsibility for identifying code segment (integer operations) to be executed in floating-point units and inserting special supporting instructions (in need of instruction set extensions) to control the offloaded integer instructions. However, our scheme, called floating-point unit sharing (FUS), is a pure hardware implementation. We exploit a unified instruction scheduler that issues instructions to both integer and floating-point functional units. In a value-captured scheduler, the results along with tags are broadcasted back to the instruction window at writeback stage.

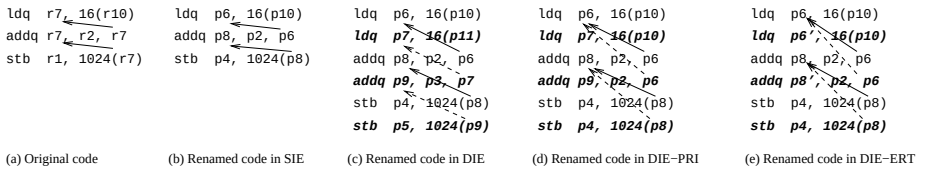

Fig. 4. A comparison of the renaming schemes in SIE, DIE, DIE-PRI, and DIE-ERT. Code in *italic* and black is the duplicate copy and arrow lines indicate the true data dependences between instructions.

In FUS, only duplicate instructions are allowed to be executed by floating-point units during the lack of integer ALUs. When offloading to floating-point units, the integer ALU instructions are assigned an additional flag to direct the floating-point units to perform the corresponding integer operations.

Since modern microprocessors already have their floating-point units supporting multimedia operations in a SIMD fashion [5], the hardware modifications required to support FUS are minimal. Basically, there are two ways to augment the floating-point ALU for integer ALU operations. One is to slightly modify the mantissa adder to support full-size integer ALU operations and bypass all unnecessary blocks such as mantissa alignment, normalization, and exponent adjustment, during integer operations. The second choice is to embed the simple small integer unit within the much larger floating-point unit. The hardware costs for both these two options at today's technology should be small. Once an integer ALU operation is issued to a floating-point unit, it either has its source operands ready (stored in its RUU entry) while waiting in the issue queue or obtains operands from the bypassing network. In the first case, the opcode and source operand values are read out from RUU entry and sent to the functional unit simultaneously. However, the latter case will require bypassing path from integer units to all floating-point ALUs and vice versa. Remember that FUS only allows the duplicate copy to be issued to the floating-point units, and with prioritizing the primary stream discussed in the next subsection, the duplicate instructions do not have out dependences and do not need to broadcast results to the issue queue. Thus, there are two design choices concerning the changes to the bypassing network. First choice is to augment the existing bypassing network with additional path from integer functional units to floating-point ALUs. The second one is to issue only duplicate instructions with already available source operands to floating-point ALUs or delay their issue until the issue queue captures all the source operands. In our design, we choose the first choice for performance sake.

4.2 Prioritizing Instructions in the Primary Stream

We observed that the two redundant streams in DIE do not have to be independent of each other in order to provide full coverage of soft errors in the sphere

of replication. The duplicate stream can directly receive source operand values from the primary stream (results produced) rather than itself since these result values will finally be verified when the producer instruction is committed. Thus the duplicate instructions form dependences on instructions in the primary stream rather than within its duplicate stream. More importantly, the duplicate instruction do not need to maintain out dependences, which means the duplicate instructions produce results only for soft-error checking against the primary copies. Since both the primary and duplicate streams are data dependent on instructions in the primary stream, the primary instructions should be issued as soon as possible for performance consideration. Once the primary instruction is issued, it wakes up instructions in both redundant streams. Delaying duplicate instructions from issue does not hurt DIE performance since no instruction depends on the results of duplicate instructions, unless it blocks instructions from retiring. Based on this observation, we propose to prioritize the primary instructions (PRI) for instruction scheduling. Prioritizing the primary instructions helps restrain the duplicate instruction (if on the mispredicted path) from competing with the primary instructions for computing resources. In this sense, it has the potential to improve DIE performance. When two instructions (one from the primary stream, the other from the duplicate stream) compete for an issue port, the primary instruction always wins because of its high priority assigned. The priority can be implemented by introducing an additional bit to the priority logic indicating whether it is a duplicate or not. Priority within each stream is same as in SIE using oldest-instruction first policy.

To support this DIE-PRI microarchitecture, we redesign the register renaming strategy where the source operands of the duplicate are renamed to the physical registers assigned to the primary instructions producing the results and the result register renaming of the duplicate is the same as in the original DIE. Figure 4 shows an example code renamed in SIE, DIE, and the new DIE. Rename/physical registers start with "p" in the code. DIE register renaming form two redundant streams that maintain dependence chains within themselves as shown in Figure 4 (c). In the new DIE microarchitecture, although both copies are assigned physical registers if a result is to be produced, the duplicate copy forms dependences on the primary copy as illustrated in Figure 4 (d). Notice that combined with PRI, the implementation of FUS can be further simplified by eliminating the result forwarding path from floating-point ALUs to integer functional units.

4.3 Early Retiring (ERT) Redundant Instructions

Notice that the DIE performance of many low IPC applications is constrained by the instruction window size rather than the integer ALU bandwidth or issue width as discussed in the previous Section. As the size of the instruction queue remains fixed when entering dual-instruction execution environment, the available instruction window is effectively reduced by a factor of two. This effectively half-sized instruction window significantly handicaps the superscalar processors from extracting ILP in the aforementioned low IPC applications.

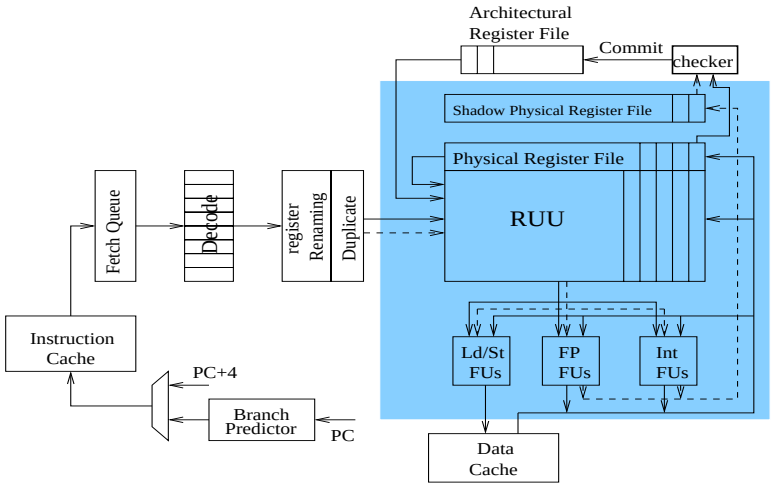

Fig. 5. DIE datapath implementing early retiring of redundant instructions (ERT)

Our third proposed optimization, Early Retiring Redundant Instructions (ERT), frees up the entry of an early issued redundant instruction (either the primary or the duplicate copy) right after its issue thus to reduce the pressure of dual redundant instructions on the instruction window. Observing the inherent redundancy of the redundant copies of an instruction, it is not necessary to maintain both copies in the instruction queue once one copy is issued for execution. Only one copy is sufficient for proper commit operation. Early retiring redundant instructions (ERT) reclaims the space in the instruction window immediately after a redundant copy is issued, and guarantees that the other copy will not be early retired. In such a way, ERT can virtually increase the instruction window size to accommodate more distinct instructions for ILP extraction.

The best case of ERT is that the instruction window is filled up with instructions only with one copy and all the redundant copies are early retired, which effectively recovers the original capacity of the instruction window. To support ERT, we add a shadow physical register file with the same size of the original one. More specifically, the original physical register file functions exactly the same as in SIE and is updated only by the primary instructions. The shadow physical register file is written only by the duplicate instructions and is only for the error-checking comparison purpose at commit stage. With asymmetric renaming, source operands are supplied by the original physical register file if needed. In ERT, the register renaming is further simplified since only the primary copy needs to perform register renaming and the duplicate instruction gets the exactly same renamings as the primary. However, the result physical register in the duplicate implicitly points to the one in the shadow register file. Figure 4 (e) shows the renamed code in DIE-ERT. The renamed result registers *p6'* and *p8'* are the shadow copy of p6 and p8. Figure 5 shows the DIE datapath supporting early retiring redundant instructions (ERT). When the instruction reaches the

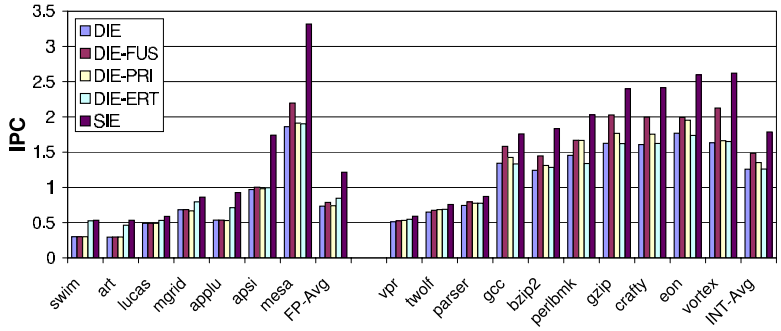

Fig. 6. Performance comparison: DIE, DIE-FUS, DIE-PRI, DIE-ERT, and SIE

commit stage, it compares its results from dual execution that are stored in the same location in the physical register file and its shadow copy before updating the architectural register file. With ERT, the renamed result register number in the remaining instruction copy may suffer from soft errors before commit. This problem can be simply solved by adding a small buffer to store the result physical register number when the redundant copy is early retired. The buffer is also indexed by the register number. Thus, both the results and result register numbers are compared to verify the absence of soft errors at commit stage. A more cost-effective solution is to add a parity bit for the result physical register number. Notice that only the result register number needs to be protected. Since the physical register number only uses 7 or 8 bits (for 128 or 256 physical registers), the cost and timing overhead of this parity logic should be minimal. Note that ERT still provides full protection within the sphere of replication.

5 Experimental Results

In this section, we evaluate the effectiveness of the proposed schemes, floating-point unit sharing (FUS), prioritizing the primary instructions (PRI), and early retiring of redundant instructions (ERT) for performance improvement in DIE.

Integer ALU contention is one of the major causes resulting in the severe performance loss of high IPC applications in DIE as compared to SIE. We propose to share idle floating-point ALUs for integer ALU operations thus to reduce the competition on integer ALUs, and more importantly to provide a virtually larger integer ALU bandwidth. The performance improvement delivered by FUS (DIE-FUS) is given in Figure 6. Applying FUS, DIE-FUS recovers the performance loss of DIE to SIE by up to 58% (*gcc*) in integer benchmarks with an average of 39%, and up to 23% (*mesa*) in floating-point benchmarks with an average of 4%.

By prioritizing the primary stream during instruction scheduling, DIE-PRI helps early resolve branch instructions in the primary stream thus reducing the number of instructions on the mispredicted path in the duplicate stream that

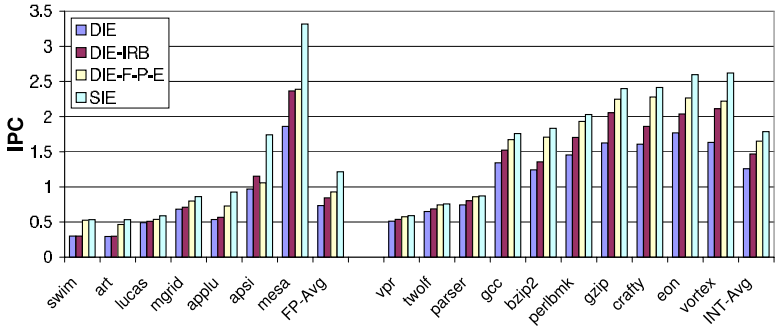

Fig. 7. Performance comparison: DIE, DIE-IRB, DIE-F-P-E, and SIE.

compete with the primary ones. Since most integer benchmarks are control intensive, DIE-PRI is expected to work well in these benchmarks, especially those with low branch prediction accuracy such as *perlbmk*. On the other hand, the ability of DIE-PRI to recover performance loss in DIE is quite limited for floating-point benchmarks as the control-flow instructions are much less frequent in those benchmarks. Figure 6 sees that DIE-PRI recovers the performance loss in DIE by up to 37% (*perlbmk*) in integer benchmarks with an average of 21%. DIE-PRI improves performance for all integer benchmarks. For floating-point benchmarks, DIE-PRI has a margin impact on the performance.

For applications heavily depending on a large instruction window to expose possible ILP, neither FUS nor PRI may work effectively since the performance bottleneck is not the functional unit bandwidth nor the issue width, rather is caused by the half-sized instruction window. This is especially true for many floating-point benchmarks such as *swim*, *art*, *lucas*, *mgrid*, and *applu*, whose performance can only be improved by enlarging the instruction window. Early retiring redundant instructions (ERT) virtually increases the instruction window size in DIE by immediately reclaiming the slot occupied by an early issued redundant instruction. As shown in Figure 6, DIE-ERT are very effective in boosting the performance of low IPC benchmarks. DIE-ERT reduces the performance loss of *swim* in DIE from 44% to 2% (a 96% performance recovery). On the average, DIE-ERT recovers 46% performance loss in DIE for floating-point benchmarks. For integer benchmarks, the average performance recovery delivered by DIE-ERT is around 9% with *vpr* up to 46%. However, if the benchmark suffers from frequent branch mispredictions, ERT might worsen the performance by fetching/dispatching more instructions along the wrong path that compete for resources against instructions earlier than the mispredicted branch instruction. This situation happened in *perlbmk*, instead of improving the performance, DIE-ERT causes an additional 20% performance degradation in DIE.

Since the three optimizations, FUS, PRI, and ERT, have very different effects on different types of benchmarks (e.g., floating-point or integer, high IPC or low IPC, high branch prediction accuracy or low accuracy), we propose to combine them together to accommodate various of benchmarks. We also compare this

new DIE-F-P-E processor (DIE augmented with FUS, PRI, and ERT) with a previous dual instruction execution instruction reuse buffer (DIE-IRB) processor [9]. The instruction reuse buffer (IRB) has the same configuration as in [9]. Our experimental results given in Figure 7 show that our combined scheme DIE-F-P-E outperforms DIE-IRB for all benchmarks except *apsi*. DIE-IRB works much better for benchmark *apsi* due to a considerable number of long latency floating-point multiplications hitting the IRB and getting the results directly from IRB. For benchmarks *swim*, *art*, *bzip2*, and *crafty*, DIE-F-P-E achieves significant performance improvement over DIE-IRB. On the average, DIE-F-P-E recovers DIE's performance loss by 78% for integer benchmarks and 54% for floating-point benchmarks, comparing to DIE-IRB's 39% recovery for integer and 15% for floating-point benchmarks, respectively. This brings us a performance loss of only 6% and 17%, an average for integer and floating-point benchmarks respectively, for providing transient error detecting dual-redundant execution in DIE-F-P-E superscalar processors (as compared to the significant performance loss, 26%/37% for integer/floating-point benchmarks in DIE).

6 Related Work

Fault detection and correction using modular redundancy has been employed as a common approach for building reliable systems. In [8], the results of the main processor and a watch-dog processor are compared to check for errors. A similar approach is presented in DIVA [2] except that the watch-dog processor is a low-performance checker processor. Cycle-by-cycle lockstepping of dual-processors and comparison of their outputs are employed in Compaq Himalaya [1] and IBM z900 [16] with G5 processors for error detection.

AR-SMT [12] and SRT [11] techniques execute two copies of the same program as two redundant threads on SMT architectures for fault detection. The slipstream processor extends the idea of AR-SMT to CMPs [18]. The SRTR work in [19] also supports recovery in addition to detection. [11] and [7] explore such redundant thread execution in terms of both single- and dual-processor simultaneous multithreaded single-chip processors.

Redundant instruction execution in superscalar processors is proposed in [6, 10] for detecting transient faults. In [6], each instruction is executed twice and the results from duplicate execution are compared to verify absence of transient errors in functional unit. However, each instruction only occupies a single re-order buffer (ROB) entry. On the other hand, the dual-instruction execution scheme (SS2) in [10] physically duplicates each decoded instruction to provide a *Sphere of Replication* including instruction issue queue/ROB, functional units, physical register file, and interconnect among them. In [9], instruction reuse technique is exploited to bypass the execution stage, and thus to improve the ALU bandwidth at the cost of introducing two comparators for source operand value comparison in the wakeup logic. A recent work [17] from the same CMU group has investigated the performance behavior of SS2 in details with respect to resources and staggered execution, and proposed SHREC microarchitecture, an

asymmetric instruction-level redundant execution, to effectively reduce resource competition between the primary and redundant streams. Our work shares many of the common issues studied in previous research efforts. However, our main focus here is to exploit and maximize the utilization of the existing resources to alleviate the competition pressure on a particular part in the datapath, thus to improve the performance in redundant-instruction execution superscalar processors.

7 Concluding Remarks

This paper has investigated the performance-limiting factors in a transient-error detecting redundant-execution superscalar processor. Based on our observation that that most floating-point ALUs were idle during execution, we proposed floating-point unit sharing (FUS) to utilize these idle FP ALUs for integer ALU operations. This reduces competition for the integer ALUs significantly and increases the total available integer operation bandwidth. To further steer these available resources for useful work (instructions along the correct path), we proposed to prioritize the primary stream for instruction scheduling in DIE. This helps restrain the duplicate instructions on the mispredicted path from competing for issue slots and functional unit bandwidth, thus leading to more efficient use of these resource and performance improvement. To address the effectively half-sized instruction window in DIE, we proposed an early-retiring scheme for redundant instructions (ERT) to increase the effective size of the instruction window. By combining these three proposals, our modified processor DIE-F-P-E is able to execute instruction-level (dual) redundant code for full coverage of soft errors in the sphere of replication while experiencing a performance penalty of only 6% for integer applications and 17% for floating-point applications. As such, it recovers 78% of the 26% performance penalty normally seen in the base DIE processor for integer applications, and recovers 54% of the 37% performance penalty seen in floating-point applications. This limited performance penalty and low hardware overhead allows redundant execution, and hence fault tolerance, to be implemented on a wide range of processors at a very low cost.

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. D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.
4. 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.

5. G. Hinton, D. Sager, M. Upton, D. Boggs, D. C. n, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technical Journal*, Q1 2001 Issue, Feb. 2001.
6. 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.
7. 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.
8. M. Namjoo and E. McCluskey. Watchdog processors and detection of malfunctions at the system level. Technical Report 81-17, CRC, December 1981.
9. 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.
10. 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.
11. 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.
12. 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.
13. S. S. Sastry, S. Palacharla, and J. E. Smith. Exploiting idle floating point resources for integer execution. In *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation*, pages 118–129, June 1998.
14. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
15. 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.
16. T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.
17. 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.
18. K. Sundaramoorthy, Z. Purser, and E. Rotenburg. 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.
19. 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.
20. 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.

This article was processed using the LaTeX macro package with LLNCS style