November 2015

Modifying Instruction Sets In The Gem5 Simulator To Support Fault Tolerant Designs

Chuan Zhang
University of Massachusetts Amherst

MODIFYING INSTRUCTION SETS IN THE GEM5 SIMULATOR

TO SUPPORT FAULT TOLERANT DESIGNS

A Thesis Presented

by

CHUAN ZHANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2015

Electrical and Computer Engineering

MODIFYING INSTRUCTION SETS IN THE GEM5 SIMULATOR

TO SUPPORT FAULT TOLERANT DESIGNS

A Thesis Presented

by

CHUAN ZHANG

Approved as to style and content by:

_____

Israel Koren, Chair

_____

C.Mani Krishna, Member

_____

Sandip Kundu, Member

_____

Christopher V. Hollot, Department Head as
Department of Electrical and Computer Engineering

To my parents.

ACKNOWLEDGMENTS

I would like to thank Gem.org for providing the platforms for all Gem5 users and developers to communicate and discuss.

I would also like to thank Professor Koren for his guidance and support over the years.

I would also like to thank my girlfriend Jane Teergele, who supports me in spirit over the years.

ABSTRACT

MODIFYING INSTRUCTION SETS IN THE GEM5 SIMULATOR TO SUPPORT FAULT TOLERANT DESIGNS

SEPTEMBER 2015

CHUAN ZHANG

B.S., BEIJING INSTITUTE OF TECHNOLOGY

M.S.M.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren

Traditional fault tolerant techniques such as hardware or time redundancy incur high overhead and are inefficient for checking arithmetic operations. Our objective is to study an alternative approach of adding new instructions to check arithmetic operations. These checking instructions either rely on error detecting code or calculate approximate results and consequently, consume much less execution time. To evaluate the effectiveness of such an approach we wish to modify several benchmarks to use checking instructions and run simulation experiments to find out their execution time and memory usage. However, the checking instructions are not included in the instruction set and as a result, are not supported by current architecture simulators. Therefore, another objective of this thesis is to develop a method for inserting new instructions in the Gem5 simulator and cross compiler. The insertion process is integrated into a software tool called Gtool. Gtool can add an error checking capability to C programs by using the new instructions.

Keywords: Gem5, compiler, error checking, ISA modification.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Objectives

Instruction set modification can be a significant challenge. For example, in the development of embedded systems where a standard Instruction Set Architecture (ISA) is often not optimal, Peymandoust et al. developed a methodology to automatically add new instructions to the Tensilica's ISA to reduce the execution time [26].

However, the majority of processor simulators and compilers do not support modified ISAs. Gem5, one of the most popular processor simulators, only supports six standard instruction sets. Cross compilers have a similar situation, and there is almost no prior work on modifying ISA in cross compilers.

In this project, we developed a new software, Gtool that allows the insertion of new instructions into a given ISA. The new instructions can be inserted into the Alpha ISA or the MIPS ISA automatically by Gtool.

Our main objective in developing Gtool is adding instructions for real-time checking of arithmetic operations. In this project, integer checking instructions rely on the residue number system, while floating-point checking instructions use truncated floating-point values. The checking procedures which include checking instructions and comparisons of the results are added to target programs by Gtool. The resulting fault-tolerant target programs may have lower overhead when compared with traditional redundancy techniques.

1.2 Related Work

Bloom presented a method of adding pseudo-instructions to Gem5 in his blog [17]. He provided a tutorial on how to add instructions to the x86 ISA and used the new instructions only in full-system simulations.

However, adding pseudo-instructions is not the same as ISA modification. Pseudo instructions are not currently supported by all types of instruction sets. Only the x86 ISA has full support for adding such instructions. Moreover, although these new instructions can be functionally regarded as actual instructions, the execution of pseudo-instructions is still different from the execution of the original instructions. In his blog, he conceded that pseudo-instructions cannot be integrated tightly with the pipeline [17]. In addition, these pseudo-instructions can only use the reserved opcodes whose number is limited. In conclusion, pseudo-instructions cannot be used for the purpose of adding error checking.

Some efforts have also been made to add customized instructions to GCC (GNU compiler collection). However, these efforts have not produced good results. One reason is that the target ISA (PISA) is not widely used [20]. Secondly, GCC cannot use the new instructions as it was not designed to use them. Instead, the user must manually insert the instructions into the inline assembly syntax.

Eibl et al. proposed the use of reduced precision floating-point values to check floating-point operations [2]. They also discussed the differences between the reduced precision results and the precise results. However, their research only focused on comparing the result of a reduced precision addition to the corresponding exact result.

Lipetz et al. studied the application of residue check to floating-point operations where the mantissa addition is checked [6]. They discussed hardware implementations and fault detection coverage of different moduli. Their research focused on reducing the cost of hardware redundancy in terms of power consumption and chip area.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 introduces the simulator and compiler. In Chapter 3, the insertion of new instructions into the cross compiler and Gem5 simulator is explained. Chapter 4 presents the parameters of the experiments, including those of the simulator and workloads. Chapter 5 presents the results of the experiments. Finally, conclusions are presented in Chapter 6.

CHAPTER 2

BASIC BACKGROUND

2.1 Compiler and simulators

In this project, Gem5 is used as the base simulator, and crosstool-NG is used to build cross compilers with GCC and Binutils.

The main reason for choosing Gem5 is that it is the most popular simulator for computer architecture research. Besides, it is a modular discrete event-driven simulator platform, which can be rearranged, parameterized, extended or replaced easily to suit project requirements [24]. Furthermore, Gem5 supports several instruction sets including Alpha, ARM, MIPS, x86, POWER and SPARC. However, these instruction sets are not equally supported in Gem5. Among these six ISAs, Alpha is the most supported one and therefore it is one of the target ISAs in our project. Table 2.1 compares Gem5 with two other popular simulators, SimpleScalar and SESC.

	SimpleScalar	SESC	Gem5
Multicore supported	No	Yes	Yes
Supported ISA	Alpha, PISA	MIPS	Alpha, x86, ARM, SPARC, PowerPC, MIPS
ISA modification	No	Yes	Only pseudo-instructions in full-system Mode

Table 2.1 Comparison of three simulators

Gem5 supports full-system and system-call modes. The operating system needs to be loaded in full-system mode simulation. On the other hand, in system-call mode simulations, system services are called only when necessary. In this project, all simulations were performed in the system-call mode.

Crosstool-NG is a software tool that is used to build cross compilers for multiple architectures. We use crosstool-NG to build cross compilers in this project. However, the recommended configuration does not work for Alpha ISA. The cross compiler for Alpha ISA in this project was built with a configuration that we have developed.

2.2 Decode process in Gem5

In Gem5, the decoding process cannot be done in a single step. It needs multiple steps that involve different parts of the instruction set structure. The instruction set structure consists of a decoder section and a declaration section, as shown in Figure 2.1. The decoder section describes the decoding process and functional behavior by providing entries for all types of instructions. It classifies and extracts the variable values from the machine code, then assigns these values to the simulator, while the declaration section explains the details of the simulations. For example, the decoder section can recognize the machine code 0x01002240 as addl $1, $2, $1. Then, it transfers addl $1, $2, $1 to the IntegerOperate part of the simulator since this is an integer arithmetic operation. In other words, the ISA description works like a dictionary for simulators. The decoder file is an index for the dictionary, while the declaration sections are definitions of words.

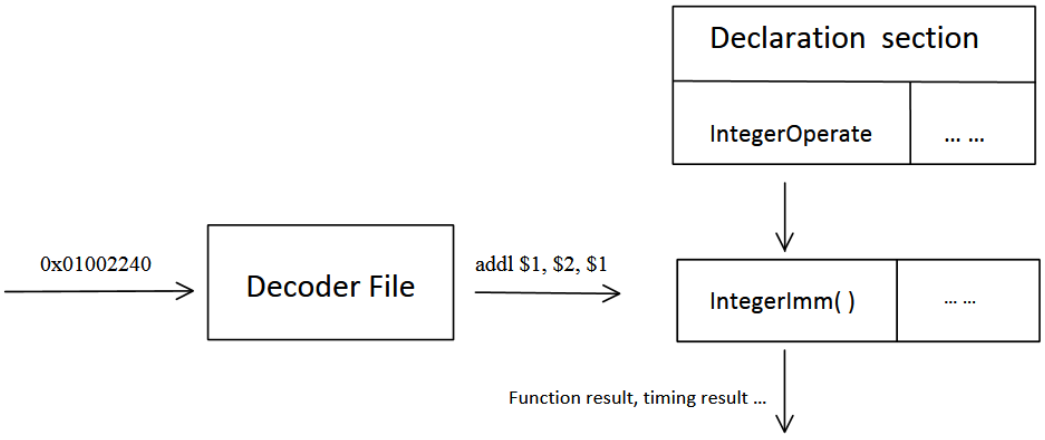

Figure 2.1 Decoding process in Gem5 (Alpha)

The declaration section defines the functionality of multiple types of instructions. In Gem5, each instruction has a unique format that is defined in the declaration section. Since the existing formats cover all types of instructions, we do not modify the declaration section in the project.

Both the decoder file and the declaration sections are written in the M5 ISA description language. This language is used for describing instruction sets and generating C++ code for simulations.

CHAPTER 3

APPROACH TAKEN

3.1 Introduction

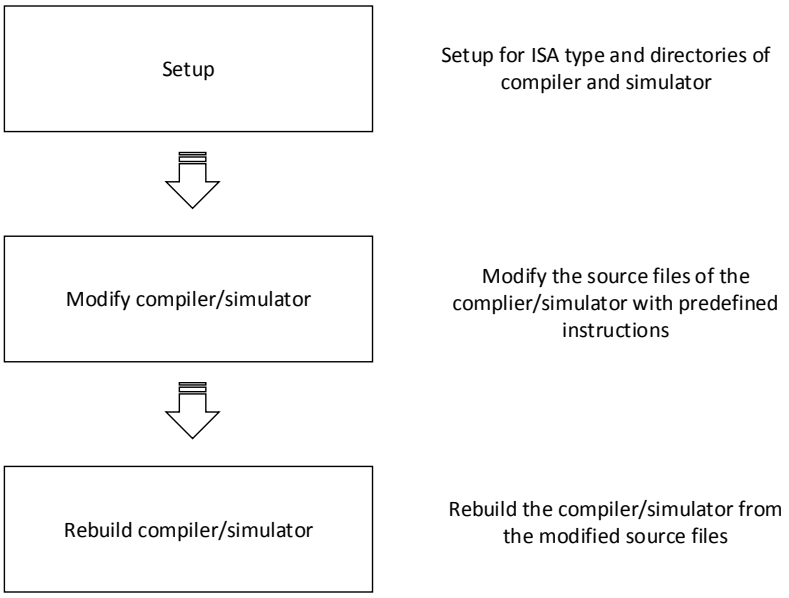

Figure 3.1 Steps for modifying the compiler and simulator

Modifying the source code is necessary in order to use the modified compiler and simulator. Since the modified source code cannot be directly processed by the compiler and simulator, it is necessary to rebuild the compiler and the simulator after modifying them as shown in Figure 3.1. Rebuilding a cross compiler is very time-consuming. It takes hours to rebuild the cross compiler for Alpha by using crosstool-NG. The rebuilding time of Gem5 depends on the amount of changes and can vary from minutes to half an hour.

Gtool users need to input the name, opcode, type and other information about the new instructions to both simulators and instructions. Gtool checks all inputs and then

modifies the source code of Gem5 and cross compiler. Figure 3.2 shows the human-computer interface of Gtool.

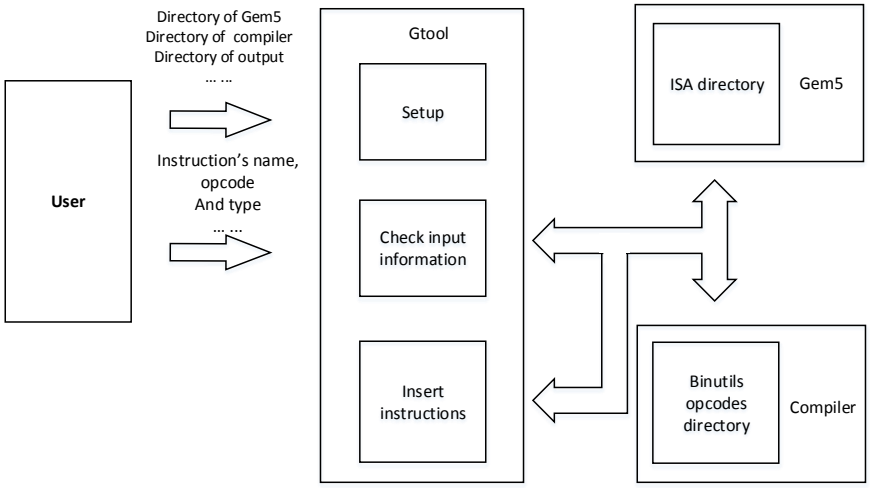

Figure 3.2 Human-computer interactions

The next part of this chapter discusses the approach taken with the cross compiler and simulator.

3.2 Insertion in the compiler

There are two types of inputs that need to be provided to the compiler during instruction insertion. One is the opcode of the instruction. The other is the name of the instruction.

Insertion is the first step of using a new instruction in the cross compiler. Since the compiler is not optimized with the new instruction, the instruction can only be used in inline assembly code. In this project, a new method of using new instructions is used for real-time checking.

Our approach combines source-code modification and insertion into the cross compiler to make use of the new instructions. The steps of compiling a test program with the new instructions are shown in Figure 3.3.

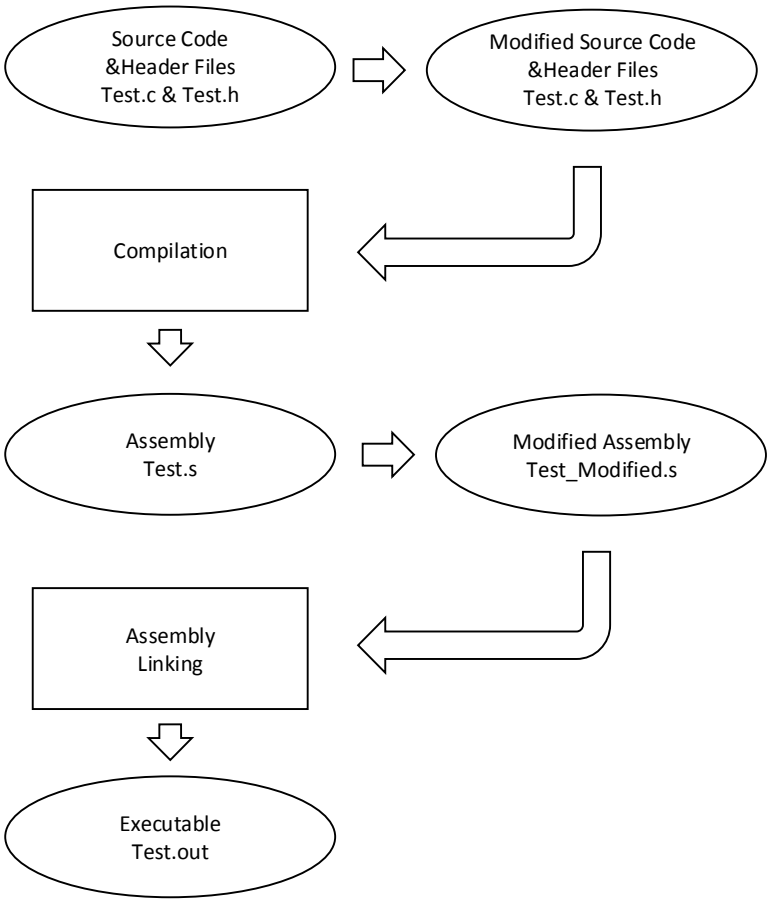

Figure 3.3 Steps of compiling a test program with new instructions

Target source code are modified by inserting a duplicate operation after each to be checked operation. Then, the modified code is compiled to assembly code. The next step is to replace the assembly line for the duplicate operation with a new instruction. The linker then links the modified assembly file to generate an executable program. In this process, the compiler only needs to know how to translate new instructions to machine code. It does not need to be aware of the functionality of the new instructions.

3.2.1 Insertion in the cross compiler

In this project, Binutils is used as the assembler and linker for the cross compiler. It stores the opcode of the supported architectures in a directory named *opcode* under the root directory of Binutils' source code. The opcode for the Alpha instructions are defined in alpha-opc.md. Therefore, adding instructions requires adding lines in alpha-opc.md. Then, the cross compiler needs to repackage the source code of Binutils and replace the original package with the new one for crosstool-NG.

3.2.2 Source-code modification

Source-code modification consists of two parts: C code modification as shown in Figure 3.4 and assembly code modification.

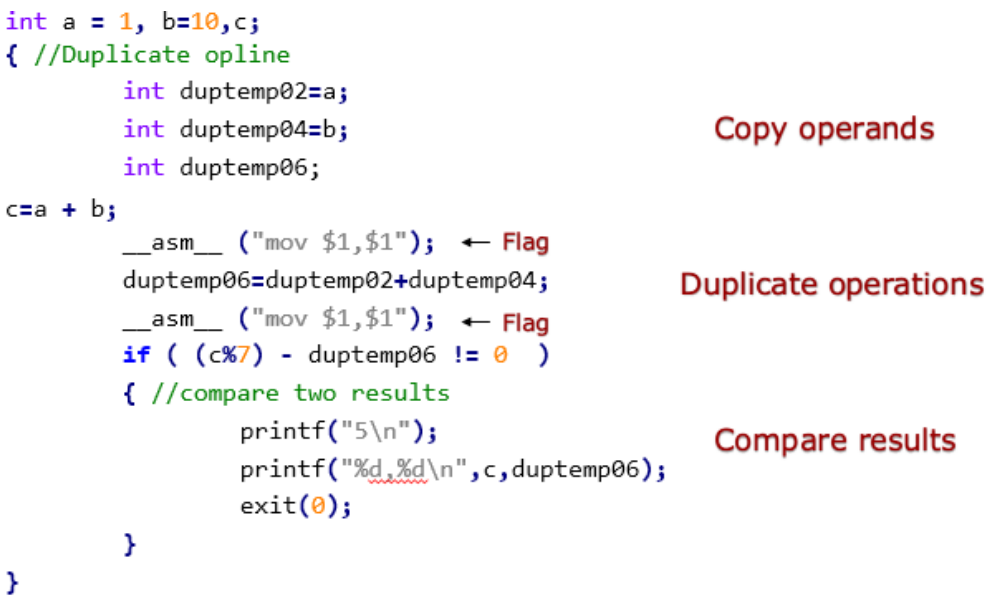

Figure 3.4 C code modifications

In the C code modification, the Gtool first defines new variables and copies values from the original variables to the new variables. Afterward, Gtool uses the original computation statements. (c=a+b; in Figure 3.4). Then, Gtool insert a flag followed by the

duplicated operations with new variables. Finally, an *if* statement is used to compare the two results. If the two results do not match (result matching does not mean that the results are identical, instead, it means that either the residues match or the difference between the results is smaller than the maximum allowed), the program will be terminated.

The assembly code that stands for *moving a value from register one to register one* is used as a flag since it is easily recognizable and does not affect the results of the program.

Gtool uses this flag to locate the instruction that needs to be replaced by a checking instruction.

3.2.3 Checking arithmetic operations

The checking method for integer operations is based on the residue number system. For integers X, Y and m, the following equations hold [14]:

$$|\boldsymbol{X}+\boldsymbol{Y}|_{\boldsymbol{m}} = ||\boldsymbol{X}|_{\boldsymbol{m}} + |\boldsymbol{X}|_{\boldsymbol{m}}|_{\boldsymbol{m}} = |\boldsymbol{x}+\boldsymbol{y}|_{\boldsymbol{m}}$$

$$|\boldsymbol{X}-\boldsymbol{Y}|_{\boldsymbol{m}} = ||\boldsymbol{X}|_{\boldsymbol{m}} - |\boldsymbol{X}|_{\boldsymbol{m}}|_{\boldsymbol{m}} = |\boldsymbol{x}-\boldsymbol{y}|_{\boldsymbol{m}}$$

$$|\boldsymbol{X}\times\boldsymbol{Y}|_{\boldsymbol{m}} = ||\boldsymbol{X}|_{\boldsymbol{m}} \times |\boldsymbol{X}|_{\boldsymbol{m}}|_{\boldsymbol{m}} = |\boldsymbol{x}\times\boldsymbol{y}|_{\boldsymbol{m}}$$

where, $|X|_m = x$ is the residue of X modulo m.

In the inserted comparison, integer results are checked by comparing the residues of the original result and the duplicate one.

Integer division constitutes a special case. Even though the result of the division X/Y can be checked through

$$|\boldsymbol{X}|_{\boldsymbol{m}} = |\boldsymbol{Y}|_{\boldsymbol{m}} \times |\boldsymbol{Q}|_{\boldsymbol{m}} + |\boldsymbol{R}|_{\boldsymbol{m}}$$

where Q is the quotient and R is the remainder. The checking result cannot be done in a C program as the remainder R is not made available. Therefore divisions can be only checked by recalculation.

It is more complicated to check floating-point results. It is obvious that the results of truncated floating-point operations are different from the original results. The question is to determine whether these differences are due to the truncation or real errors. This requires calculating an upper bound for the truncation error. The upper bound can be viewed as the reference difference. Results that have a smaller difference than the reference are marked as correct.

We next explain the truncation procedure and derive the reference difference.

Figure 3.5 Structure of truncated floating-point values

The truncated floating-point value keeps the sign bit, exponent bits and part of mantissa bits as shown in Figure 3.5. We denote by n the number of fraction bits in the truncated mantissa. In single precision, n = 8, which is the same as was used in [2]. In double precision, n = 20 as this would truncate a double precision value from 64 bits to 32 bits. We denote by ε the difference between the precise and the truncated values.

$$\varepsilon = Result_{precise} - Result_{truncated}$$

In practice, the relative value of the difference compared to the *Result* is more important. Therefore, we will estimate $\varepsilon/Result_{precise}$ instead of ε. In most cases, ε is very small compared to $Result_{precise}$. For convenience, we use $\varepsilon/Result_{truncated}$ which is almost equal to $\varepsilon/Result_{precise}$.

According to the definition of floating-point values, the original floating-point value and the truncated one can be written as:

$$\mathrm{F}_{truncated} = (-1)^{sign} \cdot 2^{exp} \cdot 1.f_{truncated}$$

$$\mathrm{F}_{precise} = (-1)^{sign} \cdot 2^{exp} \cdot 1.f_{truncated} +$$

$$(-1)^{sign} \cdot 2^{exp-n} \cdot 0.f_{removed}$$

where

$$1 < 1.f_{truncated} < 2$$

$$0 < 0.f_{removed} < 1$$

and *exp* is the exponent, $1.f_{truncated}$ is the truncated mantissa and $0.f_{removed}$ is the part of the mantissa that was removed during the truncation. As a consequence, the exponent of the second part of $\mathrm{F}_{precise}$ is much smaller than the first part. For convenience, we use $\mathrm{F}_{removed}$ to denote the difference between $\mathrm{F}_{precise}$ and $\mathrm{F}_{truncated}$.

$$\mathrm{F}_{removed} = (-1)^{sign} \cdot 2^{exp-n} \cdot 0.f_{removed}$$

Since

$$0.f_{removed} < 1$$

then

$$|\mathrm{F}_{removed}| < 2^{exp-8}\ (Single)\ or\ 2^{exp-20}\ (Double)$$

The exact value of this upper bound depends on the type of operation performed and is analyzed for each operation separately.

Addition and subtraction:

Denote by A and B the operands and by R the result.

$$R_{precise} = A_{precise} + B_{precise}$$

$$= A_{truncated} + A_{removed} + B_{truncated} + B_{removed}$$

$$= R_{truncated} + A_{removed} + B_{removed}$$

$$\varepsilon = R_{precise} - R_{truncated} = A_{removed} + B_{removed}$$

If A and B have the same sign, R will also have the same sign. Then,

$$\varepsilon = 2^{exp_A - n} \cdot 0.f_{A\_removed} + 2^{exp_B - n} \cdot 0.f_{B\_removed}\ (1)$$

$$|R_{truncated}| = 2^{exp_R} \cdot 1.f_{R\_truncated}$$

The relative error is

$$\frac{\varepsilon}{|R_{truncated}|} = 2^{exp_A - n - exp_R} \cdot \frac{0.f_{A\_removed}}{1.f_{R\_truncated}} + 2^{exp_B - n - exp_R} \cdot \frac{0.f_{B\_removed}}{1.f_{R\_truncated}}$$

Both $\frac{0.f_{A\_removed}}{1.f_{R\_truncated}}$ and $\frac{0.f_{B\_removed}}{1.f_{R\_truncated}}$ are less than 1. The equation becomes

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n}(2^{exp_A - exp_R} + 2^{exp_B - exp_R})\ (2)$$

Since

$$exp_R \geq exp_A$$

and

$$exp_R \geq exp_B$$

The term in the parenthesis in (2) is always smaller than 2.

Equation (2) becomes

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n+1}\ (0.78\%\ Single\ or\ 0.00019\%\ Double)$$

When A and B have different signs, A+B will be performed as subtraction. Then assuming that A > B.

$$\varepsilon < 2^{exp_A - n} \cdot 0.f_{A\_removed} - 2^{exp_B - n} \cdot 0.f_{B\_removed}\ (4)$$

Similar to the addition case, when $R_{truncated} \neq 0$

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n}(2^{exp_A - exp_R} - 2^{exp_B - exp_R})$$

If $exp_A = exp_B$, (4) results in

$$\varepsilon < 2^{-n}2^{exp_A}(0.f_{A\_removed} - 0.f_{B\_removed})$$

Clearly

$$\varepsilon < 2^{-n}2^{exp_A}0.f_{A\_removed} < 2^{-n}2^{exp_A}1.f_{A\_truncated} = 2^{-n}|A_{truncated}|$$

Since A>B, then $|R_{truncated}| < |A_{truncated}|$

The result indicates that errors in the mantissa bits of the result of subtraction cannot be detected under the worst case ($A \approx B$).

If $exp_A = exp_B + 1$, then exp_R is equal to either exp_A or $exp_A - 1$

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n}(2^{exp_A - exp_R}) < 2^{-n}2^1 < 2^{-n+1}$$

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n+1}\ (0.78\%\ Single\ or\ 0.00019\%\ Double)$$

If $exp_A \geq exp_B + 2$, in which case $2^{exp_A} = 2^{exp_R}$ and $2^{exp_B} \ll 2^{exp_R}$, then

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n}(0.39\%\ Single\ or\ 0.000095\%\ Double)$$

Multiplication

Let the product of A and B be denoted by R.

$R_{precise} = A_{precise}B_{precise}$

$= (A_{truncated} + A_{removed})(B_{trunated} + B_{removed})$

$= A_{truncated}B_{trunated} + A_{truncated}B_{removed} + A_{deducted}B_{removed} + A_{removed}B_{removed}$

$= R_{truncated} + A_{truncated}B_{removed} + A_{deducted}B_{removed} + A_{removed}B_{removed}$

then

$\varepsilon = R_{precise} - R_{truncated}$

$= A_{truncated}B_{removed} + A_{removed}B_{trunated} + A_{removed}B_{removed}$

Since $A_{removed}B_{removed}$ is a very small value compared to the other two, the equation can be simplified to the following:

$$\varepsilon \approx A_{truncated}B_{removed} + A_{removed}B_{truncated}$$

$$\varepsilon = 2^{exp_A+exp_B-n}1.f_{A\_truncated}0.f_{B\_removed} + 2^{exp_A+exp_B-n}0.f_{A\_removed}1.f_{B\_truncated}$$

Since

$$R_{truncated} = 2^{exp_A+exp_B}1.f_{A\_truncated}1.f_{B\_truncated}$$

then

$$\frac{\varepsilon}{|R_{truncated}|} = 2^{-n}(\frac{0.f_{B\_removed}}{1.f_{B\_truncated}} + \frac{0.f_{A\_removed}}{1.f_{A\_truncated}})$$

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n+1}(0.78\%\ Single\ or\ 0.00019\%\ Double)$$

Division

Assume

$$R_{precise} = \frac{A_{precise}}{B_{precise}}$$

then

$$R_{precise} = \frac{A_{truncated} + A_{removed}}{B_{truncated} + B_{removed}} = \frac{A_{truncated}}{B_{truncated} + B_{removed}} + \frac{A_{removed}}{B_{truncated} + B_{removed}}$$

where

$$\frac{A_{removed}}{B_{truncated} + B_{removed}} = \frac{2^{exp_A-n}0.f_{A\_removed}}{2^{exp_B}1.f_{B\_truncated} + 2^{exp_B-n}0.f_{B\_removed}}$$

$$= 2^{-n}2^{exp_R}\frac{0.f_{A\_removed}}{1.f_{B\_truncated} + 2^{-n}0.f_{B\_removed}}$$

$$\frac{A_{removed}}{B_{truncated} + B_{removed}} < 2^{-n} 2^{exp_R} < 2^{-n} |R_{truncated}|$$

The other factor which causes the difference between the precise and truncated results is the divisor's truncation.

$$\frac{A_{truncated}}{B_{truncated} + B_{removed}} = \frac{R_{truncated}}{1 + \frac{B_{removed}}{B_{truncated}}}$$

Since

$$\varepsilon = R_{precise} - R_{truncated} \approx R_{truncated} \left(\frac{1}{1 + \frac{B_{removed}}{B_{truncated}}} - 1 \right) + \frac{A_{removed}}{B_{truncated} + B_{removed}}$$

(5)

and

$$\left(\frac{1}{1 + \frac{B_{removed}}{B_{truncated}}} - 1 \right) < 0$$

then

$$\frac{\varepsilon}{|R_{truncated}|} < 2^{-n} (0.39\% \; Single \; or \; 0.000095\% \; Double)$$

The upper bounds for the relative differences between the truncated and precise results are summarized in Table 3.1.

$\left\lvert\frac{\varepsilon}{R_{truncated}}\right\rvert_{upperbound}$	Single (17/32 bits)	Double (32/64 bits)
Addition	2^{-7}	2^{-19}
Subtraction	$2^{-8}/2^{-7}/1$	$2^{-20}/2^{-19}/1$
Multiplication	2^{-7}	2^{-19}
Division	2^{-8}	2^{-20}

Table 3.1 Upper bounds for the relative precision loss.

Estimated upper bounds based on experiments

In order to verify the upper bounds for the relative precision loss, we set up experiments to find out the distribution of the precision loss. The experiments generate random floating-point values and calculate the results of addition, subtraction, multiplication and division, for precise and truncated operations. The relative differences between the precise and the truncated results were calculated. The following charts show the distributions of the relative precision loss. Each of the charts is based on results of 40, 000 operations.

The vertical axis is the frequency of the values and the horizontal axis is the relative precision loss. All the floating-point values are double precision. The dotted red line in each chart is the corresponding regression curve.

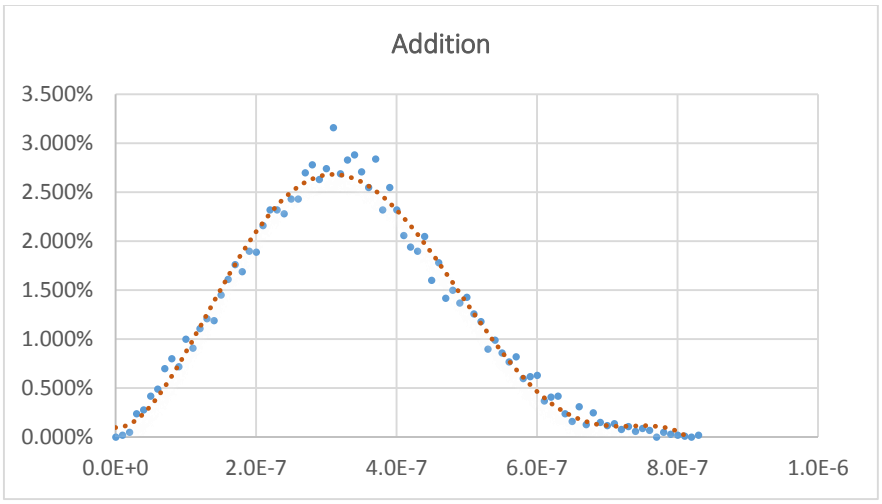

Figure 3.5 Distribution of relative precision loss in additions

It is obvious that the range of precision loss in addition is between 0 and 0.000095 in Figure 3.5. The observed upper bound is half of the calculated upper bound.

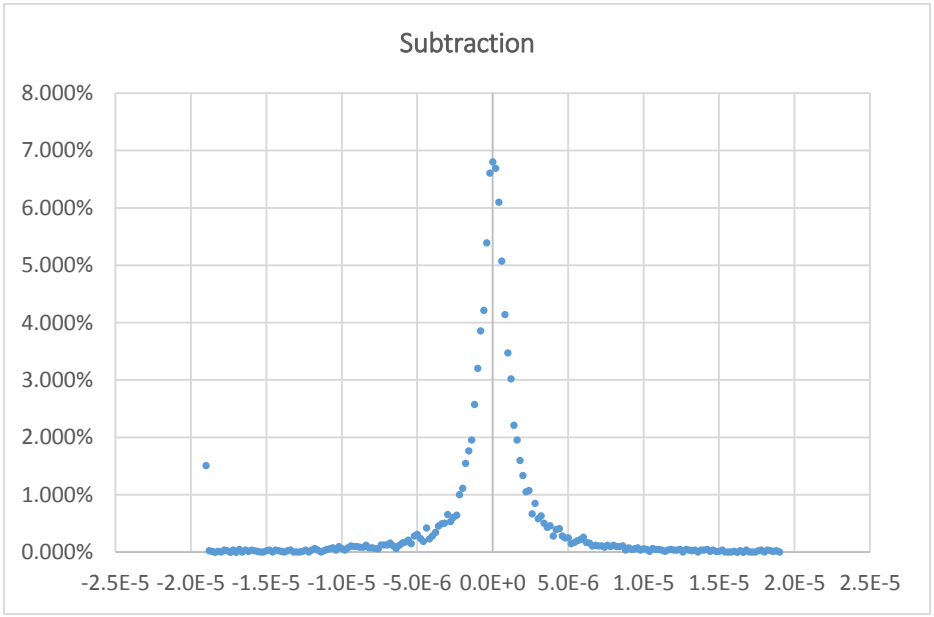

Figure 3.6 Distribution of relative precision loss in subtractions

The range of precision loss of subtraction is considerably larger than for other operations. Still, the frequencies are sharply decreased away from 0 and the values that are far away from zero are not displayed because of the low frequency.

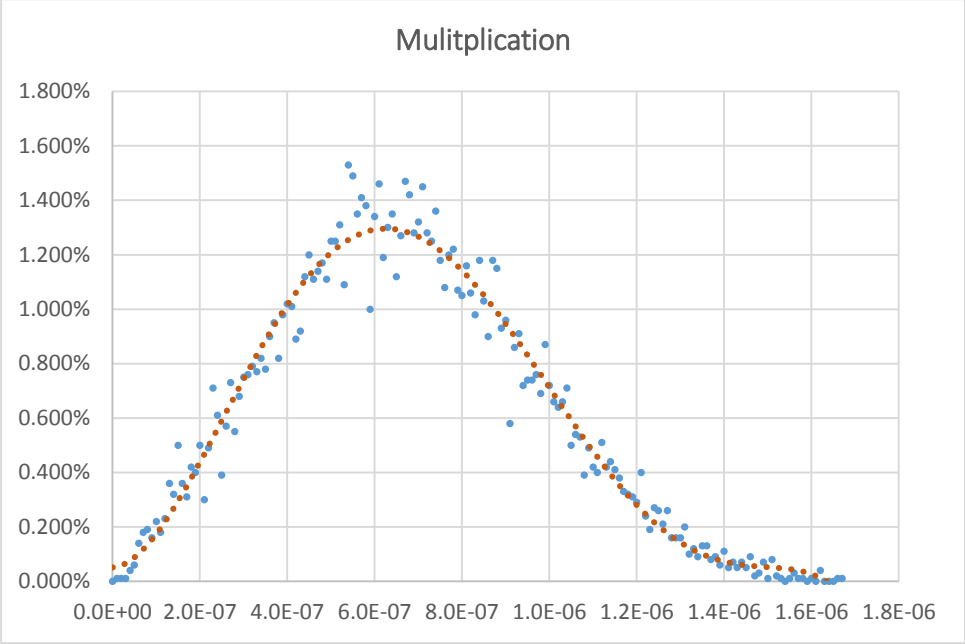

Figure 3.7 Distribution of the relative precision loss in multiplications

Figure 3.7 shows the relative precision loss distribution for multiplication. The range of the relative loss is from 0 to 0.00019%. This is equal to the upper bound we have calculated.

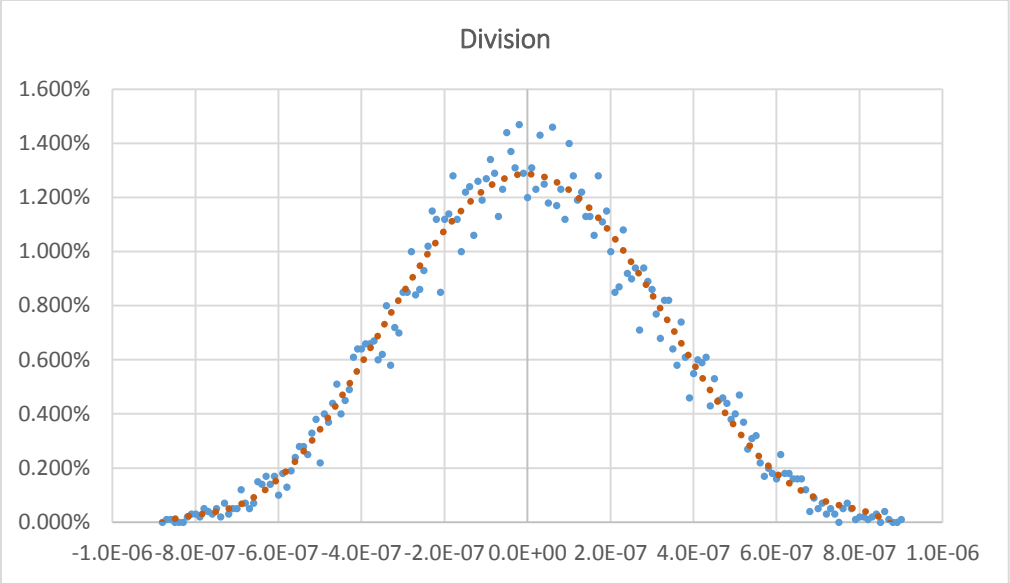

Figure 3.8 Distribution of the relative precision loss in divisions

Similar to addition, the upper bound for the relative precision loss in division can be narrowed to half of the calculated value. The range of the relative precision loss for division is from -0.000095% to +0.000095%.

Based on the above analyses, the range of the relative precision loss based on the experiments is shown in Table 3.2.

$\frac{\varepsilon}{R_{truncated}}$	Single (17/32 bits)	Double (32/64 bits)
Addition	$0 \sim 2^{-8}$	$0 \sim 2^{-20}$
Subtraction	$-2^{-8}/2^{-7}/1 \sim +2^{-8}/2^{-7}/1$	$-2^{-20}/2^{-19}/1 \sim +2^{-20}/2^{-19}/1$
Multiplication	$0 \sim 2^{-7}$	$0 \sim 2^{-19}$
Division	$-2^{-8} \sim +2^{-8}$	$-2^{-20} \sim +2^{-20}$

Table 3.2 Updated ranges of the relative precision loss

3.3 Insertion of new instructions into Gem5

Instruction insertion in Gem5 is more complicated than insertion in the cross compiler. Gtool needs to choose unused opcode for the new instructions, check instruction syntax and insert instruction blocks in the proper places in the decoder file.

Gem5 labels different instruction fields in 32-bit instructions. These labels mark instruction fields as shown in Figure 3.9. For example, OPCODE stands for the bits 25 to 31 and INTFUNC stands for the bits 5 to 15 in the instruction. Gem5 uses these as entries to decode instructions. We use these labels to locate the proper position in the decoder file for inserting the new instructions.

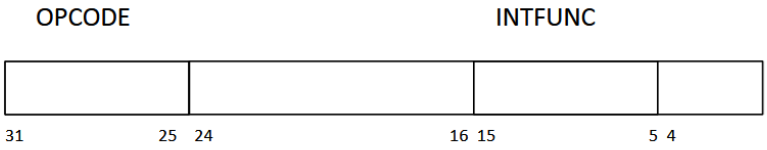

Figure 3.9 Instruction field labels

3.3.1 Difference between Alpha and MIPS

The differences between Alpha and MIPS are obvious when comparing their instruction field labels. The differences are shown in Table 3.3.

Comparison between opcode of Alpha and MIPS

	Alpha	MIPS
Number of instructions (Include reserved opcode)	248	529

Max number of instruction fields used for one instruction	5	8
The most often number of instruction fields used by instructions	2	6
Number of integer instructions that can be inserted with unused opcode	1959	772

Table 3.3 Comparison between opcode of Alpha and MIPS

MIPS has more instructions than Alpha and each MIPS instruction uses more instruction field labels than an Alpha instruction. This means there are more steps when decoding a MIPS instruction than an Alpha instruction.

3.3.2 Instruction decoding block

Each instruction has its decoding block within the decoder file. A typical decoding block is shown in Figure 3.10. It consists of the format name, the instruction name, the function field and other parts such as flags. Sometimes the labels are also included. The decoding block has to be changed when varying the ISA.

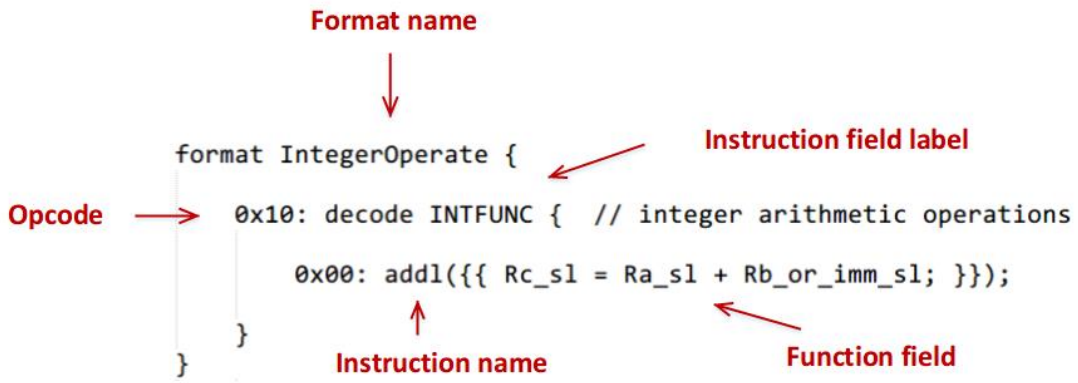

Figure 3.10 Instruction decoding block

Every instruction has its format, which appears in the decoding block. The format name and instruction name provide details about how a new instruction would be implemented in the simulator.

The function fields use a C-like code, which describes the function of the instruction. The variables in the function fields are register names or immediate-value symbols. For example, in the Alpha ISA, register names are Rc, Ra and Rb with a suffix indicating the length of the register while the MIPS ISA, usually, uses Rt, Rd and Rs with different suffixes.

Every time the user inputs information about a new instruction, Gtool prepares a decoding block for it. Then, it inserts this block into the proper location in the decoder file.

3.3.3 Insertion of new instructions

Similar to the cross compiler, inserting an instruction into Gem5 requires inserting its decoding block into the decoder file. Since the function field defines the behavior of the instruction, most of the changes in instructions are related to the function field. It is easy to design the function field by using basic C-like symbols. However, truncated floating-point operations need truncation functions for all floating-point operands. There are no such functions in the C language, and as a result, the user should create them.

In the upper-level directory of ISA in Gem5, *decoder.cc* and *decoder.hh* can be used as function definition files for the current instructions set. Gtool does not need to modify these files. In this project, we build the floating-point truncation function and error injection function for verifying the fault checking capability.

CHAPTER 4

EXPERIMENTAL SETUP

The parameters of the processor used in our experiment are shown in Table 4.1, and were obtained from [7]. We inserted into the Alpha ISA the new instructions shown in Table 4.2. In this project, we used 15 as the residue modulus, which ensures a high fault detection coverage.

Width	64 bits
Fetch/issue	6/3
I-cache	32k/64B/4-way/2 cycles
D-cache	64k/64B/4-way/2 cycles
Frequency	2GHz
L2	1MB/64B/8-way/14 cycles
Gem5 CPU model	DerivO3CPU

Table 4.1 Processor Configuration

In the fault detection coverage experiments, faults were injected through erroneous instructions. Erroneous instructions are new instructions which are similar to checker instructions. They have the same functionality as regular instructions except the extra error generation function in their output. The error generation function changes a random bit in the output to its opposite value. For example, it the correct output for a regular instruction is 1110 in binary, the error generation function would randomly flip one bit in 1110.

The flipped bit is selected randomly and in each experiment the randomly selected bit would be different. However, when comparing two checking mechanisms the same random number should be used in both experiments.

Name	Opcode	Type
mulrf	0x04 0x01	Integer Operation
addrf	0x02 0x00	Integer Operation
subrf	0x05 0x00	Integer Operation
divtr	0x21 0 0,1,5,7 0x23	Floating-Point Operation
addtr	0x21 0 0,1,5,7 0x24	Floating-Point Operation
subtr	0x21 0 0,1,5,7 0x25	Floating-Point Operation
multr	0x21 0 0,1,5,7 0x26	Floating-Point Operation

Table 4.2 Checker instructions for the Alpha ISA

For the experiments, we have selected five integer benchmarks shown in Table 4.3 [27] and six workloads listed in Table 4.4. The first five workloads in Table 4.4 use mostly floating-point operations. The sixth benchmark (edn) has both integer and floating-point arithmetic operations. The average results for floating-point workloads do not include the results of edn.

Benchmark	Description	Bytes	Lines of Code
adpcm	Adaptive pulse code modulation algorithm.	26852	879
ud	Calculation of matrixes.	6 K	163
matmult	Matrix multiplication of two 20x20 matrices.	3737	163
fdct	A lot of calculations based on integer array elements.	8863	239
insertsort	Input-data dependent nested loop with worst-case of (n^2)/2 iterations (triangular loop).	3892	92

Table 4.3 Integer workloads

Benchmark	Description	Bytes	Line of Code
fft1	A lot of calculations based on floating-point array elements.	6244	219
ludcmp	LU decomposition algorithm.	5160	147
minver	Floating value calculations in 3x3 matrix. Nested loops (3 levels).	5805	201
lcdnum	Loop with iteration-dependent flow.	1678	64

qsort-exam	Non-recursive version of quick sort algorithm.	4535	121
edn	Finite Impulse Response (FIR) filter calculations.	10563	285

Table 4.4 Floating-point workloads

In the performance results of the next chapter, the DMR (Dual modular redundancy) results were from programs that did not use the new instructions. This is time-redundancy DMR, i.e., every checked instruction is executed twice and the results are compared.

In the fault detection coverage experiments, faults were injected into the benchmarks by using erroneous instructions.

CHAPTER 5

RESULTS

We now present the performance and fault detection coverage analysis of the proposed method. We first compare the performance of using checker instructions to that of DMR. Then, we compare the fault detection coverage of these two methods.

5.1 Performance comparisons

The performance comparison between the method of using checker instructions and DMR is shown in Figure 5.1 (integer) and Figure 5.2 (floating-point). On average, the use of checker instruction reduced the execution time to 94.84% for integer workloads and to 99.43 for floating-point workloads.

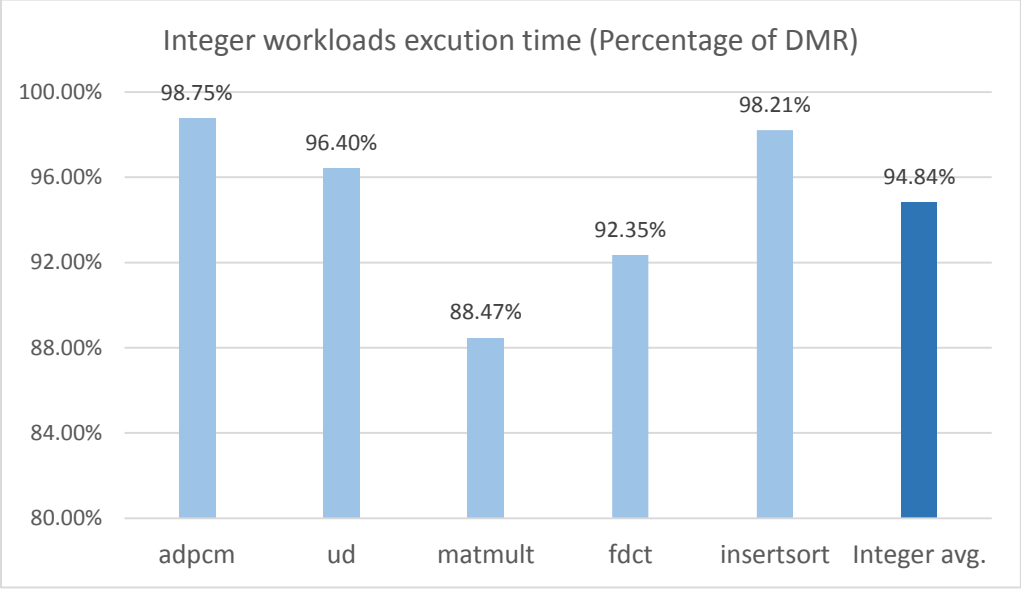

Figure 5.1 Performance comparison (integer).

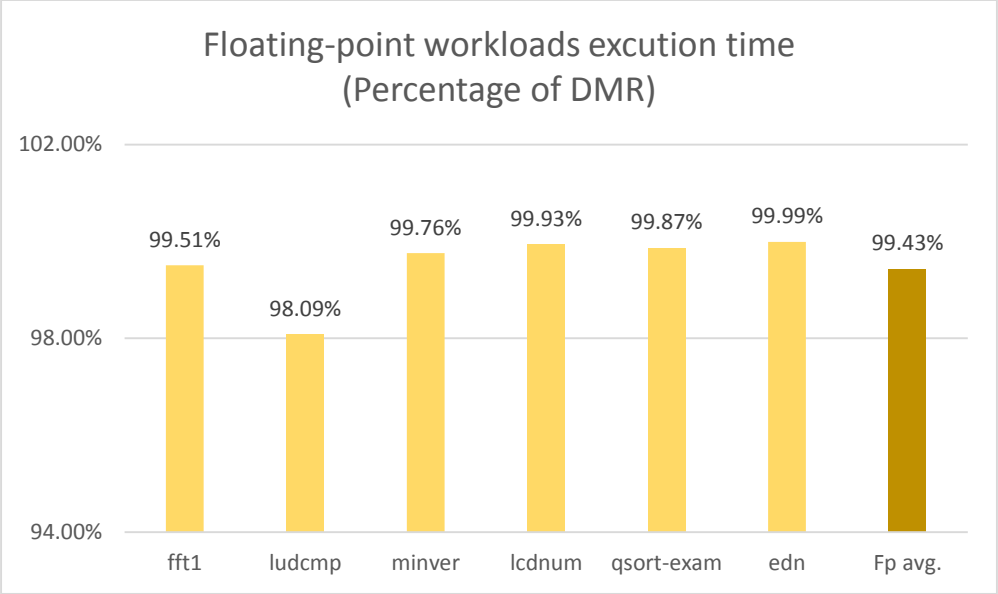

Figure 5.2 Performance comparison (floating-point).

We also measured the memory usage of these workloads during their execution and the results are shown in Figure 5.3 (integer) and Figure 5.4 (floating-point).

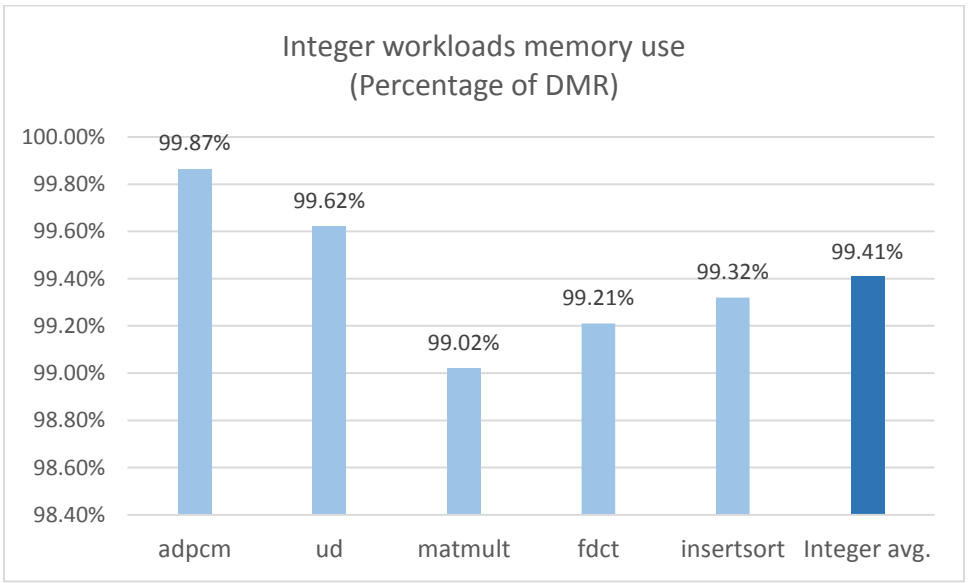

Figure 5.3 Memory use comparison (integer)

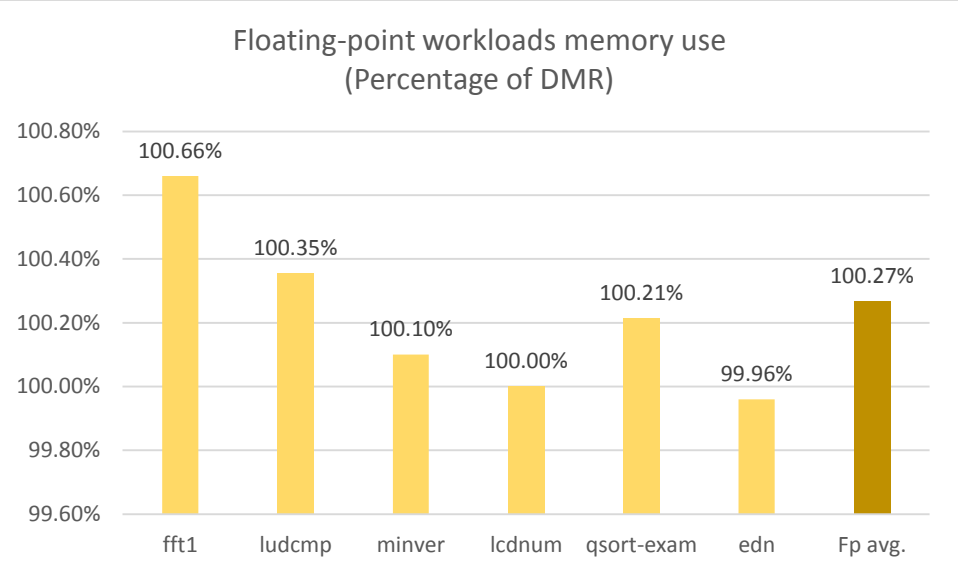

Figure 5.4 Memory use comparison (floating-point)

5.2 Fault detection comparisons

As we use residue checking in integer operations and truncated floating-point in floating-point operations, their resulting fault detection coverage would be different. The results are shown in Figure 5.5 (integer) and Figure 5.6 (floating-point).

It is obvious that the fault detection coverage of floating-point operations is lower than that for integer operations, as the checker operations could only detect the errors which were larger than the largest precision loss.

In the comparisons in Figure 5.5 and Figure 5.6, all errors are considered equally irrespective of their magnitude.

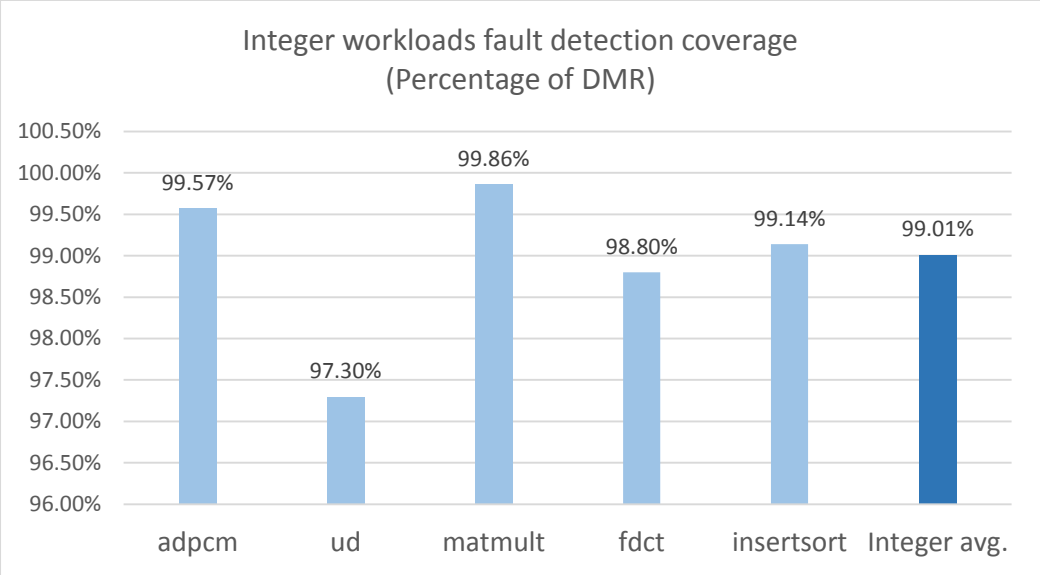

Figure 5.5 Fault detection coverage comparison (integer)

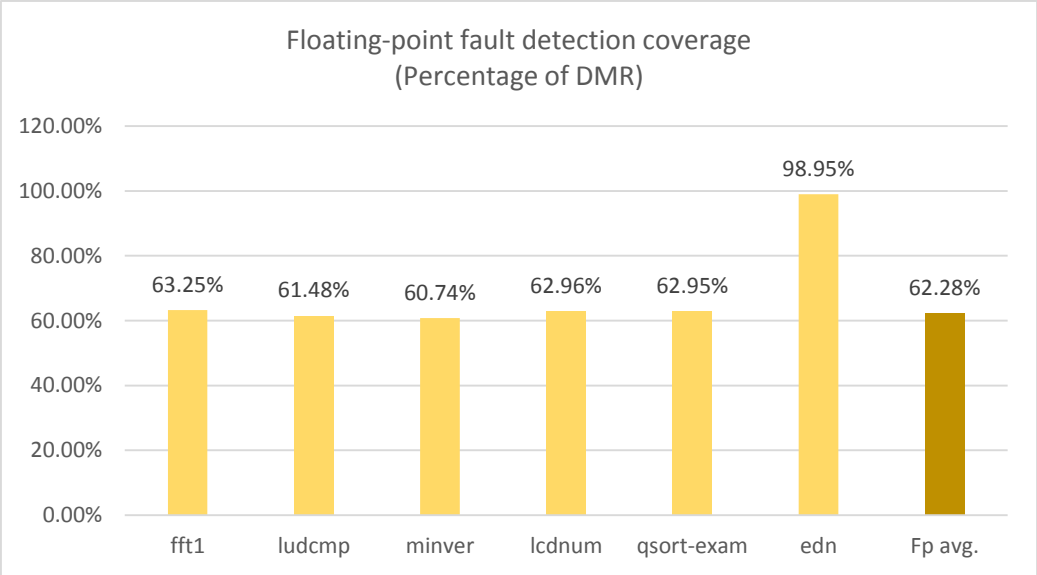

Figure 5.6 Fault detection coverage comparison (floating-point)

We therefore, performed another experiment. In this experiment, errors were weighted by their relative value compared to the correct result.

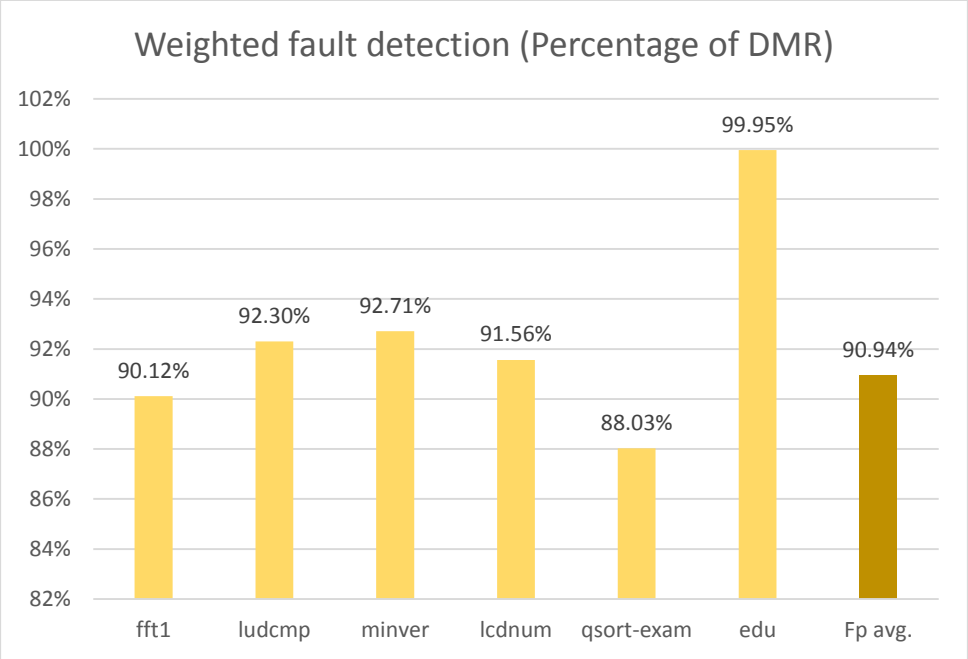

Figure 5.7 Weighted fault detection coverage for floating-point operations

As expected, the coverage in Figure 5.7 is higher than that in Figure 5.6, which indicates that the method of truncated floating-point values can detect almost all of the large errors.

CHAPTER 6

CONCLUSION

We have presented an analysis of the performance and fault detection coverage of using checker instructions and compared them to DMR. Using checker instructions can benefit integer operations as the lower execution time and reduced memory usage make the residue checking method better than DMR. In this project, we used 15 as the residue modulus. Our scheme for injecting erroneous instructions restricts the errors to be multiples of 2. This means that the fraction of undetected faults will only be 1/30 of the total number of injected errors in theory. The experiments have shown better results than predicted by the theory, since divisions are checked by the DMR method.

The truncated floating-point scheme is not as beneficial. The main reason is that the truncated floating-point operations only reduced marginally the execution time. The comparison between the truncated results and the precise results consumed the execution time that was saved in the arithmetic operation. Unless a way to further accelerate the execution of the truncated operations is found, the DMR approach will still outperform the truncated floating-point approach.

BIBLIOGRAPHY

[1] Forsati, R.; Faez, K.; Moradi, F.; Rahbar, A., "A Fault Tolerant Method for Residue Arithmetic Circuits," Information Management and Engineering, 2009. ICIME '09. International Conference on, pp.59- 63, 3-5 April 2009

[2] Eibl, P.J.; Cook, A.D.; Sorin, D.J., "Reduced Precision Checking for a Floating Point Adder," *Defect and Fault Tolerance in VLSI Systems, 2009. DFT '09. 24th IEEE International Symposium on*, pp.145-152, 7-9 Oct. 2009

[3] Piestrak, S.J., "Design of multi-residue generators using shared logic," *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pp.1435-1438, 15-18 May 2011

[4]Honda, M.; Kameyama, M.; Higuchi, T., "Residue arithmetic based multiple-valued VLSI image processor," Multiple-Valued Logic, 1992. Proceedings, Twenty-Second International Symposium on, pp.330-336, 27-29 May 1992

[5] Shivakumar, P.; Kistler, M.; Keckler, S.W.; Burger, D.; Alvisi, L., "Modeling the effect of technology trends on the soft error rate of combinational logic," Dependable Systems and Networks, DSN 2002. Proceedings. International Conference on, pp.389-398, 2002

[6] Lipetz, D.; Schwarz, E., "Self Checking in Current Floating-Point Units," Computer Arithmetic (ARITH), 2011 20th IEEE Symposium pp.73, 76, 25-27 July 2011

[7] Subramanyan, P.; Singh, V.; Saluja, K.K.; Larsson, E., "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp.1572-1577, 8-12 March 2010

[8] Smolens, J.C.; Gold, B.T.; Falsafi, B.; Hoe, J.C., "Reunion: Complexity-Effective Multicore Redundancy," Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on, pp.223-234, Dec. 2006

[9] Compaq Computer Corporation (1998). *Alpha Architecture Handbook*. 4th ed. Compaq Computer Corporation: Compaq Computer Corporation.

[10] Constantinides, K.; Plaza, S.; Blome, J.; Bin Zhang; Bertacco, V.; Mahlke, S.; Austin, T.; Orshansky, M., "BulletProof: a defect-tolerant CMP switch architecture," High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, pp.5, 16, 11-15 Feb. 2006

[11] Gomaa, M.; Scarbrough, C.; Vijaykumar, T.N.; Pomeranz, I., "Transient-fault recovery for chip multiprocessors," Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on, pp.98-109, 9-11 June 2003

[12] Jeyapaul, R.; Hong, F; Rhisheekesan, A.; Shrivastava, A.; Kyoungwoo Lee, "UnSync: A Soft Error Resilient Redundant Multicore Architecture," *Parallel Processing (ICPP), 2011 International Conference on*, pp.632-641, 13-16 Sept. 2011

[13] Koren, I, and Krishna, C.M.. Fault Tolerant Systems. San Francisco, CA: Elsevier, 2007. 36-41.

[14] Koren, I. Computer Arithmetic Algorithms. Natick, MA: K Peters, 2002. 259-277.

[15] Hennessy, J .L, Patterson A. D. and Arpaci-Dusseau C. A. Computer Architecture: A Quantitative Approach. Amsterdam: Morgan Kaufmann, 2007.196-264.

[16] Md Salim, S.I.; Sulaiman, H.A.; Jamaluddin, R.; Salahuddin, L.; Zainudin, M.N.S.; Salim, A.J. "Two-pass assembler design for a reconfigurable RISC processor," Open Systems (ICOS), 2013 IEEE Conference on ,pp.77-82, 2-4 Dec. 2013

[17] Bloom, G. (2013). *Add a pseudo instruction to gem5.* http://gedare-csphd.blogspot.com/2013/02/add-pseudo-instruction-to-gem5.html/

[18] Ortego, M.P.; Sack, P.; (2004). *SESC: SuperESCalar Simulator.* http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/

[19] Renau, J.; Fraguela, B.; Tuck, J.; Liu, W.; Prvulovic, M.; Ceze, L.; Sarangi, S.; Sack, P.; Strauss, K. and Montesinos, P. (2005). *SESC: cycle accurate architectural simulator.* http://sesc.sourceforge.net/index.html/

[20] SimpleScalar LLC. (2004). *SimpleScalar LLC to serve and project* http://www.simplescalar.com/

[21] Austin, T.; Ernst, D.; Larson, E.; Weaver, C.; Desikan, R.; Nagarajan, R.; Huh, J.; Yoder, B.; Burger, D. and Keckler, S. (2004). *SimpleScalar Tutorial.* http://www.simplescalar.com/docs/ simple_tutorial_v4.pdf/

[22] Koren, I. *Introduction to SimpleScalar.* http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm/

[23] Austin, T.; Larson, E.; Ernst, D., "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol.35, no.2, pp.59, 67, Feb 2002

[24] Saidi, A. (2013). *the gem5 Simulator System.* http://www.gem5.org/Main_Page

[25] Price, C. (1995). *MIPS IV Instruction Set Revision 3.2*. Mountain View, CA: MIPS Technologies, Inc.

[26] Peymandoust, A.; Pozzi, L.; Ienne, P. and Micheli, G. D. "Automatic instruction set extension and utilization for embedded processors." In Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors, 2003.

[27] Gustafsson, J. "SWEET Manual", Mälardalen University, Sweden, 2013