Hardware Error Detection in Multicore Parallel Programs

by

Jiesheng Wei

B.E., Harbin Institute of Technology, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2012

Abstract

The scaling of Silicon devices has exacerbated the unreliability of modern computer systems, and power constraints have necessitated the involvement of software in hardware error detection. Simultaneously, the multi-core revolution has impelled software to become parallel. Therefore, there is a compelling need to protect parallel programs from hardware errors.

Parallel programs' tasks have significant similarity in control data due to the use of high-level programming models. In this thesis, we propose BlockWatch to leverage the similarity in parallel program's control data for detecting hardware errors. BlockWatch statically extracts the similarity among different threads of a parallel program and checks the similarity at runtime. We evaluate BlockWatch on eight SPLASH-2 benchmarks to measure its performance overhead and error detection coverage. We find that BlockWatch incurs an average overhead of 15% across all programs, and provides an average SDC coverage of 97% for faults in the control data.

Preface

This thesis is based on a work conducted by myself in collaboration with Dr. Karthik Pattabiraman. The work was published as a conference paper in the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) [44]. I was responsible for coming up with the solution and validating it, evaluating the solution and analyzing the results, and writing the paper. Karthik was responsible for guiding me with the solution reasoning, experiments design and results analysis, as well as editing and writing portions of the paper.

Jiesheng Wei and Karthik Pattabiraman, BlockWatch*: Leveraging Similarity in Parallel Programs for Error Detection, the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks, 2012*

Table of Contents

List of Tables

List of Figures

List of Acronyms

CFC Control Flow Checking

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

IR Intermediate Representation

LLVM Low Level Virtual Machine

MPI Message Passing Interface

MTTF Mean Time To Failure

MTTR Mean Time To Repair

SDC Silent Data Corruption

SIMD Single Instruction Multiple Data

SPMD Single Program Multiple Data

SSA Static Single Assignment

Acknowledgements

First of all, I would like to thank my advisor Dr. Karthik Pattabiraman for his support during the past two years. Karthik constantly motivated me to think out of the box independently, and at the same time he gave me guidance and suggestions at the appropriate time. Without his support, this thesis would not have been possible, and I would not have improved so much in creative and critical thinking. Karthik is also a mentor to me. His passion, enthusiasm and consistency is always something I am trying to learn.

I would like to thank my colleagues in the Computer Systems Reading Group (CSRG) for their feedbacks on my practice talks. The weekly paper discussion meeting is also a place where I got the big picture in this area. Special thanks to my labmates for making the lab an enjoyable place to work, and for their advice and suggestions on my work and paper writing.

I am grateful to my dear friends in Canada and China, for helping me with some real problems or to talk about life.

Finally, special thanks to my family. They do not know what I am working on exactly, but they have always provided me support. I would never be whom I am today without their constant support.

Dedication

To my parents

Chapter 1

Introduction

1.1 Motivation

The continued scaling and lower power consumption of Silicon devices have exacerbated their unreliability and error-proneness. A recent study has found that during an eight month period, machines with more than 30 days of accumulated CPU time have 1/190 chance of crashing due to CPU hardware faults. Further, machines that crashed once have a probability of 1/3.3 of crashing a second time [29]. Moreover, microprocessors are expected to experience significantly higher rates of hardware faults in the near future because of the decrease in manufacturing process and voltage [7]. For instance, studies have shown that alpha-induced transient faults increase 30 times when the manufacturing process goe from $250nm$ to $180nm$ and the voltage drops from $2V$ to $1.6V$ [18]. Processor faults have hitherto been masked from software through redundancy at the hardware level [38] (e.g., dual modular redundancy). However, as power consumption becomes a first class concern in computer systems, hardware-only solutions become infeasible due to their high power costs [47]. Therefore, software applications must be designed to tolerate hardware faults.

On another front, the microprocessor industry has adopted the multi-

core paradigm, or the integration of multiple cores on a single die. Already, eight-core processors are available on the market, and the number of cores is expected to increase in future generations [8]. The multi-core paradigm has revolutionized software development, and industry experts have predicted that parallel programs will become the de-facto standard in the future [40]. Therefore, parallel programs that run on future multi-core processors will need to be capable of detecting and recovering from hardware errors. While error recovery for parallel programs has received considerable attention [14], efficient error detection remains a challenge.

With the two issues above in mind, the research question we address in this thesis is as follows: *How do we detect hardware faults that propagate to parallel programs at the software level?* The solution we provide is to leverage the similarity across tasks (i.e., threads) of the parallel programs for error detection, which will be further explained in the following section.

1.2 Proposed Solution

In this thesis, we explore the use of similarity among threads of a parallel program for runtime error detection. The similarity arises as a result of high-level programming models. For instance, Single-Program-Multiple-Data (SPMD) paradigm, which is the most commonly used style for parallel programming [13], has the same control flow graph and some shared data among threads. Our approach statically extracts the similarity through compiler-based analysis, and inserts runtime checks in the program. The runtime checks compare the behaviors of the tasks at runtime, and flag any

deviation from the statically extracted similarity as an error. Because we leverage similarity among a group of tasks for error detection, we call our approach BlockWatch[1].

While there are many sources of similarity in an SPMD program [25], we focus on the similarity of control-data to detect faults that corrupt the control-data. Control-data is data that is used to make branch and loop decisions, and we define two threads as exhibiting control-data similarity at a branch if the behavior of a thread for the branch is constrained by the behavior of the other threads for the same branch. We focus on control-data because: (1) control-data is critical for the correctness of a program, and errors in this data lead disproportionately to Silent Data Corruptions (SDCs)[2] [41], (2) SPMD programs exhibit substantial similarity in the control-data (Chapter 5), and (3) no software technique other than duplication can protect this class of program data. (duplication has some disadvantages as mentioned in Chapter 6).

BlockWatch consists of two parts: static analysis and runtime checking. We implement static analysis with the LLVM compiler infrastructure [24]. Our analysis statically extracts the control-data similarity and instruments the program for collecting the runtime behaviours of different threads at its branches. We implement the runtime checking by developing a runtime monitor that checks the collected runtime behaviours for errors.

[1] BlockWatch is a program for crime prevention by residents watching for suspicious activities in a neighbourhood and reporting them.

[2] An SDC is a deviation from the output in an error-free execution.

1.2.1 Advantage of BlockWatch

We are not the first to observe that parallel programs exhibit similarity among their tasks - other techniques have used parallel programs' similarity for error detection [9, 27]. BlockWatch differs from these techniques in two ways. First, the other techniques learn the similarity by observing the program at runtime, and may consequently incur false-positives because they cannot distinguish between an unexpected corner case and a deviation due to an error. In contrast, BlockWatch is based on the static characteristics of the program, which by definition, incorporates a superset of the dynamic runtime behaviours, and hence has *no false positives*. This is especially important in production settings where a false-positive can trigger wasteful recovery or diagnosis. Secondly, BlockWatch operates at the granularity of individual branches in the program while the other techniques operate at the function or region granularities. As a result, BlockWatch can detect errors that affect a single branch, even if the error does not cause deviations at other granularities. To our knowledge, BlockWatch is the *first* technique to statically extract the similarity among a parallel programs' tasks, and leverage it for runtime error detection.

1.3 Contributions

The main contributions we make in this thesis are as follows:

1. We identify the generic code patterns that characterize control-data similarity in parallel programs, and divide the code patterns into four

categories (Chapter 2).

2. We develop compiler techniques to statically extract the control-data similarity patterns, and instrument the program with runtime checks corresponding to the categories of the patterns (Chapter 3).

3. We build a scalable, lock-free monitor for dynamically executing the runtime checks inserted by the compiler (Chapter 3).

4. We evaluate BlockWatch on seven SPLASH-2 benchmark programs [45]. The results of our empirical evaluation show that BlockWatch, (1) finds considerable control-data similarity in the programs (50% to 95%), (2) incurs average performance overheads of about 15% across the programs (for 32 threads on a 32-core machine), and (3) provides average coverage of 97% for transient errors in the control-data.

BlockWatch has three aspects that make it practical. First, BlockWatch does not require any modifications to the hardware, and can work on today's multi-core systems. Secondly, it does not require any intervention from the programmer, and is fully automated. Finally, BlockWatch incurs *no false positives* (i.e., does not detect an error unless one occurs in the program).

1.4 Related Work

In this section, we present prior work related to error checking and parallel program's similarity pattern identification. We classify related work into six

broad categories. Because we will discuss duplication in detail in Chapter 6, we do not consider it here.

Control-flow checking: Control-flow Checking (CFC) techniques such as ECCA [2], PECOS [5] and CFCSS [30] check the conformance of the program's control-flow to its static control flow graph. However, CFC techniques cannot detect errors that propagate to the control data and lead to a valid but incorrect branch outcome, i.e., control-data errors that result in the branch going the other way than its error-free behavior. BlockWatch detects this class of errors.

Statistical techniques: AutomaDeD [9] uses Semi-Markov Models (SMMs) to find parallel tasks that deviate from other tasks' behavior. AutomaDeD is similar to BlockWatch in that both techniques consider deviations as detections. However, AutomaDeD differs from BlockWatch in three ways. First, AutomaDeD requires the programmer to annotate their code with region identifiers which are used as the building blocks of the SMMs. Second, AutomaDeD is targeted towards software bugs during debugging, and not at runtime hardware errors. Finally, AutomaDeD learns SMMs at runtime, and can incur false-positives.

Mirgorodskiy et al. [27] use statistical techniques based on function execution times in parallel programs' tasks to detect outliers. However, this approach does not detect errors that do not cause a noticeable difference in the execution times of functions. Their approach also incurs false-positives as the execution times are learned at runtime.

Invariant based Checks: DMTracker [15] leverages invariants on data movement to find bugs in MPI-based parallel programs. They leverage the

observation that MPI programs have regular communication patterns, which gives rise to invariants on the transfer of data among the different tasks. DMTracker differs from BlockWatch in three ways. First, the invariants are specific to MPI-based programs, and do not apply for shared memory parallel programs. Second, the invariants derived by DMTracker pertain to the messages sent by the program, and not necessarily to the control-data. Finally, DMTracker attempts to learn the pattern of data transfer at runtime and may hence incur false-positives.

FlowChecker [10] also finds errors by tracking invariants on communication operations in MPI parallel programs. FlowChecker extracts message intentions, which are matching pairs of sends and receive MPI calls, and checks whether the message flows in the underlying MPI library match the extracted intentions. The goal of FlowChecker is to find bugs in MPI libraries that cause data loss or lead to mismatched messages, rather than detect runtime hardware errors.

Static and Dynamic Analysis: Static analysis has been extensively used for verifying in parallel programs [28, 43]. In these cases, the goal is to find bugs in the program, rather than detect runtime errors arising in hardware. Pattabiraman et al. [32] use static analysis to derive runtime error detectors for sequential programs. Their technique differs from ours in three ways. First, they confine themselves to critical variables that have high fanout in the program. Second, they duplicate the backward slice of the critical variable, and compare the value computed by the slice with that in the program. This approach will not work for non-deterministic parallel programs. Finally, they use support from the hardware to track control-flow

within the program, and hence require hardware modifications.

Dynamic analysis techniques detect errors by learning invariants over one or more executions [17, 19, 36]. These techniques target only sequential programs, and hence do not consider similarity across threads . Yim et. al. [46] propose a technique to learn invariants for GPU programs, and use the invariants for detecting errors. However, their focus is on errors that can cause large deviations in the output as GPU programs are inherently error-tolerant. A generic problem with all dynamic techniques is that of false-positives, which can trigger unwanted detection and recovery.

Algorithmic techniques: Algorithm-based Fault Tolerance (ABFT) is an error detection technique for specialized parallel computations such as matrix manipulation and signal processing [21, 33]. Sloan et al. [37] develop error-resilient gradient descent algorithms for stochastic processors, or processors that allow variation-induced errors to occur by drastically shaving off design margins in order to save power. Finally, Geist et al. develop a class of naturally fault-tolerant algorithms for certain classes of iterative parallel computations [16]. While these techniques are efficient, they only protect programs of the specific type they target. In contrast, BlockWatch targets general-purpose parallel programs.

Similarity based performance improvement: Long et al. [25] exploit the similarity in SPMD applications for performance improvement. They merge instruction fetching if certain instructions are the same among different threads and merge instruction execution if the instructions and their input operands are shared among different threads. However, they do not leverage the similarity for error checking.

1.5 Dissertation Organization

The rest of this thesis is organized as follows: Chapter 2 discusses the BlockWatch approach with an example, while Chapter 3 details its implementation. Chapter 4 introduces the experimental setup, and Chapter 5 presents the evaluation. Chapter 6 quantitatively compares BlockWatch to software-based duplication, and proposes some techniques to further improve BlockWatch. Finally, Chapter 7 concludes and proposes some future directions.

Chapter 2

Approach

2.1 Introduction

This chapter describes the high-level approach of BlockWatch. Section 2.2 presents the fault model for BlockWatch, while Section 2.3 lists the assumptions we make about the parallel program. Section 2.4 uses an example parallel program to illustrate the kinds of similarity considered by BlockWatch. Section 2.5 illustrates the runtime checks introduced by BlockWatch on the example program.

2.2 Fault Model

We consider transient or intermittent hardware faults that affect at most one processor or core in a multi-processor or multi-core processor. The fault can occur in the processor data path, control logic or memory elements in the core (e.g., caches). However, we assume that no more than one core or processor is affected by a fault at any time. This is reasonable as hardware faults are rare events (relative to the total time of execution of a parallel program).

Our fault model also captures certain kinds of software errors such as

rare race conditions and memory corruption errors that result in a thread deviating from its static semantics. However, we do not consider software errors in this thesis.

2.3 Assumptions on Parallel Program

We make three assumptions regarding the parallel program. First, we assume that it is written using a shared memory model, which is the common case with multi-core processors today. We have implemented BlockWatch for *pthreads* style parallel programs, though it can be extended for other kinds of shared memory parallel programs (e.g., CUDA programs). Second, we assume that the parallel program is written in an SPMD style. This ensures that the code to be executed by each thread is identical, and hence it suffices to analyze the common code to identify the similarity of branch runtime behaviour among threads. Finally, we assume that the entire source code of the program is available for analysis by BlockWatch. If this is not the case, BlockWatch will not be able to statically extract the program's similarity characteristics.

2.4 Control-data Similarity in Parallel Programs

We use Figure 2.1 to illustrate the presence of similarity in the control-data of a parallel program. In Figure 2.1, the program starts from function *main()*, which spawns *nprocs* threads, all of which execute the function *slave()* concurrently. The *slave()* function first assigns a unique thread ID *procid* to each thread in line 17 - 20 in Figure 2.1. It then executes four

```
1  int id = 0;
2  long im = DEFAULT_N;
3  struct global_private *gp;
4  int nprocs;
5
6  int main(int argc, char *argv[]) {
7    int i;
8    nprocs = argv[1];
9    for (i = 0; i < nprocs; i++)
10     gp[id].num = rand();
11   for (i = 0; i < nprocs; i++)
12     pthread_create((void *)slave);
13 }
14
15 void slave() {
16   int private, procid;
17   pthread_mutex_lock();
18   //procid is the thread id
19   procid = id++;
20   pthread_mutex_unlock();
21   //Branch 1: threadID
22   if (procid == 0) {
23     ...
24   }
25   ...
26   //Branch 2: shared
27   for (i = 0; i <= im - 1; i = i + 1) {
28     ...
29   }
30   ...
31   //Branch 3: none
32   if (gp[procid].num > im - 1) {
33     private = 1;
34   } else {
35     private = -1;
36   }
37   ...
38   //Branch 4: partial
39   if (private > 0) {
40     ...
41   }
42 }
```

Figure 2.1: Sample pthreads parallel program to illustrate the static similarity among all threads in the program. The comments indicate the similarity categories for each branch according to the classification in Table 2.1.

branches labelled 1 through 4 in the figure. The bold italic variables in the *slave()* are either constants or global variables that are shared among all threads. In this paper, we include loops in our definition of branches.

Table 2.1: Branch condition similarity category definition

Similarity Category	**Static characteristics of control data**	**Branch runtime behavior similarity**
shared	All operands of the instruction are shared variables among threads	All threads take the same decision at the branch.
threadID	One operand depends on thread ID, and the remaining operands are shared variables	The branch decision is related to thread ID - threads of certain thread IDs take the same decision. For example, if the condition comparison statement is an equality comparison between thread ID and shared variables, one thread follows one path and the remaining threads follow the other path at run time.
partial	Local variables, but these local variables are assigned with one of a small subset of shared variables	The threads which are assigned to the same shared variabl take the same decision.
none	Local variables that cannot be statically inferred to be similar across threads	No known similarity in branch runtime behavior among the threads.

We now illustrate the control-data similarity among the program's threads in Figure 2.1 for each of the four branches in the *slave()* function. The generic code patterns that result in the similarity are shown in Table 2.1. The similarity of the control-data in the four branches are as follows:

1. **Branch 1**: The branch condition tests equality of thread ID and a constant *0*. Because the constant is the same for all threads, and the thread ID is different, at most one thread will take the branch in a correct execution. This would be classified as *threadID* according to Table 2.1.

2. **Branch 2**: The variable i shares the same initial value, increment value and end value among all threads. Assuming there are no *break* statements in the loop, all threads execute the same number of loop iterations. This would be classified as *shared* according to Table 2.1.

3. **Branch 3**: The variable *gp[procid].num* is thread local and may be different for different threads. This would be classified as *none* according to Table 2.1.

4. **Branch 4**: The variable *private* is also thread local. However, it's value is either *1* or *-1*, depending on the outcome of branch 3. Therefore, threads in which *private* takes the same value will make the same decision in this branch. This is classified as *partial* according to Table 2.1.

Thus, the control-data for each of the four branches above belongs to a different similarity category according to Table 2.1. The table also illustrates the type of similarity exhibited by the branches belonging to each category. This similarity is encoded as a runtime check in Section 2.5.

Note that the similarity inference only relied on static analysis of the program's code, and did not require us to execute it. In this example, we

showed the analysis on the program's source code for simplicity. In reality, the analysis is done on the program's intermediate code generated by the compiler (Section 3.2).

2.5 Runtime Checking

In the previous section, we saw how to statically identify the similarity of the control data used in the branches in Figure 2.1. In this section, we illustrate how the similarity can be encoded as a runtime check within the program.

The basic idea is as follows: the statically inferred branch similarity behaviour among threads is consistent with the actual runtime branch behaviour similarity in an error-free execution. However, if a hardware error propagates to the branch condition data of one thread and causes the branch's outcome to flip, the program will deviate from the statically inferred behaviour. BlockWatch detects the deviation and stops the program.

As an example, we use *branch 1* in Figure 2.1 to explain the runtime checks. As we show in Section 2.4, *branch 1* belongs to category *threadID* according to the classification in Table 2.1. This means that no more than one thread (thread 0 in this case) takes the branch. To check this constraint, we insert a call to the checking code immediately after the branch decision to record its status. Assume that a hardware error propagates to *procid* variable in thread 2, thus causing it to take the branch. This violates the constraint that no more than one thread takes the branch, and is hence detected by the check.

2.6 Summary

This chapter proposed a high-level approach of BlockWatch. The approach contains identifying similarity patterns in SPMD programs and using the similarity for runtime checking. Section 2.4 identified the control-data similarity patterns and divided the patterns into four categories: *shared*, *threadID*, *partial* and *none*, as shown in Table 2.1. For the first three categories, variables depend on either shared variables or thread ID, and hence we are able to infer their runtime behaviour similarities through static analysis. Therefore, we focused on error detection for control-data whose categories belong to one of the first three categories, and Section 2.5 presented a method to compare their inferred similarity with actual runtime similarity for error checking at run time.

Chapter 3

Implementation

3.1 Introduction

This chapter describes the detailed implementation of BlockWatch. The implementation consists of two steps. The first step is to infer the branches' similarity category through static analysis at compile time, and is described in Section 3.2. The second step is to compare the actual runtime behaviours' of the branches with the inferred behaviour according to the branches' similarity categories using a runtime monitor, and is described in Section 3.3.

3.2 Similarity Category Identification

In this section, we introduce an algorithm to identify the branches' similarity categories. Our algorithm is implemented as part of an optimizing compiler. The algorithm assumes that the program has been translated into a low-level intermediate representation (IR) by the compiler's front-end. Therefore, all the branches in the program, including those in loops, have been explicitly represented as branch instructions prior to the algorithm. Further, we assume that the IR uses Static Single Assignment (SSA) form [12], which requires that a variable be assigned exactly once in the program i.e., every

variable in the program has a unique instruction that assigns to it.

As we show in Chapter 2, the similarity category of a branch depends upon the nature of the variables used in the branch condition i.e., whether they are shared, dependent on the thread ID or local to the thread. Therefore, in order to infer the similarity category of a branch, we need to find the similarity categories of the operands used in the branch instruction. However, the operands may themselves be produced by other instructions, and hence we need to determine the operand type of *all* instructions in the program. This determination is based on whether each operand is derived from a shared variable (*shared*), a variable containing the thread ID[3] (*threadID*), or from a local variable that can only take one of a small number of shared variables (*partial*).

Initially, all instructions in the program are assigned a classification of "NA", or "Not Assigned". Then instructions that are directly assigned from the thread ID variable are assigned to the category *threadID*. Similarly, instructions that are directly assigned from a shared variable are assigned to the category *shared*. After this step, the similarity categories are propagated to other instructions in the program as follows: (1) if it is a unary instruction, the similarity category of the instruction is the same as that of its (only) operand, (2) if it is a binary or ternary instruction, we consider each operand separately and update the similarity category of the instruction based on the rules in Table 3.1.

Propagation Rules: Before we present the overall algorithm, we first

[3]We look for common code patterns that compute the thread ID. These can be customized for different libraries.

explain Table 3.1. The rows of Table 3.1 correspond to the current instruction's similarity category, while the columns correspond to the operand's similarity category. The entries in the table indicate the similarity category to which the instruction should be assigned after processing the operand. Because we process each operand separately and update the instruction's similarity category after doing so, the same table applies for both binary and ternary instructions.

Table 3.1: Rules to infer instruction's similarity category from its current category and the operand's category

operand / current inst.	NA	*shared*	*threadID*	*partial*	*none*
NA	NA	*shared*	*threadID*	*partial*	*none*
shared	NA	*shared*	*threadID*	*partial*	*none*
threadID	NA	*threadID*	*threadID*	*none*	*none*
partial	NA	*partial*	*none*	*partial*	*none*
none	NA	*none*	*none*	*none*	*none*

We explain the rationale behind Table 3.1 with an example. Assume that the current instruction's similarity category is *partial*. This corresponds to the fifth row in Table 3.1. If the next operand belongs to category NA, then the instruction's category is set to NA and the inferring process ends for this instruction (the instruction will be revisited later). If the next operand is *shared* or *partial*, the instruction's category is set to *partial* because the instruction continues to depend on local variables that may come from one of the shared variables. If the next operand belongs to *threadID*, the instruction's category is set to *none* because the instruction depends neither exclusively on one of several shared variables nor the thread ID, and hence does not satisfy either category. If the next operand belongs to *none*, then

the instruction's category also becomes *none* as it depends on private variables. Note that the inference rules are conservative: even if a single operand belongs to category *none*, the instruction is updated to this category (see *optimizations* for how to mitigate this effect).

One case where we deviate from the rules in Table 3.1 is when a local variable is assigned with a shared value in one path of an if-else branch but not assigned in another, or is assigned different shared values in both paths. We update its category to *partial* instead of *shared* at the convergence point of the branch (i.e., the phi instruction in the SSA form). This is because the shared value is only one possible value that the variable may take at runtime. An example of this case occurs in the variable *private* in Figure 2.1, which is assigned to one of the two different constants 1 and -1 in the two outcomes of *branch 3*. Hence, its category is assigned to *partial*.

Multiple Instances: Because a static branch in the program may be executed multiple times e.g., if it is inside a loop or the function containing it is called multiple times, its similarity category may vary depending on the way we group the runtime instances to check. We illustrate this case with an example in Figure 3.1, which is adapted from FFT in the SPLASH-2 Benchmark Suite [45].

In Figure 3.1, there are two functions *slave()* and *foo()* that are executed by each thread. The *slave()* function calls *foo()* in two different places. Consider *branch 1* which is inside function *foo()*. The function is called at two different places in *slave()*, each time with a different value. However, in each invocation of the function, the local variable used in the branch condition is the same, namely *arg*. The variable *i* in *branch 1* also have

```
1  bool test;
2  void slave() {
3     ...
4    foo(1);
5     ...
6    if (test) {
7      foo(2);
8    }
9     ...
10 }
11 void foo(int arg) {
12   //Branch 2
13   for (int i = 0; i < 5; i = i + 1) {
14     //Branch 1
15     if (i < arg) {
16       ...
17     }
18   }
19 }
```

Figure 3.1: Example code of multiple runtime instances of the same branch

similar issues across different loop iterations.

There are two ways to classify the similarity of this branch. We can classify it as *shared* in which case we need to track the value at each call site and loop iteration number separately and ensure that we are comparing the values from each loop iteration of each call site separately. Another possibility is to merge the values across the call sites and across all iterations in the loop, and treat the branch as belonging to category *partial*, as it is derived from multiple shared variables. In this case, we need not track each invocation and the loop iteration number separately. We adopt the former policy in spite of the additional performance overhead it entails, as it allows us to perform tighter checks on the branch.

Algorithm: We now present the overall algorithm for inferring each

```
1  map categorymap;
2  int main() {
3    bool changed = true;
4    while (changed) {
5      changed = false;
6      for (inst in program) {
7        changed = visitInst(inst) || changed;
8      }
9    }
10
11   for (branch in program) {
12     if (branch in categorymap) {
13       branchcategory =
14         categorymap[branch];
15     } else {
16       branchcategory = ''none'';
17     }
18   }
19 }
20
21 bool visitInst (inst) {
22   Category category = NA;
23   for (op in operands) {
24     if (op is shared) {
25       category = lookupTable(
26         category, ''share'');
27     } else if (op is thread id) {
28       category = lookupTable(
29         category, ''threadID'');
30     } else if (op in categorymap) {
31       category = lookupTable(
32         category, categorymap[op]);
33     } else {  // op is NA
34       return false;
35     }
36   }
37
38   Category old = categorymap[inst];
39   categorymap[inst] = category;
40   return (category != old);
41 }
```

Figure 3.2: Pseudo-code to show the similarity category identification algorithm

instruction's similarity category in Figure 3.2. The algorithm iterates over all instructions in the program and updates the similarity category of each instruction by calling the *visit* function (lines 4 - 9) on the instruction. This process is repeated until there are no more changes in the instructions' similarity categories. The *categorymap* contains the inferred categories of all similar branches at the end of the iterations. The other branches are assigned to *none* in line 18.

The *visitInst* function(lines 23 - 36) takes an instruction as an argument, and walks through each of its operands in turn. For each operand, it infers the similarity category based on the category of the operand or by looking up the operand in the *categorymap*. Then it calls function *lookupTable* (not shown in figure) with the current instruction's category as well as the category of the operand. The *lookupTable* function uses Table 3.1 in Chapter 2 to find the similarity category of the current instruction and update it accordingly.

Note that the algorithm terminates in a finite number of iterations (say k) because the number of similarity categories is finite and the updated categories in Table 3.1 flow monotonically (i.e., in one direction only). Also, each iteration is proportional to the number of instructions in the program (say N). In the worst case, 'k' can be at most equal to 'N', and hence the worst-case complexity of the algorithm is O(N^2). In practice, 'k' is less than ten for the programs we studied.

Example: We illustrate the algorithm in Figure 3.2 with the example in Figure 3.1. Table 3.2 shows the similarity categories of the variables and branches in the example after each iteration of the algorithm. The variables

are used as proxies for the instructions that define them (these are not visible at the source code level) [4]. The algorithm converges within three iterations in this example. Note that the categories of the two branches in the first iteration are NA because in SSA form, the definition instruction of variable i has two operands: 0 and $i+1$, and $i+1$ is executed after the *branch 1* and *branch 2*. Therefore, when we visit the two branches in the first iteration, the category of i is still NA and hence the branches' categories are not updated. Later in this iteration, the category of i is determined as *shared* and the two branches' categories are changed in the 2nd iteration, after which there are no more changes and hence the process is terminated.

Table 3.2: Example of category propagation algorithm on Figure 3.1

Variables and Branches	Initial	1st iteration	2nd iteration	3rd iteration	Final category
test	shared	shared	shared	shared	shared
arg	NA	shared	shared	shared	shared
i	NA	shared	shared	shared	shared
Branch 1	NA	NA	shared	shared	shared
Branch 2	NA	NA	shared	shared	shared

Optimizations: We perform two optimizations over the base algorithm in Figure 3.2 to improve the coverage and the performance of the technique.

Because the algorithm for inferring static branch similarity is conservative, it will label some branches as *none* even if there is a single operand that it determines as private (not shared). However, in practice we find that considerable similarity exists even in these branches, as the private variable may

[4] In SSA form, instructions and variables are synonymous with each other.

have the same value across threads. We therefore promote such branches to the *partial* category and only compare the threads which have the same value for the private variable.

In some cases, a branch can be executed by no more than one thread at a time (e.g., branches inside critical sections). We remove the checks on such branches as BlockWatch needs a minimum of two threads to detect errors that violate the threads' similarity. Checking such branches would incur runtime overheads while providing no coverage benefit. We assume that the program has no race conditions which violate this constraint.

3.3 Runtime Checking

This section details the implementation of a runtime monitor to check the statically inferred similar branches in Section 3.2. The monitor is spawned as a separate thread in the program (BlockWatch adds instrumentation to spawn the monitor thread), and has three design goals as follows.

1. *Asynchronous:* The monitor must interfere minimally with the program's execution. In particular, it should not be in the critical path of the program, and must execute asynchronously with the program's threads.

2. *Unique branch identifier and fast lookup*: The monitor must assign a unique identifier for each runtime branch. Moreover, given a specific branch identifier, it must be possible to do a fast lookup of the branch's runtime characteristics of different threads. The two requirements are

important for correlating the information across multiple threads when storing the branches' runtime behaviors.

3. *Lock freedom*: The monitor must acquire no locks, as doing so may introduce deadlocks in the program, and also lead to unnecessary serialization of the program.

Architecture: We achieve goals 1 and 3 through separate *lock-free* front-end queues adapted from Lamport's algorithm [23] for each thread to send its branch information. The monitor thread asynchronously scans the queues and processes the information without using any locks. We achieve goal 2 through the use of a back-end hash table to store the branches based on their identifiers. The architecture of the monitor is illustrated in Figure 3.3.

Operation: The operation of the monitor is as follows:

- When a branch is executed by a thread in the program, it will execute an instrumentation function that transfers the branch's information to the monitor. This function is inserted by the compiler for the branches identified as similar by the algorithm in Section 3.2.

- The function appends the branch information to the thread-specific front-end queue of the monitor (recall that in a shared memory architecture, the entire address space is visible to all the threads), without taking a lock. The function returns immediately after the insertion.

- The monitor thread asynchronously removes the branch information from the thread-specific front-end queues in round robin fashion. No

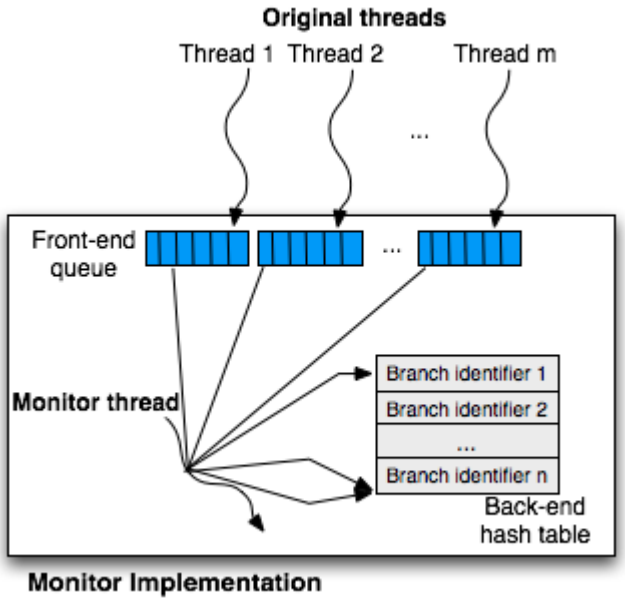

Figure 3.3: Architecture of the runtime monitor in BlockWatch

lock is required as the removal is done from the front of the queue while the insertion is done at the back. Further the queues are of fixed length[5], so there is no need to dynamically allocate memory.

- The monitor thread inserts the branch information into the back-end hash-table using the identifier of the branch as the key (see below). Thus, all instances of a given branch across different threads will occupy the same entry in the hash table.

- Once all threads have reported the outcomes of a specific branch, the monitor checks them by reading the hash table entry corresponding to the branch.

[5]We set the queue length to a sufficiently large value to prevent it from being a bottleneck. This value can be modified if needed.

Instrumentation: We instrument the similar branches identified by the static analysis algorithm in Section 3.2 with calls to our custom library, which send the branches' runtime behaviours to the monitor.

We illustrate the instrumentation with an example. Figure 3.4 shows the instrumentation added for *branch 4* in Figure 2.1. Recall that this branch belongs to the *partial* category. The library calls are highlighted with boldface in Figure 3.4, and consist of the following two functions.

- *sendBranchCondition*: Sends the branch condition to the monitor, so that the monitor can check if all threads for which the condition variable is identical, have the same branch outcome.
- *sendBranchAddr*: Sends the branch address to the monitor, so that the monitor can compare the target addresses of all threads for which the condition is the same.

In both cases, the functions send the static branch identifier, the outer loop iteration number, and the thread ID. The former two fields are used to find the hash table key of the branch, while the thread ID is used to identify which thread sends the data.

Hash table Key: The hash table key of a branch is obtained by combining its static identifier with a runtime identifier. The static identifier encodes the static position of the branch in the program. Each branch within a function or loop is assigned the same static identifier. The runtime identifier distinguishes among different instances of the branch in different loop iterations and at different call sites (through instrumentation). This is obtained by dynamically encoding the call stack corresponding to the parent

```
1  void slave() {
2    ...
3    sendBranchCondition(4 /*static branch ID*/, procid,
4      private /*condition*/, loop_iter );
5    /* loop_iter here means the loop iteration
6    number of all outer loops*/
7
8    //Branch 4: Partial
9    if (private > 0) {
10     sendBranchAddr(4 /*static branch ID*/, procid,
11       TAKEN /*behavior*/, loop_iter);
12     ...
13   } else {
14     sendBranchAddr(4 /*static branch ID*/, procid,
15       NOTTAKEN /*behavior*/, loop_iter);
16   }
17 }
```

Figure 3.4: Example code to show the instrumented program

function's invocation and the loop iterations of outer loops. The combination of the static and runtime identifier yields a unique hash table key for each runtime instance of a branch. This key is used to store the information about the branch in the hash table by each thread that executes it.

As shown in Figure 3.5, we implement the hash table as a two level table. In the first level, the function's call site ID (added by instrumented code) and the static branch identifier is used to generate the key. In the second level, the loop iteration number of all outer loops is used to generate the key. We separate the function's call site IDs and the loop iteration numbers to achieve better utilization of the memory and reduction of access times.

Performance Optimizations: For performance considerations, the monitor executes asynchronously and does not affect the program execution. The only thing that the original program needs to do is to send the branch

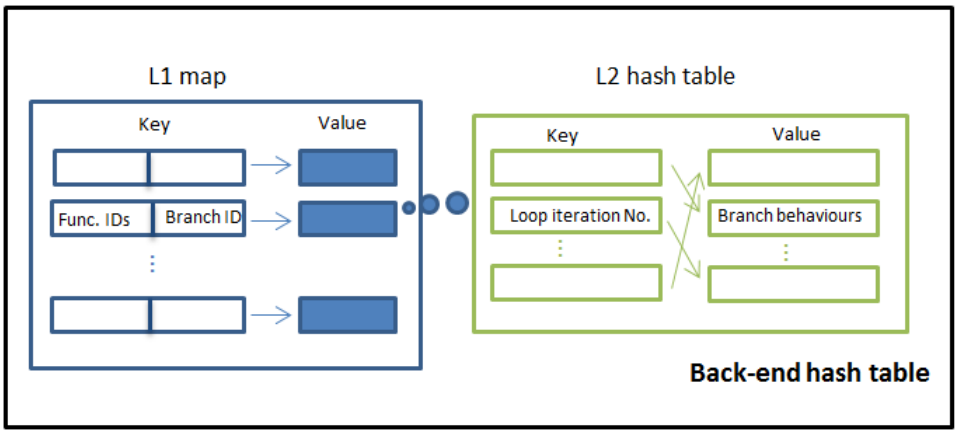

Figure 3.5: Architecture of hash table of the monitor

runtime behaviours to the front-end queue of the monitor. In order to further improve performance, we do two main optimizations to reduce the time to send the branch behaviours:

1. *Multi-core optimization:* Since we need the loop iteration numbers and call site identifiers of different threads to generate runtime identifiers for the hash table, we store the information of all threads in the same object. In this case, data of different threads may belong to the same cache line, and there will be multiple copies of the cache line when different threads access this line from different processor cores. According to the cache coherence protocol [4], a thread in one processor core will invalid the cache lines of another thread on another core when it writes to the cache lines which have another copy on that core. Because of the invalidation of cache line, other threads have to re-fetch the data in the memory. This is called false sharing and it leads to the increase of execution time [4]. To address this problem,

the multi-core optimization aligns the data of different threads to different cache lines and removes the false sharing of the loop iteration number and call site identifiers of different threads.

2. *Multi-processor optimization:* Multi-core optimization removes the false sharing in cache across different threads and reduces the overhead of BlockWatch on multi-core processors. However, when the program runs on a multi-processor machine, threads running on different processors might store their neighbours' loop iteration number and call site identifier information on the processor's last-level cache. Therefore, there exists false sharing in the last-level cache and a change in the information in one processor invalids the information of other processors. Similarly, threads running on different processors have to spend extra time to re-fetch the data in the memory. Therefore, we use thread-local object to store data of different threads and ensure they are not put on the same last-level cache line, which further improves the performance on multi-processor machines.

3.4 Summary

In Chapter 2, BlockWatch identified the control-data similarity patterns in SPMD programs and divided them to four categories, and this chapter detailed the implementation of studying the similarity categories of the branches and using them for runtime error checking. The first step in the implementation is the static analysis. In this step, because the branches that we focused on depend on either shared variables or thread ID (See Table 2.1

in Chapter 2 for details), we inferred the branches' similarity category by studying the propagation of shared variables of thread ID in a compiler at compiling time. The second step is runtime checking. In this step, we first instrumented the branches at static time to send their runtime behaviours. At the same time, we implemented a asynchronous, lock-free monitor as an separate thread which receives the runtime behaviours sent by different threads and compare the actual runtime behaviours' of the branches with the inferred behaviour according to the branches' similarity categories.

Chapter 4

Experimental Setup

4.1 Introduction

In this chapter, we first describe the tools used in implementing BlockWatch. Then we describe the benchmarks used to evaluate BlockWatch. Finally, we discuss how we evaluate the performance and the error coverage of BlockWatch.

4.2 Implementation Tools

We implement BlockWatch using the LLVM compiler infrastructure [24]. LLVM is a compilation infrastructure for lifelong program analysis and transformation. It has an intermediate representation (IR) that uses Static Single Assignment (SSA) form. The IR is manipulated by our custom passes before being compiled to machine code. We first compile the program to LLVM IR and apply BlockWatch's static analysis to: (1) analyze the program's IR and find the similarity category for each branch; (2) instrument the program's IR with calls to our custom library. For each of the benchmarks, the static analysis and instrumentation passes take less than 1 second on a quad-core core i7 machine with 8 GB RAM. Finally, we compile

the instrumented IR to machine code on our target platform. We also use the Boost library's hash table in the runtime monitor's implementation [22].

In the static analysis pass, we study the propagation of shared variables and thread ID (see Section 3.2 for details). To study the propagation, it is essential to get the definition-use chain of the program variables. This is intuitive for non-pointer variables, while it is a challenge for pointers. The reason is that pointer is not statically binded to a object and it might point to different objects for different runs. For instance, a variable a is binded to the object a, while a pointer p may point to the object a or the object b depending on runtime configurations. Since a pointer may point to several objects, we need to collect the objects the pointer may point to and understand the definition-use chain of each variable. Resolving the objects a pointer may point to requires a compiler technique named pointer alias analysis.

There are some widely-used alias analyses, such as Andersen's alias analysis [3], Steensgaard's alias analysis [39] and Choi's alias analysis [11]. We choose Andersen's alias analysis as our alias analysis tool, because studies [20] show that Andersen's alias analysis is precise (compared with Steensgaard's analysis) and efficient (compared with Chois's analysis). In order to use Andersen's analysis, we upgrade an existing Andersen's alias analysis to the latest LLVM. The upgrade mainly consists of two parts. First, we add support for dynamically allocated object (e.g. allocation through *malloc*() function) in the existing analysis; Second, we create a mode that is enabled when the analysis is performed on the whole program during compilation. The mode is more precise because it assumes the all objects in the program

can be seen at the compilation stage and hence makes a more aggressive assumption.

4.3 Benchmarks

We use the SPLASH-2 Benchmark Suite [45] for evaluating BlockWatch. The SPLASH-2 Benchmark Suite has been extensively used for studies of shared memory parallel programs. There are nine applications in the SPLASH-2 Benchmark Suite, and we choose six of them for the evaluation. Table 4.1 describes each application and explains why the other three applications are not included. There are also four kernels which are smaller in the suite, and we include two kernel programs FFT and radix sort for evaluation because they are small and can be used for more detailed study (See Chapter 5 for details).

Table 4.1: Description of application programs of SPLASH-2 benchmark suite

Program name	**Description**	**Included in evaluation? If not, why?**
Barnes	The application implements the Barnes-Hut algorithm to simulate the interaction of a system of bodies (galaxies or particles, for example) over a number of time-steps.	No. The application does not have any output, so we cannot collect SDC result, which is the focus of our evaluation (Section 4.5).

Continuous ocean	The application studies the ocean movements under the influence of eddy and boundary currents. The implementation uses dynamically allocated 4-D array for grid data storage.	Yes
FMM	The application also performs N-body simulation, but the algorithm is adaptive Fast Multipole Method.	Yes.
Non-continuous ocean	The application solves the same problem as continuous ocean, but the implementation uses statically allocated 2-D array for grid data storage.	Yes.
Radiosity	The application computes the equilibrium distribution of light using iterative hierarchical diffuse radiosity method.	No. The output is in binary format and is non-deterministic. Therefore, we are not able to distinguish between an SDC and correct output.

Raytrace	The application use ray tracing to render a 3-D scene.	Yes.
Volrend	The application uses a ray casting technique to render a 3-D volume.	No. In order to reduce the time and space overhead in using loop iteration number to calculate the monitor's hash table identifier (Chpater 3), we made an implementation choice that we check branches whose loop depth is less than 5. This is applicable for most programs, but it does not hold for Volrend.
Water-nsquared	The application evaluates forces and potentials that occur over time in a system of water molecules. The algorithm is an $O(n^2)$ algorithm.	Yes.

Water-spatial	The application solves the same problem as water-nsqared, but it is an $O(n)$ algorithm	Yes.

We use the default configurations of the suite except that we vary the number of threads in order to study the scalability of BlockWatch. Table 4.2 describes the characteristics of the evaluated programs. In the table, the parallel section refers to the part of the program which is executed concurrently by two or more threads. Because BlockWatch relies on the similarity across threads to detect errors, we focus on the parallel section of the program in reporting the similarity categories assigned to branches.

Table 4.2: Characteristics of benchmark programs

Program name	**Total lines of code (LOC)**	**LOC in parallel section**	**Total number of branches**	**Number of branches in parallel section**
continuous ocean	5329	4217	876	785
FFT	1086	561	110	44
FMM	4772	3246	395	321
non-continuous ocean	3549	2487	543	478
radix	1112	441	99	35
raytrace	10861	7709	726	268
water-nsquared	2564	1474	144	103
water-spatial	2756	1154	202	143

4.4 Performance Evaluation

We evaluate the performance overhead of BlockWatch on a 32-core processor that contains four 8-core AMD Opteron 6128 processors running at 2 Ghz each. In order to study the performance overhead and the scalability of BlockWatch, we vary the number of threads from 1 to 32 and measure the time spent in the parallel section of the program, both with and without BlockWatch. We do not measure the checking time of monitor thread, as the monitor thread is executed asynchronously and hence does not have a significant effect on the execution time of the program's parallel section. The SPLASH-2 programs can scale to at least 64 threads [45].

To measure the performance with 32 threads, we disable the monitor thread during the execution of the main program so as not to interfere with it. This is because our machine has only 32 cores and we need 33 threads to execute the program with the monitor [6]. We have verified that the difference in execution times is negligible under this scenario for the 16 thread case. Note that the threads still send the branch information to the front-end queues of the monitor - the only difference is that the monitor does not do anything with the information.

4.5 Coverage Evaluation

We evaluate the error detection coverage of BlockWatch through fault injection studies. Specially, we focus on detections of Silent Data Corruptions

[6] We cannot set the thread number to 31 because the SPLASH-2 benchmarks require the number of threads to be a power of 2.

(SDCs). SDCs are failures in which the program finishes executing but the output deviates from the golden result in an error-free run. In this paper, we focus on SDCs because crashes and hangs can be easily detected through other means (e.g., heartbeats). Further, the program can be restarted from a checkpoint upon a crash or a hang, and continued.

We build a fault injector with the PIN tool [34]. PIN is a dynamic instrumentation framework for programs on X86 processors. The goal of the fault injector is to simulate transient hardware faults that propagate to a *branch* instruction in exactly one thread of the program. We focus on branch instructions because BlockWatch targets hardware faults that propagate to the control data of programs (i.e., data used by branches) in this study.

The fault injection procedure consists of three steps. First, we instrument an m-thread program using PIN and record the number of branches executed by each thread of the program at runtime (say n_i where $0 < i < m$). In the second step, we randomly pick a thread from 1 to m, say j, and choose the j^{th} thread to inject faults. Then we select a number from 1 to n_j, say k, and choose the k^{th} branch of j^{th} thread at runtime to inject. Thirdly, we flip a single bit in either the flag register or condition variable of the chosen branch instruction of j^{th} thread. The former fault leads to the branch being flipped, i.e., going the wrong (but legal) way. This is to verify the correctness of BlockWatch in detecting branch runtime behaviour deviations. The latter fault may or may not lead to the branch being flipped. For example, a fault in a branch condition that flips the least significant bit of the condition variable, may not affect the comparison being performed by the

branch. However, the corruption introduced in the condition variable will persist even after the execution of the branch, and is more representative of hardware faults in the control data. This is to verify that the effectiveness of BlockWatch in detecting control-data errors. Only one fault is injected in each run of the program to ensure controllability.

Because PIN can monitor all executed instructions in the program, the fault injection considers *all branches* in the program, and is not restricted to those that are instrumented by BlockWatch. However, we do not consider the instrumentation added by BlockWatch for injection, as errors that affect these branches can at worst lead to additional crashes or hangs, but not to SDCs, as they do not affect the program.

After injecting the fault, we track its activation and whether it is detected by the monitor. If not, we let the program execute to completion (if it does not crash/hang), and compare the results with the golden result to measure the SDC percentage.

For each experiment, we inject 1000 faults of each type and count how many faults are activated (over 75% of the injected faults are activated in our experiments). We calculate the coverage as the probability that an activated fault will not lead to an SDC [1]. In other words, $coverage = 1 - SDC_f$, where SDC_f is the fraction of activated faults that lead to an SDC. Thus the coverage includes faults that lead to program crashes or hangs as well as masked faults. In reality, even an unprotected program will typically have non-zero coverage due to natural redundancies and memory protections provided by the operating system, and hence we measure the coverage of the program both with and without BlockWatch.

False Positives: Since BlockWatch relies on static analysis for runtime checking, BlockWatch has no false positives. This means that if there is no faults propagating to original program and monitor, the monitor does not report an error.

To verify there are no false positives, we perform 100 error-free runs for each program instrumented by BlockWatch and check if there are errors reported by it. The results show that BlockWatch does not report any errors, i.e., there are no false positives.

4.6 Summary

In this chapter, we described the experimental setup for evaluating BlockWatch. Section 4.2 presented the implementation tool. We chose LLVM as the compiler tool for BlockWatch's static analysis and instrumentation. Section 4.3 presented the benchmark programs (SPASH-2) for evaluating BlockWatch and described the characteristics of the programs. Section 4.4 talked about procedure of performance evaluation and Section 4.5 presented the procedure of using fault injection studies to evaluate fault tolerance of BlockWatch.

Chapter 5

Results

5.1 Introduction

In this chapter, we first present the relative frequencies of the branch similarity categories in the benchmark programs as discovered by BlockWatch. Then we present the performance overheads and error detection coverage of BlockWatch. Finally, we trace back from the final output of two kernel programs FFT and radix for different steps to further understand the relation of backtrace and the performance overhead and coverage of BlockWatch.

5.2 Similarity Category Statistics of Branches

We run the static analysis part of BlockWatch on the eight SPLASH-2 programs. Table 5.1 shows the number of branches in each program that fall into the similarity categories in Table 2.1, as discovered by the static analysis phase of BlockWatch. We also calculate the percentage of the branches that belong to each similarity category based on the total number of branches in the program's parallel section.

The results in Table 5.1 are as follows. In general, between 49% to 98%

Table 5.1: Similarity category statistics of the branches in the benchmark programs

Program	**Total**	**No.(%) of branches of each category**			
		shared	*threadID*	*partial*	*none*
continuous ocean	785	30 (4%)	12 (2%)	723 (92%)	20 (2%)
FFT	44	14 (32%)	11 (25%)	18 (41%)	1 (2%)
FMM	321	51 (16%)	8 (2%)	98 (31%)	164 (51%)
non-continuous ocean	478	22 (5%)	116 (24%)	329 (69%)	11 (2%)
radix	35	11 (31%)	9 (26%)	7 (20%)	8 (23%)
raytrace	268	12 (4%)	4 (1%)	117 (44%)	135 (51%)
water-nsquared	103	34 (33%)	12 (12%)	26 (25%)	31 (30%)
water-spatial	143	36 (25%)	11 (8%)	41 (29%)	55 (38%)

of the branches fall into the *shared*, *threadID* and *partial* categories. This means the BlockWatch is able to statically identify at least 50% of the branches as similar across the seven programs. FMM and raytrace have relatively fewer similar branches, as many branches in these programs have both variables in the branch conditions to be local variables. These branches are identified as belonging to category *none* according to the propagation rules in Section 3.2.

Thus we see that a significant fraction of branches in each program are identified as similar by the static analysis phase of BlockWatch, and are hence eligible for checking in the runtime phase. This shows that BlockWatch can be applied to commonly used parallel programs. Note that our static analysis is rather conservative and hence these are lower bounds on the number of similar branches in a program.

5.3 Performance Overheads

Figure 5.1 shows the execution times of the eight SPLASH-2 programs with BlockWatch for 4 threads and 32 threads. The results are normalized to the execution time of the program without BlockWatch (for the same number of threads), and hence the baseline is 1.0. We name the normalized execution time *slowdown*.

From Figure 5.1, the geometric mean of the *slowdown* of BlockWatch is 2.01X with 4 threads, and 1.15X with 32 threads. Thus *the performance overhead of* BlockWatch *with 32 threads is only 15%*, and is lower than that for 4 threads (see below for why).

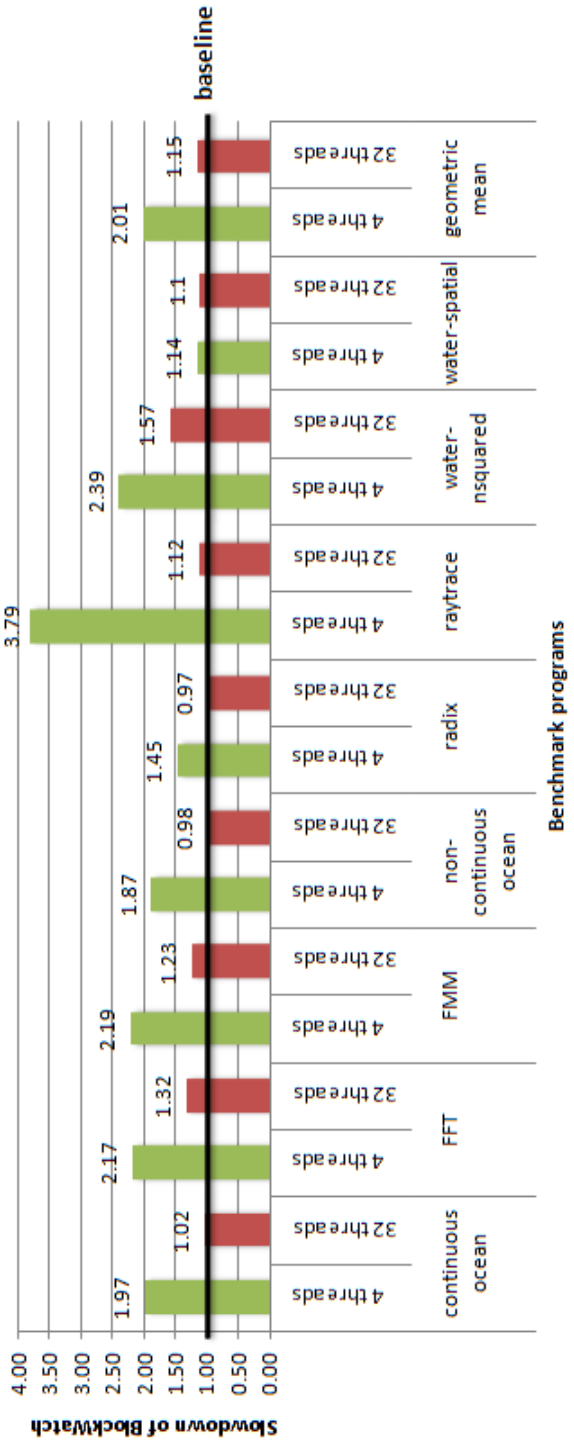

Figure 5.1: Slowdown of BlockWatch. Lower is better

5.3.1 Scalability

We study the scalability of BlockWatch by considering the variation of the geometric mean of the performance overheads (across all eight programs) with the number of threads. The results are shown in Figure 5.2 as the number of threads is varied from 1 to 32.

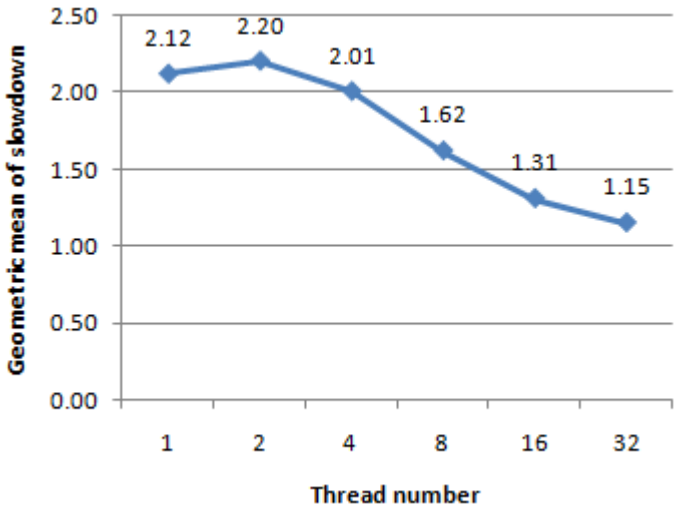

Figure 5.2: Geometric mean of the slowdown of BlockWatch Vs. number of threads

In Figure 5.2, we find that the overhead of BlockWatch first increases as the number of threads increases from 1 to 2, and then decreases as the number of threads increases from 2 to 32. The reason for the overhead increase from 1 to 2 threads is that the machine we use consists of four 8-core processors and is not fully symmetric. This asymmetry causes the memory access time to depend on where the threads execute. When we increase the number of threads from 1 to 2, the operating system assigns the 2 threads to cores in different processors. Thus, the threads cannot share

data at the cache level and the memory access time increases. This hurts the program with BlockWatch more than the original program, and the overhead of BlockWatch increases.

The reason for the decrease of overhead from 2 to 32 threads is that when the number of threads doubles, the work done by each thread reduces by half and so does the number of branches executed by each thread. However, due to communication and waiting among threads, the reduction in execution time of the program is less than 2X. Nonetheless, when the number of threads increases, the relative time spent by BlockWatch reduces and so does the overhead of BlockWatch (up to 32 threads in Figure 5.2).

5.4 Error Detection Coverage

We study the coverage of BlockWatch under two kinds of faults: branch-flip faults and branch-condition faults. The former type of fault is guaranteed to flip the branch but does not corrupt any program data directly. The latter type of fault corrupts the branch's condition data but does not necessarily lead to branch flip.

The results are shown in Figure 5.3 and Figure 5.4. Note that the *coverage* of y axis in both figures start from 50%. In the figures, $coverage_{original}$ is the coverage of the original program, and $coverage_{\text{BlockWatch}}$ is the coverage of the program protected by BlockWatch.

5.4.1 Coverage results for branch-flip faults

Figure 5.3 shows the *coverage* with and without BlockWatch for all programs under branch flip faults. Across the programs, the average $coverage_{original}$ is 86%, while average $coverage_{\text{BlockWatch}}$ is 98%. Other than raytrace, all programs have a coverage value between 99% - 100% when protected with BlockWatch, whereas without BlockWatch, their coverage value is between 60% (radix) and 98% (FMM). *In other words,* BlockWatch *detects almost all branch-flip faults that cause SDCs* for seven of the eight programs.

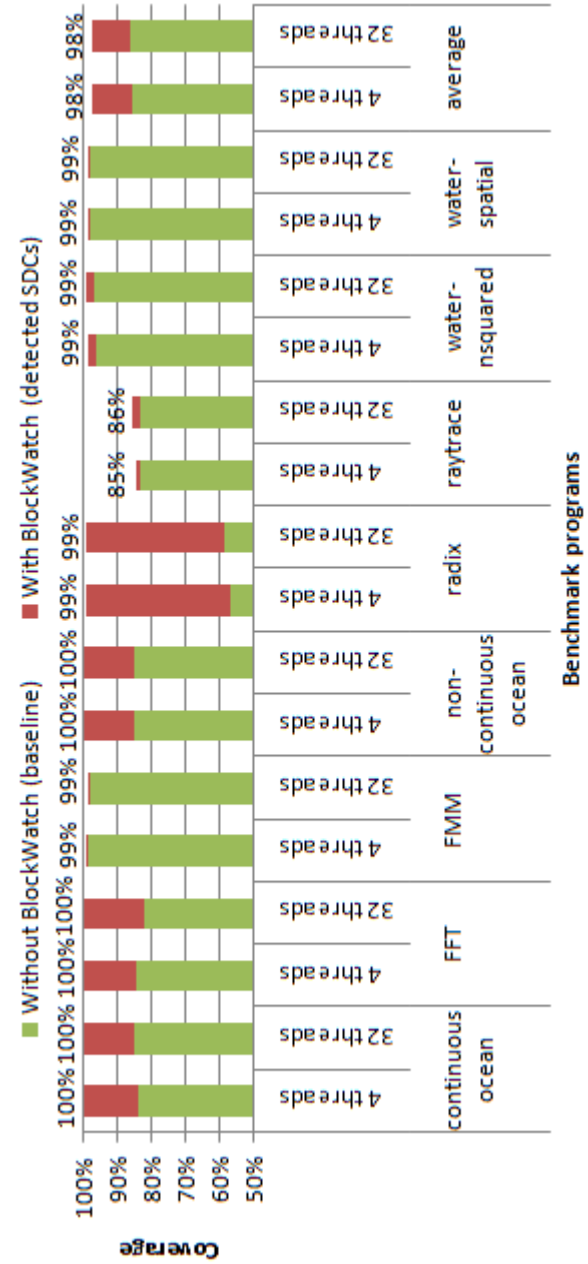

Figure 5.3: $Coverage_{original}$ (baseline) and $coverage_{\text{BlockWatch}}$ (aggregated number) for branch-flip faults: The dark part is due to the detection provided by BlockWatch. Higher is better.

For raytrace, the coverage with BlockWatch is about 85%, which is comparable to the coverage obtained without BlockWatch (for both 4 and 32 threads). Thus, the coverage benefit provided by BlockWatch for this program is negligible. There are two main reasons for this result. First, raytrace makes extensive use of function pointers, that may point to different functions for different threads at runtime. Therefore, the number of threads that execute the same function is low, and hence BlockWatch does not have enough threads to compare at runtime. Second, As we mentioned in Chapter 4, due to overhead considerations, we choose to only check the branches whose nesting levels are smaller than six. In other words, any branch that occurs in loops deeper than six levels of nesting is not checked by BlockWatch. Raytrace has some loops deeper than six levels of nesting which are not checked.

5.4.2 Coverage results for branch-condition faults

Figure 5.4 shows the results of *coverage* of the eight programs both with and without BlockWatch, when faults are injected into the branch's condition data. The results are similar to those in Figure 5.3. For example, when BlockWatch is used, the coverage increases from 91% to 97% for the 4-thread case and from 92% to 98% for the 32-thread case. However, the average $coverage_{original}$ value is between 91% and 92%, which is much higher than the $coverage_{original}$ for branch-flip faults (average 86%). This is because unlike branch-flip faults, branch-condition faults may or may not cause the branch to flip, and branch flips are more likely to lead to SDC in the programs.

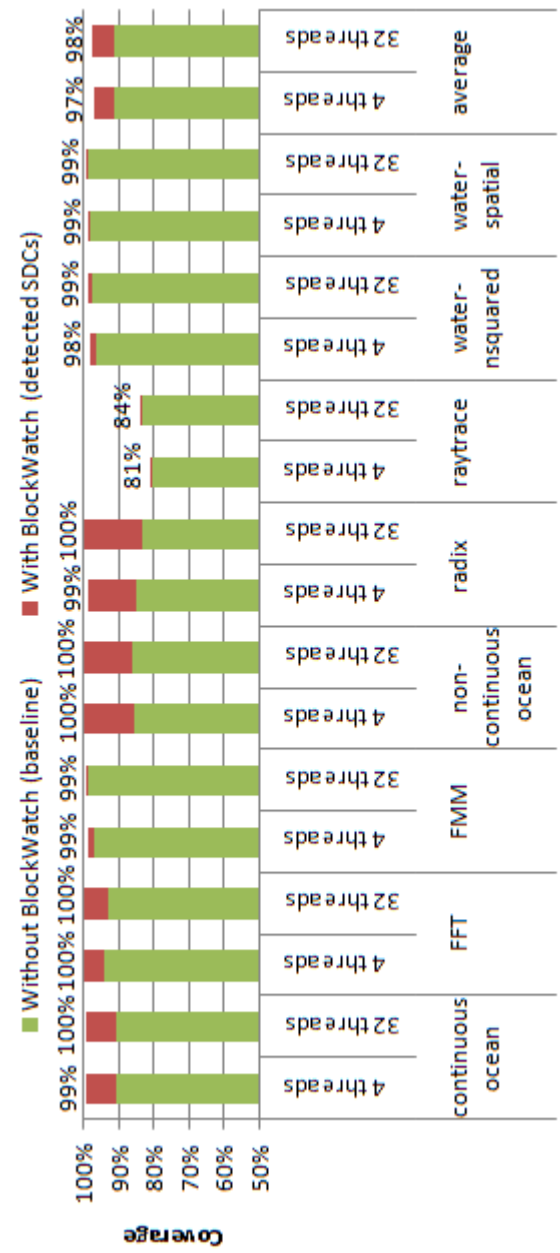

Figure 5.4: $Coverage_{original}$ (baseline) and $coverage_{\text{BLOCKWATCH}}$ (aggregated number) for branch-condition faults: The dark part is due to the detection provided by BLOCKWATCH. Higher is better

Implication of the coverage increase: Figure 5.4 shows that the average coverage increases from 93% to 99% for all programs except raytrace when it is protected with BlockWatch. This means that the average SDC percentage of the programs decreases by 7 times (from 7% to 1%) if the detected SDCs by BlockWatch are recovered. If we assume programs are already protected from crashes and hangs, the system failure rate decreases by 7 times. Since transient faults in computation units are memoryless, we assume the failure distribution is exponential [42]. For exponential model, the mean time to failure (MTTF) of the system is $\frac{1}{failure\ rate}$ [42], and hence the MTTF of the system increases by 7 times.

Availability is an important metric to measure the quality of service (QoS) of a system, and it is expressed as Equation 5.1 [42], in which MTTR is acronym of mean time to repair.

$$availability = \frac{MTTF}{MTTF + MTTR} \tag{5.1}$$

We express the availability of the original system ($availability_{original}$) and availability of the system protected with BlockWatch ($availability_{\text{BlockWatch}}$) as Equation 5.2 and 5.3.

$$availability_{original} = \frac{MTTF_{original}}{MTTF_{original} + MTTR} \tag{5.2}$$

$$\begin{aligned} availability_{\text{BlockWatch}} &= \frac{MTTF_{\text{BlockWatch}}}{MTTF_{\text{BlockWatch}} + MTTR} \\ &= \frac{7 \times MTTF_{original}}{7 \times MTTF_{original} + MTTR} \end{aligned} \tag{5.3}$$

From Equation 5.2 and 5.3, we conclude the relation of $availability_{\text{BLOCKWATCH}}$ and $availability_{original}$ can be expressed as Equation 5.4.

$$availability_{\text{BLOCKWATCH}} = \frac{7 \times availability_{original}}{6 \times availability_{original} + 1} \tag{5.4}$$

If the $availability_{original}$ is 0.99, the $availability_{\text{BLOCKWATCH}}$ is 0.999. This means that when BLOCKWATCH is deployed, the system availability has a big increase, which improves the QoS [26].

5.5 Detailed Study

In this section, we perform a detailed study on two SPLASH-2 kernel programs FFT and radix sort to understand the correlation of final output backtrace and BLOCKWATCH's coverage and overhead. We pick the two programs because their sizes are small and it is easy to understand the results on them. Backtrace here means obtaining the list of instructions by tracing back from the final output of a program for several steps and only detect immediate branches of these instructions. A branch of k backtrace step number means there is at least one instruction in the branch that is used to compute the final output within k steps. The goal of this study is to understand the breakdown of the BLOCKWATCH's coverage and overhead so that programs can choose to get x coverage at the price of y performance overhead when they are protected with BLOCKWATCH. The reason we choose backtrace step number to break down the coverage and overhead of BLOCKWATCH is because intuitively we believed that the closer a instruc-

tion is to the final output, the more likely an error that propagates to the branch outside and affects the instruction will lead to silent data corruption of the final output, since there are fewer opportunities for errors in the closer instructions getting masked. The rationale for protecting the immediate branches of the instructions is that we assume the errors propagated to immediate branches are more likely to impact a instruction compared with other branches.

The experiment is done as follows: we take FFT and radix, and choose backtrace step number 1, 2 and 3. For each backtrace number k, we trace back k steps and get the instructions that are used to compute the final output *within* k steps (including k). Then we find the immediate branches of the instructions and instrument error checking code for the branches that are similar. Finally, we generate the executable for k backtrace steps and evaluate the coverage and performance following the procedure in Chapter 4.

5.5.1 Correlation of output backtrace and coverage

Figure 5.5 shows the coverage of FFT and radix when protected with BlockWatch for different backtrace numbers. For FFT, the coverage increase is almost linear when we increase the backtrace step number. For radix, the coverage increases more when we increase the step number from 2 to 3 when compared with when we increase the step number from 1 to 2. The results are counter-intuitive as they show that detecting instructions that are far away from the final output can be equally (FFT) or more beneficial (radix) in error detection coverage.

In Figure 5.6, we study the coverage of FFT and radix when protected

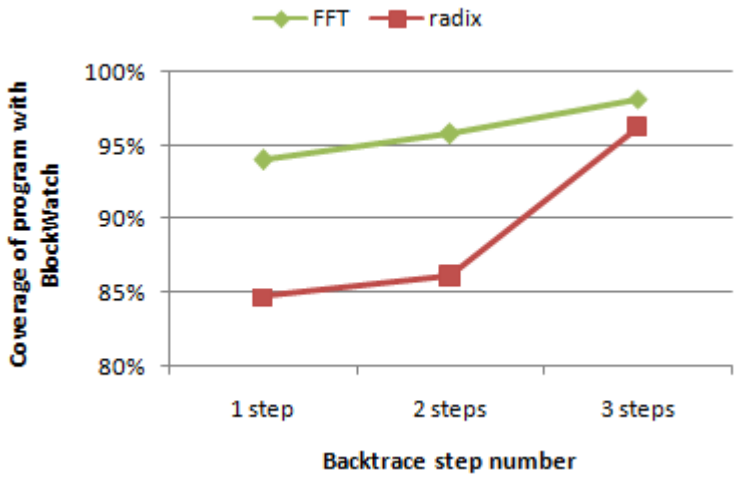

Figure 5.5: $Coverage_{\text{BlockWatch}}$ for branch-condition faults Vs. backtrace step number. Thread number is 32

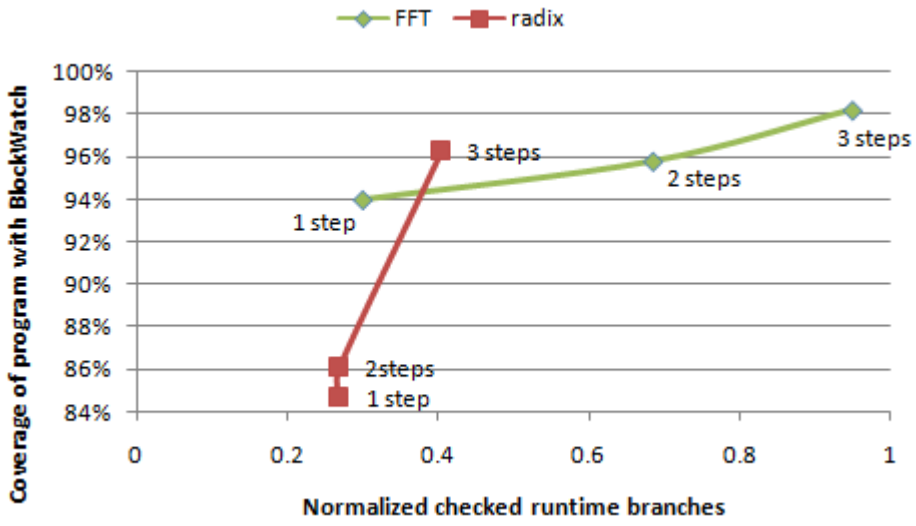

Figure 5.6: $Coverage_{\text{BlockWatch}}$ for branch-condition faults Vs. normalized checked runtime branches. Thread number is 32

with BlockWatch for different checked runtime branches. The goal of this study is to understand whether each runtime branch is equally like to lead to SDCs. The checked runtime branches in Figure 5.6 are normalized to the total runtime branches in each program. From the figure, we see that for FFT, when we increase the checked branch number, the coverage also increases linearly. For radix, however, the coverage increases by about 2% when we check *negligible* more branches from backtrace 1 step to 2 steps, while the coverage increases less dramatically when we backtrace from 2 steps to 3 steps. The results show that protecting some branches provides higher coverage benefit than others.

From Figure 5.5 and 5.6, we conclude that protecting some branches provides higher coverage than others, but the importance of the branch is not related to whether the immediate instructions within the branch are closer to the final output. Therefore, if we want to get more coverage by protecting a proportion of the branches, it is beneficial to find these kinds of branches first and protect them. One future direction that we plan to explore is to loosen the restriction of detecting immediate branch to detecting all branches outside of the selected instructions.

5.5.2 Correlation of output backtrace and performance overhead

Figure 5.7 shows the slowdown of BlockWatch for different step number. The variation of the performance overhead is so small that we are not able to infer any overall trend.

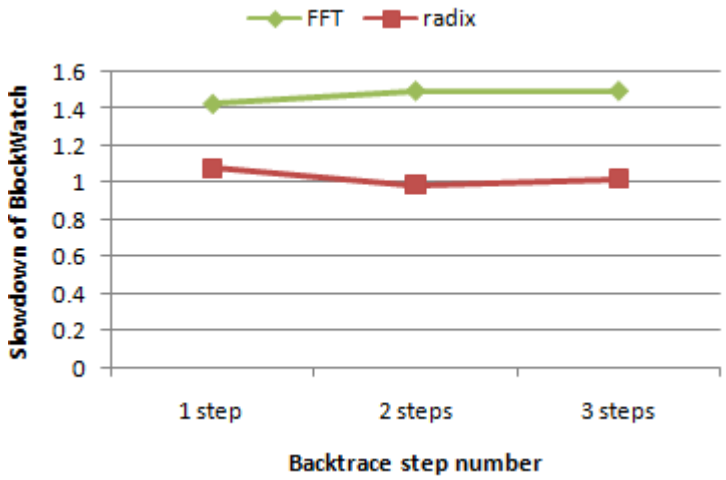

Figure 5.7: *Slowdown* of BlockWatch Vs. backtrace step number. Thread number is 32

5.6 Summary

In this chapter, we presented the result of the evaluation. We found that BlockWatch is able to identify between 50% and 97% of branches as similar and can be checked. Then we presented the performance overheads and error detection coverage evaluation and found that BlockWatch is able to provide more than 98% coverage with the performance overhead as low as 15% (for 32-thread case). Finally, we studied the correlation of the step number of final output backtrace and the coverage and overhead with two kernels in SPLASH-2. The result showed that protecting some branches is more beneficial than others in terms coverage, but the benefit does not depend on whether a branch is an immediate branch of instructions which have shorter distance to the final output.

Chapter 6

Discussion

6.1 Introduction

In this chapter, we compare the error detection coverage and performance overhead of BlockWatch with that of software-based duplication. Duplication, or running two copies of a program and comparing their outputs, has been used to detect errors in sequential programs [35]. The main advantage of duplication is that it is simple to apply and requires no knowledge of the application. However, duplication has two main disadvantages when applied to parallel programs. First, parallel programs are often non-deterministic, and duplicated versions of a parallel program may yield different results, thus rendering them ineffective for error detection. We illustrate it with a simplified *pthread* program in Figure 6.1. When duplication is directly applied to detect errors in control-data of branch 1, the same thread of the original version and duplicated version may acquire the lock differently and hence lead to different i for them. The difference in i may lead original version and duplicated version to make dfifferent decision at branch 1 even in an error-free run. Second, duplication requires twice the amount of hardware resources, and hence reduces the resources available for the actual program, thus leading to significant slowdowns [47]. In this chapter, we

compare BlockWatch with duplication in terms of error detection coverage and overhead. Although duplication is a general technique that can protect programs from a large class of errors, we focus on control-data errors in this chapter as this is the focus of BlockWatch.

```
1  #define N 10
2  int i = 0;
3  int a[N];
4  void slave() {
5    pthread_mutex_lock();
6    i++;
7    pthread_mutex_unlock();
8    // branch 1
9    if (a[i] < N) {
10     ...
11   }
12 }
```

Figure 6.1: An example to show why duplication cannot be directly applied to parallel programs

6.2 Coverage

Our results show that BlockWatch improves the SDC coverage of the SPLASH-2 programs under both branch-flip faults and branch-condition faults. Other than raytrace, all programs have a coverage value between 98% and 100% for errors in the control data. This indicates that when the program is protected with BlockWatch, the percentage of SDCs is less than 2% for seven of the eight programs. To our knowledge, duplication is the only other generic technique that can provide near 100% coverage for SDCs. However, it has other disadvantages (see below).

The coverage results can be improved in several ways: for example, we use a fairly conservative method to classify the branches' category in this study, the result of which is that there are some branches that may have runtime similarities but are not checked by BlockWatch. Therefore, it is possible to improve the coverage of BlockWatch by using a more aggressive static analysis or by incorporating the program's dynamic information in the classification of the branches.

6.3 Performance Overhead

The average performance overhead of BlockWatch is 101% for 4 threads and 15% for 32 threads. In contrast, software-based duplication incurs overheads of 100% to 200% for sequential programs [47]. Although this overhead can be reduced through the use of speculative optimizations [47], doing so is not straightforward for parallel programs due to their non-determinism. Thus, the overhead of BlockWatch is comparable to that of software-based duplication in the 4-thread case, *but is almost an order of magnitude lower in the 32-thread case.*

Further, BlockWatch is scalable while duplication is not. This is because duplication requires program determinism, which may not hold for parallel programs. This problem can be solved by using determinism inducing techniques [6, 31]. However, determinism inducing techniques require the replica threads and the programs' threads to follow the same execution order. Forcing execution order among threads incurs communication and waiting overheads that are proportional to the number of threads in the

program, and does not scale. In contrast, BlockWatch scales as it neither requires program determinism nor locking.

BlockWatch can be optimized to further reduce its overhead. For example, our current implementation adds checks for every branch that is eligible for checking. However, there may be many branches that depend on the same set of variables, and faults propagating to the data will affect all of them. Therefore, it is sufficient to check one of the branches. Moreover, we showed in Section 5.5 that protecting some branches are more beneficial and hence we can protect a subset of the branches and reduce the overhead while getting good enough coverage.

As we scale BlockWatch to higher numbers of threads, it is possible that the monitor itself becomes a bottleneck. To alleviate this, we can have multiple monitor threads structured in a hierarchical fashion, each of which is assigned to a sub-group of branches. Figure 6.2 shows one possible solution to solve the problem. In this case, we have multiple monitor threads, and each of them is responsible for different static branches, which can be encoded at static time or decided at runtime. This solution alleviates each monitor thread's pressure while comparing all threads' runtime behaviours on the same branch together, which helps BlockWatch maintain the same coverage. This is a direction for future work.

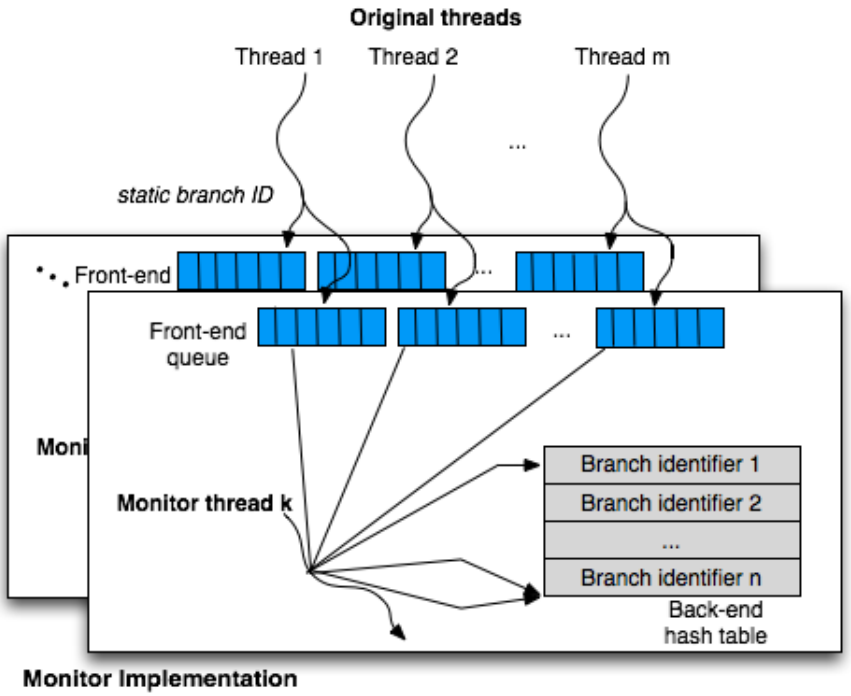

Figure 6.2: One possible improved version of the current monitor

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis presented BlockWatch to detect control-data errors in SPMD parallel programs. BlockWatch statically infers the similarity of the program's control-data across threads, and checks their conformance to the inferred similarity at runtime. Upon detecting a violation, it raises an exception and reports the error. We implement the static analysis part of BlockWatch with LLVM and the runtime checking part by creating an asynchronous monitor thread. Experimental results on SPLASH-2 programs show that BlockWatch increases the average SDC coverage across eight programs from 86% (91%) to 98% for branch-flip faults (branch-condition faults), while incurring only 15% overhead in the 32 thread case (on a 32 core machine). BlockWatch is automated, incurs zero false-positives and can run on unmodified hardware, thus making it suitable for today's multi-core processors.

7.2 Future Work

Future work will improve BlockWatch in three directions: extending BlockWatch to other classes of parallel programs (than pthreads-style programs), and to other program data (in addition to control-data). We will also explore optimizations to reduce the performance overhead of BlockWatch even further.

As mentioned in Chapter 2, the similarity that BlockWatch leverages for error checking comes from the SPMD program structure and the shared data across tasks (threads). This kind of similarity exists in other SPMD or Single-Instruction-Multiple-Data (SIMD) programs, such as CUDA programs (through global variables) and MPI programs (through message passing). Therefore, we plan to extract the similarity in these programs for error detection.

Although the current implementation of BlockWatch focuses on control-data, it can be extended to detect faults that propagate to regular instructions. Studies have shown that around 80% of the runtime instructions in SPMD parallel programs exhibit similarity [25], which means they can be used by BlockWatch for error detection. In the future, we will extend BlockWatch to protect other program data.

In this study, the overhead of BlockWatch is about 15% when the thread number of the programs is 32 (on a 32 core machine). However, as we mentioned in Chapter 6, this overhead can be further reduced. In the future, we plan to adopt the solutions mentioned in Chapter 6 to further reduce the performance overhead of BlockWatch.

Bibliography

[1] R. Alexandersson and J. Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 303–314, 2011.

[2] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.

[3] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, the University of Copenhagen, 1994.

[4] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer System*, 4:273–298, 1986.

[5] S. Bagchi, Z. Kalbarczyk, R. Iyer, and Y. Levendel. Design and evaluation of preemptive control signature (PECOS) checking. *IEEE Transactions on Computers*, 2003.

[6] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *IEEE Symposium on Reliable Distributed Systems*, pages 250–255, 2002.

[7] S. Borkar and A.A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[8] Shekhar Borkar. Thousand core chips: a technology perspective. In *the Design Automation Conference*, pages 746–749, 2007.

[9] G. Bronevetsky, I. Laguna, S. Bagchi, B.R. de Supinski, D.H. Ahn, and M. Schulz. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 231–240, 2010.

[10] Z. Chen, Q. Gao, W. Zhang, and F. Qin. Flowchecker: Detecting bugs in MPI libraries via message flow checking. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[11] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[12] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[13] Frederica Darema. The SPMD model: Past, present and future. In *the European PVM/MPI Users' Group Meeting*, page 1, 2001.

[14] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Survey*, 34:375–408, 2002.

[15] Q. Gao, F. Qin, and D.K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[16] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors. *Journal of Parallel and Distributed Computing*, 2002.

[17] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *the International Conference on Software Engineering*, pages 291–301, 2002.

[18] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and Changhong Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *2001 Symposium on VLSI Technology*, pages 73–74, 2001.

[19] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 135–144, 2002.

[20] Michael Hind and Anthony Pioli. Which pointer analysis should i use?

In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[21] K.H. Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, pages 518–528, 1984.

[22] B. Karlsson. *Beyond the C++ standard library.* Addison-Wesley Professional, 2005.

[23] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[25] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *IEEE/ACM International Symposium on Microarchitecture*, pages 337–348, 2010.

[26] D.A. Menasce. Composing web services: A QoS view. *IEEE Internet Computing*, 8(6):88–90, 2004.

[27] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller. Problem diagnosis in large-scale computing environments. In *ACM/IEEE Conference on Supercomputing*, pages 88–100, 2006.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 41(6):308–319, 2006.

[29] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In *Proceedings of Conference on Computer systems*, pages 343–356, 2011.

[30] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[31] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ACM SIGPLAN Notices*, volume 44, pages 97–108, 2009.

[32] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *IEEE International On-Line Testing Symposium*, pages 211–216, 2007.

[33] J.S. Plank, Y. Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *the International Symposium on Fault-Tolerant Computing*, pages 351–360, 1995.

[34] V.J. Reddi, A. Settle, D.A. Connors, and R.S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *the Workshop on Computer Architecture Education*, 2004.

[35] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software implemented fault tolerance. In *the International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[36] S.K. Sahoo, M.L. Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 70–79, 2008.

[37] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 161–170, 2010.

[38] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Morgan & Claypool Publishers, 2009.

[39] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[40] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[41] D.D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F.T. Chong. Characterization of error-tolerant applications when protecting control data. In *IEEE International Symposium on Workload Characterization*, pages 142–149, 2006.

[42] Kishor S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., 2nd edition edition, 2002.

[43] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B.R. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2010.

[44] J. Wei and K. Pattabiraman. BlockWatch: Leveraging similarity in parallel programs for error detection. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[45] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36, 1995.

[46] K.S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *IEEE Parallel & Distributed Processing Symposium*, pages 287–300, 2011.

[47] Y. Zhang, J.W. Lee, N.P. Johnson, and D.I. August. DAFT: decoupled acyclic fault tolerance. In *the International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, 2010.