Leveraging Compiler Optimizations to Reduce Runtime Fault Recovery Overhead

Fateme S.Hosseini, Pouya Fotouhi, Chengmo Yang, and Guang R. Gao
Department of Electrical and Computer Engineering, University of Delaware
{fateme, pfotouhi, chengmo, ggao}@udel.edu *

ABSTRACT

Smaller feature size, lower supply voltage, and faster clock rates have made modern computer systems more susceptible to faults. Although previous fault tolerance techniques usually target a relatively low fault rate and consider error recovery less critical, with the advent of higher fault rates, recovery overhead is no longer negligible. In this paper, we propose a scheme that leverages and revises a set of compiler optimizations to design, for each application hotspot, a smart recovery plan that identifies the minimal set of instructions to be re-executed in different fault scenarios. Such fault scenario and recovery plan information is efficiently delivered to the processor for runtime fault recovery. The proposed optimizations are implemented in LLVM and GEM5. The results show that the proposed scheme can significantly reduce runtime recovery overhead by 72%.

1. INTRODUCTION

While smaller feature size, lower supply voltage, and faster clock rates have blessed modern computer systems with higher computational power, these trends also make silicon devices more vulnerable to various types of faults [14]. *Transient faults* caused by external events such as alpha-particle strikes, cosmic rays, or radiation from radioactive atoms [11] are expected to rise in facing the reduced noise margin, while process variation, voltage and temperature fluctuation, and in-progress wear-out [5, 2] expose systems to frequent irregular faults known as *intermittent faults*. These transient and intermittent faults usually lead to program failures, making fault resilience one of the most important design concerns.

A system-level fault tolerant solution typically integrates three functions: fault detection, execution checkpointing, and error recovery. Previously when the fault rate was low, many fault tolerance approaches focused on optimizing fault detection [12, 10] or checkpointing strategies [21, 18] since their overheads were consistently imposed onto the system regardless of whether a fault occurred or not. Those techniques performed recovery in a straightforward manner and sometimes even traded off recovery to reduce fault detection overhead [7, 15, 9]. Unfortunately, those approaches are insufficient and inefficient for future systems, especially the ones with high fault rates or with real-time constraints. In order for the computation to progress and meet deadline constraints before encountering another fault, the ability to quickly recover from a detected fault is critical.

*This work is supported by NSF grant #1253733.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '17, June 18-22, 2017, Austin, TX, USA

© 2017 ACM. ISBN 978-1-4503-4927-7/17/06...\$15.00

DOI: <http://dx.doi.org/10.1145/3061639.3062273>

The goal of this paper is to reduce recovery overhead without affecting overheads of the other two fault tolerance functions. We propose a scheme which leverages a set of compile-time optimizations to design a smart recovery plan for each application hotspot (i.e., a frequently executed loop). Our scheme targets hotspots because their code sizes are small but the runtime benefit is significant. For each hotspot, the proposed scheme analyzes instruction dependencies to identify the minimal set of instructions to be re-executed in different fault scenarios. Registers are also statically renamed to mitigate ambiguity in fault diagnosis. Such fault scenario and recovery plan information is efficiently delivered to the hardware for quick runtime fault recovery. The proposed compile-time and runtime optimizations are respectively implemented in LLVM [8] and GEM5 [1]. Experiments show that the scheme is capable of not only efficiently detecting faults, but also quickly recovering from them.

The rest of this paper is organized as follows: Section 2 briefly reviews previous work on error recovery. Section 3 outlines the fault tolerance framework and the technical motivation. Section 4 presents the proposed compiler-directed runtime recovery scheme. Section 5 describes the experimental framework and presents the results, while Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

When a fault is detected, recovery is necessary, either by preventing faults from modifying computation states, or by rolling the execution back to a previously saved clean checkpoint. One technique of the first category is presented for chip multiprocessors in [7]. To detect faults, two threads of the same program are run in parallel. The trailing thread repeats and checks the computation performed by the leading thread. Instructions of the trailing thread are not committed until their correctness is verified, and the register state of this thread is used for recovery. This scheme imposes high detection overhead on system because they need to constantly check and buffer the results for every instruction. To reduce this overhead, a technique that dynamically creates the dependence chain among instructions and only check the last instruction in the chain is presented in [15]. Yet the reduction in comparison overhead is insignificant since it can only create simple dependence chains. In sum, both recovery techniques need to execute all the faulty instructions even if they may not affect the correctness of program output.

The second type of recovery techniques allow faulty results to be committed to registers. Once a fault is detected, the computation is rolled back to the last checkpoint and re-executed from there. To detect faults, some instructions are selected as “detection points”. For example, the technique in [20, 13] only compares the values and addresses of store instructions. These schemes incur fewer comparison operations, but the increased

(c) Recovery Information
Figure 1: Recovery process example

distance between two comparison points leads to higher recovery overhead. In [4], only the live¹ registers are compared and checkpointed, while faults in dead registers are ignored. A following work [3] aims to balance detection and recovery overhead through adapting checkpoint frequency according to runtime fault rates. In sum, these recovery techniques need to execute all the instructions between the last checkpoint and the detection point, without any discrimination between them.

3. A MOTIVATING EXAMPLE

As reviewed before, most existing fault tolerant solutions perform error recovery in a straightforward manner by rolling the computation back to the last checkpoint and re-executing all the instructions from there. This straightforward process neither reduces the amount of re-execution nor reuses any clean but not checkpointed value.

To illustrate the potential for reducing recovery overhead, consider the hotspot code shown in Figure 1(a). The program is duplicated into two threads for redundant execution. T1 is the leading thread whose results are periodically verified by T2, the trailing thread. Fault detection is performed by comparing store values and addresses and periodically checkpointing the live registers. Checkpoints are placed at the loop entry before instruction 1. When T2 reaches instruction 6, since it is a store, T2 checks its value and address and detects a fault. In the default recovery approach, registers of both threads are restored to their last checkpointed values, and instructions 1–6 all need to be re-executed. This recovery strategy is independent of the fault origin. In other words, no matter which and how many instructions are faulty, the recovery overhead is the same.

However, a detailed examination of the Data Dependency Graph (DDG) indicates that upon detecting a fault, not all the instructions following the last checkpoint need to be re-executed. The DDG of the code in Figure 1(a) is shown in Figure 1(b). Nodes represent instructions, while an edge (u, v) represents a register value produced (written) by u and consumed (read) by v . Each edge is labeled with the register operand passed between the two instructions. The checkpointed regis-

¹At any point of the code, a register is *live* if its subsequent access is a read, and *dead* if its subsequent access is a write.

Figure 2: System Framework

ter set which includes the live-in registers $r0$, $r4$, and sp is also depicted in the figure.

Now consider the scenario of a fault originated in instruction 5 and detected at instruction 6. The default recovery policy has two limitations. First, it re-executes all instructions 1–6 indiscriminately, while according to the DDG, only instructions 5 and 6 are definitely faulty. The ones not located on the faulty instruction’s dependence chain, such as instruction 4, can potentially be skipped during re-execution. Second, all the data produced by instructions 1–6 in the faulty run are thrown out, although some of them might still be correct and useful, such as results in $r2$, $r1$, and $r4$ produced by instructions 1, 2, and 4, respectively.

The example reveals the general guidelines for reducing recovery overhead. Specifically, the recovery process only needs to re-execute the detected faulty instruction(s) as well as the instructions that either directly, or through a dependence chain, produce inputs for the faulty instruction(s). Instructions whose results have not been affected by the fault can be skipped during re-execution, and their results could potentially be used for executing other instructions. Motivated by this observation, we propose a scheme to minimize the recovery overhead in application hotspots.

4. PROPOSED SCHEME

4.1 Framework Overview

The proposed scheme improves recovery efficiency through a collaboration between dynamic and static optimizers, as shown in Figure 2. The *dynamic optimizer* performs fault detection, checkpointing, and recovery under the guide of the *static optimizer*, which is responsible for analyzing the code and generating recovery plans for the dynamic optimizer to use.

4.1.1 Dynamic optimizer

At runtime, the application is duplicated into two loosely-coupled threads for redundant execution. Checkpoints are created at loop entries and include all the registers which are live-in to the loop. Algorithm 1 shows the runtime fault detection process performed by the trailing thread. Store instructions and checkpoints serve as detection points (dctPnt). At a checkpoint, all the live registers are verified by the trailing thread, while upon a store, its value and address are compared. If no fault is detected, the trailing thread continues execution. If any fault is detected, the entire register sets of both threads are compared to identify all the registers with inconsistent values. Then the processor recovers the computation in the way shown in Algorithm 2. For every mismatch detected, the processor accesses the *recovery plan*, implemented as a direct mapped cache that accepts partial bits of dctPnt as the index and the rest of dctPnt alongside two other inputs as the tag, to get the

Algorithm 1 Fault Detection Process

```
1: if dtcPnt is a checkpoint then
2:   compare the set of live registers to detect fault;
3: elseif is store
4:   compare the store address and value to detect fault;
5: end if
6: if no mismatch then //fault-free
7:   continue;
8: else
9:   compare the entire register set to identify all mismatches;
10: end if
```

Algorithm 2 Fault Recovery Process

```
1: for every mismatched register detected do
2:   find MRS and RstRegs from recovery plan table
3:   add them to the overall MRS and RstRegs
4: end for
5: restore registers in RstRegs
6: restart execution from the checkpoint according to MRS
```

Minimum Re-execution Set (MRS) of instructions alongside the registers to be restored to their last checkpointed values (RstRegs).

The MRS uses a sequence of 0s and 1s to represent instructions that can be skipped during the recovery process and the ones that have to be re-executed. The bottom part of Figure 2 shows how to selectively re-execute instructions according to MRS. Specifically, when re-execution starts from the checkpoint, each time a new PC is calculated, MRS is right-shifted as well. The rightmost bit of MRS is the selection signal of the multiplexer, which decides whether a new instruction should be fetched or to be replaced with a No-Operation (NOP).

To ensure the correctness of the detection and recovery process, three constraints are enforced. First, the branch outcomes of the leading thread are buffered and delivered to the trailing thread for comparison, so as to detect faults in the control flow. Second, the leading thread waits for the trailing thread at the checkpoint, and both threads resume their executions after creating a new checkpoint. Third, store instructions are not committed until a new checkpoint is created. This is to avoid memory inconsistency caused by a load followed by a store to the same memory location [19].

4.1.2 Static optimizer

As depicted in Figure 2, entries of the recovery plan provide MRS and RstRegs for a particular register state vector (RegStat), identified at a specific detection point, reached from a specific path (i.e., a certain Branch Prediction History (BPH)) in the hotspot. Such information is extracted by the static optimizer. One challenge faced by the static optimizer is the inability to precisely locate runtime faults at compile time. To cover all possibilities, a recovery plan is needed for each possible combination of faulty registers. In order to minimize the amount of information to be extracted and delivered to the dynamic optimizer, the static optimizer defines **prime fault scenarios** which are indecomposable and can be used to derive the remaining non-prime fault scenarios.

To identify the MRS for each prime fault scenario, the instructions that have to be re-executed during the recovery process should be determined, and the input availability of these instructions should be checked to form the RstRegs. Overall, instructions can be classified into the following three categories:

- *Clean*: instruction’s output register is not overwritten by any other instruction and is clean.
- *Faulty*: instruction’s output register is not overwritten by any other instruction and is faulty.

- *Ambiguous*: instruction’s output register is overwritten by another instruction.

As a concrete example, consider the fault scenario shown in Figure 1. The RegStat at the detection point and the instruction classification are shown in Figure 1(c). Instructions 1 and 2 are *Clean* because their outputs *r2* and *r1* are clean at the detection point and have not been overwritten by any other instruction. Instruction 5 is *Faulty* since its output *r0* is faulty. Instruction 3 is *Ambiguous* because its output *r0* is overwritten by instruction 5. At runtime, *Clean* instructions can be skipped during recovery while both *Faulty* and *Ambiguous* instructions have to be re-executed. Although an ambiguous instruction might not be faulty, there is no way to identify it for sure, and hence the static optimizer adopts a conservative strategy.

Given this instruction classification, the goal of the static optimizer is to minimize MRS by reducing the size of the ambiguity set. As shown in Figure 2, it leverages two compiler optimizations to generate an efficient MRS: a *Minimum Recovery Set Selector (MRSS)* which classifies instructions and generates the MRS, and an *Ambiguity Set Minimizer (ASM)* which minimizes the size of the ambiguity set through register renaming. These two optimizations are described below.

4.2 Minimum Recovery Set Selector (MRSS)

4.2.1 MRSS algorithm

MRSS, implemented as an LLVM pass [8], differentiates instructions to be skipped during recovery from the ones to be re-executed. It classifies instructions not based on its operation or its relation with the others, but based on its status in the fault scenario. For each fault scenario, it first classifies the instructions that form a dependence chain with the faulty instruction and then processes the remaining instructions.

Algorithm 3 shows the MRSS algorithm executed per each fault scenario. It uses a stack to hold all the instructions selected for re-execution and colored them in *red*. It first pushes all the faulty instructions on the stack (Lines 2-6). A faulty instruction is the Last Writer (LW) of a faulty register before the detection point. For example, in Figure 1, *r0* is detected as faulty and instruction 5 is its LW. The algorithm traverses the predecessors of faulty instructions in the reverse Depth-First-Search (DFS) order in Lines 8-28. For each faulty instruction, this process continues until reaching a clean instruction or the end of the dependence chain. A predecessor instruction is added to the MRS (lines 15–20) if its output is known faulty (a *faulty* instruction) or if it is not the LW of its destination register and hence its output cleanliness is unknown (an *ambiguous* instruction).

Algorithm 3 also ensures, for each instruction added to the MRS, the availability of its input registers. An instruction gets inputs either from its predecessors in DDG or directly from the checkpoint (i.e., a loop-carried dependence). For the former case, input availability is ensured in the DFS search process. For the latter case, if the instruction’s input register(s) are overwritten in the loop body, they need to be restored to their last checkpointed values. These registers form a modified set, as formulated in Equation (1):

$$ModSet = Live\text{-}in(hotspot) \cap Def(hotspot) \quad (1)$$

Definitions of “Live-in”, “Def”, and other functions used in this paper are summarized in Table 1. Based on this formula, RstRegs is formed by identifying instructions in MRS which gets their input(s) directly from the checkpoint, and the input register(s) are in *ModSet* (Line 22-27).

Algorithm 3 MRSS Algorithm

Require:

```

    DDG, dtcPnt, RegStat, LW
1: Mark all hot-spot's instructions Gray
2: for all regs do
3:   if (RegStat(regs) = faulty) then
4:     DFS_stack.push(LW(regs));
5:   end if
6: end for
7: DFS_stack.push(dtcPnt);
8: while DFS_stack is not empty do
9:   instr  $\leftarrow$  stack.pop();
10:  if (instr.color = Red) then continue;
11:  end if
12:  instr.color  $\leftarrow$  Red;
13:  for all Pred of instr in DDG do
14:    reg  $\leftarrow$  Pred.dstReg;
15:    if (RegStat[reg] = faulty or Pred != LW(reg)) then
16:      DFS_stack.push(Pred);
17:      if (Pred != LW(reg)) then
18:        DFS_stack.push(LW(reg));
19:      end if
20:    end if
21:  end for
22:  for all instr.src in ModSet do
23:    if instr.src comes from checkpoint then
24:      RstRegs.insert(instr.src);
25:      DFS_stack.push(LW(instr.src));
26:    end if
27:  end for
28: end while
29: Add all Red instructions to MRS;

```

Table 1: Variable and function definitions

Term	Definition
Live-in(<i>L</i>)	Registers live at the entry of loop <i>L</i>
Def(<i>L</i>)	Registers written in loop <i>L</i>
Used(<i>L</i>)	Registers read in loop <i>L</i>
LW(<i>r</i>)	last producer of register <i>r</i> before detection point
FP(<i>r</i>)	first producer of register <i>r</i> in loop <i>L</i>

Finally, a correct and complete recovery scheme should not only recover the faulty registers, but also preserve the value of clean registers at the detection point. The value of a clean register may change during recovery only if it is selected as a RstRegs or its value will be overwritten by an instruction selected for re-execution. To avoid changing any clean register, these two cases are checked in Algorithm 3 and the LW of any register of these two cases are pushed on stack and added to MRS at Line 25 and 18, respectively.

As an example, we outline the procedure of Algorithm 3 when applied to the fault scenario shown in Figure 1(b), assuming the fault origins at instruction 5 and is detected at instruction 6. Instructions are processed in the order of $\{6,5\} \rightarrow \{1,3\} \rightarrow \{2,4\}$. Instruction 5 is added to MRS as the LW of the faulty register *r0*. Instructions 1 and 3 are predecessors of 5, and instruction 2 is the predecessor of 3. When 3 is added to MRS as an *ambiguous* instruction, the algorithm detects that it needs the checkpointed value of *r4* for re-executing 3, and *r4* is in *ModSet*. Therefore, instruction 4 as the LW of *r4* is also added to the MRS, and is processed to find further instructions on its dependence chain. The process terminates with MRS containing instructions 3,4,5, and 6 and RstRegs containing *r4*, as shown in Figure 1(c).

4.2.2 Covering all fault scenarios

While theoretically the MRSS pass should determine a recovery plan for each possible combination of faulty registers, such amount of information is exponential since for *n* registers there is a total number of 2^n fault scenarios. To minimize the

amount of information to be extracted and delivered to the dynamic optimizer, the MRSS pass only generates recovery plans for **prime fault scenarios**, defined as indecomposable cases in which only one register is faulty. The other non-prime fault scenarios that contain more than one faulty registers can be considered as a combination of multiple prime fault scenarios. Their recovery plans can be derived at runtime, by forming the union of recovery sets of those prime fault scenarios. Equations (2) and (3) show the way to derive MRS and RstRegs using prime fault scenarios in a system with *N* registers $r_1, r_2, \dots, r_N$. Only if register *r_i* is faulty will its MRS and RstRegs of the prime scenario be included.

$$MRS = \cup_{i=1}^N \{MRS(r_i) \mid (RegStat(r_i) = \text{faulty})\} \quad (2)$$

$$RstRegs = \cup_{i=1}^N \{RstRegs(r_i) \mid (RegStat(r_i) = \text{faulty})\} \quad (3)$$

4.3 Ambiguity Set Minimizer (ASM)

Modern register allocators carefully examine variable live ranges and employ different heuristics to minimize the number of registers spilled to memory. It is very likely to write a physical register multiple times inside the hotspot. Although this policy works in favor of code performance, it increases the size of the *ambiguity set*. The more registers overwritten in the hotspot, the less chance of minimizing the MRS.

The ASM pass runs counter to the current register allocator's goal: it minimizes the *ambiguity set* by avoiding overwriting registers inside hotspots as much as possible. While this goal can be achieved via Static Single Assignment (SSA)², this approach works only if there are enough free registers to hold all the assignments inside the hotspot. Instead, the ASM pass creates a *pseudo-SSA* form inside hotspots. It applies two optimizations: creating as much free registers as possible and, when overwriting a register is unavoidable, prioritizing the decisions according to the recovery overhead.

First, in the ASM pass, registers which are not defined or used inside the loop body are considered free, as shown below:

$$R_{FreeSet} = allRegs - (R_{Used(Loop)} \cup R_{Def(Loop)}) \quad (4)$$

To preserve program data flow, live-in registers that are not accessed inside the loop are stored to memory before entering the hotspot, and loaded from memory upon exiting the hotspot. Such overhead is trivial since these store and load instructions are executed only once outside the loop.

Once the FreeSet is obtained, the ASM pass starts to rename registers inside the hotspot, by processing instructions from the bottom of the DDG in a reverse Breadth First Search (BFS) order. This processing order pushes potential register overwrites to the beginning of the DDG. It helps reduce recovery overhead in worst case scenarios since usually more instructions have to be re-executed to recover from a fault originated at the upper levels of DDG, compared to faults that are closer to the detection points.

In the renaming process, if the target instruction has a destination, it is renamed to a register obtained from the FreeSet. The define-use chain is preserved by renaming all the subsequent usage of that renamed register. This process repeats until either there is no instruction left to be renamed or the FreeSet is empty. Since one definition for each register is still allowed, for each register the ASM pass marks one instruction as its First Producer (FP) and skips it during the rename process. Note that if a register forms a loop-carried dependence, not the first but the last assignment instruction is marked as

²A program is in SSA form if each variable is assigned exactly once, and every variable is defined before it is used [6].

(a) Original code & DDG (b) Renamed code & DDG

All Regs.	<i>sp, r0, r1, ..., r8, r9</i>
Live_in (Chkpt. Set)	<i>sp, r0, r4, r5, r6, r8, r9</i>
FreeRegs	<i>r3, r7, r8, r9</i>
Live_in & not used	<i>r8, r9</i>
Faulty Set	<i>r0, r2</i>
Faulty Instr.	<i>6, 8, 9</i>
Amb. Instr.	<i>1, 2, 3, 4, 7</i>
Clean Instr.	<i>5</i>
MRS	<i>1, 2, 3, 4, 5, 6, 7, 8, 9</i>
RstRegs	<i>r0</i>

(c) Information of (a)

Live_in (Chkpt. Set)	<i>sp, r0, r4, r5, r6</i>
Faulty Set	<i>r0, r8</i>
Faulty Instr.	<i>6, 8, 9</i>
Amb. Instr.	<i>2</i>
Clean Instr.	<i>1, 3, 4, 5, 7</i>
MRS	<i>6, 8, 9</i>
RstRegs	

(d) Information of (b)

Figure 3: ASM Process Example

FP. This property is shown in Figure 3, which presents the original and renamed versions of a loop body. Nodes marked with * are FPs and those instructions are not renamed, as can be seen in Figure 3(a)(b). For *r0* in 3(a), not instruction 2 but instruction 8 is marked as its FP since 8 in iteration *i* provides input to 2 in iteration *i* + 1. By retaining these FPs, names of the loop's live-in registers are preserved as well.

Figure 3 illustrates the ASM process and its effect on the ambiguity set and MRS size. The fault scenario is a fault originated at instruction 6 and detected at instruction 9. For the original code, The MRS in Figure 3(c) shows that every instruction has to be re-executed; even the clean instruction 5 has to be re-executed to re-produce *r1*. In comparison, Figure 3(d) shows the recovery information for the renamed code as well as the instruction processing order. As can be seen, the ASM pass processes instructions in the reverse BFS order, and renames the ones not marked with * except for instruction 2. Since there are only 4 free registers which are used to rename instructions 5, 6, 7 and 4, at the time when instruction 2 is processed, the FreeSet is empty. Although instruction 2 is ambiguous, it is not added to MRS since its successor instruction 3 is known clean. Overall, this example shows that the ASM pass successfully reduces the ambiguity set to 1, and as a result, MRS size is decreased from 9 to 3 and RstRegs becomes $\emptyset$.

5. EXPERIMENTAL EVALUATION

5.1 Experiment Setup

The static optimizer is implemented in LLVM 3.7 [8]. Both the MRSS and ASM passes are developed for ARM architecture. MiBench benchmarks are used in the experiment. For each benchmark, two sets of recovery plans are generated respectively for the baseline code (without ASM) and converted code (with ASM).

The runtime fault detection, checkpointing, and recovery

Table 2: Effect of ASM on SSA ratio and MRS size

Benchmark	SSA ratio (%)		MRS size reduction (%)		
	w/o ASM	w/ ASM	Min	Average	Max
Basicmath	30.5	89.4	0	8	33
Bitcount	24.4	89.5	0	15	66
Dijkstra	22.5	72.7	0	6	66
FFT	35.8	69.3	0	8	45
Patricia	21.8	73.1	0	4	50
Qsort	30.9	88.8	0	18	60
Search	31.3	88.7	0	10	66
Sha	21.8	73.2	0	5	71
Susan	17.2	44.7	0	6	66
Average	26.9	75.9	0	9	58

functions are simulated in GEM5 [1], under *SE* mode with L1 and L2 caches, using *simple timing* CPU model for the ARM architecture. Each benchmark is duplicated into two threads: a clean thread that executes normally, and a faulty thread with faults injected. According to [16], 70% of faults in the processor lead to inconsistencies in the register file. Therefore, our experiments inject faults into the register file, by randomly selecting one instruction and injecting a single bit-flip into one of its source or destination registers. A faulty thread is injected with 1000 faults inside its hotspots. Fault detection is implemented in the post-execution stage of GEM5. Two threads are compared at store instructions and at checkpoints, in the way described in Algorithm 1.

5.2 Results

Impact of ASM: Table 2 reports the percent of hotspot code in the SSA form, before and after applying ASM. On average, the ASM pass effectively increases the SSA ratio of the code by 49%. Variations across benchmarks are due to the differences in their hotspot sizes and the number of free registers.

By increasing the SSA ratio of the code, ASM is able to reduce the size of the ambiguity set and hence the MRS size. Columns 4–6 of Table 2 respectively show the min, average, and max reductions in MRS size. On one hand, the average maximum MRS size reduction of 58% confirms the effectiveness of ASM. On the other hand, the minimum reduction is 0% since ASM has no effect on those hotspots already in the SSA form. Existence of such cases pulls the average reduction down to 9%.

Fault classification: A classification of all the injected faults is performed to demonstrate the fault coverage of the experiment framework and quantify the rate of injected faults to which the proposed recovery scheme is applicable. Specifically, faults are classified into six categories: *Crashed* faults lead to unsuccessful termination of the program, *Benign* faults are neither detected nor manifested as an error in program output, *SDC* (Silent Data Corruption) faults corrupt program output without being detected, *Control Faults* are detected and they change program control flow, *DUE* (Detected Unrecoverable) faults are detected but cannot be recovered, while *Recoverable* faults are detected and recovered successfully.

Figure 4 shows the classification of injected faults for different benchmarks. The average rates of Benign, Crashed, Control faults, and SDC are 13%, 26%, 4%, and less than 1%, respectively. The remaining 56% of faults are all Recoverable, and there is no DUE fault in the system. These results show that the proposed framework is applicable to 56% of faults.

Our experiments also confirm that there is no difference between the baseline code and the converted code in terms of fault coverage. The converted code never changes the instruction dependence chain and registers omitted from the checkpointed set, if any, never affect fault propagation and detection

Figure 4: Fault classification

Figure 5: Recovery overhead reduction

in the hotspot. In other words, in the two versions of code, a fault always propagates through the same path and is detected at the same detection point.

Recovery latency: Figure 5 depicts the recovery overhead reduction achieved by the proposed scheme compared to the traditional recovery scheme, represented as the number of clock cycles in GEM5 dedicated to the recovery process. Even for the baseline code without ASM pass being applied, the MRSS pass still reduces recovery time by 68%. The ASM pass reduces recovery overhead by an additional 4%, leading to a total reduction of 72%. Effect of ASM on different benchmarks is in line with the average MRS size reduction results in Table 2. Since faults are injected randomly in GEM5, a run-time fault scenario could be a combination of multiple prime scenarios. The relatively small reduction achieved by ASM indicates that many of the prime scenarios used at runtime are best case scenarios whose size cannot be significantly decreased by ASM.

Recovery plan table Size: As shown in Figure 2, the recovery plan is stored in a direct mapped cache for the dynamic optimizer to access. Size of the cache, which is constrained by the largest hotspot in each benchmark, can be computed using the following equation:

$$Size = 2^{\lceil \log_2^{entries} \rceil} * (PC_{Tag} + BPH + RegStat + MRS + RstRegs) \quad (5)$$

Here, *entries* is the total number of entries in the recovery plan, *RegStat* and *RstRegs* are both 16-bit values with each bit corresponding to an ARM register, *PC_{Tag}* is the portion of PC bits used as tag in Figure 2, *BPH* is the minimum number of bits needed to identify a specific path, and *MRS* size is the number of instructions in the largest hotspot. These values and the cache size are reported in Table 3. It can be seen that a 16KB cache is sufficient to store recovery plans of all the benchmarks. According to CACTI [17], the access latency of this cache is 0.3ns which is usually shorter than one cycle.

6. CONCLUSIONS

In this paper, a fault resilience scheme that minimizes the recovery overhead in application hotspots has been proposed. Compile-time optimizations are leveraged to generate recovery plans for different fault scenarios to guide runtime error recovery. These optimizations, implemented in LLVM, identify the minimum set of instructions to be re-executed during the recovery process and exploit renaming techniques to further reduce the re-execution set. To evaluate the proposed scheme,

Table 3: Overhead for Storing Recovery Plans

	Entries	PC index+tag	BPH (bits)	MRS (bits)	Size(KB) tag+data
Basicmath	159	8+5	2	24	0.7+1.25
Bitcount	89	7+6	2	19	0.4+0.5
FFT	277	9+4	2	40	1.4+3.5
Dijkstra	162	8+4	4	38	0.7+1.7
Patricia	34	6+6	2	77	0.2+0.7
Qsort	42	6+4	2	15	0.2+0.3
Search	85	7+7	2	29	0.3+0.7
Sha	115	7+5	4	35	0.4+0.7
Susan	1280	11+5	4	37	6+13.5

intensive fault injection studies have been conducted in GEM5. Results show that on average the recovery overhead can be reduced by 72%. Such a low-overhead recovery process enables current and future systems to efficiently tolerate the elevated and varying rates of faults.

7. REFERENCES

- [1] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, pages 1–7, 2011.
- [2] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *DAC*, pages 338–342, 2003.
- [3] H. Chen and C. Yang. Boosting efficiency of fault detection and recovery through application-specific comparison and checkpointing. In *LCTES*, pages 13–20, 2013.
- [4] H. Chen and C. Yang. Fault detection and recovery efficiency co-optimization through compile-time analysis and runtime adaptation. In *CASES*, pages 22:1–22:10, 2013.
- [5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.
- [6] R. Cytron et al. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.
- [7] M. A. Goma et al. Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83, 2003.
- [8] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.
- [9] S. Mukherjee, S. Reinhardt, and J. Emer. Incremental checkpointing in a multi-threaded architecture, 2005. US Patent App. 10/651,376.
- [10] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA*, pages 99–110, 2002.
- [11] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *HPCA*, pages 243–247, 2005.
- [12] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, pages 25–36, 2000.
- [13] J. Sharkey et al. Trade-offs in transient fault recovery schemes for redundant multithreaded processors. In *HiPC*, pages 135–147, 2006.
- [14] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, 2002.
- [15] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *ISCA*, pages 87–98, 2002.
- [16] N. J. Wang et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN*, pages 61–70, 2004.
- [17] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits*, 31:677–688, 1996.
- [18] M. Xie et al. Checkpoint-aware instruction scheduling for nonvolatile processor with multiple functional units. In *ASP-DAC*, pages 316–321, 2015.
- [19] M. Xie et al. Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor. In *DAC*, pages 1–6, 2015.
- [20] C. Yang and A. Orailoglu. A light-weight cache-based fault detection and checkpointing scheme for mpocs enabling relaxed execution synchronization. In *CASES*, pages 11–20, 2008.
- [21] C. Yang and M. R. Varella. Qualifying non-volatile register files for embedded systems through compiler-directed write minimization and balancing. In *VLSISOC*, pages 86–91, 2015.