Reducing Resource Redundancy 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. Redundant multi-threading (*RMT*) is an attractive approach for concurrent error detection and recovery. However, redundant threads significantly increase the pressure on the processor resources, resulting in dramatic performance impact.*

In this paper, we propose reducing resource redundancy as a means to mitigate the performance impact of redundancy. In this approach, all the instructions are redundantly executed, however, the redundant instructions do not use many of the resources used by an instruction. The approach taken to reduce resource redundancy is to exploit the runtime profile of the leading thread to optimally allocate resources to the trailing thread in a staggered RMT *architecture. The key observation used in this approach is that, even with a small* slack *between the two threads, many instructions in the leading thread have already produced their results before their trailing counterparts are renamed. We investigate two techniques in this approach (i)* register bits reuse *technique that 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, and (ii)* register value reuse *technique that attempts to use the same register for a main instruction and a distinct redundant instruction, if both the instructions produce the same result. These techniques, along with some others, are used to reduce redundancy in register file, reorder buffer, and load/store buffer. The techniques are evaluated in terms of their performance, power, and vulnerability impact on an* RMT *processor. Our experiments show that the techniques achieve about 95% performance improvement and about 17% energy reduction. The vulnerability of the* RMT *remains the same with the techniques.*

Keywords: Concurrent Error Detection, Reducing Redundancy, Register File, Redundant Multi-threading

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 [4, 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, 11]. 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 multi-threading (*RMT*) [14, 6, 4, 19, 1, 13, 15, 16, 21, 22, 7]. In this approach, the same application is run multiple times and the errors are detected by corroborating the redundant results. Studies [22, 7] have shown that a *staggered RMT*, where one thread *slacks* behind the other thread, results in better performance because the trailing thread does 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. In this paper, we propose reducing resource redundancy to mitigate the performance impact of *RMT*. In this approach, full instruction redundancy is provided for full error coverage, however, many redundant instructions do not use all the resources used by an instruction. This reduces the pressure on the resources, thus improving performance and reducing energy consumption. In addition, we observed that reducing redundancy in just one resource simply shifts the pressure from the optimally allocated resource to another resource, and does not result in significant performance improvement. Hence, in this paper, we attempt to simultaneously reduce redundancy in multiple resources to achieve significant improvement. The key observation used in reducing the redundancy is that many instructions in the leading thread produce their results before the trailing thread is renamed. This enables us to exploit the leading thread's runtime profile for reducing redundancy in the trailing thread. These techniques are very well suited particularly for a *staggered RMT* execution.

The first technique in this approach — *register bits reuse (RBR)* — exploits the sizes of the results produced by the leading instructions 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 its 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, thus effectively reducing the use of additional registers by the redundant trailing instructions. We also discuss novel ways of defining a narrow width value, which result in a significant number of narrow values (even floating-point values). We extend the *RBR* technique to also reduce the redundant allocation of reorder buffer (ROB) entries to the trailing thread. This is possible because many instructions have exactly the same ROB entry values for the leading and the trailing counterparts when these instructions use the same register mapping. Even with reducing the redundancy in the register file and the ROB, we observed that the load/store buffer (LSB) became a bottleneck for many benchmarks. Hence, we propose a novel technique to reduce the pressure on the LSB and obtain an average performance improvement of about 62%. We also evaluate the techniques in terms of their energy and vulnerability impact and observe an average of about 17% in energy. The vulnerability of the *RMT* remains almost the same.

For the normal-sized results, we investigate *register value reuse (RVR)* technique that exploits data value locality observed in instructions' results. In this technique, if two leading instructions produce the same value, then the trailing counterpart of the second leading instruction is assigned the register of the first leading instruction. This technique further reduces the register file pressure, especially for the floating-point benchmarks, resulting in overall performance improvement of about 96%.

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 *RBR* technique. Section 4 presents the experimental results and analysis. Section 5 discusses further enhancements to the *RBR* technique. Section 6 presents sensitivity study for the *RBR* technique. Section 7 presents the implementation details and experimental results for the *RVR* technique. Section 8 presents related work. Finally, in Section 9, 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 an instruction 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 [7].

The entries for the instructions of the two threads are fixed in the ROB and the LSB, to facilitate finding multiple copies of an instruction at 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 [7]. 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.

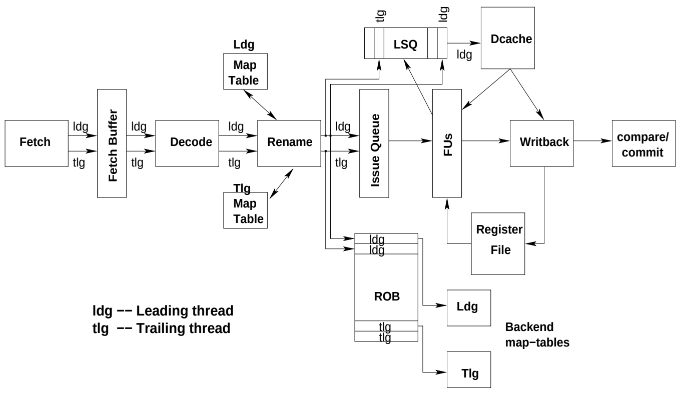


Figure 1. Diagram of a Reliable Processor

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 is the best for a *slack* of 64 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 redundancy reduction techniques, we measured the IPCs (presented in Figure 2) of 3 configurations — base single-thread execution (*BST*), base redundant multi-threading with one redundant thread (*RMT*), and *RMT* where the trailing instructions do not consume any registers, LSB, and ROB entries (*RMT-TNR*). The IPC reduces dramatically from *BST* to *RMT*. For *RMT-TNR*, the IPC increases by about 150%, compared to *RMT*. Drop in IPC from *BST* to *RMT-TNR* is due to redundancy in 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 *RMT* and the *RMT-TNR* configurations is not the same for all the benchmarks. The difference in IPC between the *RMT* and the *RMT-TNR* configurations depends on the IPC of the benchmark and the actual resource pressure observed during program execution. If the IPC is lower, then other shared resources such as the issue queue can also become a bottleneck. If the register/ROB/LSB pressure is lower, then avoiding allocation of these resources (*RMT-TNR*) may not benefit significantly.

As discussed in introduction, it may not be sufficient to reduce the redundancy in just one resource. Figure 3(a) shows

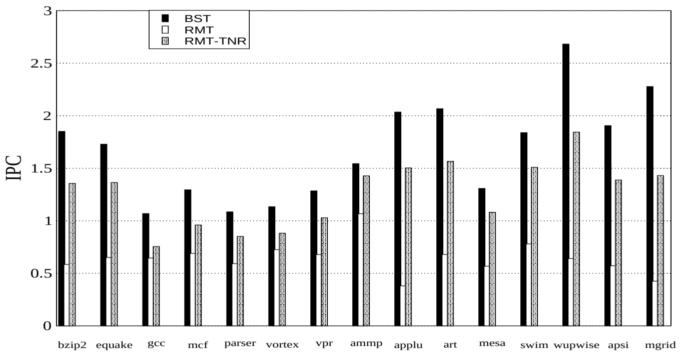


Figure 2. IPCs for 3 configurations showing the benefits from reducing resource redundancy for trailing instructions

the normalized percentage distribution of cycles (out of the total cycles where decode/dispatch was attempted) in which the instructions were stalled due to unavailability of register file, LSB and ROB buffer. When measuring the stalls, if multiple resources are not available, register file is given the highest priority, followed by ROB and then by LSB. The measurements are made when the redundant instructions do not allocate registers (*Rep-NoReg*), ROB entries (*Rep-NoROB*), or LSB entries (Rep-NoLSB). Normalization is performed with respect to the *Rep-NoLSB* configuration. The rest of the hardware parameters are shown in Table 1. As seen in Figure 3(a), 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 LSB 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. Figure 3(b) further gives the Instructions per Cycle (IPC) count for the four – *Rep-NoReg, Rep-NoROB*, Rep-NoLSB, and *RMT-TNR* – configurations. Figure 3(b) shows that the *RMT-TNR* configuration performs better than the other three configurations for most of the benchmarks. For some benchmarks (such as `mesa`, `parser`, and `vpr`), we observed that *RMT-TNR* performed slightly worse, primarily due to a reduction in branch mispredictions and load-alias misspeculations. Figures 3(a) and (b) show that it is imperative to have a comprehensive technique that attempts to reduce the redundancy in multiple resources.

3 Reducing Resource Redundancy

3.1 Reducing Register Redundancy with Register Bits Reuse (RBR)

Previous studies [8, 9, 2] have shown that many narrow-sized data values are produced in a program. 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 zero bits 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.

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 can be used to store the trailing instruction's result, avoiding a separate register for the redundant trailing instruction. 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 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 $0xfa25fa25$.

To pass the leading instruction's result's size and its register identifier to the trailing instruction, we use an additional *size bit* for each ROB entry in the leading thread's ROB section (ROBs 1 - N in Figure 4(b)) 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 trailing instruction will be next renamed. When leading instructions are dispatched, the corresponding size bits are *reset*. Since, the dispatch is in-order, a single port (with multiple bit-lines activated) is enough to *reset* the bits. When a leading instruction produces a narrow result, it *sets* its *size bit*. In this case as well, a single port is enough to set the appropriate bits for all the instructions producing narrow results in a cycle. Hence, the *size bits* bit-vector requires 2 write ports and a single read port. The trailing instructions that are getting renamed in a cycle read the size bits following the bit pointed to by the *replica pointer*. The *rename pointer* is then incremented.

The register to be allocated to the trailing instructions (if their leading counterparts produce a narrow result) can be determined from the ROB entries pointed to by the *replica pointer*. Note that, each ROB entry holds the register identifier used by an instruction to update the *backend map tables* at commit. However, this will necessitate an increase in the number of read ports into the register identifier portion of the ROB. This increase in the number of read ports can be avoided as we will discuss in Section 3.2. If the *size bit* is set, the same register is allocated to the trailing instruction, else another register is sought. The *replica pointer* is updated as the instructions are committed or squashed. Figure 4(b) shows the renaming and register allocation for instructions i_x and i_{xR} that produce a narrow result. Each ROB entry for the trailing instruction is also provided with a *check bit* which is set (at dispatch) if the trailing instruction reuses the register of the leading instruction. The *check bit* is used to guarantee the correct functioning of the *RBR* technique. At commit, the identifiers of the registers allocated to the two instructions are compared and the *check bit* of the trailing instruction is checked. If the register identifiers match and the *check bit* is set, or the register identifiers mismatch and the *check bit* is reset, then the correct operation is assumed. A single *check bit* is enough to detect errors caused by single event upsets (SEUs).

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). For a leading instruction's register, the *width* bit is set if the result is narrow, the *location* bit is set if the result has non-significant

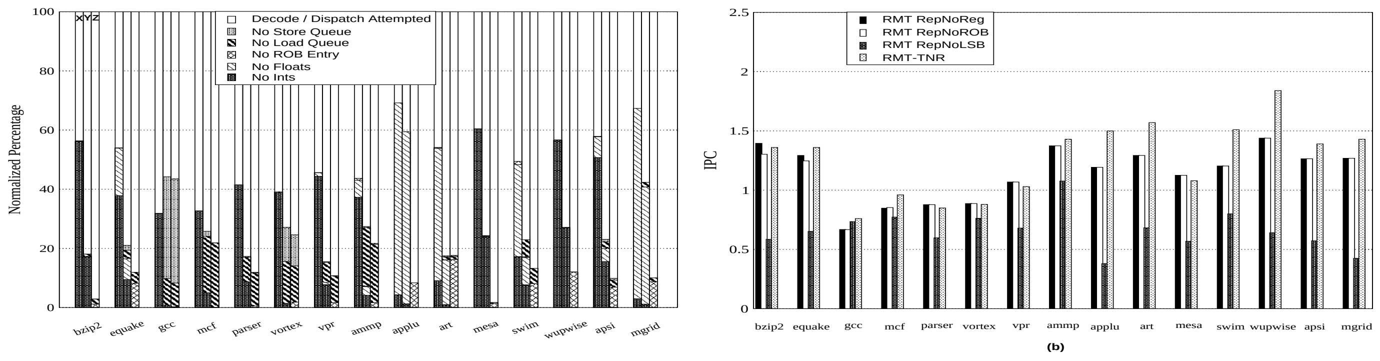


Figure 3. (a) Percentage Distribution of decode/dispatch stalls for (X) No LSB (RepNoLSB); (Y) No ROB (RepNoROB); and (Z) No Register (RepNoReg); (b) IPCs for the different confi gurations

data in the front, and the *value* bit is set if the non-significant data is all ones. For example, in Figure 4(c), the *width*, *location*, and *value* bits for $0xfa25ffff$ are “101”. To mitigate faulty execution because of errors in these status bits, a *parity bit* is used to detect errors in the *width*, *location*, and *value* bits. *RBR*’s functions correctly even if a trailing instruction is renamed before the leading instruction generates a narrow result. Since the copies of instructions are committed and squashed simultaneously, the registers used by these instructions are de-allocated simultaneously.

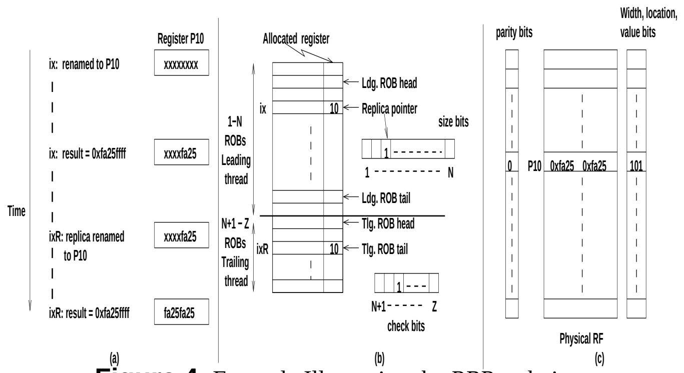


Figure 4. Example Illustrating the *RBR* technique

The *RBR* technique reduces the register file pressure and 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. It also reduces the energy consumption in the additional value buffer (refer Section 2.1) because if the *check bit* of the trailing instruction’s ROB entry is set, then a single register is read from the additional value buffer.

3.2 Reducing ROB Redundancy with RBR

The ROB entries of the same instruction in the two threads differ only in the register mapping information. The mapping information stored in the ROB entry can only be the current mapping (for checkpointed branch misprediction recovery) or both current and previous mappings (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 their ROB entries to be exactly similar. For instance, if the *defi ner* and the *redefi ner* of a register produce narrow results, then the trailing counterpart of the *redefi ner* will have exactly the same ROB entry as the leading instruction. ROB allocation for the trailing counterpart can be avoided in this case. 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. On a branch misprediction, all the *map bits* are reset. To further reduce the redundancy in ROB allocation, control type instructions (such as branch and jump instructions) are also allocated a single ROB entry for their leading and trailing counterparts.

This scheme will necessitate that the ROB entries are protected using parity bits, as an error in an ROB entry may not detected because that ROB entry may be used by both the copies of an instruction. The parity bits are generated after rename and before dispatch, in parallel to other pipeline stages. To cover faults that may occur after rename but before parity generation, the ROB entry is written by the leading instruction, whereas the parity bit for the entry is generated for the trailing instruction and written into the *parity bits buffer*. Parity bit generation is required for only those trailing instructions that do not use a separate ROB entry, thus limiting the additional energy as well. Each *parity bit buffer* entry is also provided with a *valid-parity bit* which also signifies whether a separate ROB entry is used by the trailing instruction. At commit, if the *valid-parity bit* is set, then the corresponding parity bit detects errors in the ROB entry. The *map* and *valid-parity* bits may be duplicated to avoid faulty execution on errors in these bits. ROB redundancy reduction will also save energy by reducing the number of writes and reads from the ROB.

Reduction in the ROB entries that are read at commit can facilitate increasing the commit width. However, to limit the increase in ROB read ports (refer Section 3.1), we limit the commit width (from a maximum of four leading and four trailing instructions) to a maximum of four instructions. This will commit four distinct instructions if the copies of all the four instructions share the ROB entries. If only two or three instructions (out of the four instructions) share their ROB entries, then only three distinct instructions can be committed. The determination of the number of instructions to be committed is performed one cycle prior to the commit of the instructions.

3.3 Reducing LSB Redundancy

Traditionally, LSB is implemented in a way 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. The trailing load and store instructions simply write their addresses (and value for the store instruction) into the LSB entry allocated to them. However, if the leading load and store instructions have already produced their addresses by the time their trailing counterparts are dispatched, then the same broadcast and compare hardware can be used to perform the comparisons for error detection. To detect whether a leading memory instruction has generated its address, a *LSB pointer* similar to the *replica pointer* is used to point to the next memory instruction whose trailing counterpart will be dispatched. On dispatch of trailing memory instructions, if the *valid bits* of the LSB entries of their leading counterparts are set, then the trailing instructions are not allocated LSB entries.

In this technique, a trailing load instruction (whose leading counterpart has already generated its address) generates its address on the store's address generation unit (AGU). This results in its address being broadcasted in the load buffer and compared with the leading counterpart's entry in the load buffer. A trailing store instruction is executed in a similar fashion. In case there is a mismatch in the addresses generated by the two counterparts, the instruction is marked as being faulty. If the leading load instruction has not have generated the address by the time its trailing counterpart is dispatched, the trailing load instruction is made dependent on the leading load instruction to serialize their execution. This is possible because most of the load instructions have a single register operand. If a load instruction has two register operands, then the dispatch is stalled till the leading counterpart produces its address. However, a trailing store instruction is allocated a separate store buffer entry if its leading counterpart has not produced its address. This scheme replaces the write and read of the address of the trailing instructions, with a single broadcast of its address, thus saving energy as well.

This technique may remove all the redundancy in the load buffer due to trailing instructions. However, store instructions also have to store the value 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. Note that, this technique will require *parity* and *valid-parity* bits to protect the store buffer entries' addresses (not the values) once the comparison has been performed.

3.4 Limited RBR

To reduce the additional *width*, *location*, *value*, and *parity* bits required for each register in the *RBR* technique, we propose a *limited RBR* technique where a value is narrow if its significant part is 14 bits or less. In this case, each 32-bit register can either hold a normal value or two narrow values and the duplicated *location* and *value* bits for the two values. The *width* bit has to be provided and duplicated outside of the register to determine the size of the values. In *limited RBR*, the number of additional bits reduce from four to two.

4 Experimental Results

4.1 Experimental Setup

The hardware parameters for the base *RMT* superscalar processor are given in Table 1. Our base pipeline (for the *BST* configuration) consists of 8 front-end stages. For the *RMT* 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 after the register read to re-construct the correct values from the compressed values read from the register file. Overall, the branch misprediction penalty increases by 1 cycle because of the additional pipeline stage before the register read.

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. 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 trailing instructions may stall due to too few entries provided for them. We use a single *parity bit* each for the ROB entry, store buffer entry, and the additional status bits in the register file to protect from a single event upset (SEU). The number of *parity bits* can be increased to protect from multi-bit upsets.

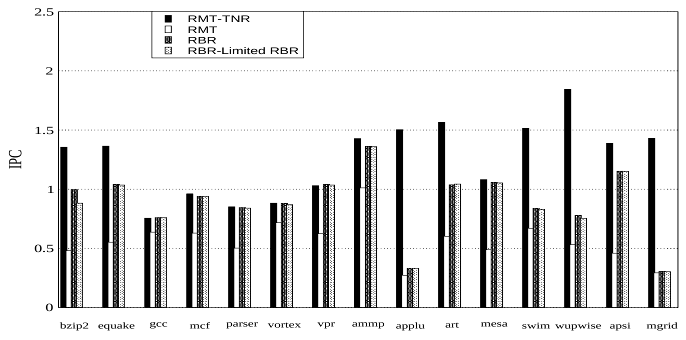
4.2 Results

Figure 5(a) shows the IPC results of the *RBR* and *Limited RBR* techniques, compared to *RMT* and *RMT-TNR* configurations. We also experiment with two more configurations – *RMT-Add* and *RMT-Add-Ltd* – where the additional bits required for each structure in the *RBR* technique are used to increase the capacity of the structures in the base *RMT* configuration. For instance, the *width*, *location*, *value*, and *parity* bits are used to increase the number of registers by 16 in the *RMT-Add* configuration and the additional *width* bits increase the number of registers by eight in the *RMT-Add-Ltd* configuration. Depending on the additional bits, the ROB size can be increased by 10 entries. Similarly, the load buffer increases by one entry and the store buffer increases by two entries. Figure 5(b) shows the IPCs of *RBR* and *Limited RBR* compared to *RMT-Add* and *RMT-Add-Ltd*. *Note that an increase in the structures' sizes will significantly impact their access time, which is not affected in the RBR technique, where the bits are used as separate status bits.*

As can be seen from Figure 5(a), IPC difference between *RBR* and *Limited RBR* techniques is negligible. However, *RBR* technique increases the IPC of the *RMT* configuration by about 62% with a maximum reaching 144% for `apsi`. As

Parameter	**Value**	**Parameter**	**Value**
Fetch/Decode/ Commit Width	8 instructions	*FP FUs*	3 ALU, 1 Mul/Div
Unifi ed 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



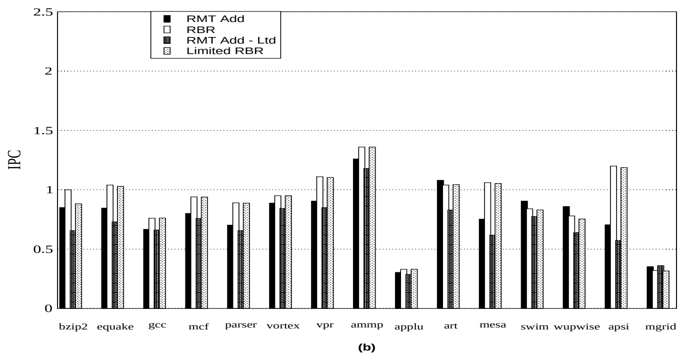


Figure 5. (a) IPCs for *RMT*, *RMT-TNR*, *RBR*, and *Limited RBR* confi gurations; (b) IPCs for *RBR*,*Limited RBR*, *RMT-Add*, *RMT-Add-Ltd* confi gurations

expected, we observe that the techniques perform better for integer benchmarks, where more narrow values are encountered. The IPC with *RBR* is better than that for *RMT-TNR* for `gcc` and `vpr` because the stalls in these benchmarks are almost reduced to that of *RMT-TNR*, because of a high percentage of narrow values. However, lower branch misprediction and load-alias misspeculation gives slightly better results for the *RBR* technique. Figure 5(b) shows that *limited RBR* is significantly better than *RMT-Add-Ltd*, where the additional hardware in *limited RBR* is instead used to increase the size of the structures. However, when the number of registers are increased by 16, *RMT-Add* performs slightly better than *RBR* for `art`, `swim`, `wupwise`, and `mgrid` because of fewer narrow values encountered in these benchmarks.

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 integer register pressure, then the techniques are not expected to be very effective. For instance, consider `applu` and `swim` in Figure 6. Figure 6 presents the reduction in register, ROB, and store buffer redundancy. Load buffer redundancy reduction is 100% and hence not shown. The bars in Figure 6 present the percentage of trailing instructions (out of those that require a resource) for which separate allocation of that resource was avoided. For instance, bar 1 shows the percentage of integer result producing trailing instructions for which integer register allocation was avoided. `Applu` and `Swim` are constrained by floating-point registers. The *RBR* scheme is able to save more than 40% of integer registers for these two benchmarks, whereas only about 15% of floating point registers. Hence, *RBR* scheme does not perform that well for these two benchmarks.

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. The absence of the bar for floating-point registers is due to the absence of floating-point instructions.

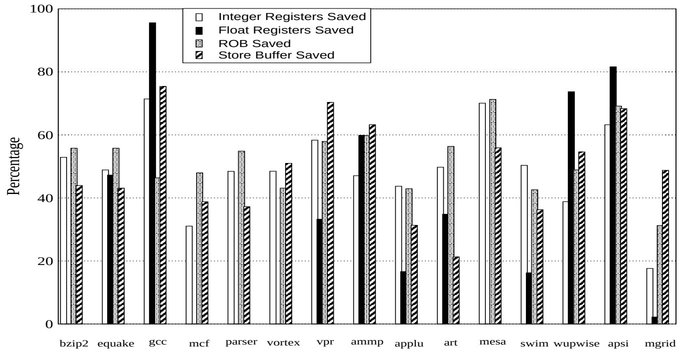


Figure 6. Percentage Savings of Integer and Floating-point registers, ROB and Store buffer entries for the trailing thread with *RBR*

4.3 Power Results

We use the cacti tool [17] 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. 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. However, additional energy is consumed in the additional *width*, *location*, *value* and *parity* bits. Energy consumption in the ROB and LSB is reduced primarily 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. Additional energy is also consumed in the ROB and the LSB from *parity* and *valid-parity* bits. Figure 7 shows the percentage savings in dynamic energy consumption (w.r.t *RMT*) 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 shows that, in general, energy savings is more in integer benchmarks than the floating-point benchmarks, which is expected.

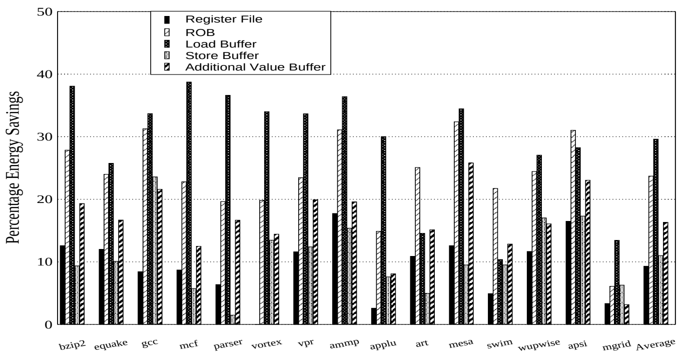


Figure 7. Percentage energy saving in RF, ROB, Load/Store Buffer, and AVB for the *RBR* configuration

4.4 Load Value Buffer

In staggered execution, the values loaded by the leading thread are forwarded to the trailing thread using a load value buffer. The load value buffer size will depend on the average number of load instructions in the *slack*. 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. We call this technique *load value buffer reduction (LVBR)*. The *replica pointer* is used to determine whether a trailing counterpart of a leading load instruction has been renamed or not. 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. For the *RMT* configuration, the average number of load instructions that require a load buffer entry (across all the benchmarks) is about 30, and with *LVBR* scheme, this number reduces by about 46%.

We performed experiments with a 32-entry load value buffer in the *RMT* configuration, a 32-entry load value buffer in the *RBR* configuration, and a 16-entry load value buffer in the *RBR-LVBR* technique. We observed that the IPC reduction was about 5% in going from *RBR* to *RBR-LVBR* technique. However, the reduction in power consumption in the load value buffer was about 65%. The structure of the load value buffer assumed was a fully associative buffer with a register identifier field and a loaded value field. The trailing load instruction broadcasts the register identifier of the leading load instruction, and on a match reads the value. On a mismatch, the trailing load instruction waits to be woken up by the leading load instruction. The structure and the working of the load value buffer will remain the same in the *RMT* and the *RBR-LVBR* techniques. The reduction in energy consumption is significantly greater than the reduction in size because the savings are achieved from both the reduction in size and accesses.

4.5 Vulnerability Impact of RBR

As discussed in the previous sections, with the addition of *parity bits* and the duplication of selective status bits, the *RBR* will be able to detect all the single event upsets (SEUs) that result in a single bit flip. However, the probability of a multi-bit error in the same ROB entry, store buffer entry, and *width*, *location*, and *value* bits going undetected may increase slightly for the *RBR* technique. In addition, 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 trailing instruction only writes half of its result bits into the upper half of the register. 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.

We measure the vulnerability of the *RMT* architecture in terms of the number of errors that are incurred in the execution of a program. Of course, these errors will be detected in the *RMT* architecture. If all the errors are assumed to be independent of each other, the number of errors will be given by $N_e = P_i \times N_i$, where P_i is the probability of an error in a committed instruction and N_i is the number of committed instructions. N_i remains the same in the base *RMT* and *RMT* with *RBR*. Hence the vulnerability of these two configurations depends on P_i. P_i will be given by $P_i \;\; \alpha \;\; P_e \times P_{ip} \times P_{ep}$, where P_e is the probability that an error occurs, P_{ip} is the probability that the instruction i is in the processor when the error occurred, and P_{ep} is the probability that the error occurred in the hardware being used by the instruction i. This equation assumes that all the three events are independent. To first order approximation, P_i will be
$P_i \;\; \alpha \;\; A \times C_i/C_t \times 1/W^2 \;\; \alpha \;\; C_i/C_t$
where, A is the area of the processor, C_i is the number of

cycles spent by an instruction in the pipeline, C_t is the total cycles taken by the program, and W is the width of the processor. We assume that A almost remains constant when going from *RMT* to *RBR*. Hence, we measured the average number of cycles spent by an instruction in the pipeline and the total cycles spent for executing the code, for the *RMT* and the *RBR* configurations. We observed that the number of cycles for which a committed instruction remains in the pipeline reduces by about 33% for the *RBR* configuration, while the total number of cycles reduce by about 35%, suggesting that the probability of an instruction incurring an error remains almost the same.

5 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 *RMT* and *RBR* configurations that were allowed to violate the slack condition if the trailing instructions could go forward. Note that the slack is violated only if the leading instructions are stalled. Figure 8 shows that IPCs of the *RMT* and the *RBR* configurations with and without the violating the slack condition. It can be seen from Figure 8, that the performance of the *RMT* configuration does not improve when violation of the slack condition is allowed, whereas, the performance of the *RBR* technique increases by about 7%.

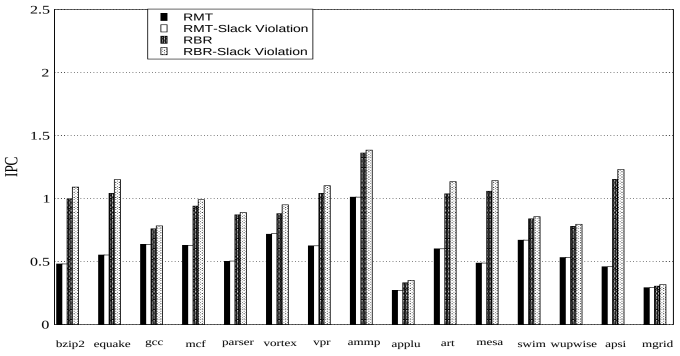


Figure 8. IPCs for *RMT* and *RBR* configurations with and without slack violation

6 Sensitivity Study

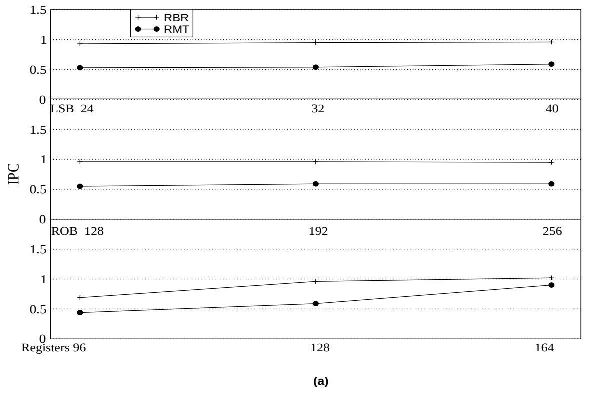
In this section, we measure the IPCs of the *RMT* and the *RBR* configurations as the register file size is changed to 96 and 164, the ROB size is changed to 128 and 256, and the Load/store buffers are changed to 32 and 24 entries. In these experiments, all the other hardware parameters remain the same. For instance, when changing the register file size, ROB and LSB are also kept at default values. When ROB size is varied, the difference in the number of leading and trailing entries is maintained at 64 for *RMT*, and the trailing entries are halved for *RBR*. When the LSB size is varied, the 3:1 ratio in the number of leading and trailing entries is also maintained for *RMT*, and the store buffer entries are halved for *RBR*. Figure 9 presents the integer and floating-point benchmarks' average IPC, as the resource sizes are varied. When the ROB is increased beyond 192 entries, the IPCs of both the *RMT* and the *RBR* configurations remain the same, as ROB is no longer a bottleneck. However, for 128-entry ROB, the average integer IPC reduces slightly for *RMT* whereas it remains the same for *RBR*. When the *LSB* entries are reduced, the *RMT* IPC reduces whereas the *RBR* remains almost the same. As the number of registers are varied from 164 to 128, *RMT* IPC reduces much faster than *RBR* IPC. In fact for the integer benchmarks, the *RBR* IPC almost remains the same. However, when they are reduced from 128 to 96, *RBR* IPC reduces faster than *RMT* IPC. This is because at register file size of 96, registers start to become a bottleneck for the *RBR* technique as well, whereas at register file size of 128, the registers are not much of a bottleneck. This is especially true for the integer benchmarks. For *RMT*, the registers are a bottleneck at both 96 and 128, and hence the decrease in IPC in going from 128 to 96 is not large.

7 Register Value Reuse (RVR)

RBR technique reduces redundancy only if the results generated by instructions are narrow. We investigate *register value reuse (RVR)* technique to reduce redundancy even for normal-sized values. The *RVR* technique can only reduce redundancy in the register file. We also observed that about 19% of the normal results generated by instructions were repeated when considering the previous eight unique normal results. In the *RVR* technique, if a leading instruction produces a normal result that is already present in another register (written to by another leading instruction), then the trailing counterpart of that instruction is renamed to the register already holding the value. Figure 10 explains the *RVR* technique. As seen in Figure 10, instructions i_1, i_x, and i_y generate the same normal result X, where i_1 is the first instruction that generates X. If i_x generates the result by the time i_{xR} (trailing instruction of i_x) is renamed, i_{xR} can be renamed to the register used by i_1 (*i. e.* $P10$). Note that the leading instruction has already broadcast its register identifier in the issue queue (to wake up dependent instructions) by the time the trailing counterpart is renamed to that register.

The *RVR* technique will not increase the vulnerability of the processor to SEUs. Consider that an error occurs in one of the instructions i_x and i_{xR} in Figure 10, whereas i_y and i_{yR} compute correct results. In *RVR* technique, there are two cases: (i) register used by i_x is not overwritten by i_{yR} by the time i_x and i_{xR} commit, and (ii) register used by i_x is overwritten by i_{yR} by the time i_x and i_{xR} commit. In the first case, the error will be detected when the results of i_x and i_{xR} are compared at commit time. In the second case, if an error occurred in i_{xR}, then the error will again be detected when the results of i_{yR} (which overwrites the result of i_x) and i_{xR} are compared at commit time. If i_x errs (*i. e.* i_x erroneously produced a result equal to that of i_1 and i_y), the error is again detected when i_{yR} and i_{xR} are compared at commit.

In this paper, we employ the *RVR* technique on top of the *RBR* technique. To compare the result values of different leading instructions and also to forward the register identifiers to



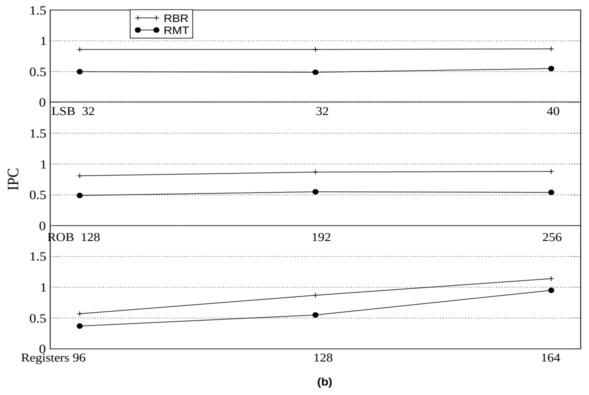


Figure 9. IPCs for *RMT* and *RBR* as the Register File, ROB and LSB sizes are varied for (a) Integer; (b) Floating-Point benchmarks

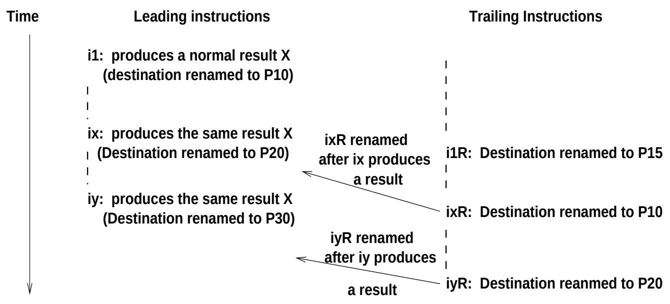

Time
Leading instructions
Trailing Instructions
i1: produces a normal result X (destination renamed to P10)
ix: produces the same result X (Destination renamed to P20)
iy: produces the same result X (Destination renamed to P30)
ixR renamed after ix produces a result
iyR renamed after iy produces a result
i1R: Destination renamed to P15
ixR: Destination renamed to P10
iyR: Destination reanmed to P20

Figure 10. Example Illustrating the Basic Idea behind the *Register Value Reuse* technique

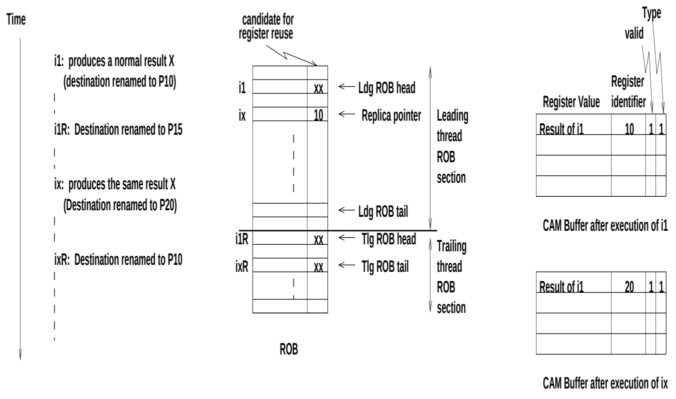

Time
i1: produces a normal result X (destination renamed to P10)
i1R: Destination renamed to P15
ix: produces the same result X (Destination renamed to P20)
ixR: Destination renamed to P10
candidate for register reuse
i1
ix
i1R
ixR
xx
10
Ldg ROB head
Replica pointer
Ldg ROB tail
Tlg ROB head
Tlg ROB tail
Leading thread ROB section
Trailing thread ROB section
ROB
Type valid
Register Value
Register identifier
Result of i1
10
20
CAM Buffer after execution of i1
CAM Buffer after execution of ix

Figure 11. Example Illustrating the *RVR* technique

the trailing counterparts, we use a small CAM (Context Addressable Memory) buffer. The buffer is accessed when the leading instructions produce a value and when a leading redefiner instruction of a register commits. When a redefiner instruction commits, the entry holding the register it redefines is invalidated. When leading instructions produce normal results, unique results are compared with the values in the CAM buffer, and on a match the register identifiers in the ROB (candidate register identifier in Figure 11), and the CAM buffer are updated accordingly. If there is no match, then the least recently filled entry is updated. When a trailing instruction is renamed, it obtains the reuse candidate through the *replica pointer*. Figure 11 illustrates the working of the *RVR* technique when instructions i_1 and i_x produce the same result. When instruction i_1 writes back its result, the CAM buffer is updated with i_1's result. Consequently, when i_x writes the result, the CAM buffer entry is updated with the register identifier of i_x and the ROB entry of i_x records register P_{10} as a reuse candidate for i_{xR}. To handle register deallocation accurately in the *RVR* technique, 2 *register usage* bit-vectors (each of size equal to the number of physical registers) are used, one for the leading instructions and the other for the trailing. A register is deallocated only when its bit in both the bit-vectors is *reset*. A parity bit-vector is required to protect these two bit-vectors. As explained earlier in the section, any errors in the CAM buffer or the additional bits in the ROB will not result in a faulty detection and the error will be detected. This technique will not hamper in the working of *RBR*.

The only potential problem in this technique will arise if a register gets recorded as a candidate for reuse, but is deallocated and reallocated before it is reused. To handle these situations, we use a *candidate valid* bit-vector (with one bit for each physical register), where the bits are set when a physical register becomes a reuse candidate for reuse and reset when the register is reused. However, if a leading instruction allocates a register with the *candidate valid* bit set, the instruction does not access the CAM buffer even if it produces a normal value. When reusing a register for a trailing instruction, the *candidate valid* bit should be "1", otherwise another register is sought.

Results: In this section, we present the IPC results of *RMT-TNR*, *RBR*, and *RBR+RVR* configurations in Figure 12. We use a fully associative 8-entry CAM structure for *RVR*. As seen in the figure, the *RVR* technique results in further performance improvement, an average of about 20% (over the *RBR* technique), especially for the floating point benchmarks that have a larger percentage of normal values.

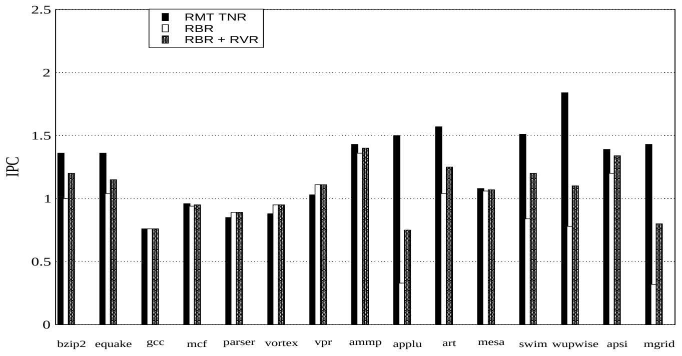


Figure 12. IPCs of *RMT-TNR*, *RBR*, and *RBR + RVR*

8 Related Work

Techniques that simultaneously execute multiple copies of the same instructions have been proposed for concurrent error detection and recovery [14, 1, 13, 15, 16, 21, 22, 7]. Ray, Hoe, and Falsafi [14] 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 [15, 16, 22, 7] use the inherent hardware redundancy in simultaneous multithreading and chip multiprocessors for concurrent error detection. Patel and Fung [13] propose transforming the input operands between redundant computations to expose a persistent fault.

Smolens et. al. [20] study 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 values in a register has been discussed using speculation techniques for a single threaded by Oguz et. al. [12]. 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. Exploiting value locality in the register file for a single threaded processor in [2] can be difficult to implement, especially in the presence of exceptions, unless only a small subset of a priori decided values are used. In our proposal, value locality is exploited to avoid redundant register allocation, eliminating many of the implementation difficulties (such as reference counters, updating the renaming of dependent instructions, etc.) when used for a single thread.

Shubhendu, et. al. [10] 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.

9 Conclusion

Ensuring reliability in systems using 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 reduce resource redundancy in Register File, ROB, and LSB. The techniques in this paper are based on a key observation: in *staggered execution*, the leading instructions usually produce their results before their trailing counterparts are renamed. The *register bits reuse* technique allocates a single register to both copies of an instruction, if the result produced by the instruction is narrow. 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 redundancy in ROB entries for many trailing instructions. Innovative scheme is also proposed to reuse the load/store buffer hardware to avoid redundancy in LSB entries.

We observed that the *RBR* technique produces about 62% performance improvement over the base *RMT* configuration. The power consumption reduces by about 10-30% in the various hardware structures. We also proposed reusing the registers for normal values. In this case if two leading instructions produce the same normal result, then the trailing counterpart of the second is renamed to the register used by the first leading instruction. This technique improves the performance to about 95% more than the base case.

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

[5] 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.

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

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

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

[9] S. Kumar, P. Pujara and A. Aggarwal, “Bit-Sliced datapath for energy-effi cient high performance microprocessors,” *Workshop on PACS*, 2004.

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

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

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

[13] 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.

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

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

[16] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors,” *Proc. 29th Intl. Symp. on Fault-Tolerant Computing Systems*, 1999.

[17] P. Shivakumar, and N. Jouppi, “CACTI 3.0: An Integrated Cache Timing Power, and Area Model,” *Technical Report, DEC Western Research Lab*, 2002.

[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., “Effi cient 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.