Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures

Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA 15213
{jsmolens, jangwook, jhoe, babak}@ece.cmu.edu
http://www.ece.cmu.edu/~truss

Abstract

Previous proposals for soft-error tolerance have called for redundantly executing a program as two concurrent threads on a superscalar microarchitecture. In a balanced superscalar design, the extra workload from redundant execution induces a severe performance penalty due to increased contention for resources throughout the datapath. This paper identifies and analyzes four key factors that affect the performance of redundant execution, namely 1) issue bandwidth and functional unit contention, 2) issue queue and reorder buffer capacity contention, 3) decode and retirement bandwidth contention, and 4) coupling between redundant threads' dynamic resource requirements. Based on this analysis, we propose the SHREC microarchitecture for asymmetric and staggered redundant execution. This microarchitecture addresses the four factors in an integrated design without requiring prohibitive additional hardware resources. In comparison to conventional single-threaded execution on a state-of-the-art superscalar microarchitecture with comparable cost, SHREC reduces the average performance penalty to within 4% on integer and 15% on floating-point SPEC2K benchmarks by sharing resources more efficiently between the redundant threads.

1. Introduction

A soft error occurs when the voltage level of a digital signal is temporarily disturbed by an unintended mechanism such as radiation, thermal, or electrical noise. Previously, soft errors have been a phenomenon associated with memory circuits. The diminishing capacitance and noise margin anticipated for future deep-submicron VLSI has raised concerns regarding the susceptibility of logic and flip-flop circuits. Recent studies have speculated that by the next decade, logic and flip-flop circuits could become as susceptible to soft errors as today's unprotected memory circuits [6,8,18]. In light of increasing soft-error concerns, recent research has studied the soft-error vulnerability of the microarchitecture as a whole [11]. A number of studies have investigated ways to protect the execution pipeline by means of concurrent error detection; this protection requires 1) instructions to be executed redundantly and independently and 2) the redundant outcomes to be compared for correctness. Among recent studies, a popular approach has been to leverage the inherent redundancy in today's high-performance superscalar pipelines to support concurrent error detection [10,13,14,15,16,20].

Performance Penalty. A minimally modified superscalar datapath can support concurrent error detection by executing an application as two independent, redundant threads. A drawback of this approach is that the doubled workload from redundant execution incurs a significant IPC penalty on superscalar datapaths tuned for single thread performance. Understandably, an application that achieves high utilization and high IPC as a single thread on a given superscalar datapath will saturate the available resources and lose performance when running as two redundant threads on the same datapath. *More interestingly, however, applications with low IPC as a single thread can still perform poorly as two redundant threads, despite the apparently ample resources to support both low-IPC threads.* In this paper, we explain the performance bottlenecks that contribute to this paradoxical behavior and offer a microarchitecture design that improves redundant execution performance in these cases.

Performance Factors. We study four factors in superscalar datapath design that affect the performance of redundant execution. The first three factors represent the different resources that support high performance execution. The factors are 1) issue and functional unit bandwidth, 2) issue queue (ISQ) and reorder buffer (ROB) capacity, and 3) decode and retirement bandwidth. Our analysis indicates that the bottleneck imposed by forcing two redundant threads to share issue and functional unit bandwidth has a first-order effect on performance. However, in the cases where issue bandwidth is not clearly saturated, sharing the ISQ and ROB between two threads—

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

effectively halving the out-of-order window size—also imposes a significant performance bottleneck, particularly in floating-point applications. Decode and retirement bandwidth may be a performance limiting factor for microarchitectures that fetch or decode instructions redundantly. The fourth factor in this study is a design choice of whether to allow the progress of two redundant threads to stagger elastically. Staggered re-execution has been previously shown effective [9,15,16,20]. Our analysis further shows that the effects from allowing stagger and relieving the ISQ and ROB contention are independent and can be combined to achieve better performance than either factor alone.

SHREC Microarchitecture. SHREC (SHared REsource Checker) is an efficient concurrent error detection microarchitecture that requires only limited modifications to a conventional superscalar microarchitecture. SHREC employs asymmetric redundant execution to relieve the sharing pressure on ISQ and ROB capacity without physically enlarging those structures. It is also naturally amenable to permitting an elastic stagger between redundant executions of the same instruction. Our evaluation shows that asymmetric redundant execution and stagger together enable SHREC to more efficiently utilize available issue bandwidth and functional units, thus reducing the performance impact of sharing between two redundant threads. Assuming a comparable state-of-the-art baseline superscalar microarchitecture, SHREC reduces the average performance penalty of redundant execution to within 4% on integer and 15% on floating-point applications. This performance is in comparison to prior proposals that experience an IPC loss of 15% on integer and 32% on floating-point applications.

Related Work. In asymmetric redundant execution, an instruction is only checked after the second execution using input operands already available from the first execution. This idea has been proposed previously by Austin in the DIVA microarchitecture [2]. In DIVA, the initial execution and the redundant instruction checking do not share issue and functional unit bandwidth; a second dedicated set of functional units is added to the checker. In this study, we assume the resource configuration of the underlying superscalar microarchitecture is fixed by technology and market forces and cannot be easily increased. Therefore, the initial execution and redundant checking in SHREC share the same set of functional units. An important issue addressed in our study is whether functional units can be shared efficiently.

Previously, Mendelson and Suri proposed the O3RS approach for redundant execution of register update unit-based (RUU) superscalar microarchitectures [10]. Redundant execution of the same instruction is initiated from the same RUU entry (from the same ISQ and ROB entries in the context of recent superscalar microarchitectures). This is another way to resolve the sharing pressure on ISQ and ROB capacity. However, the O3RS approach is not amenable to supporting stagger between redundant executions because the ISQ and ROB occupancy times would be necessarily increased (entries are held until the second execution). Without stagger, O3RS cannot fully capture the performance opportunity available to SHREC. Conversely, staggered execution has been shown to be effective in improving performance in several SMT-based concurrent error detection designs, but these proposals do not address the bottleneck imposed by sharing ISQ and ROB capacity. In Section 3, we present more detailed analysis and comparison of designs that only support staggered re-execution or only resolve the ISQ and ROB bottleneck.

Contributions. This paper offers two key contributions:

- We investigate and explain the performance behavior of redundant execution on superscalar microarchitectures.
- We propose the SHREC soft-error tolerant superscalar microarchitecture, which maximizes the performance of redundant execution with minimized changes to a conventional baseline superscalar microarchitecture.

Paper Outline. This paper is organized as follows. Section 2 provides background and defines assumptions regarding our baseline superscalar microarchitecture and redundant execution superscalar microarchitecture. Section 3 presents a factorial analysis of key design choices impacting the performance of redundant execution on superscalar microarchitectures. Section 4 describes and evaluates SHREC. We offer our conclusions in Section 5.

2. Baseline Microarchitectures

In this section, we first define our assumptions about the baseline superscalar microarchitecture. Next, we explain previous soft-error tolerance proposals that extend superscalar microarchitectures to redundantly execute a program as two concurrent threads. Finally, we establish the performance penalty associated with the prior proposals.

2.1. SS1: Baseline Superscalar Microarchitecture

The starting point of all soft-error tolerant designs in this paper is a balanced superscalar speculative out-of-order microarchitecture. In this paper, we assume a conventional wide-issue design where:

1. architectural register names are renamed onto a physical register file
2. an issue queue (ISQ) provides an out-of-order window for dataflow-ordered instruction scheduling to a mix of functional units (FUs)

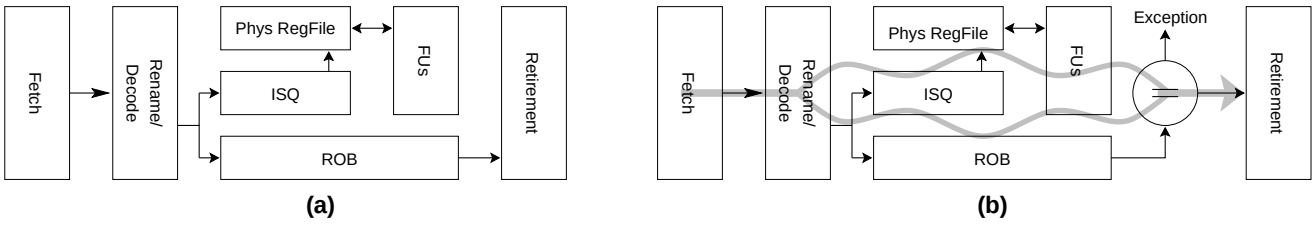


FIGURE 1. The (a) SS1 baseline superscalar and (b) SS2 redundant execution microarchitectures.

TABLE 1. Baseline SS1 parameters.

OoO Core	128-entry ISQ, 512-entry ROB, 64-entry LSQ, 8-wide decode, issue, and retirement
Memory System	64K 2-way L1 I/D caches, 64-byte line size, 3-cycle hit, unified 2M 4-way L2, 12-cycle hit, 200-cycle memory access, 32 8-target MSHRs, 4 memory ports
Functional Units (latency)	8 IALU (1), 2 IMUL/IDIV (3/19), 2 FALU (2), 2 FMUL/FDIV (4/12), all pipelined except IDIV and FDIV
Branch Predictor	Combining predictor with 64K gshare, 16K/64K 1st/2nd level PAs, 64K meta table, 2K 4-way BTB, 7 cycle branch misprediction recovery latency

3. a reorder buffer (ROB) tracks program order and execution status for all in-flight instructions
4. a load/store queue (LSQ) holds speculative memory updates and manages out-of-order memory operations

Figure 1(a) shows the layout of the baseline microarchitecture with the key datapath elements and their interconnections. Table 1 lists the salient parameters assumed for this paper. We refer to this reference microarchitecture configuration as SS1. SS1 nominally is an 8-way superscalar design. The settings in Table 1 are extrapolated from the Alpha EV8 [7].[1] Overall, SS1 should reflect an aggressive superscalar design that pushes today's state-of-the-art. Consequently, we assume the parameters in Table 1 cannot be increased further without substantial cost. For instance, the functional units in the EV8 occupy a substantial amount of die area, similar in size to 1MB of L2 cache storage [12].

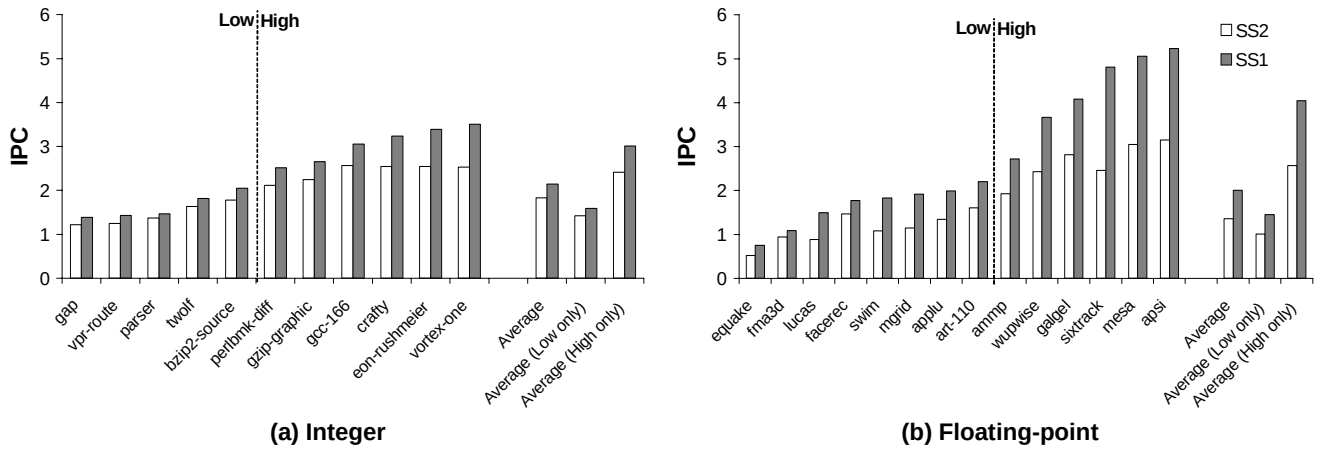
We assume all memory arrays in SS1 (such as register files, caches and TLBs) are protected against soft errors by error correcting codes (ECC). The storage overhead to protect a 64-bit value with single error correction, double error detection ECC is roughly 13%. We assume the instruction fetch stages are protected by circuit-level soft-error tolerance techniques; these in-order stages can be more deeply pipelined to overcome the associated performance overhead. We assume circuit-level techniques are also responsible for protecting non-performance critical portions of the datapath (e.g., memory management state machines). The main focus of this paper is on providing soft-error protection for the out-of-order pipeline. Neither ECC nor circuit-level techniques are applicable within the out-of-order pipeline due to the pipeline's structural complexity and extreme sensitivity to the critical path delay.

2.2. SS2: Symmetric Redundant Execution

Register renaming and dataflow-driven execution in an out-of-order pipeline enable fine-grain simultaneous execution of two independent instruction streams—this is the key premise behind the simultaneous multithreading approach [19]. Figure 1(b) illustrates a simple approach to execute a program redundantly as two data-independent threads for concurrent error detection [14]. The key changes from SS1 are concentrated in the decode and retirement stages. In Figure 1(b), the instruction fetch stream is duplicated into an M-thread (main thread) and an R-thread (redundant thread) at the decode stage. Register renaming allows the M- and R-threads to execute independently between the decode and the retirement stages. This portion of the pipeline operates identically to SS1, with the exception of memory instructions. Load and store addresses are calculated redundantly. The memory access required by a load instruction is only performed by the M-thread. The returned load value is buffered in a load-value queue [15] for later consumption by the corresponding load instruction in the R-thread. This structure ensures that two executions of the same load instruction result in the same load value. In the retirement stage, the redundantly computed results of the duplicated instructions are checked against each other in program order. If an error is detected, program execution can be restarted at the faulty instruction by triggering a soft-exception. We refer to this 2-way symmetric redundant superscalar pipeline as SS2.

Although SS2 is based on a superscalar datapath, the SS2 discussion in this paper is also applicable to SMT-based designs that fully re-execute a program as two data independent threads (e.g., AR-SMT [16], SRT [15], Slip-

1. Although EV8 is an SMT, it is emphasized that EV8 is tuned for single-threaded superscalar execution.



(a) Integer **(b) Floating-point**

FIGURE 2. Performance impact of redundant execution on SS2.

stream [13]). A key commonality between them is the resource pressure from concurrently supporting two threads. We assume SS2 has the same level of resources as SS1 (Table 1). For every instruction executed in SS1, not only must the instruction use functional units twice in SS2, but the instruction must also consume twice the bandwidth throughout the SS2 pipeline (i.e., decode, issue, writeback and retirement). Furthermore, each instruction also occupies two separate entries in SS2's ISQ and ROB, effectively halving the out-of-order execution window size.

2.3. SS2 Performance Penalty

Figure 2 reports the IPC of SPEC2K benchmarks from executing their corresponding SimPoints [17] on cycle-accurate performance simulators of SS1 and SS2. Throughout this paper, we report performance results based on 11 integer and 14 floating-point SPEC2K benchmarks[1] running their first reference input. To estimate IPC for each benchmark, we simulate an out-of-order core and measure IPC for up to eight 100-million-instruction SimPoint regions (as published in [17]) that are supposed to be representative of the benchmark's complete execution trace. We follow SimPoint's prescribed procedure to weight and combine the IPCs of individual SimPoint regions to estimate the entire application's average IPC. Average IPCs over multiple benchmarks are computed as harmonic means.

The performance simulators used in this paper are derived from the sim-outorder/Alpha performance simulator in Simplescalar 3.0 [4]. For this study, we modify sim-outorder's stock RUU-based model to support the issue queue-based SS1 datapath shown in Figure 1, MSHRs and memory bus contention. Unless otherwise noted, the configuration in Table 1 is used consistently in this paper.

Graphs (a) and (b) in Figure 2 separately report the IPCs of integer and floating-point benchmarks. In each graph, the benchmarks are sorted according to their SS1 IPC. The benchmarks are further divided into high-IPC versus low-IPC subsets; each graph is summarized in terms of the overall average IPC as well as separate averages of the high-IPC and low-IPC subsets. Overall, the SS2 IPC is 15% lower than SS1 for integer benchmarks and 32% lower for floating-point benchmarks. The difference in IPC is more significant for the high-IPC benchmarks. For benchmarks with high IPC on SS1, SS2 IPC suffers noticeably because the doubled workload of redundant execution saturates the already highly utilized 8-wide superscalar datapath. In this scenario, the SS2 performance penalty cannot be reduced without increasing the issue bandwidth and functional units.

It is interesting to note that significant performance loss is also experienced by SS2 for benchmarks with an SS1 IPC well below four—half the issue width. There are several reasons why this can happen. First, a low-IPC benchmark may nonetheless demand high utilization of the SS1 pipeline if a large number of mispredicted wrong-path instructions are part of the overall execution. Similarly, a low IPC benchmark may require the full capacity of the ISQ and ROB to extract available instruction level parallelism (ILP). Several of the low- and moderate-IPC floating-point benchmarks also experience significant floating-point functional unit contention. A benchmark with a low average IPC can include short program phases with high IPC; SS2 bottlenecks during these instantaneously high IPC regions can further lower the average IPC. In all of these scenarios, it is possible to reduce the performance penalty of re-execution through more efficient scheduling

1. We exclude the integer benchmark *mcf* from our study because it is insensitive to all design parameters in this paper; its IPC is within 1% of 0.17 for all experiments.

and management of re-execution. In the next section, we present an analysis to help clarify the performance behavior of SS2 and the effects of the different resource bottlenecks.

3. Factorial Design Analysis

The performance difference between SS2 and SS1 is not caused by any single bottleneck. The contributing bottlenecks vary from benchmark to benchmark, and even between different phases of the same benchmark. The analysis in this section establishes the importance of individual bottlenecks in SS2, as well as their interactions.

3.1. SS2 Performance Factors

We reduce the factors contributing to performance bottlenecks in SS2 into three resource-related factors and one mechanism-related factor:

- X: issue width and functional unit bandwidth
- C: ISQ and ROB capacity
- B: decode and retirement bandwidth
- S: staggering redundant threads (mechanism related)

X-factor: The X-factor represents the issue and functional unit bandwidth, a limit on raw/peak instruction execution throughput of a superscalar pipeline. We combine issue and functional unit bandwidth into a single factor because they must be scaled together in a balanced design. A fundamental property of the X-factor is that if an application on average consumes more than one-half of the issue and functional unit bandwidth in SS1, it is impossible for the same application to achieve the same level of performance in SS2. However, what happens to an application that on average consumes less than one half of the issue and functional unit bandwidth in SS1 is not immediately apparent, since whether the available issue and functional unit bandwidth can be used effectively in SS2 depends on the other factors.

C-factor: The C-factor represents the ISQ and ROB capacity. The C-factor governs a pipeline's ability to exploit instruction-level parallelism by executing instructions out-of-order. Again, we combine the ISQ and ROB capacity into a single factor because they are scaled together in commercial designs.[1] In SS2, the out-of-order window size is effectively halved. Applications that depend on a large out-of-order window to expose ILP lose performance on SS2, independent of the X-factor. On the other hand, applications that are insensitive to the out-of-order window size perform predictably on SS2, as governed by the X-factor.

B-factor: The B-factor represents the various bandwidth limits at the interface to the ISQ and ROB. These include the decode bandwidth, the completion bandwidth and the retirement bandwidth. In SS2, these bandwidth limits impose a bottleneck because they must be shared between the redundant M- and R-threads. We expect the B-factor to have the least observed impact on SS2 performance, because in a balanced design the X and C factors are generally tuned to become limiting factors before saturating the other bandwidth limits.

S-factor: Besides the three resource-related factors, we introduce a fourth factor, stagger (S-factor), that improves the performance of SS2 but is not applicable to SS1. Several previous proposals for redundant execution on SMT advocate maintaining a small stagger (128~256 instructions) between the progress of the leading M-thread and the trailing R-thread [15,16]. In SMT-based designs, the stagger serves to hide cache-miss latencies from the R-thread and to allow the M-thread to provide branch prediction information to the trailing thread. Allowing some elasticity between the progress of the two threads also allows better resource scheduling. By giving static scheduling priority to the M-thread, the resulting effect is that the R-thread makes use of the slack resources left over by the M-thread rather than competing with the M-thread for resources. The reported benefit of allowing stagger is typically around a 10% increase in the IPC of redundant execution [15].

3.2. SS2 Design Space

The X, C, and B factors are potential bottlenecks that cause SS2 to lose performance relative to SS1. The bottleneck associated with a factor is removed if the resources associated with that factor are doubled from the assumed configuration in Table 1. To understand the different contributing factors to SS2's performance, we evaluate sixteen SS2 configurations, based on enumerating all possible combinations of settings for X, S, C and B. In each configuration, the factors X, C, and B are either kept the same as Table 1 or doubled and an elastic stagger of up to 256 instructions is either enabled or disabled.

The resulting IPC improvements relative to SS2 for the sixteen possible configurations are summarized in Table 2. We order the factors in columns 1 through 4 by their overall impact on performance. In each configuration, we report the percentage improvement relative to SS2 for the integer and floating-point SPEC2K benchmarks and separately for the high and low-IPC integer and floating-point benchmarks.

1. A recent study has suggested, however, that there may be advantages to scaling the ROB capacity disproportionally [1].

TABLE 2. Percentage increase in IPC relative to SS2 for all combinations of factors. The notes column identifies mappings to designs referenced in this paper.

Factor				Integer % Increase			Floating-point % Increase			Notes
X	S	C	B	All	High	Low	All	High	Low	
-	-	-	-	0	0	0	0	0	0	SS2
-	-	-	B	3	4	2	0	1	0	
-	-	C	-	3	1	4	28	9	34	
-	-	C	B	5	4	6	28	11	35	SS2+C+B ≈ O3RS [10]
-	S	-	-	9	13	6	11	9	12	SS2+S ≈ SRT [15]
-	S	-	B	10	15	7	11	10	11	
-	S	C	-	12	14	11	33	15	40	
-	S	C	B	13	16	11	34	16	40	SS2+S+C+B ≈ SHREC
X	-	-	-	15	25	9	12	36	6	
X	-	-	B	15	26	9	12	38	6	
X	-	C	-	19	28	14	46	58	43	SS2+X+C ≈ SS1 ≈ DIVA [2]
X	-	C	B	20	30	14	48	61	44	
X	S	-	-	16	27	9	19	46	13	
X	S	-	B	17	29	10	21	51	15	
X	S	C	-	20	30	14	49	62	45	
X	S	C	B	22	33	15	52	71	47	

Several general trends can be observed from Table 2. First, a comparison of the upper and lower halves of the table shows that the X-factor has a large performance impact on all benchmark classes. Comparing the neighboring quarters of the table shows that stagger also improves the performance of all benchmarks. Only the floating-point benchmarks are highly sensitive to the C-factor. Finally, from comparing neighboring rows, the B-factor has a minimal effect on all benchmarks.

3.3. 2-k Factorial Analysis

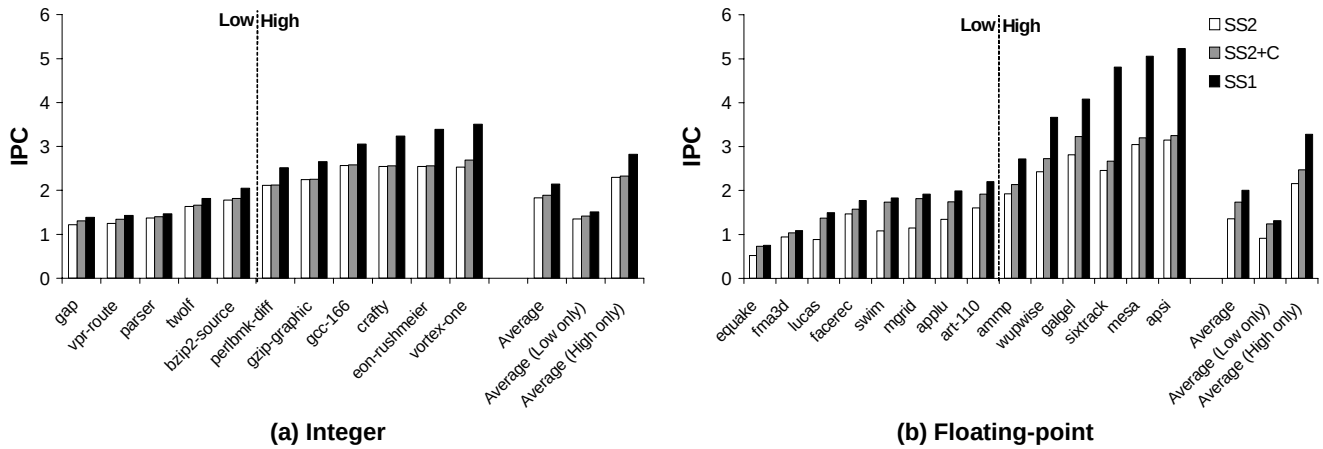
To examine the SS2 design space more systematically, we apply 2-k factorial analysis to the CPI[1] of the sixteen configurations to disentangle the performance impact of each factor and the impact from interactions between multiple factors. (Refer to [3] for a detailed explanation of 2-k factorial analysis.) The result of the 2-k factorial analysis is a quantitative measure of the average effect of changing each of the four factors individually. In addition, factorial analysis also gives the effect due to interactions between changing any two factors, three factors, or all four factors together. Table 3 summarizes the significant effects (> 3%) for high and low-IPC integer and floating-point benchmarks. For rows showing an individual factor, the value under the "Effect" column signifies the average percentage CPI decrease (i.e., performance increase) between all configurations with the factor enabled versus all configurations with the factor disabled.

TABLE 3. The main factors and interactions that increase performance in redundant execution.

	Factor	Effect
Integer: High	X	17.1
	S	7.1
	X+S	-5.2
Integer: Low	X	5.2
	C	4.3
	S	3.1
Floating-point: High	X	33.5
	C	9.9
	S	6.4
Floating-point: Low	C	27.0
	S	6.2
	X	4.6
	S+C	-3.9

For example, the significant factors to increase performance on high-IPC floating-point benchmarks are X, C and S. The interactions between the three factors (not shown) are insignificant. Their effects sum nearly linearly

1. This is because CPI is additive when normalized with respect to the instruction count, while IPC is not.



(a) Integer **(b) Floating-point**

FIGURE 3. The effect of relieving ISQ and ROB contention in SS2 (C-factor).

when more than one factor is enabled. In another example, the two most significant factors for high-IPC integer benchmarks are X and S. In this case, X and S have a significant and negative interaction. In other words, the effect of implementing both X and S is that the added performance improvement will be less than the simple sum of the two effects.

Although the X-factor has significant effects for all classes of benchmarks, we do not explore the X-factor further, since we cannot increase the issue bandwidth or functional units. We will show, however, that the S-factor improves performance by helping alleviate the issue bandwidth and functional unit pressure. One should note that from Table 2, doubling the issue bandwidth and functional units alone (SS2+X) cannot improve SS2's performance to SS1 levels. We also do not discuss the B-factor since it is not a significant factor in any benchmark class. We concentrate on the C and S factors next.

3.4. Capacity of ISQ and ROB

The 2-k factorial analysis suggests that the capacity of the ISQ and ROB (C-factor) in SS2 can have a major impact on the performance of low-IPC floating-point benchmarks. Although one cannot physically double the capacity of the ISQ and ROB due to complexity and area constraints, microarchitectural techniques exist to achieve the same effect in the context of redundant execution.

Mendelson and Suri proposed the O3RS approach for redundant execution in a superscalar microarchitecture [10]. A unique feature of O3RS is that an instruction occupies just one entry of the ISQ and ROB after decode.[1] An instruction is issued twice from the same ISQ entry before it is removed; similarly, one ROB entry keeps track of the two redundant executions of the same instruction. In other words, given the resources in Table 1, O3RS would perform similarly to SS2 with twice the ISQ and ROB capacity. Figure 3 compares the IPC of SPEC2K benchmarks on SS1, SS2 and SS2 with twice the ISQ and ROB capacity (SS2+C). As suggested by the factorial analysis, the IPCs of SS2 and SS2+C are comparable on integer benchmarks, and SS2+C shows a 28% improvement over SS2 for floating-point benchmarks.[2] The previous 32% floating-point performance gap between SS2 and SS1 has been reduced to 25% by the C-factor.

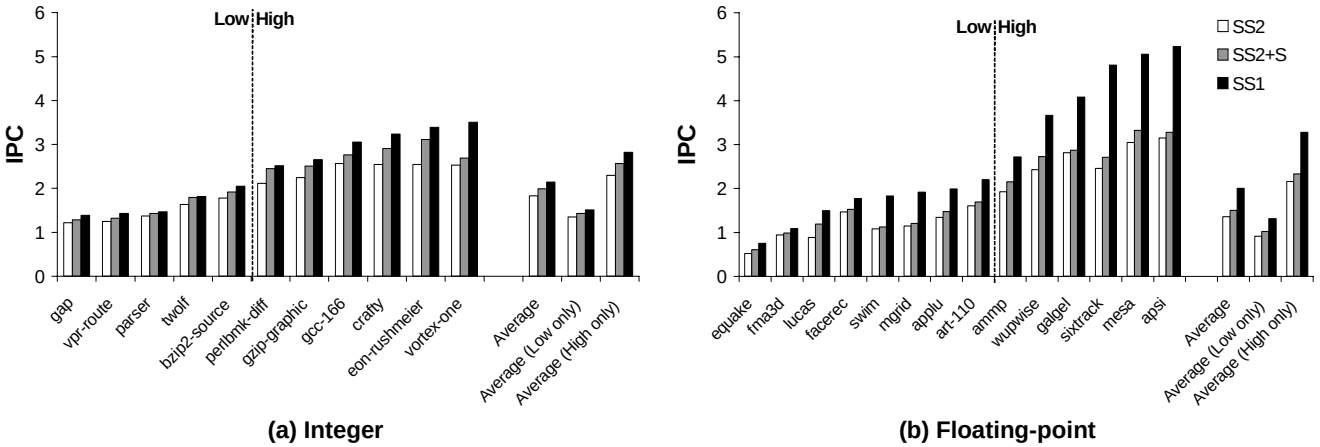
A limitation of the O3RS approach is that it cannot be easily modified to support stagger. In O3RS, redundant instructions should issue from the ISQ in rapid succession to not increase the effective occupancy of the ISQ. Increasing this interval to support stagger has the effect of reducing the out-of-order window, offsetting O3RS's original benefit from the C-factor.

3.5. Staggered Re-execution

Figure 4 compares the IPC of SPEC2K benchmarks on SS1, SS2, and SS2 with stagger (SS2+S). In accordance with previous stagger studies, SS2+S results in a noticeable IPC improvement over SS2, 9% on integer and 11% on floating-point benchmarks. However, notice in Table 2 that also relieving the C-factor (i.e., SS2+S+C) can yield another 3% improvement on integer benchmarks and 22% on floating-point benchmarks beyond SS2+S.

1. The original O3RS is based on an RUU. Here, we describe the spirit of the proposal in the context of an ISQ and ROB.

2. It is important to note that the effect given by the factorial analysis is an average over all settings of other factors; the actual change in performance from changing one variable may be higher or lower, depending on the state of the other factors.



(a) Integer **(b) Floating-point**

FIGURE 4. The effect of allowing a stagger of 256 instructions in SS2 (S-factor).

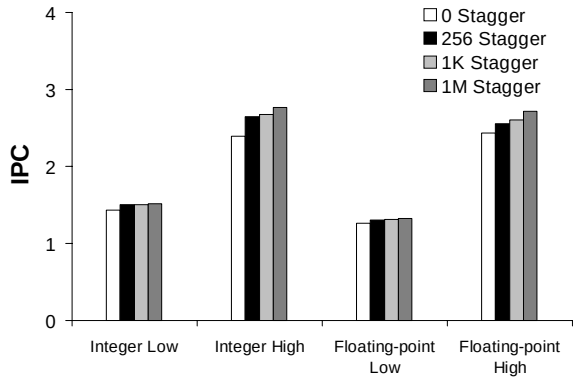


FIGURE 5. Applications are not sensitive to degrees of stagger over 256 instructions.

We conclude this section with an analysis of the impact of varying the degree of elastic stagger. The allowed degree of stagger impacts performance because it limits the granularity of IPC variation that can be effectively hidden. For example, if the maximum degree of stagger is longer than an L2 miss latency, it is likely that the R-thread can continue to make forward progress on backlogged instructions while the M-thread is stalled on a L2 miss. However, if the maximum degree of stagger is much shorter than an L2 miss latency, the R-thread will also stall due to data dependencies on the same L2 miss.

We evaluate SS2+S+C over different maximum staggers of 0, 256, 1K, and 1M instructions. The trends of IPC versus degree of stagger are reported in Figure 5 for the high/low-IPC integer and floating-point benchmarks. Figure 5 shows that, with the exception of high-IPC floating-point benchmarks, a moderate stagger of up to 256 instructions is sufficient to capture nearly all the benefits. Low-IPC applications, in particular, have the potential to be latency-bound. With an IPC of 1.5, an application would need a stagger of up to 300 instructions to completely overlap a 200-cycle main memory access. Enabling 1K or 1M stagger—longer than the longest system latency—has little additional benefit. Very large stagger could be effective if a program exhibited IPC variations across large program phases, but the associated hardware cost is indefensible for the incremental performance benefit.

4. SHREC

In this section, we present the details of SHREC and evaluate its performance. The SHREC design reduces the performance penalty of redundant execution through more efficient scheduling and management. First, a key feature of the SHREC design is that R-thread instructions are processed differently from M-thread instructions. The R-thread instructions are only checked using input operands produced by the M-thread. This asymmetric handling of re-execution restores the full capacity at the ISQ and ROB to the M-thread. R-thread instruction checking is efficiently handled by a separate, simple in-order scheduler. Thus, bottlenecks associated with the ISQ and ROB are avoided without physically enlarging those structures. Second, the progress of the threads can stagger elastically, up to the size of the ROB. This added scheduling flexibility allows the R-thread to fill in the slack issue bandwidth and functional units left idle by the M-thread, rather than constantly competing with the M-thread. Delaying R-thread execution from the corresponding M-thread instruction also hides both cache miss latency and branch misprediction latency from the corresponding R-thread instruction. Thus, the issue bandwidth and functional unit bottlenecks can be ameliorated by means of more efficient utilization, rather than providing additional issue bandwidth or functional units.

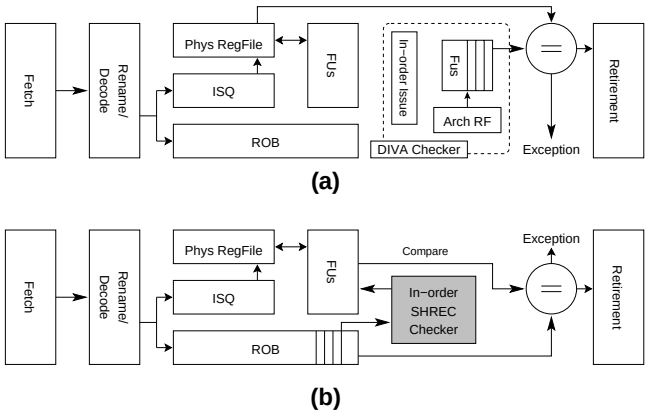


FIGURE 6. (a) DIVA and (b) SHREC microarchitectures.

4.1. Asymmetric Re-execution on Dedicated FUs

To aid our discussion of SHREC, we first review DIVA, a prior proposal for asymmetric re-execution [2]. The previously proposed DIVA microarchitecture augments a conventional superscalar pipeline (e.g., SS1) with a highly abbreviated checker pipeline [2, 5, 21]. Figure 6(a) depicts the datapath of the DIVA microarchitecture. In DIVA, before the completed instructions in the main out-of-order pipeline can retire, the instructions must first proceed in program-order through the checker pipeline, accompanied by their previously computed results. The checker pipeline re-computes each instruction individually; a mismatch between the original result and the re-computed result triggers an exception in the main out-of-order pipeline.

To minimize the throughput mismatch, the DIVA checker pipeline is assumed to have the same issue bandwidth and functional unit mix as the main out-of-order pipeline. Since the instructions in the checker pipeline are not constrained by data dependencies, the checker issue logic is in-order and the functional units can be deeply pipelined. In other words, the checker pipeline, although greatly simplified, can match the throughput of the main out-of-order pipeline.

DIVA's performance should closely follow SS1, since DIVA's main out-of-order pipeline supports only the initial execution of an instruction stream on essentially the same hardware as SS1. However, DIVA's good performance is, to a large degree, also bolstered by a second set of functional units at a significant cost over SS1, as discussed in Section 2.1.

4.2. Asymmetric Re-execution on Shared FUs

As illustrated in Figure 6(b), a SHREC pipeline is comprised of a primary out-of-order pipeline (essentially SS1) and an in-order checker pipeline (shaded in gray). The out-of-order pipeline and the checker pipeline in SHREC share access to a common pool of functional units. In other words, SHREC adopts DIVA's asymmetric execution scheme to relieve capacity and bandwidth contention at the ISQ and ROB, but shares functional units to reduce the hardware cost. This design point is important because soft-error tolerance may not be needed under all circumstances (e.g., Doom vs. TurboTax). In DIVA, soft-error tolerance comes at a fixed hardware overhead. SHREC's soft-error tolerance comes with a performance overhead but can be disabled to regain performance in non-critical applications.

The SHREC pipeline is similar to the DIVA pipeline except in the design surrounding the functional units and issue bandwidth of the checker pipeline. Based on the assumed SS1 configuration in Table 1, the derived SHREC microarchitecture can issue only up to eight instructions per cycle to the original set of functional units, whether from the out-of-order or the checker pipelines. The out-of-order pipeline has static priority for the instruction issue bandwidth and functional unit selection; only unused issue bandwidth and functional units are available to the in-order issue window of the checker pipeline.

Naturally, the out-of-order issue logic in SHREC is more complicated than in SS1. However, by 1) keeping the number of issue slots the same, 2) assigning fixed priority to the out-of-order pipeline, and 3) issuing from only a small in-order window of instructions from the checker pipeline, the increase in complexity can be kept to a minimum. SS1 originally supports a 128-entry ISQ. The equivalent SHREC checker pipeline is allowed an eight-entry in-order issue window, and the SHREC ISQ is reduced commensurately to 120 entries. Thus, the total number of entries feeding into the issue selection logic remains 128, as in SS1. No additional ports are added to the out-of-order issue queue, since the in-order issue window is a logically and physically separate structure.

4.3. Microarchitecture Operation

SHREC's operation is similar to DIVA and can leverage many of the same optimizations proposed for DIVA [5, 21]. Below, we describe the basic operations of the SHREC pipeline.

Out-of-order Pipeline: The main SHREC pipeline is a conventional superscalar out-of-order design. An instruction stream proceeds through the out-of-order pipeline normally, except completed instructions must be re-executed

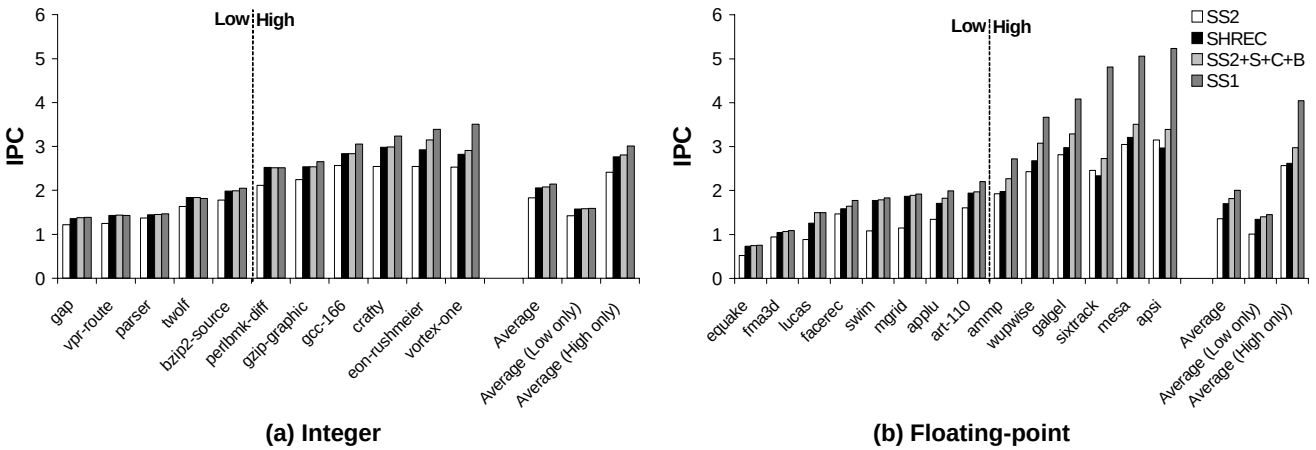


FIGURE 7. The IPCs of SS2, SHREC, and SS1. SS2+S+C+B models an idealized SHREC model.

by the checker pipeline before they can retire from the ROB. The main pipeline needs to be augmented by a result buffer that records the result of each instruction. These values are later re-executed by the checker. This result buffer operates in tandem with the ROB, but has its own read and write ports, thus it does not increase ROB complexity or port requirements.

Checker Pipeline: As in DIVA, we assume information about an instruction's initial execution in the out-of-order pipeline is conveyed to the checker pipeline without error via the result buffer (protected by ECC and circuit-level techniques). The checker pipeline considers at most eight consecutive completed instructions in the ROB for re-execution in program order, subject to the availability of surplus issue slots and functional units not scheduled by the main pipeline. When issued, the checker fetches operands from the same physical registers that provided operands to the initial execution. Since the total number of instructions issued per cycle is never more than in SS1, additional register file ports are not needed.

Verification and Retirement: Following re-execution, instructions are considered for program-order retirement. The checker-computed result is immediately compared against the original result computed by the out-of-order pipeline. If the values agree, the instruction retires from the ROB. Otherwise, a soft-exception restarts execution at the corrupted instruction.

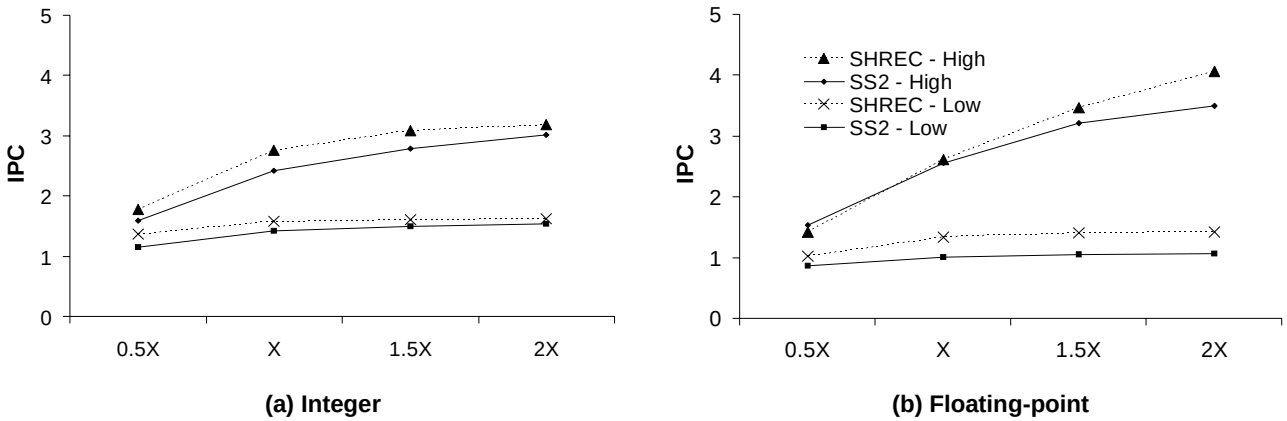
4.4. SHREC Performance

Figure 7 compares the IPC of SPEC2K benchmarks on SS2, SHREC, SS2+S+C+B, and SS1 models. Overall, SHREC reduces the performance penalty relative to SS1 to 4% on integer and 15% on floating-point benchmarks. This penalty is in comparison to SS2's performance penalty of 15% on integer and 32% on floating-point benchmarks. The SHREC result generally agrees with the performance of the idealized SS2+S+C+B model (included in Figure 7 for comparison). The small difference between SHREC and SS2+S+C+B is caused by SHREC's in-order issue restriction on the R-thread; SS2+S+C+B still supports out-of-order issue for the R-thread.

SHREC is especially effective in reducing the performance penalty for benchmarks with low to moderate IPC on SS1. SHREC matches the SS1 IPC for low-IPC integer benchmarks and comes to within 8% for low-IPC floating-point benchmarks. In this scenario, SHREC's main out-of-order pipeline operates identically to SS1 because there is ample issue bandwidth and functional units to support equal progress by both the main pipeline and the checker pipeline. The same cannot be claimed for SS2 because, even with ample issue bandwidth and functional units, the contention for other resources still reduces IPC.

On high-IPC benchmarks, SHREC loses performance relative to SS1 because SHREC can only sustain half of the peak performance of SS1 (when the issue bandwidth and/or functional units are saturated). In these scenarios, exemplified by the high-IPC floating-point benchmarks, SHREC and SS2 perform comparably since both suffer from the same bottlenecks in issue bandwidth and functional units. In *apsi*, *sixtrack*, and *art*, SHREC is even slightly slower than SS2. This slowdown is attributed to the SHREC checker pipeline's in-order issue, which introduces more stalls than SS2 when there is contention for the floating-point units.

In many benchmarks, performance improvement opportunities also exist when the program exerts fluctuating pressure on issue slots and functional units. IPC fluctuation can be caused by stall events such as cache misses, pipeline refills after branch mispredictions, long latency operations, or inherent IPC variations. During high-IPC cycles, the



(a) Integer **(b) Floating-point**

FIGURE 8. Over a range of functional unit mix sizes, SHREC uses the supplied units more efficiently than SS2.

SHREC out-of-order pipeline momentarily keeps pace with SS1, while the checker pipeline is starved from issue. If the out-of-order pipeline has sustained high-IPC (i.e., in high-IPC benchmarks) the out-of-order pipeline will eventually stall because the ROB backs up with instructions awaiting checking. Ideally, high-IPC program phases are interspersed with sufficiently frequent low-IPC phases such that the checker pipeline catches up. In these fluctuating IPC scenarios, SHREC will be more efficient than SS2.

Although SHREC does not increase issue bandwidth or the functional unit count, it reduces the pressure of sharing issue bandwidth and functional units through more efficient utilization. Here, we present additional data to show that SHREC's utilization efficiency is scalable when additional issue bandwidth and functional units are allowed. Figure 8 plots IPC versus the X-factor (available issue bandwidth and functional units) for high/low integer and floating-point benchmarks. '2X' means twice the issue bandwidth and functional units as specified in Table 1. Between 0.5X and 2X, SHREC matches or outperforms SS2, except under high contention for floating-point units. The integer benchmarks operate in a mode where they become ILP-bound with additional functional units. At that point, the benefit of SHREC's efficient scheduling diminishes; the IPC of SS2 and SHREC converge for both high and low-IPC integer benchmarks. The floating-point benchmarks tend to have high ILP, but experience functional unit bottlenecks. As the number of functional units increases, this bottleneck diminishes and the effect of more efficient SHREC scheduling increases. The performance of SHREC and SS2 should eventually converge, however, as even more functional units are introduced.

5. Conclusion

Recent proposals for soft-error tolerance in superscalar microarchitectures suffer a significant performance loss by requiring the instruction stream to be executed redundantly. In a balanced superscalar design, supporting a second redundant thread incurs a severe performance loss due to contention for bandwidth and capacity in the datapath, (e.g., issue bandwidth and the number of functional units, issue queue, reorder buffer size, and decode/retirement bandwidth). Relaxing any single resource bottleneck cannot fully restore the lost performance.

The proposed SHREC microarchitecture effectively relieves all of the above bottlenecks, except issue bandwidth and the number of functional units. In SHREC, the main thread executes normally, as on a conventional superscalar datapath. The redundant thread's instructions, however, are only checked using input operands already computed by the main thread. The redundant thread does not compete with the main thread for out-of-order scheduling resources because checking can be efficiently managed by a separate, simple in-order scheduler. Furthermore, the progress of the main and redundant threads can stagger elastically up to the size of the reorder buffer. This added scheduling flexibility allows better utilization of issue bandwidth and functional units, reducing the performance pressure from sharing. The above techniques enable SHREC-style re-execution to share resources more efficiently and to achieve higher IPC over previous shared-resource concurrent re-execution proposals.

Acknowledgements

We would like to thank the anonymous reviewers and members of the Impetus research group at Carnegie Mellon for their helpful feedback on earlier drafts of this paper. This work was supported by NSF CAREER award CFF-037568, NSF award ACI-0325802, and by Intel Corporation. Computers used in this research were provided by an equipment grant from Intel Corporation.

References

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[3] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley and Sons, Inc., 1978.

[4] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[5] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient checker processor design. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 87–97, December 2000.

[6] Neil Cohen, T.S. Sriram, Norm Leland, David Moyer, Steve Butler, and Robert Flatley. Soft error considerations for deep-submicron CMOS circuit applications. In *IEEE International Electron Devices Meeting: Technical Digest*, pages 315–318, December 1999.

[7] Joel S. Emer. EV8: the post-ultimate alpha. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001. Keynote address.

[8] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron CMOS logic circuits. *IEEE Journal of Solid State Circuits*, 30(7):830–834, July 1995.

[9] Shih-Chang Lai, Shih-Lien Lu, Konrad Lai, and Jih-Kwon Peir. Ditto processor. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[10] Avi Mendelson and Neeraj Suri. Designing high-performance and reliable superscalar architectures: The Out of Order Reliable Superscalar O3RS approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[11] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[12] Ronald P. Preston, et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, February 2002.

[13] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

[14] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[15] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[16] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[17] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[18] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[19] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[20] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[21] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. In *IEEE International Conference on Dependable Systems and Networks*, July 2001.