

Self-Checking Instructions — Reducing Instruction Redundancy for Concurrent Error Detection

Sumeet Kumar
Electrical and Computer Engineering
Binghamton University
Binghamton, NY 13902
skumar1@binghamton.edu

Aneesh Aggarwal
Electrical and Computer Engineering
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. However, redundant thread execution has a significant impact on performance and energy consumption in the chip.

In this paper, we propose reducing instruction redundancy (the instructions that are redundantly executed) as a means to mitigate the performance and energy impact of redundancy. In this paper, we experiment with an *decoupled RMT* approach where the frontend pipeline stages are protected through error codes, while the backend pipeline stages are protected through redundant execution. In this approach, we define two categories of instructions — *self-checking* and *semi self-checking* instructions. *Self checking* instructions are those instructions whose results are checked for any errors when their “main” copies are executed. These instructions are not redundantly executed. *Semi self-checking* instructions are those instructions for which a major part of their results is checked when the “main” copies are executed, and the remaining part of the instructions is checked using a small amount of additional hardware. Reducing instruction redundancy with this approach has the same fault coverage as the base architecture where all the instructions are redundantly executed. The techniques are evaluated in terms of their performance, power, and vulnerability impact on the *RMT* processor. Our experiments show that the techniques reduce instruction redundancy by about 58% and recover about 51% of the performance lost due to redundant execution. Our techniques also recover about 40% of the energy consumption increase in the key data-path structures.

Categories and Subject Descriptors

C.1.1 [Processor Architectures]: Single Data Stream Architectures—*Pipeline Processors, RISC/CISC, VLIW Ar-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PACT’06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

chitectures; C.4 [Processor Architectures]: Performance of Systems—*Fault Tolerance*

General Terms

Performance, Design, Reliability

Keywords

Concurrent Error Detection, Reducing Instruction Redundancy, Self-checking Instructions, 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, 15] that occur randomly due to various reasons such as alpha particle radiations, power supply fluctuations due to ground bounce, etc. Current trends suggest that transient errors will be an increasing burden for microprocessor designers [20, 9]. Transient hardware errors are troublesome because they elude most of the current testing methods. A popular approach to detect transient errors is to redundantly execute the instructions [11, 5, 4, 16, 1, 10, 12, 13, 18, 19, 6]. In this approach, the same application is run multiple times and the errors are detected by corroborating the redundant results (called *Redundant Multi-threading (RMT)*).

There are three different flavors of the *RMT* approach in a single core microprocessor. The first method (*SMT approach*) uses the inherent redundancy in a simultaneous multi-threaded environment [12, 13, 19, 6], where the two threads are executed entirely independently of each other (except may be some profile information being transferred from one thread to another). The second approach (*super-scalar approach*) uses the data-path of a single core super-scalar processor [11] to execute the redundant instructions by using some additional hardware (such as additional PC generation) to ensure reliability. The third approach – *decoupled RMT (dRMT)* – [1, 17] protects the front end of the pipeline using error codes and protects the execution core logic through redundant execution. The first two approaches may have a higher performance impact (in terms of instructions per cycle (IPC)) than the third approach, while the third approach is likely to be more expensive in terms of hardware than the first two approaches (mainly because of protecting the frontend through error codes). In either

of the approaches, studies have shown that a *staggered* execution (where one thread runs ahead of the second thread by a few instructions) performs better than a *simultaneous* execution (where both the threads are run in a lock-stepped manner) [6, 17]. This is because the trailing thread does not incur many of the branch mis-prediction and the load miss penalties incurred by the leading thread. However, redundantly executing each instruction in an application can result in significant performance degradation as well as dramatic increase in energy consumption. Our experiments show that the *dRMT* approach results in 21% reduction in IPC and about 75%-135% increase in energy consumption in key structures in the backend data-path (such as the issue queue, register file, and ROB, which form a major fraction of the total chip power consumption [7]). The performance degradation in the other two approaches is much more significant.

In this paper, we experiment with the *dRMT* approach, and propose simple-to-implement techniques to reduce *instruction redundancy* (instructions that need to be executed redundantly for fault tolerance) without having an impact on fault coverage of the processor. We experiment with a MIPS ISA to find the instructions whose redundant execution can be avoided in the *dRMT* approach. Similar techniques may be used for other ISAs and other *RMT* approaches. Our first technique exploits the observation that many of the instructions have at least one operand whose value is "zero". The results produced by such instructions (except a few such as multiply, divide, AND, etc) will be exactly the same as their "non-zero" operands, and the two are compared with each other for error detection. Such instructions, called *self-checking* instructions, are then not executed redundantly. We observed that about 38% of the instructions executed were such self-checking instructions. Our experiments show that avoiding redundant execution of these self-checking instructions improves the performance by about 8% (which is almost 25% of the performance lost due to redundant execution) and reduces the average energy consumption in the key backend data-path structures by about 17%.

We extend this technique to include *semi self-checking* instructions to further reduce the performance and energy consumption impact of *dRMT*. This technique exploits the observation that many of the instructions have at least one operand (either immediate or register) that is small in size (uses a small number of lower significant bits for representation). The higher significant bits of the results of such instructions are frequently the same as the higher significant bits of their "non-small" operands. This approach compares the higher significant bits of an instruction's result with that of its "non-small" operand to detect errors in these bits. The remaining bits are checked for errors using a small amount of additional hardware. We also propose innovative techniques to further extend this approach to also include small negative operands. Our experiments show that exploiting the self-checking and the semi-self-checking instructions prevents re-execution of about 58% instructions in the spec2K benchmarks, recovering about 51% of the performance loss observed in *dRMT*. These techniques also reduce the energy consumption impact by about 40%.

The rest of the paper is organized as follows. Section 2 discusses the background and presents the performance and energy consumption impact of redundant execution. Section

3 discusses the technique for self-checking instructions. Section 4 presents the experimental results and analysis. Section 5 discusses the technique for semi self-checking instructions. Section 6 presents sensitivity study for the different processor configurations. Section 7 presents related work. Finally, in Section 8, we conclude.

2. BACKGROUND AND MOTIVATION

2.1 Background

The schematic diagram of the *dRMT* configuration is shown in Figure 1. The frontend of the pipeline is the same as the original pipeline (without redundant execution), but protected using error codes (which could be a single parity bit to detect a single event upset (SEU)). The parity bits for instructions (protecting the instructions in the instruction cache) are fetched along with the instructions. The decoding logic partitions an instruction into multiple fields (such as opcode, source registers, immediate value, etc.). Hence, to recover from an error in the decoding logic, each field of an instruction may have to be separately protected using parity bits. When the instructions are dispatched to the issue queue, ROB, and the load/store buffers (for load and store instructions), all the fields of the instruction are checked for errors. We call these instructions as "main" instructions. In this configuration, we assume that the control logic is protected either through parity bits (for microprogramming based control signal generation) or through hardware duplication.

Once the instructions are dispatched, the backend pipeline is protected through redundant execution. For this, the "main" instructions are re-dispatched from the ROB back into the issue queue. We call these instructions "redundant" instructions. For re-dispatch: (i) the "main" instructions should have finished execution, (ii) all the older instructions should have been re-dispatched (the re-dispatch is performed in-order), and (iii) the *slack* should have been satisfied (*i.e.* the number of younger instructions is greater than or equal to the minimum *slack* instructions). Since, the instructions are re-executed, each ROB entry (shown in Figure 1) also stores the execution control signals, the source register mappings, the immediate value (if any), and PC (for branch instructions), along with the destination register identifier and its current and previous mappings. The "redundant" instructions use the information in the ROB entry for re-execution. The results produced by the "redundant" instructions (addresses for the load/store instructions) are compared with that of the "main" instructions for error detection. Note that, only the "main" load/store instructions access the transient fault-tolerant caches, but the addresses are generated redundantly for error detection. To avoid additional register file ports for result comparison, the results of the "main" instructions are also stored in the ROB entry [17], as shown in Figure 1. The *result field* also stores the branch target address for a branch instruction, and the effective address for the load/store instructions (to avoid additional load/store buffer (LSB) ports). The "redundant" instructions do not store the redundantly generated results and addresses in the register file and the LSB. Hence, the register file and the LSB are also protected through parity bits. In an ROB entry, only the destination register identifier and its mappings need to be protected through parity bits; the other fields are protected through re-execution.

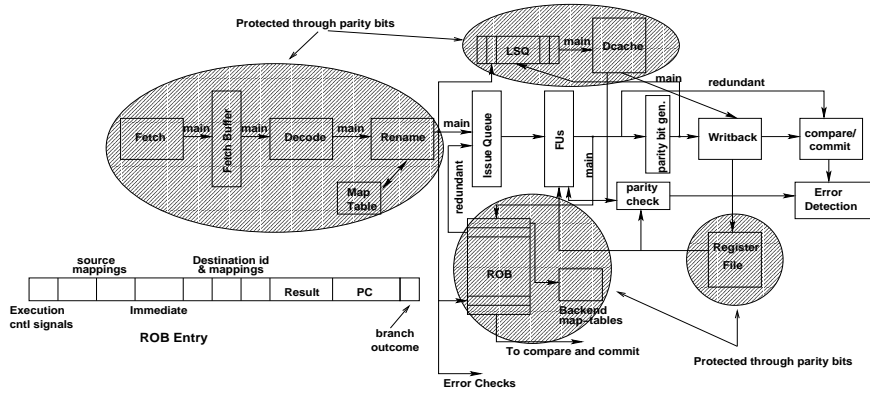


Figure 1: Schematic Diagram of a Decoupled RMT Processor (*dRMT*)

Deadlocks may occur if the “main” instructions fully utilize a resource (such as register file or load/store buffer) without the *slack* being satisfied because the instructions will not be able to re-dispatch, and hence not commit. Deadlocks are avoided by keeping a counter that counts the number of “redundant” instructions that have been re-dispatched but not re-executed. If any resource is full and the counter is *zero*, then a potential deadlock is avoided by re-dispatching the “redundant” instructions, 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. For our base configuration given in Table 1, we choose an instruction *slack* of 64 instructions between the threads. If the conditions for re-dispatch are not satisfied, only the “main” instructions are dispatched. When the conditions are satisfied, equal number of “main” and “redundant” instructions are dispatched. However, on branch mis-prediction, irrespective of the *slack* condition, the “redundant” instructions (prior to the mis-predicted branch instruction) are dispatched to the issue queue.

2.2 Motivation

Each redundantly executed instruction in the *dRMT* approach vies for an issue queue entry, an issue slot, and a functional unit, resulting in increased pressure on these resources and hence, in performance loss. In addition, the processor resources are accessed more number of times resulting in an increase in energy consumption. Furthermore, the size of the structures (such as the ROB) also increases, further increasing the energy consumption. Figure 2(a) shows the instructions per cycle (IPC) count for the base (non-RMT) superscalar processor and the *dRMT* approach. Figure 2(b) shows the percentage increase in energy consumption in the key backend data-path structures (such as the register file, ROB, and the issue queue) for the *dRMT* approach, as compared to the base superscalar processor. For the energy consumption readings, we measured the energy consumption of each type of access using a modified version of the cacti tool [14] for a $0.18\mu\text{m}$ technology. We multiplied the energy consumption of each access with the number of accesses for the total energy consumption. Note that additional energy is also consumed in the functional units, the select logic, error detection logic, frontend pipeline stages, etc. that is not shown in Figure 2(b). As can be seen in Figure 2, an average 21% reduction in IPC and about 75%-135% increase in en-

ergy consumption (in register file, ROB, and issue queue) is observed for the *dRMT* approach. The experimental setup for these readings is provided in Section 4.1. The ROB entry for the *dRMT* approach is shown in Figure 1, whereas for the base superscalar processor, we assume that each ROB entry only holds the destination register identifier and its mappings (the only information required for branch mis-prediction recovery in a processor). In addition, the ROB energy consumption in Figure 2(b) has been optimized for the *dRMT* approach by only activating those bit-lines that are required to read and write the appropriate fields in the ROB entry. The issue queue energy consumption only considers the read and write accesses to the issue queue (and not the wakeup and select energy consumption), which forms a major fraction of the energy consumption in the issue queue.

3. REDUCING INSTRUCTION REDUNDANCY

3.1 Self-checking Instructions

A self-checking instruction in the *dRMT* approach is an instruction whose result is the same as at least one of the operands, so that the result can be compared against that operand for error detection. Except for a few instructions such as multiply, divide, AND, etc., for self-checking instructions, one of the operands typically has a value of “zero”. These include move instructions, and memory address calculation and other instructions that use a “zero” operand. In our implementation, we do not consider the instructions such as multiply, divide, AND, etc. as self-checking instructions. Branch instructions are also not considered as self-checking instructions. This is because, branch instructions perform two operations – comparing operands to determine the outcome of the branch and computing the branch target address using the PC and immediate value that are almost always *non-zero* values. Hence, the branch target address has to be recomputed for full fault coverage. We measure the percentage of instructions (out of the total committed) that are self-checking instructions. For the store instructions, these measurements only consider operands used for address calculation. The results are shown in Figure 3. As can be seen in Figure 3, an average of about 38% instructions fall into the category of self-checking instructions. As expected, the percentage of self-checking instructions is relatively lower for floating-point benchmarks.

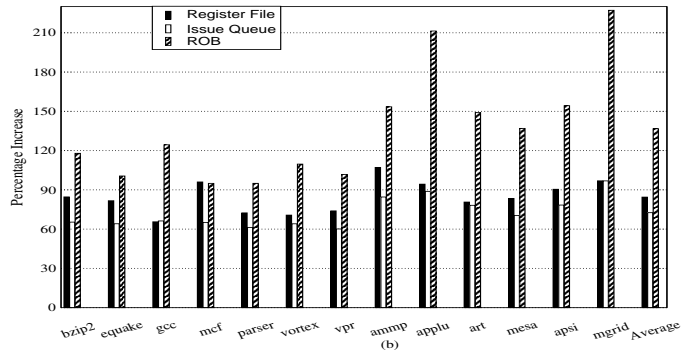
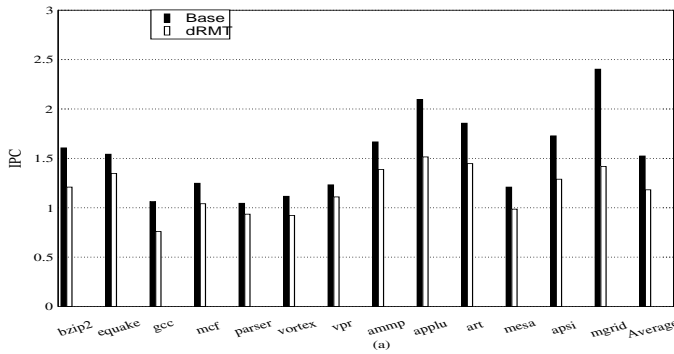


Figure 2: (a) IPCs for the base superscalar processor and the *dRMT* approach; and (b) Percentage increase in energy consumption in key backend hardware structures due to the *dRMT* approach

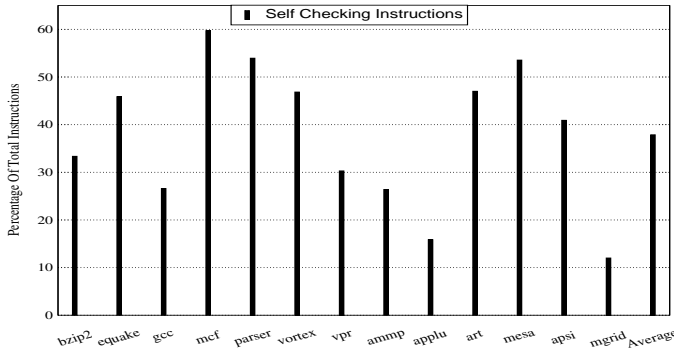


Figure 3: Percentage of total instructions that are *self-checking instructions*

3.2 Implementing Self-checking in *dRMT*

In this section, we discuss detailed implementation of the *dRMT* approach where self-checking instructions are not re-executed (called *dRMT-SC*). Figure 4(a) shows the original *dRMT* processor backend pipeline and Figure 4(b) shows the pipeline with the *dRMT-SC* technique. In the original *dRMT* pipeline, the result (including the branch target address and load/store effective address) is written into the *result field* of the ROB entry. The result produced by a “main” instruction (including the effective address for load/store instructions) goes through parity bit generation. This is followed by a writeback (of the results and their parity bits) into the register file and the LSB. If the current instruction is a “redundant” instruction, its result is forwarded to the *result comparison* logic, which compares this result with that of the “main” instruction read from the ROB entry. The result of the “redundant” instruction is not written back.

In the *dRMT-SC* pipeline, two *instruction identifier* bits are used for marking the self-checking instructions, where a “00” indicates a *non-self-checking instruction*, a “01” indicates a *self-checking instruction* with operand one being *zero*, and a “10” indicates a *self-checking instruction* with operand two being *zero*. If both the operands of an instruction are *zero*, then any operand can be chosen arbitrarily. These checks are performed only for those instructions that can be considered as self-checking instructions. In Figure 4(b), the check for “zero” operand is performed in parallel to the *parity bit check/execute* stage. The results and the

non-zero operands of the self-checking instructions are forwarded to *result comparison*. However, in Figure 4(b), the *result comparison* is shifted by one stage to protect against errors between *result comparison* and *parity bit generation* for self-checking instructions. If the result of an instruction does not match the *non-zero* operand, an error is flagged. In this pipeline, the instructions that are not marked to be self-checking instructions are re-executed as in the original *dRMT* pipeline. For the “redundant” instructions, the *parity bit generation* logic is clock gated to save energy.

In the *dRMT-SC* approach, only the destination register identifier and its mappings need to be written into the ROB for a self-checking instruction. However, at the time of dispatch when the ROB is written, it is not known whether the instructions is self-checking. For MIPS ISA, some self-checking instructions use register *R0* or an immediate value of “zero” as one of the operands, which can be detected in decode. Hence, the *instruction identifier* bits of such instructions are accordingly set in decode and fewer bits are written into ROB, saving more energy. If an instruction is marked as self-checking after issue, then it still writes the appropriate fields in the ROB entry at the time of dispatch. This optimization will not result in any performance benefits, but will save more energy. For self-checking instructions, the result is never written into the ROB entry. In the *dRMT-SC* pipeline, a slack of 64 instructions is still maintained between the redundant threads, unless a branch misprediction or deadlock is encountered.

3.3 Instruction Vulnerability

In this section, we discuss the vulnerability of the self-checking instructions that are not redundantly executed. The non-self-checking instructions are executed redundantly and their vulnerability will not change as compared to the original *dRMT*. Since the LSB, the register file, and the ROB (at least the destination identifier and its mappings) are protected through parity bits, and the execution is protected through comparison of the result with an operand, the self-checking instructions will become vulnerable only due to errors in the issue queue. Note that any errors in the “zero” checking hardware will be detected in *result comparison*. For instance, the opcodes, the source register identifiers, or the immediate values of the self-checking instructions can change in the issue queue resulting in an erroneous result. However, the parity bit for each of these fields (such as opcode, register identifiers, etc.) are already available

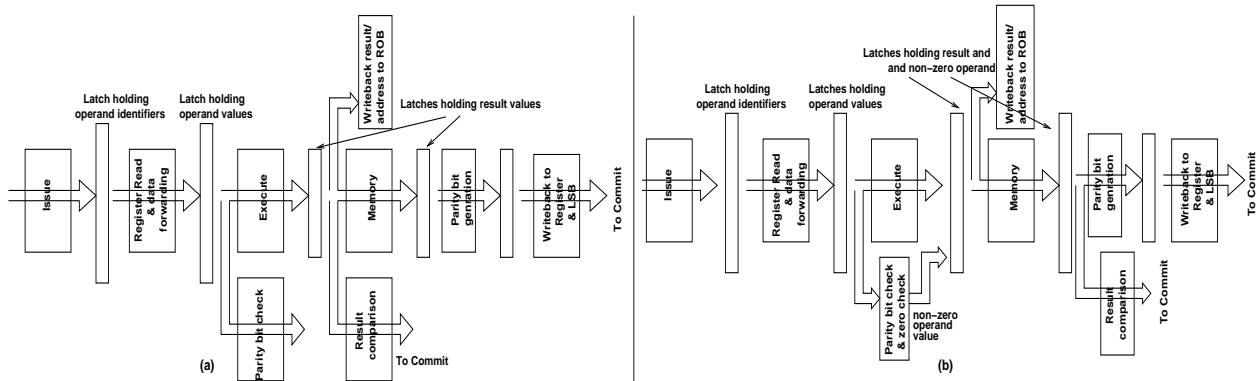


Figure 4: backend Pipelines for the (a) original *dRMT* approach; and (b) *dRMT-SC* approach

for each instruction at dispatch to issue queue (refer Section 2.1). Hence, these parity bits can also be stored along with the instruction in the issue queue, protecting the issue queue entries of self-checking instructions from errors.

More interestingly, self-checking instruction may become vulnerable to errors in the bypass network. For instance, even if a “main” instruction X produces a correct result, an error may occur in the value in the bypass network so that the value received by its dependent self-checking instruction is erroneous. Such an error cannot be detected because the parity bits have not yet been generated for the forwarded value, and the self-checking instruction is not redundantly executed. Note that a value is available from the forwarding network only for a few cycles and probability of such an error occurring may be very minuscule. Nevertheless, to detect such errors, we store the parity bits generated for the most recently forwarded values. If a forwarded value is used by a self-checking instruction, then the parity bit for that value should be zero and/or equal to that of the result of the consumer self-checking instruction. However, this technique will require duplication of decoders used to address the registers to protect against errors in them for self-checking instructions.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

The hardware parameters for the base superscalar processor are given in Table 1. Our superscalar model consists of separate integer and floating point subsystems. Our original *dRMT* pipeline consists of 8 frontend stages, with the backend stages shown in Figure 4(a). The number of stages remains the same for the new *dRMT* pipelines as well. We use a modified SimpleScalar simulator [2], 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 7 FP (*ammp*, *equake*, *applu*, *art*, *apsi*, *mgrid*, and *mesa*) benchmarks. The statistics are collected for 500M instructions after skipping the first 1B instructions.

We use a single *parity bit* each for a ROB entry, a load/store

buffer entry, a register, and a rename table entry. The frontend pipelines are protected using a *parity bit* for each field (such as opcode, immediate, source and destination identifiers) of an instruction. A single *parity bit* is able to protect from a single event upset (SEU). The number of *parity bits* can be increased to protect from multi-bit upsets.

4.2 IPC Results

Figure 5 shows the IPC results of the *dRMT-SC* approach with respect to the base *dRMT* configuration and compared to the base processor without redundant execution. We perform the experiments with two *dRMT* approaches – *fixed IQ* and *non-fixed IQ*. In the *fixed IQ* approach, the issue queue (IQ) entries that can be occupied by “redundant” instructions are fixed (six in the integer subsystem and four in the floating point subsystem)¹. In addition, the lowermost entries in the IQ are fixed for the “redundant” instructions, giving them a lower priority than the “main” instructions. In the *non-fixed IQ* approach, the “redundant” instructions can occupy as many entries in the IQ as warranted by the current *slack*. These entries can be present anywhere in the IQ giving the “redundant” instructions the same priority as the “main” instructions.

Figure 5(a) presents the results for the *non-fixed IQ* approach and Figure 5(b) presents the results for the *fixed IQ* approach. The first observation that can be made from Figure 5 is that the IPC results are generally consistent with the percentage of self-checking instructions shown in Figure 3. Overall, *dRMT-SC* gives about 7% and 8.5% improvement in IPC over the *dRMT* configuration for the *non-fixed IQ* and the *fixed IQ* configurations respectively. More interestingly, the IPC improvement with *dRMT-SC* reduces the performance impact of redundant execution by about 25% for the *fixed IQ* approach. The *dRMT* configuration gives lower IPC results with the *non-fixed IQ* approach because the “redundant” instructions have the same priority as the “main” instructions, delaying the fulfillment of conditions for re-dispatch. In addition, the *non-fixed IQ* approach may result in more stalls where the “main” instructions are stalled and fill up the issue queue and the “redundant” instructions cannot enter the issue queue (even if the *slack* has been satisfied). Henceforth, we only present the results with a *fixed IQ dRMT* approach.

¹The number of fixed entries are chosen for a commit width of six instructions.

Parameter	Value	Parameter	Value
<i>Fetch/Decode/Commit Width</i>	6 instructions	<i>FP FUs</i>	3 ALU, 1 Mul/Div
<i>Unified Phy. Register File</i>	128 INT/128 FP entries, 2-cycle acc. lat. 1-cycle inter-subsystem lat.	<i>Int. FUs</i>	4 ALU, 2 AGU 1 Mul/Div
<i>Issue Width</i>	4/2 INT/FP instructions	<i>Issue Queue</i>	48 INT/32 FP Instructions
<i>Branch Predictor</i>	Gshare 4K entries	<i>BTB Size</i>	4K entries, 2-way assoc.
<i>L1 - I-cache</i>	32K, direct-map, 2 cycle latency	<i>L1 - D-cache</i>	32K, 4-way assoc., 2 cycle latency, 2 r/w ports
<i>Memory Latency</i>	100 cycles first word 2 cycle/inter-word	<i>L2 - cache</i>	unified 512K, 8-way assoc., 10 cycles
<i>ROB size</i>	192	<i>LSB size</i>	64 entries

Table 1: Baseline Processor Hardware Parameters for the Experimental Evaluation

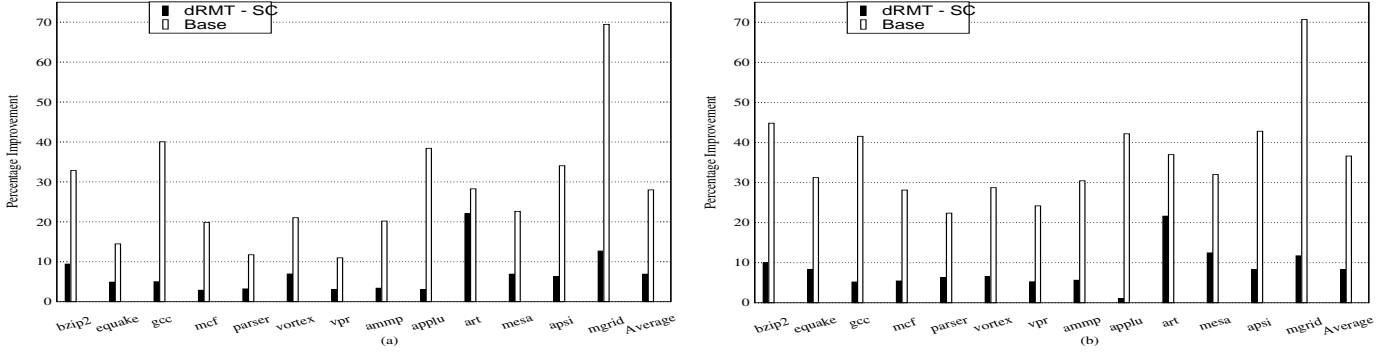


Figure 5: Percentage Increase in IPC for *dRMT-SC* and Base Superscalar Processor for the (a) *non-fixed IQ*; and (b) *fixed IQ*; approaches

4.3 Energy Results

Figure 6 gives the percentage reduction in dynamic energy consumption in the register file, issue queue, and the ROB for the *dRMT-SC fixed IQ* configuration, with respect to the original *dRMT fixed IQ* configuration. These measurements also include the energy consumption in the additional *instruction identifier* bits that need to be written in the ROB. Energy consumption in the register file reduces mainly because of a reduction in the register reads for the “redundant” instructions that are prevented from getting re-executed. Note that almost all the “redundant” instructions read their operands from the register file. Similarly, energy consumption in the issue queue (which also includes the energy consumption in writing the parity bits (refer Section 3.3)) is reduced mainly because of fewer writes and reads to the issue queue. Energy consumption in the ROB is reduced primarily because fewer fields are written into the ROB entry for a self-checking instruction, and the ROB entry of a self-checking instruction is not read for re-dispatch and comparison of results. The small amount of energy consumed in the additional hardware for the *dRMT-SC* configuration (such as the checker for “zero” operands and the additional bits in the inter-pipeline stage latches) is not considered in Figure 6. As seen in Figure 6, about 17% energy savings is achieved in the register file, about 22% in the ROB, and about 13% in the issue queue. Importantly, the *dRMT-SC* configuration reduces the energy consumption impact due to redundant execution by about 23%.

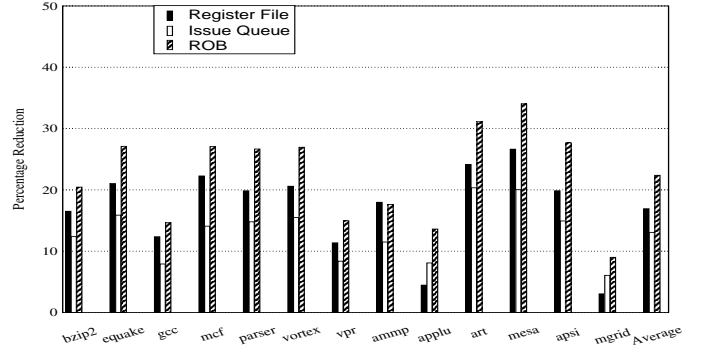


Figure 6: Percentage energy saving in Register File, ROB, and Issue Queue for the *dRMT-SC* configuration

4.4 Vulnerability Impact of Self-checking Instructions

We measured the processor vulnerability of the *dRMT* and the *dRMT-SC* approaches in terms of *ACE* bits [9] in the two approaches. *ACE* bits are the average number of valid bits in a processor per cycle, which when flipped will cause an error in the program output. As discussed in Section 3.3, the errors will be detected in both the approaches. However, even though errors may not affect the correct execution of the application, they will affect the number of times the execution has to be stopped and restarted due to detection

of an error, and hence the performance of the processor. Hence, it is beneficial to reduce the processor vulnerability by reducing the *ACE* bits, especially for higher error rates.

As compared to the original *dRMT* approach, the *dRMT-SC* pipeline of Figure 4(b) will have additional *ACE* bits in checks for “zero” operands. The *dRMT-SC* pipeline will also have additional *ACE* bits in the inter-pipeline stage latches, and in the issue queue. On the other hand, the *dRMT-SC* pipeline has fewer *ACE* bits in the ROB because the entire ROB entry is written only for the non-self-checking instructions and in the issue queue because the self-checking instructions are not re-dispatched to the issue queue for re-execution. The lifetimes of the register values also reduce because the values are no longer required if the consumer instructions are self-checking instructions. In addition, because of an improvement in performance, the values stay for a shorter duration in the issue queue, ROB, register file, load/store buffer etc., further reducing the *ACE* bits. In our experiments, we measure the *ACE* bits every cycle for both the original *dRMT* and the *dRMT-SC* pipelines. In our measurements, we do not include the bits in the memory hierarchy of the processor and in the branch predictor. We observed that the *dRMT-SC* approach reduces the *ACE* bits and hence, the frequency of error occurrence for a given run of program, on an average, by about 5%.

5. SEMI SELF-CHECKING INSTRUCTIONS

We define semi self-checking instruction as an instruction for which a major fraction of its computation is checked when the instruction executes, and the remaining fraction of the computation is checked using re-execution. This technique is based on the observation that many instructions have at least one operand that is of small size. We call the small-sized operand as *small operand*, and the other operand as *normal operand*. We define a *small operand* as one that has a small positive value (1 to 31; we will see later why the number 31 is chosen). If both the operands are small in size, then a *small operand* can be arbitrarily chosen. For such instructions, the upper bits of the computed result are frequently the same as the upper bits of the *normal operand*, and can be checked when the instructions execute. Hence, an instruction is called a semi self-checking instruction if it has a *small operand*, and the upper bits of the result match that of the *normal operand*. Floating-point instructions are not considered semi self-checking instructions.

When implementing semi self-checking instructions in the *dRMT-SC* pipeline (we call this pipeline *dRMT-SC&SSC*), three *instruction identifier* bits are used. If an instruction is self-checking, then it is marked “001” or “010”, and a *semi self-checking instruction* is marked “101” or “110”, depending on the operand number of the *small operand*. Figure 7 shows the backend pipeline stages of the *dRMT-SC&SSC* pipeline. The check for a *small operand* is performed in parallel to the check for a “zero” operand. Note that the immediate values can also be checked for their size at decode. If an instruction is marked “000”, “001” or “010”, the pipeline functions as the one in Figure 4(b). For a *semi self-checking instruction*, the execution control signals, the source identifiers, and the immediate value (if any) are also placed in the ROB entry at dispatch. If the instruction is marked “101” or “110”, and the upper 27 bits of the *normal operand* match that of the result, then the lower five bits of the operands and the result are stored in the 16-bit *immedi-*

ate field in the ROB entry (in the *writeback to ROB* stage). The *immediate field* is chosen to avoid any ROB read port implications. Hence, an operand is considered small if its value is between 1 and 31 because a maximum of five bits are available to store its value. The *PC* field could also be chosen in which case an operand with a larger value can be considered small. It may also happen that the upper 27 bits of the *normal operand* do not match that of the result, in which case the entire result is written into the ROB entry, and the *instruction identifier* bits are reset to “000”. Hence, the *writeback to ROB* stage has been shifted after the *result comparison* in Figure 7. When a *semi self-checking instruction* is re-executed, the five bit operand values are read from the ROB entry, re-executed, and compared with the five bits of the result.

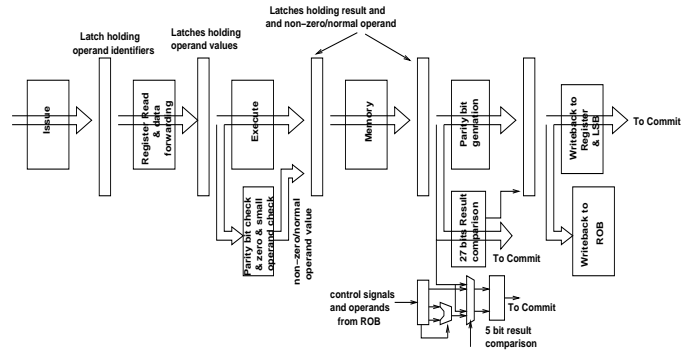


Figure 7: Pipeline for the *dRMT-SC&SSC* configuration

In Figure 7, a separate 5-bit functional unit is used for the re-execution of semi self-checking instructions. To keep this FU simple, we do not consider multiply and divide instructions as semi self-checking instructions. Note that some logic operations such as AND, XOR, etc. that could not be considered as self-checking instructions can be considered as semi self-checking instructions if the higher order bits of the result match those of the *normal operand*. The 5-bit re-execution and comparison can be pipelined, but we assume a single cycle re-execution and comparison because of a very simple 5-bit computation and comparison. An error is detected in a semi self-checking instruction if the 5-bit computation generates a carry or the results do not match. Note that, in the original *dRMT* configuration (Figure 4(a)), a separate functional unit cannot be used for re-execution because the operands need to be read from the register file. A 32-bit comparator in Figure 4(b), is now partitioned into a 27-bit comparator and another 5-bit comparator in Figure 7. The comparators can be used to compare the upper 27 bits for a semi self-checking instruction and the lower 5 bits for a different re-executing semi self-checking instruction (to increase the comparison bandwidth), or the two comparators can be combined to compare the 32-bit results for self-checking and other “redundant” instructions. In case of a conflict, when a 5-bit comparison and a 32-bit comparison are to be performed simultaneously, priority is given to the 32-bit comparison.

Branch instructions that cannot be considered as self-checking instructions (because they perform at least one operation on two non-zero operands — PC and immediate value) can be considered as semi self-checking instructions

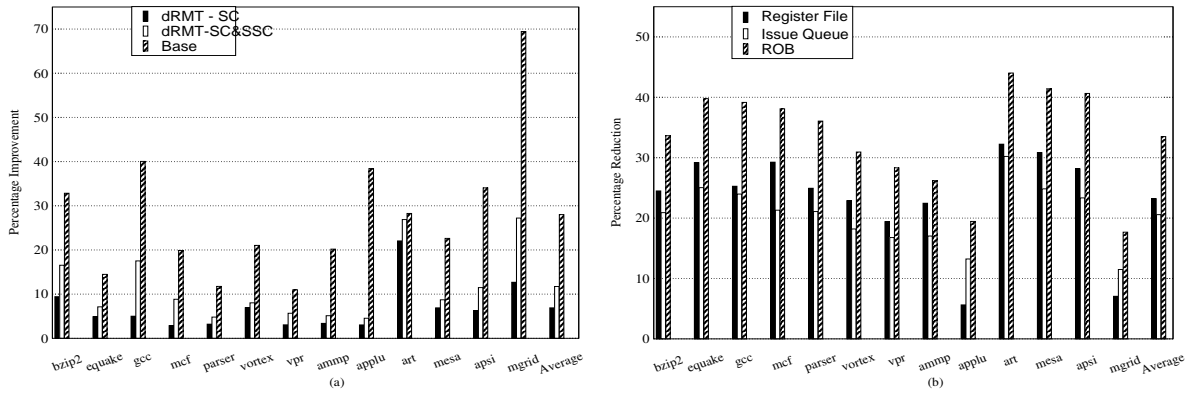


Figure 8: (a) Percentage IPC Improvement for $dRMT-SC$ and $dRMT-SC&SSC$; (b) Percentage reduction in energy consumption for $dRMT-SC&SSC$; with *fixed IQ* configurations and with respect to the original $dRMT$ configuration

if the immediate value (to be added to the PC) is a small positive value. For this one of the operands involved in determining the branch outcome should be *zero*. Hence, for branch instructions implemented as subtraction operation (the outcome of subtraction determining the branch outcome), and having one operand *zero*, errors in the “main” branch instruction outcome evaluation are detected using the self-checking principles and those in target computation using semi-self-checking principles.

Figure 9 presents the percentage of instructions (out of the total instructions) that are potential candidates for self-checking and semi self-checking instructions in the first bar, and the percentage of instructions that are actually saved from re-execution in the second bar. Note that none of the self-checking instructions are re-executed. Figure 9 shows that more than 60% of the instructions can be potentially prevented from re-execution, still maintaining the same fault coverage as the basic $dRMT$ configuration. Overall, about 52% of the instructions are prevented from re-execution.

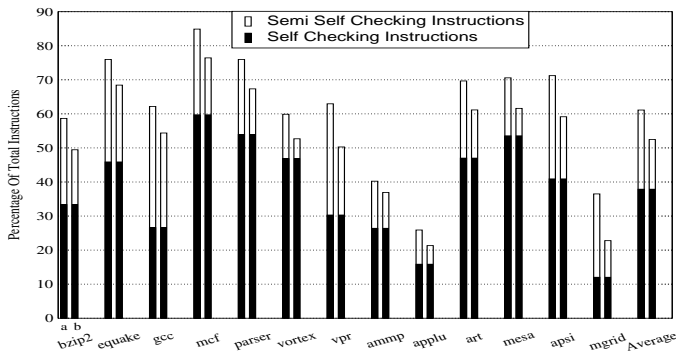


Figure 9: Percentage of total instructions that are (a) Potential candidates; and (b) Actually saved from re-execution; as self-checking and semi self-checking instructions in $dRMT-SC&SSC$

Figure 8(a) presents the percentage IPC improvement with the $dRMT-SC$ and the $dRMT-SC&SSC$ configurations with respect to the original $dRMT$ pipeline. The results are only presented for the *fixed IQ* configuration. It can be seen from Figure 8(a) that the $dRMT-SC&SSC$ configuration gives

about 12% IPC improvement (about 4% more than that of the $dRMT-SC$ configuration) over the original $dRMT$ configuration. In fact, the $dRMT-SC&SSC$ configuration recovers almost 43% of the IPC improvement obtained with the base superscalar processor without redundant execution (about 96% for *art*). Figure 8(b) shows the percentage reduction in the energy consumption in the ROB, the issue queue, and the register file for the $dRMT-SC&SSC$ configuration with respect to the original $dRMT$ pipeline. In going from $dRMT-SC$ to $dRMT-SC&SSC$, the percentage reduction in energy consumption increases from about 17% to about 24% for the register file, from about 13% to about 21% for the issue queue, and from about 22% to about 34% for the ROB. Overall, about 39% of the energy consumption impact (due to redundant execution) in these structures is recovered.

In our experiments so far, only small positive operands (either immediate or register) are considered small. This definition of *small operands* precluded the small negative operands. The higher order bits of a value may remain the same even when operated with a small negative value. Operations with small negative values may be frequently encountered when branching back for a loop, or during address computations for load and store instructions. Hence, we also included the instructions that have at least one small negative operand (value from -1 to -31) and the upper 27 bits of the result match the upper 27 bits of the *normal operand*, as semi self-checking instructions. However, when re-executing the instruction with the five lower bits of a negative operand, there will always be a carry generation for arithmetic operations (if the upper 27 bits remain the same), which will result in an incorrect detection of an error. Hence, for arithmetic instructions, the small negative operand is stored as the corresponding small positive operand, operated with the lower five bits of the instruction’s result and compared with the other *normal operand*’s lower five bits. For instance, if the lower five bits of the *normal operand* signify 25_d (value 25 in decimal), the small negative operand is -5_d , and the instruction is ADD, then the lower five bits of the result signify 20_d . When re-executing this ADD instruction, the ADD operation is performed on the result 20_d with operand 5_d , resulting in the other operand 25_d . For other instructions such as logic and shift operations, the original operands are stored. We call this technique $dRMT-SC&SSCN$. Fig-

Figure 10 shows the percentage of instructions (out of the total instructions) that are potential candidates (and that are actually prevented from re-execution) for self-checking and semi self-checking instructions, including the instructions with small negative operands. As can be seen in Figure 10, an average of about 7% instructions are semi self-checking instructions with small negative operands, and about 86% of such instructions are prevented from re-execution. Interestingly, a higher percentage of semi self-checking instructions (out of the potential candidates) are saved for small negative operands as compared to small positive operands.

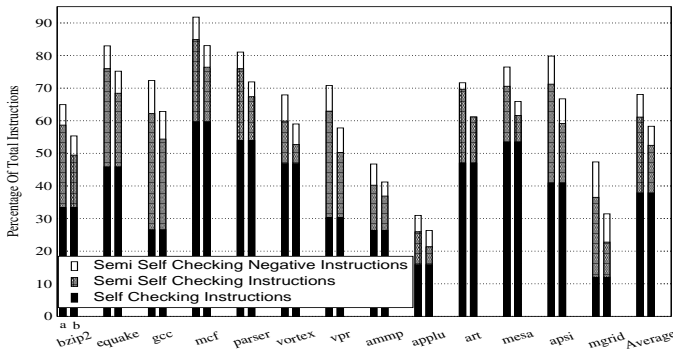


Figure 10: Percentage of Instructions that are (a) Potential candidates; and (b) Actually saved from re-execution; for self-checking and semi self-checking in $dRMT-SC&SSCN$

Figure 11(a) presents the percentage IPC improvement with the $dRMT-SC&SSCN$ configuration with respect to the original $dRMT$ pipeline. An IPC improvement of about 14% is obtained with the $dRMT-SC&SSCN$ configuration over the original $dRMT$ pipeline (about 2% more than the $dRMT-SC&SSC$ configuration). The $dRMT-SC&SSCN$ configuration recovers almost 51% of the IPC improvement obtained with the base superscalar processor without redundant execution. However, Figure 11(b) shows that the percentage reduction in energy consumption for $dRMT-SC&SSCN$ improves only by a very small amount when compared to $dRMT-SC&SSC$. We also observed that the processor vulnerability (in terms of ACE bits) reduced by about 9% with the $dRMT-SC&SSCN$ configuration.

6. SENSITIVITY STUDY

In this section, we measure the IPCs of the $dRMT-SC&SSCN$, the $dRMT$, and the base superscalar configurations as the issue queue size is varied from 40 entries to 160 entries, the issue width is varied from 3 to 8, and the *slack* is varied from 32 to 96 instructions. In these experiments, all the other hardware parameters remain the same. For instance, when changing the issue queue size, *slack* and issue width are also kept at default values. Figure 12 presents the average percentage increase in IPC for the $dRMT-SC&SSCN$ and the base superscalar configurations with respect to $dRMT$. As expected, the percentage improvement of the $dRMT-SC&SSCN$ technique (with respect to original $dRMT$) improves with fewer issue queue entries and issue slots, because the $dRMT-SC&SSCN$ technique relieves the pressure on these resources. The performance difference between $dRMT-SC&SSCN$ and base superscalar also increases with

fewer issue queue entries and issue slots because $dRMT-SC&SSCN$ still increases the pressure on these resources to some extent, whereas the base superscalar does not. However, the *slack* did not have any impact on the performance of the $dRMT-SC&SSCN$ technique as compared to the $dRMT$ configuration.

7. RELATED WORK

Techniques that simultaneously execute multiple copies of the same instructions have been proposed for concurrent error detection and recovery [11, 1, 10, 12, 13, 18, 19, 6]. Ray, Hoe, and Falsafi [11] 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. Smolens et. al. [17] perform studies to measure the performance impact of redundant execution. Both [1, 17] propose an architecture similar to the $dRMT$ architecture in this paper. The fault-tolerant architectures in [12, 13, 19, 6] use the inherent hardware redundancy in simultaneous multithreading and chip multiprocessors for concurrent error detection. Patel and Fung [10] propose transforming the input operands between redundant computations to expose a persistent fault.

Parashar et al. [24] explore instruction reuse to alleviate the ALU bandwidth problem in the SMT-style RMT approach. They don't redundantly execute instructions which were executed with the same input operands before. The techniques proposed in this paper are orthogonal to those in [24]. Gomaa et. al [21] use the same technique to provide imperfect fault coverage by avoiding re-execution of instructions that have the same operand values. In this case, the result of an instruction is checked with that of a previously executed instruction with the same operand values. Sumeet et al. [8] explore optimizing the resources used by the redundant instructions to mitigate the negative effects of RMT.

Shubhendu, et. al. [9] 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 [3].

Researchers [22, 23] have proposed eliminating the *move instructions* (which are ADD instructions with one operand as $R0$) in an instruction stream by renaming the registers. These techniques can only remove a small percentage of a specific kind of instruction that uses $R0$ as an operand. The techniques in this paper avoid redundant execution of many more instructions than just the *move instructions*.

8. CONCLUSION

Reliability in systems is usually ensured by corroborating the results of redundant threads. Redundant thread execution has a significant impact on the performance and energy consumption of the processor.

In this paper, we investigate techniques to reduce instruction redundancy – the instructions that are redundantly executed for concurrent error detection – to mitigate the negative effects of redundancy. For this, we define two categories of instructions – self-checking and semi self-checking. Self checking instructions are those instructions whose results

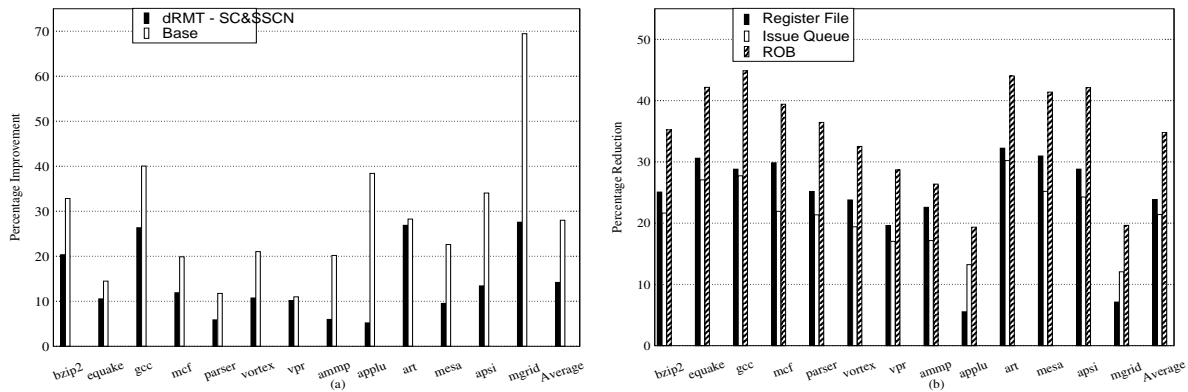


Figure 11: (a) Percentage IPC Improvement; (b) Percentage reduction in energy consumption; for *dRMT-SC&SSCN* with respect to the original *dRMT* configuration

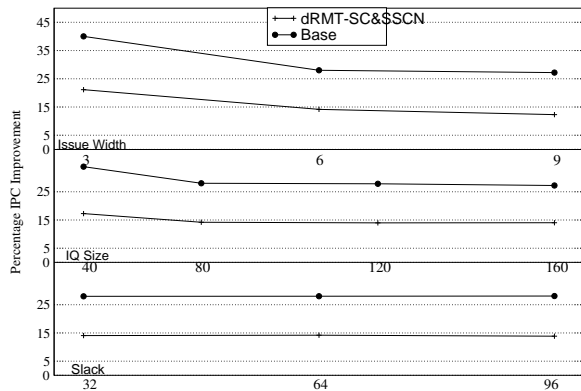


Figure 12: Percentage IPC Improvement of *dRMT-SC&SSCN* and Base Configurations, with respect to original *dRMT*, as machine parameters are varied

are checked for any errors when their “main” copies are executed. Semi self-checking instructions are those instructions for which a major part of their results is checked when the “main” copies are executed, and the remaining part of the instructions is checked using a small amount of additional hardware. Our experiments showed that almost about 58% of the instructions were prevented from re-execution, while still having the same fault coverage as the original processor. With the prevention of redundant execution of these instructions, the performance impact was reduced by about 51%, and the energy consumption impact in the key structures was reduced by about 40%.

9. REFERENCES

- [1] T. Austin, “DIVA: a reliable substrate for deep submicron microarchitecture design,” *Proc. Micro-32*, 1999.
- [2] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” *Computer Arch. News*, 1997.
- [3] J.A. Butts and G. Sohi, “Dynamic dead instruction detection and elimination,” *ASPLOS*, 2002.
- [4] Compaq Computer Corp., “Data integrity for Compaq Non-Stop Himalaya servers,” <http://nonstop.compaq.com>, 1999.
- [5] J. G. Holm, and P. Banerjee, “Low cost concurrent error detection in a VLIW architecture using replicated instructions” *Proc. ICPP-21*, 1992.
- [6] M. Goma, et. al., “Transient-Fault Recovery for Chip Multiprocessors,” *Proc. ISCA-30*, 2003.
- [7] M. K. Gowan, et. al., “Power Considerations in the Design of the Alpha 21264 Microprocessor,” *Proc. DAC*, 1998.
- [8] S. Kumar, et. al., “Reducing Resource Redundancy for Concurrent Error Detection Techniques in High Performance Microprocessor,” *Proc. HPCA*, 2006.
- [9] S. Mukherjee, et. al., “A Systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” *Micro-36*, 2003.
- [10] J. H. Patel, and L. T. Fung, “Concurrent error detection in ALU’s with recomputing with shifted operands,” *IEEE Transactions on Computers*, 31(7):589-595, July 1982.
- [11] J. Ray, J. Hoe, and B. Falsafi, “Dual use of superscalar datapath for transient-fault detection and recovery,” *Proc. Micro-34*, 2001.
- [12] S. Reinhardt, and S. Mukherjee, “Transient fault detection via simultaneous multithreading,” *Proc. ISCA-27*, June 2000.
- [13] 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.
- [14] P. Shivakumar, and N. Jouppi, “CACTI 3.0: An Integrated Cache Timing Power, and Area Model,” *Technical Report, DEC Western Research Lab*, 2002.
- [15] D. P. Siewiorek and R. S. Swarz, “Reliable Computer Systems Design and Evaluation,” *The Digital Press*, 1992.
- [16] T. J. Slegel, et. al. “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, 19(2):12-23, March/April 1999.
- [17] J. Smolens, et. al., “Efficient Resource sharing in Concurrent error detecting Superscalar microarchitectures,” *Proc. Micro-37*, 2004.
- [18] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, “Slipstream processors: Improving both performance and fault tolerance,” *In Proc. Micro-33*, December 2000.
- [19] T. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” *Proc. ISCA-29*, 2002.
- [20] C. Weaver, et. al., “Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor,” *Proc. ISCA-31*, 2004.
- [21] M. Goma and T. N. Vijaykumar, “Opportunistic Transient-Fault Detection,” *Proc. ISCA-32*, 2005.
- [22] V. Petric, et. al., “Reno: A Rename-Based Instruction Optimizer,” *Proc. ISCA-32*, 2005.
- [23] B. Fahs, et. al., “Continuous Optimization,” *Technical Report, UIUC*, 2004.
- [24] A. Parashar, et. al., “A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy” *ISCA-31*, 2004.