Optimal Resource Allocation for Concurrent Error Detection Techniques in High Performance Microprocessors

Sumeet Kumar
ECE Department
Binghamton University
Binghamton, NY 13902
skumar1@binghamton.edu

Aneesh Aggarwal
ECE Department
Binghamton University
Binghamton, NY 13902
aneesh@binghamton.edu

Abstract

With reducing feature size, increasing chip capacity, and increasing clock speed, microprocessors are becoming increasingly susceptible to transient (soft) errors. Ensuring reliability in microprocessors typically involves executing replicated threads for concurrent error detection. Redundant threads significantly increase the pressure on the processor resources, resulting in dramatic performance impact.

In this paper, we propose a technique to reduce the pressure on the resources when executing redundant threads for concurrent error detection. The technique — register bits reuse — attempts to use the same register (but different bits) for both the copies of the same instruction, if the result produced by the instruction is of small size. This may result in the Re-order buffer (ROB) entry for both the instructions to be exactly the same, which can be exploited to avoid allocation of separate ROB entries to the two instructions. We also propose novel ways of defining a small-sized value to increase their percentage out of the total values produced. The technique is based on two very important observations: (i) many of the values produced in a processor are of small size, and (ii) if the redundant thread is running behind the main thread (which has been shown to be more effective than running both the threads in tandem) by a few instructions, then the leading instructions usually produce their results before their trailing counterparts are renamed. Our experiments showed that we obtain about 64.4% performance improvement and about 18% energy reduction, over the base case with one redundant thread for concurrent error detection.

1 Introduction

With the current trends in transistor size, voltage and clock frequency, microprocessors are becoming increasingly susceptible to hardware failures. Hardware errors in the current technology are predominantly transient errors [5, 18] that occur randomly due to various reasons such as electromagnetic influences, alpha particle radiations, power supply fluctuations due to ground bounce, crosstalk or glitches, and partially defective components and loose connections. Current trends suggest that transient errors will be an increasing burden for microprocessor designers [23, 12]. Transient hardware errors are troublesome because they elude most of the current testing methods. A popular approach to detect transient errors is to use redundant threads [15, 7, 5, 19, 1, 14, 16, 17, 21, 22, 8]. In these techniques, the same application is run multiple times and the errors are detected by corroborating the redundant results. Studies [22, 8] have shown that a *staggered* execution can result in better performance, because the trailing thread may not incur many of the branch misprediction and the load miss penalties incurred by the leading thread.

Running multiple threads places a significant pressure on the processor resources, resulting in a considerable performance loss (almost 62%). Hence, it is important to investigate techniques that can reduce the pressure on the resources, and improve performance. In this paper, we investigate clever allocation techniques to optimally allocate resources to the trailing instructions. These techniques are very well suited particularly for a *staggered* exectuion, because of 2 very important observations: (i) many of the values produced in a processor are of small size, and (ii) the leading instructions usually produce their results before their trailing counterparts are renamed. Hence, we can exploit the the leading instruction's profile for optimal resource allocation.

In this paper, we investigate a technique in this spirit. The technique — *register bits reuse (RBR)* — exploits the result sizes for optimal resource allocation.

If the value produced by a leading instruction is narrow, the renamer allocates the same register (as used by the leading instruction) to the trailing counterpart. In this technique, the lower bits of the register hold the leading instruction's result, and the higher bits of the register hold the trailing instruction's result. We also discuss novel ways of defining a narrow width value, which result in a significant number of narrow values (even floating-point values). This technique can also facilitate avoiding ROB entries for the trailing instructions. We also discuss detailed implementations of this technique. We investigate novel techniques to reduce the load store buffer (LSB) pressure as well. Overall, the technique in this paper give an average performance improvement of about 64.4% and an average power reduction of about 18%.

The rest of the paper is organized as follows. Section 2 discusses the background and provides the motivation for our techniques. Section 3 discusses the details of our technique. Section 4 presents the experimental results and analysis. Section 5 presents sensitivity study. Section 6 presents related work. Finally, in Section 7, we conclude.

2 Background and Motivation

2.1 Background

In this paper, we consider a reliable microprocessor configuration running one redundant thread for concurrent error detection, shown in Figure 1. The threads are fetched independent of each other, using multiple PCs. One thread is always ahead of the other by a few instructions (*staggered* execution) [22]. However, instructions record a bit indicating whether they are leading or trailing instructions. The threads are decoded and renamed concurrently. In the rename stage, different map tables are used for the two threads. Once renamed, the instructions are dispatched to the issue queue, ROB, and load/store buffer (for load and store instructions). The threads are then executed concurrently. The results of the multiple copies of the same instruction, from the two threads, are compared for error detection when the instructions commit. Since, we use a unified register file, at commit, the instructions also update the backend map-tables. To reduce the register file port requirements for error detection, we use an *additional value buffer*, which also stores the results stored in the register file and from which the values are read for comparison at commit time [8].

The entries for the instructions of the two threads are fixed in the ROB and the LSB, to facilitate finding multiple copies of the same instruction at the time of commit. In our processor, only the leading load and store instructions access the memory, which is assumed to be transient fault tolerant (by using Error Detection and Correction Codes). The value loaded by the leading load instruction is also forwarded to the register allocated to the trailing load instructions, using a separate buffer to store the loaded values [8]. However, the addresses of the loads and the stores (and the value to be stored for the stores) are generated multiple times, and checked during commit for any errors.

Deadlocks are avoided by keeping counters that count the number of trailing instructions in the pipeline. If any resource is full without any trailing instructions, then a potential deadlock is avoided by squashing the younger leading instructions and fetching the trailing ones, irrespective of the *slack* condition. The *slack* between the threads depends on the size of the various buffers (such as ROB, LSB, RF, etc.) provided in the processor. We measured the IPCs with *slacks* of 32, 64, 96, and 128 instructions. We found that the IPC improves considerably when going from a *slack* of 32 to a *slack* of 64 instructions, and either saturates at a *slack* of 64 instructions or reduces for a *slack* of 96 or 128 instructions. For the rest of the paper, we choose an instruction *slack* of 64 instructions between the threads.

2.2 Motivation

Performance degrades because the resources (such as ROB, issue queue, LSB, and register file entries, and dispatch/issue/commit slots) are shared among the threads. To motivate the resource pressure reduction techniques, we measured the IPCs (presented in Figure 2) of 3 configurations — base single-thread execution (*BST*), base reliable dual-thread execution (*DTE*), and reliable dual-thread execution where the trailing instructions do not consume any registers, LSB, and ROB entries (*DTE-TNR*). The IPC reduces dramatically from *BST* to *DTE*. For *DTE-TNR*, the IPC increases by about 142%, compared to *DTE*. Drop in IPC from *BST* to *DTE-TNR* is due to the reduction in the availability of other resources such as the dispatch/issue slots, issue queue entries, etc. An important observation that can be made from Figure 2 is that the difference in IPC of the *DTE* and the *DTE-TNR* configurations is not the same for all the benchmarks. The difference in IPC between the *DTE* and the *DTE-TNR* configurations depends on the IPC of the benchmark and the actual resource pressure observed during program execution. For instance, the register pressure could be

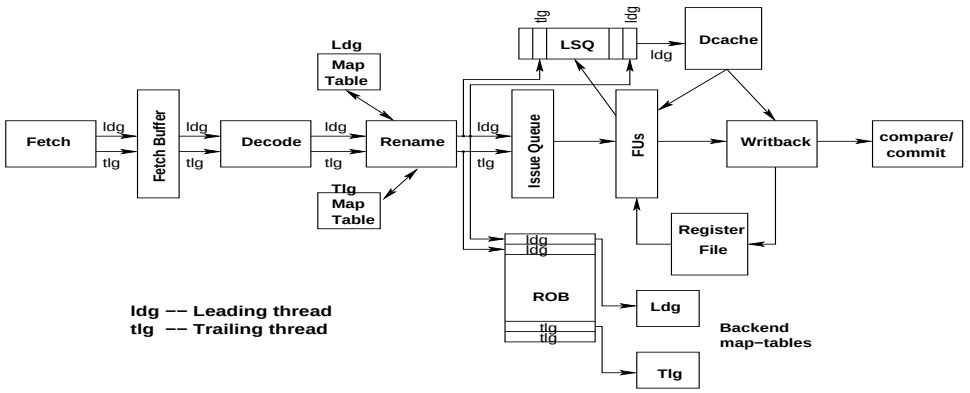

Figure 1: Schematic Diagram of a Reliable Processor

lower because of the presence of a larger percentage of branch and store instructions. If the IPC is lower, then other shared resources such as the issue queue can become a bottleneck. If the register/ROB/LSB pressure is lower, then avoiding allocation of these resources (*DTE-TNR*) may not benefit significantly.

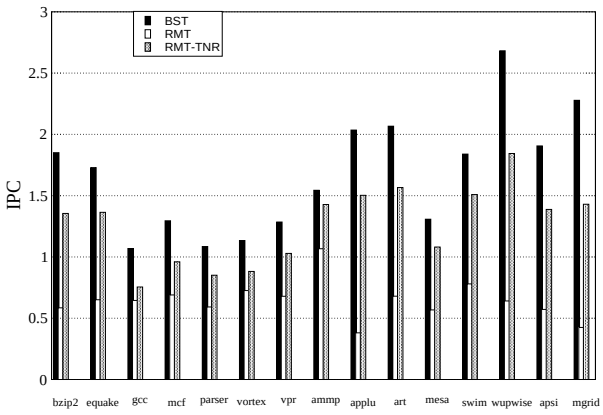

Figure 2: IPCs for 3 configurations showing the benefits from optimal resource allocation to trailing instructions

Previous studies [9, 10, 2] have shown that data values produced in a program tend to be of narrow widths. Traditionally, values are categorized as narrow only if their leading bits are all zeros or ones. However, this categorization will fail to include most of the floating-point values because of the *IEEE 754* standard used for their representation. Intuitively, for a floating-point value, the least significant portion of the significand may have a higher probability of being all zeros (for instance, a value 0.5). Hence, in our studies, any value whose at least 16 (for a 32-bit word) leading or trailing bits are zeros or ones is categorized as narrow. We observed, there are a considerable number of values with at least 16 trailing zeros and non-zero bits in the upper 16 bits. For instance, `bzip2` had about 10% values with at least 16 trailing zeros. Overall, about 50% of the results could be categorized as narrow. This statistic can be exploited to reduce the pressure on the resources.

It is important to note that it is not sufficient to reduce the pressure on just one resource. For instance, alleviating register file pressure can increase the pressure on LSB and ROB, and the benefits obtained will be limited. Figure 3 shows percentage distribution of cycles (out of the total cycles where the resources were requested[1]) in which the instructions were stalled due to unavailibility of the different resources. As seen in Figure 3, if the pressure on a resource is alleviated, the stalls shift from that resource to another. For instance, for `gcc`, the stalls are due to load/store buffer entries if the trailing instructions are not allocated either ROB entries or registers, and the stalls shift to integer registers if the trailing instructions are not allocated LSB entries. Hence, it is imperative to have a comprehensive technique that attempts to remove the stalls in all the resources.

3 Optimal Resource Allocation

3.1 Register Bits Reuse (RBR)

If a leading instruction produces a narrow result, the result can be compressed into the lower 16 bits of a register. In such a case, the upper 16 bits of the register do not store significant data, which can be used for the trailing instruction's result. The correct value is constructed by appending 16 zeros or ones either in front or behind the lower or upper 16 bits of the register. Figure 4 illustrates the working of the *RBR* technique for the instruction i_x that produces a narrow value $0xfa25ffff$. In Figure 4(a), by the time the trailing counterpart of i_x (i_{xR}) is renamed, i_x has

[1] Cycles where the resources are not requested (such as cycles were nothing is decoded or dispatched) are not counted. In addition, when measuring the stalls, if mutliple resources are not available, register file is given the highest priority, followed by ROB and then by LSB.

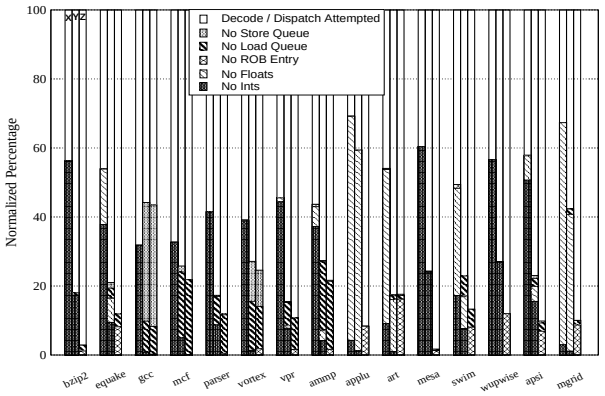

Figure 3: Percentage Distribution of stalls for (X) No LSB (RepNoLSB); (Y) No ROB (RepNoROB); and (Z) No Register (RepNoReg); allocation to trailing thread in DTE configuration

already produced a value and stored it in the register P_{10}. When i_{xR} produces the result $0xfa25ffff$, the value stored in the P_{10} is $0xfa25fa25$.

The proper functioning of the *RBR* technique hinges on the following issues: (i) how is the original instruction's result's size, and its register, passed to the replica instruction?, (ii) how are the registers read, written, committed, and squashed?, and (iii) how are the errors detected? To handle the first issue, we use an additional *size bit* in each ROB entry and an additional *replica pointer* for the ROB (Figure 4(b)). The *size bit* indicates the size of the instruction's result, and the *replica pointer* points to the ROB entry whose replica will be next renamed. When a trailing instruction is renamed, the *replica pointer* is used to obtain the size of its leading counterpart's result (if any), and the register allocated to it. Note that, each ROB entry holds the register identifier used by an instruction to update the *backend map tables* at commit. If the *size bit* is set, the trailing instruction is allocated the same register as the read from the ROB and the *size bit* in its ROB entry is also set, else another register is sought for the replica. Figure 4(b) shows the renaming and register allocation for instructions i_x and i_{xR} that produce a narrow result.

Width, *location*, and *value* bit-vectors (each of size equal to the number of physical registers) are used to appropriately read and write the registers, as shown in Figure 4(c). If a leading instruction writes a register then, if the result is narrow, the *width* bit of that register is set, if the result has non-significant data in the front, then the *location* bit is set, and if the non-significant data is all ones, then the *value* bit is set. For the example in Figure 4(c), the *width, location*, and *value* bits for register P_{10} are "101" as the result produced by i_x is $0xfa25ffff$. In some cases, it may happen that a trailing instruction is renamed before the leading instruction generates a narrow result. In such cases, the leading instructions still compress and decompress their results. However, it will not affect the correctness of the technique. Since the copies of instructions are committed and squashed simultaneously, the registers used by these instructions are de-allocated simultaneously. For error detection, if the *size* bit of the replica instruction's ROB entry is set, then a single register is read from the additional value buffer (refer Section 2.1).

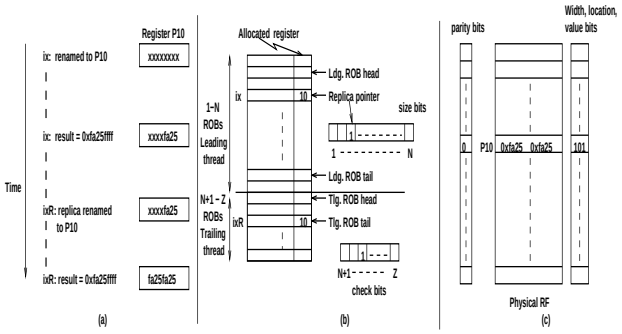

Figure 4: Example Illustrating the *RBR* technique

In addition to reducing register file pressure, the *RBR* technique has the potential of significantly reducing the energy consumption in the register file, by reducing both the size and the number of values read from and written into the register file.

3.2 Optimal ROB Allocation

The ROB entries of the same instruction in the two threads differs only in the register mapping information. The mapping information can be just the current mapping (for checkpointed branch misprediction recovery) or both current and previous mapping (for ROB-walk based branch misprediction recovery). With the *RBR* scheme, it is possible that the ROB entries of the same instruction in the two threads may be the same. We experiment with a more pathological case of ROB-walk based branch misprediction recovery, for which both the current and the previous mappings of an instruction should be the same, for the ROB entries to be exactly similar. To implement this scheme, an additional *map bit* is maintained in the map table for the trailing instructions. If a trailing instruction is using the same mapping as its leading counterpart, then the *map bit* is set. For a trailing instruction, if the *map bit* is set and the current mapping is also the same its leading counterpart, then that instruction is not allocated an ROB entry. The *map bit* is recovered, on a branch misprediction, while walking the ROB. This scheme will necessitate that

the ROB entries are protected using error codes[2], as an error in an ROB entry may not detected because that ROB entry may be used by both the copies of an instruction.

3.3 Optimal LSB Allocation

Traditionally, LSB implementation is such that load instructions broadcast their addresses in the store buffer and store instructions broadcast their addresses in the load buffer. These broadcasts are followed by comparisons to detect any load alias misspeculation and store-to-load forwarding. However, if the leading load and store instructions have already produced their addresses by the time their trailing counterparts are dispatched, then the same hardware can be used to perform the comparisons for error detection. To detect whether a leading memory instruction has generated its result, another bit is provided in the ROB which is set when a memory instruction has generated its result. In this technique, a trailing load instruction (whose leading counterpart has already generated its result) generates its address on the store's address generation unit (AGU). This results in its address being broadcast in the load buffer and compared with the leading counterpart's entry in the load buffer. Similarly, a trailing store instruction can generate its address in the load's AGU, and compare with the leading counterpart's entry in the store buffer. In case the instruction mismatches the address generated by the leading counterpart, the instruction is marked as being faulty. In some cases, the leading load instruction may not have generated the address by the time its trailing counterpart is renamed. In that case, the trailing counterpart can be given a separate Load Buffer entry. However, we follow an alternative approach, where the trailing load instruction is made dependent on the leading load instruction. This is possible because most of the load instructions have a single register operand. In the rare case that a load instruction has two register operands, then it is allocated a separate load buffer entry. Note that, for the sake of simplicity, load alias misspeculation is handled as a branch misspeculation. This scheme replaces the write and read of the address of the trailing instruction, with a single broadcast of its address, thus saving energy. Note that, this technique will require error codes to protect the store buffer entries once the comparison has been performed.

This technique may remove all the pressure on the load buffer due to trailing instructions. However, store instructions also have the value, to be stored, in the store buffer entry to support store-to-load forwarding. Hence, allocation of store buffer entries to the trailing store instructions can only be avoided for the instructions that store a narrow value, so that the space for the value can be shared among the instructions from both the threads.

[2]The base reliable processor of Figure 1 does not need error codes for ROB for only error detection. However, it still requires error codes for error correction.

3.4 Vulnerability Impact of RBR

Assuming that the *DTE* configuration is impervious to all soft errors, how does the *RBR* technique affect the vulnerability of the processor? Firstly, any error in the additional *width*, *location*, and *value* bits can result in a fault. Hence, these bits may have to be duplicated to prevent from such errors from occuring. The additional bits in the ROB will be protected by the error codes used for the ROB entries. Secondly, there is a possibility that an error may not be detected if a leading instruction erroneously produces a smaller-sized value, and its trailing counterpart is allocated the upper half of the register. The trailing instruction may only write half of its result bits into the upper half of the register, erroneously assuming that the other half bits hold either zeros or ones. To address this issue, if an instruction marked as producing a small-sized result, produces a normal-sized value, then the error is detected by tagging the instruction as faulty in the ROB. Thirdly, having the multiple copies of a value in the same register may in fact reduce the overall probability of an error going undetected. It is usually the case, that soft errors will occur in bits close to each other. If the two copies of a value are allocated contiguous registers, then the same bits are physically closer to each other than when a single register holds the two values. Hence, the probability of a bit-flip in the same location may also reduce.

3.5 Limited RBR

As discussed in the previous section, the additional bits may have to be duplicated to make them impervious to errors. To reduce the additional bits required for the *RBR* technique, we investigated a *limited RBR* technique where a value is considered narrow if its significant part is 14 bits or less. In this case, each register can either hold a normal value or 2 narrow values and the duplicated *location* and *value* bits for the 2 values.

3.6 Dead Value Removal (DVR)

Previous studies [4, 11] have shown that a program execution generates first-level dead values, which are values that are over-written without being used. This observation can be used to further optimize resource allocation, by not allocating resources for the trailing counterparts of "dead" instructions. However, the implementation of the *DVR* technique will be much more complicated than the *RBR* technique because of branch mispredictions and exceptions. This is because, if the trailing counterpart of a leading instruction has already been declared "dead", and the leading instruction is found to be "not dead" because of branch misprediction, the trailing instruction will have to be re-activated. Hence, it cannot be guaranteed that a value is dead until the redefiner of the value commits, which can only happen after both the copies of the "dead" instruction is committed. This will require allocation of resources to the "dead" trailing instruction. However, we still investigated the performance of the *DVR* technique, where resources are not allocated to "dead" trailing instructions. We do not discuss the detailed implementation of the *DVR* technique to conserve space. We observed that the *DVR* technique resulted in only about 1% improvement in IPC when it was used in isolation. In combination with the *RBR* technique, the improvement reduced to about 0.6%, because some of the "dead" values were narrow. It is important to note that the IPC improvements were obtained when almost 80-90% of the dead values were captured and resulted in no resources being allocated to the "dead" trailing instructions.

4 Experimental Results

4.1 Experimental Setup

The hardware parameters for our superscalar processor are given in Table 1. Our base pipeline (for the *BST* configuration) consists of 8 front-end stages. For the *DTE* configuration without the *RBR* technique, one pipeline stage is inserted before the commit stage to check the values. For the *RBR* technique, one pipeline stage is inserted after execution and before writeback to check the size of the result values. Another pipeline stage is inserted before the register read to re-construct the correct values from the compressed values read from the register file. Information (from the ROB entry pointed to by the *replica pointer*) for optimal replica register allocation is obtained one cycle before the rename stage (in parallel to decode). Overall, the branch misprediction penalty increases by 1 cycle because of the additional pipeline stage before the register read. In addition, additional pipeline stages further increase the register file pressure by delaying the release of registers (caused by delayed instructions commit).

We use a modified SimpleScalar simulator [3], simulating a 32-bit PISA architecture. In our simulator, we use a unified physical and architectural register file where the architectural registers are committed in the physical register file itself. Two registers are allocated to an instruction producing a long or a double result value (requiring 64 bits for representation). For benchmarks, we use 6 SPEC2000 integer (`vpr`, `mcf`, `parser`, `bzip2`, `vortex`, and `gcc`), and 9 FP (`wupwise`, `ammp`, `swim`, `equake`, `applu`, `art`, `apsi`, `mgrid`, and `mesa`) benchmarks. The statistics are collected for 500M instructions after skipping the first 1B instructions.

For the *RBR* technique, the number of ROB, Load buffer, and Store buffer entries for the trailing instructions are reduced to 32, 0, and 5 respectively. The remaining entries are provided for the leading instructions. Note that this distribution of entries may not be the best possible distribution (which can be determined empirically), because the replicas may stall due to too few entries provided for them.

4.2 Results

Figure 5 shows the IPC results of the *RBR* and *Limited RBR* techniques, compared to *DTE* and *DTE-TNR* configurations. As can be seen from Figure 5, IPC difference between *RBR* and *Limited RBR* technqiues is negligible. However, *RBR* technique increases the IPC of the *DTE* configuration by about 64.4% with a maximum reaching 144% for `apsi`. As expected, we observed that the techniques perform better for benchmarks where more narrow values are encountered. In fact, the IPC with the *RBR* technique is only about 2% - 77% less than the optimum configuration of *DTE-TNR*.

The performance improvement obtained from the *RBR* technique is not equal for all the benchmarks, because it depends on the amount by which the pressure is reduced. For the benchmarks that are register constrained, the performance improvement obtained from the *RBR* technique also depends on the type of registers for which the pressure is reduced. For instance, if a benchmark is floating-point register constrained and the techniques reduce the register file pressure on the integer registers, then the techniques are not expected to be very effective.

Figure 6 presents the stalls due to unavailability of registers, ROB and LSB entries for *DTE*, *DTE-TNR*,

Parameter	Value	Parameter	Value
Fetch/Decode/ Commit Width	8 instructions	*FP FUs*	3 ALU, 1 Mul/Div
Unified Phy. Register File	128 INT/128 FP entries, 2-cycle acc. lat. 1-cycle inter-subsystem lat.	*Int. FUs*	4 ALU, 2 AGU 1 Mul/Div
Issue Width	5/3 INT/FP instructions	*Issue Queue*	96 INT/64 FP Instructions
Branch Predictor	Gshare 4K entries	*BTB Size*	4K entries, 2-way assoc.
L1 - I-cache	32K, direct-map, 2 cycle latency	*L1 - D-cache*	32K, 4-way assoc., 2 cycle latency, 2 r/w ports
Memory Latency	100 cycles first word 2 cycle/inter-word	*L2 - cache*	unified 512K, 8-way assoc., 10 cycles
ROB size	128 leading 64 trailing	*Load buffer size* *Store buffer size*	30 leading, 10 trailing 30 leading, 10 trailing

Table 1: Baseline Processor Hardware Parameters for the Experimental Evaluation

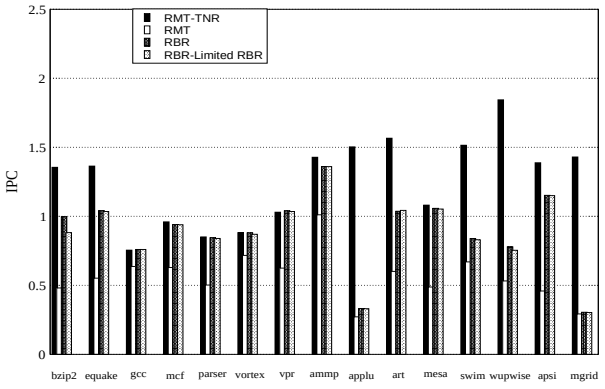

Figure 5: IPCs for various configurations

and *RBR* configurations. Figure 6 is similar to Figure 3 in representation. As seen in Figure 6 the number of stalls decrease for most of the benchmarks. Note that, the stalls could also increase because of an increase in the number of instructions vying for the resources (due to an increase in IPC). This is the reason for increase in stalls for some of the benchmarks.

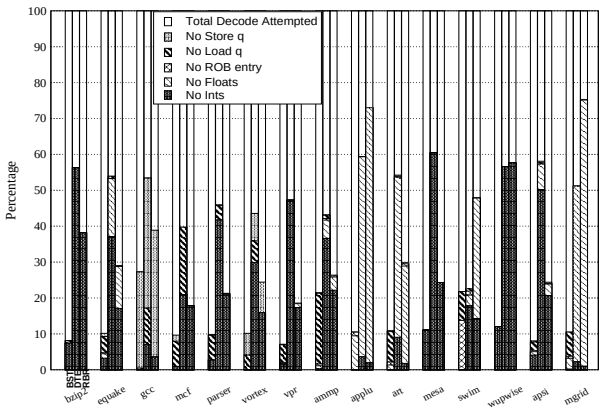

Figure 6: Percentage distribution of cycles with stalls for the various configurations

Overall, for about 45% of result producing instructions of the trailing thread register allocation was avoided, for about 50% of trailing instructions ROB allocation was avoided, for about 50% of trailing store instructions store buffer allocation was avoided, and for almost 100% of the load instructions load buffer allocation was avoided.

Power Measurements: We use the cacti tool to perform the energy measurements for the register file, ROB, LSB, and the additional value buffer. For the measurements, we measure the energy consumption for each access and the type of access and multiply it by the number of such accesses obtained from the simulations. Energy consumption in the register file and additional value buffer (*AVB*) will reduce because for many instructions, fewer bit-lines are activated to write and read smaller values from the register file. Additional energy is consumed in the additional *width*, *location*, and *value* bits for both the leading and the trailing instructions. When instructions that share a single register commit, a single register is read from the *AVB* also saving energy by reducing the number of commit-time reads. Energy consumption in the ROB and LSB is reduced mostly because of fewer writes and reads from these structures. However, some of the energy saved in the LSB is compensated by the additional broadcast and compare for the trailing instructions. Figure 7 shows the percentage savings in dynamic energy consumption (w.r.t *DTE*) obtained in the register file, ROB, LSB, and the additional value buffer, respectively. These measurements also include the energy consumed in the additional bits. As seen in Figure 7, about 10% energy savings is achieved in the register file, about 25% in the ROB, about 30% in the load buffer, about 10% in the store buffer, and about 18% in the additional value buffer.

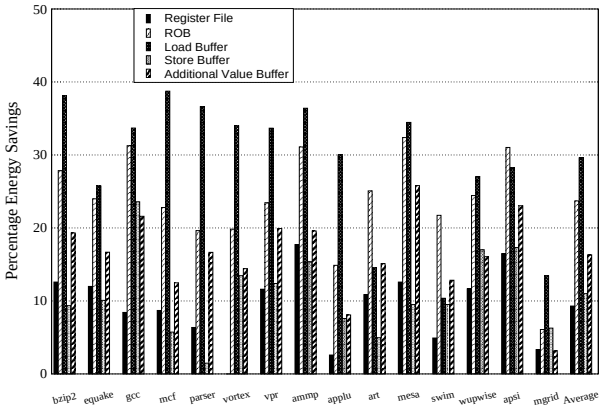

Figure 7: Percentage saving in RF, ROB, Load/Store Buffer, and AVB for the *RBR* configuration

Load Value Buffer Size: In the staggered execution model, the values loaded from the leading thread are forwarded to the trailing thread, using a load value buffer. The load value buffer size will depend on the number of load instructions in the 64 instructions by which the leading thread is ahead of the trailing thread. In the *RBR* technique, if the leading load instruction loads a narrow value by the time its trailing counterpart has been renamed, then the two instructions will share the same register. Hence, to reduce the size of the load buffer required, when a leading load instruction loads a narrow value and its trailing counterpart has not been renamed, the value is replicated in the register and no load value buffer entry is allocated for that value. The trailing load instruction is accordingly notified. For the measurement of the load buffer size required, we measured the average number of load instructions in the slack of 64 instructions for each benchmark. With this scheme, we observed that the average load buffer size could be reduced by about 46%.

Instruction Error Probability: In this section, we measure the probability (to first order approximation) of an error being introduced in the execution of an instruction. The approximation model is based on the number of cycles spent by an instruction in the pipeline, *i.e.* if an instruction spends more time in the processor pipeline, the probability of an error being induced in its execution increases proportionately. Hence, we measured the average number of cycles spent by an instruction in the pipeline for the *DTE* and the *RBR* configurations. We observed that the number of cycles for which a commited instruction remains in the pipeline reduces by about 33% for the *RBR* configuration, suggesting that the probability of an instruction incurring an error reduces by about 33%.

Violating the Slack In all the experiments so far, the slack between the leading and the trailing instructions has been fixed at 64 instructions. In such situations, if the leading thread stalls due to unavailability of resources, the trailing thread also stalls even if it had resources available. For instance, a leading instruction stalled because of a filled load buffer can stall a trailing instruction that does not require a load buffer entry. Such cases become much more prominent in the *RBR* technique where the trailing instructions may not even require any additional resources. Hence, we experimented with *DTE* and *RBR* configurations that were allowed to violate the slack condition if the trailing instructions could go forward with the leading instructions stalled. Figure 8 shows that IPCs of the *DTE* and the *RBR* configurations with and without the violating the slack condition. It can be seen from Figure 8, that the performance of the *DTE* configuration does not improve when violation of the slack condition is allowed, whereas, the performance of the *RBR* technique increases by about 1.3%.

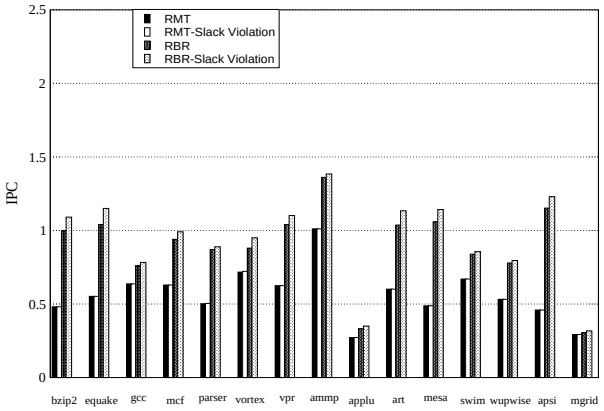

Figure 8: IPCs for *DTE* and *RBR* configurations with and without slack violation

5 Sensitivity Study

The performance of the *RBR* technique, compared to that of *DTE*, depends on the resources in the processor. We measure the IPCs of the *DTE* and the *RBR* techniques as the register file size is changed to 96 and 164, and the ROB size is changed to 128 and 256. In these experiments, all the other hardware parameters remain the same. Figure 9 presents the results for the measurements. As seen in Figure 9, generally, increasing the number of resources reduces the difference in IPC between the *DTE* and the *RBR* configurations. This is because, the resources are no longer bottlenecks.

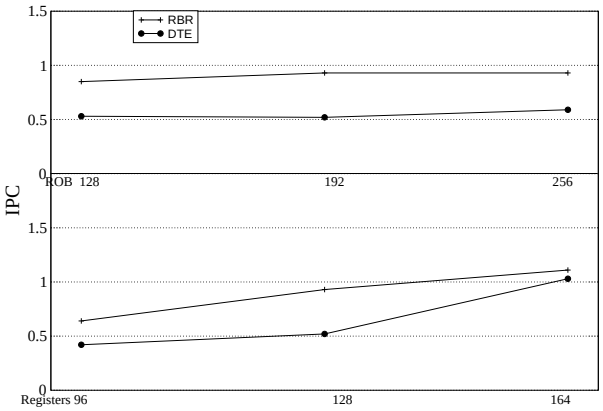

Figure 9: IPCs for *DTE* and *RBR* as the Register File and ROB sizes are varied

6 Related Work

Techniques that simultaneously execute multiple copies of the same instructions have been proposed for concurrent error detection and recovery [15, 1, 14, 16, 17, 21, 22, 8]. Ray, Hoe, and Falsafi [15] use the same superscalar datapath to execute the multiple copies of an instruction for fault-tolerance. Austin proposes a very different fault-tolerant scheme [1] which comprises of an aggressive out-of-order superscalar processor checked by a simple in-order checker processor. The fault-tolerant architectures in [16, 17, 22, 8] use the inherent hardware redundancy in simultaneous multithreading and chip multiprocessors for concurrent error detection. Patel and Fung [14] propose transforming the input operands between redundant computations to expose a persistent fault.

Smolens et. al. [20] perform studies to measure the performance impact of redundant execution. They focus their studies on the issue logic and the ROB, as they do not allocate registers to the trailing instructions. However, their techniques may be susceptible to errors in the pipeline frontend, such as errors in rename.

Packing multiple narrow values in a single register for a single threaded processor has been discussed by Oguz et. al. [13]. However, they have to use speculation techniques to decide whether to pack the result or not. Such speculative techniques can have a significant misspeculation penalty. The techniques discussed in this paper are non-speculative because when running the trailing thread at a *slack*, the information from the leading thread is readily available and can be exploited.

Shubhendu, et. al. [11] suggest that "dead" values reduce the vulnerability of an architecture to soft failures. However, they do not explore the possibility of utilizing the "dead" values to improve the performance of a reliable processor. Eliminating dynamically dead instructions from the execution stream has been studied for a single thread in [4].

7 Conclusion

Reliability in systems is usually ensured by corroborating the results of redundant threads, where the one thread runs ahead of the other thread by a few instructions. Redundant threads place a significant pressure on the processor resources, especially the register file, ROB, and LSB, thus impacting the performance.

In this paper, we investigate techniques that attempt an optimal allocation of resources to instructions of the trailing thread. The techniques in this paper are based on 2 very important observations: (i) many of the values produced in a processor are of small size, and (ii) if the one thread is running behind the other by a few instructions, then the leading instructions usually produce their results before their trailing counterparts are renamed. The *register bits reuse* technique exploits the narrow width of the results produced in a program, and attempts to allocate a single register to both copies of an instruction, if the result produced by the instructions is of narrow width. We also discussed detailed implementations of the technique. To enhance the number of narrow width values produced in a program, we propose a novel way of defining a narrow value as one that has either leading or trailing zeros or ones. The *RBR* technique is extended to also avoid the allocation of ROB entries for many trailing instructions. Innovative scheme is also proposed to resue the load/store buffer hardware to avoid allocation of LSB entries to all the loads and about 50% of the stores of the trailing instructions

We observed that the *register bits reuse* technique produces about 64.4% performance improvement over the base case running one redundant thread for concurrent error detection. The power consumption reduces by about 10-30% in the various hardware structures.

References

[1] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," *Proc. Micro-32*, 1999.

[2] S. Balakrishnan, et. al., "Exploiting Value Locality in Physical Register Files," *Proc. Micro-36*, 2003.

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, 1997.

[4] J.A.Butts and G.Sohi, "Dynamic dead instruction detection and elimination," *ASPLOS*, 2002.

[5] Compaq Computer Corp., "Data integrity for Compaq Non-Stop Himalaya servers," *http://nonstop.compaq.com*, 1999.

[6] G. Hinton, et al, "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, Nov. 2001.

[7] J. G. Holm, and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions" *Proc. ICPP-21*, 1992.

[8] M. Gomaa, et. al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. ISCA-30*, 2003.

[9] G. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," *Proc. Micro-35*, 2002.

[10] S. Kumar, P. Pujara and A. Aggarwal, "Bit-Sliced datapath for energy-efficient high performance microprocessors," *Workshop on PACS*, 2004.

[11] S. Mukherjee, et. al., "A Systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Micro-36*, 2003.

[12] S. Mukherjee, et. al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. Micro-36*, 2003.

[13] O. Ergin, et. al., "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure," *Proc. Micro-37*, 2004.

[14] J. H. Patel, and L. T. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, 31(7):589-595, July 1982.

[15] J. Ray, J. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," *Proc. Micro-34*, 2001.

[16] S. Reinhardt, and S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. ISCA-27*, June 2000.

[17] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. of the 29th Intl. Symp. on Fault-Tolerant Computing Systems*, June 1999.

[18] D. P. Siewiorek and R. S. Swarz, "Reliable Computer Systems Design and Evaluation," *The Digital Press*, 1992.

[19] T. J. Slegel, et al. "IBM's S/390 G5 microprocessor design," *IEEE Micro*, 19(2):12-23, March/April 1999.

[20] J.Smolens, et. al., "Efficient Resource sharing in Concurrent error detecting Superscalar microarchitectures ," *Proc. Micro-37*, 2004.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *In Proc. Micro-33*, December 2000.

[22] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," *Proc. ISCA-29*, 2002.

[23] C. Weaver, et. al., "Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor," *Proc. ISCA-31*, 2004.